David Mitchell Jennings
JXSHELL: a Web-based expert system platform
(Under the direction of Dr. Donald Nute)

The logic programming principles of Expert Systems (ESs) allow for the construction of an ES engine that both functions correctly from a theoretical point of view and complies with the generally accepted principles of good Web design. Thus I built JXSHELL, a Web-based ES architecture, as a platform for multiple ES deployment over the Web. JXSHELL includes a clear model for ES development that strictly separates knowledge base logic and interface logic, and JXSHELL-based deployments involve a minimum of server-side and client-side processor load.

Index words: Expert systems, Logic programming, Web applications, Prolog, Java, XML, XSL, XSLT

JXSHELL: a Web-based expert system platform

by

David Mitchell Jennings

B.A., St. John's College, 1993

A Master's Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science in Artificial Intelligence

Athens, Georgia

2002

JXSHELL: a Web-based expert system platform

by

David Mitchell Jennings

Acknolwedgements

If it had been up to me, this thesis would have never been written. I owe a debt of gratitude to those who encouraged and at times cajoled me to get this thing done: my mother Barbara Jennings, my sister Nancy Jennings Aburto, and my boss and friend Rick Hitchcock. My advisor Dr. Don Nute helped me paragraph by paragraph, and his rare combination of theorectical knowledge and good sense was, and is always, an inspiration. My other committee members, Dr. Charles Cross and Dr. Michael Covington, exhibited Job-like patience. And finally Melissa, though she never set eyes on this work, has helped me more than she knows.

Preface

This paper introduces the JXSHELL expert system architecture. JXSHELL is a stateless, Web-based architecture that can be used by developers to create expert systems delivered over the Web to users. Typically, users will use a Web browser, but the only constraint on JXSHELL clients is that they be able to interpret HTML and be able to return HTTP compliant GET strings to the server or servers running a JXSHELL application. JXSHELL applications avoid many potential problems of other Web-based applications, such as high server overhead, and can leverage many server technologies, such as load distribution. In addition, JXSHELL allows for a clean separation of duties between interface development, based on XSLT processing, and knowledge base development, based on a small subset of the Prolog programming language.

In this paper, I first discuss the design principles of the JXSHELL architecture in light of how both expert systems and the Web function. Then I go on to provide practical guidelines for developing JXSHELL applications.

Tim Berners-Lee, the original designer of the HTTP protocol and one of the Web's chief theorists, has famously claimed that the Web is a "semantic Web," but that it is not AI (Artificial Intelligence). We hope to show through JXSHELL that though the Web may not be AI, it is particularly well-suited to hosting development in the AI subdiscipline of Knowledge Based Systems.

Because expertise is by its nature limited, expert knowledge will generally be limited in its applicability to real life situations because of the simple fact that experts are greatly outnumbered by those who may benefit from expertise. Thus *expert systems* have been developed within the field of Artificial Intelligence as a way of allowing experts to disseminate the benefit of their judgements to people who would otherwise not have access to those judgements. Expert systems are computer programs that generate solutions to problems based on input combined with an expert's knowledge. *Consultative* expert systems are expert systems that generate user interactions of some sort for eliciting input. *Rule-based* expert systems are expert systems that codify expert knowledge as a set of rules. In the following we will consider consultative, rule-based expert systems specifically, and the term "expert system" will apply to these, unless stated otherwise.

In a consultative, rule-based expert system, the expert's knowledge is realized programmatically as a set of rules that the program processes in such a way that appropriate questions are asked of the user until a decision is reached. The process of query and answer mimic the steps by which a naive user approaches an actual expert to resolve a problem or answer a question. The set of rules that represent expert knowledge is known as a *knowledge base.*

Typically, expert knowledge is embodied in a program as follows: an expert, perhaps together with a knowledge base developer or by using a software tool, generates a set of rules. An expert system developer incorporates that set of rules into a larger program. The resulting program processes the rules by establishing which rules, if any, hold for a given set of conditions that are supplied by a user. Generally, a rule includes a *rule head* that represents a complete answer and a set of one or more *conditions*, each of which when processed by the program generates a user interaction. The user's interaction, for example answering "yes" to a yes and no question, establishes whether the condition in question is satisfied. If it is, then the system is one step closer to establishing that the given rule holds. If the condition is not satisfied, then the system will automatically proceed to the next rule in an attempt to establish whether it holds. The end result is either a satisfied rule resulting in an expert recommendation, or a “no rules satisfied” state, meaning that no answer can be returned based on user input. Additionally, expert systems are often set up to allow users to try to reach multiple recommendations based upon their input, and sometimes expert systems are designed to give the user the choice of seeing what rule from beginning to end was satisfied to generate a given answer. (JXSHELL expert systems can include both of these features.)

The basic procedural constraints on an expert system software is that it “remember” user input (thus avoiding needlessly repeating an interaction); that it process rules in some order so that once a rule is established *not* to hold it is never revisited (thus avoiding loop conditions); that it generate appropriate interactions; and that it have some way of reading user input. The basic logical constraint on an expert system is that program state together with user supplied information uniquely determine a next user interaction or an end state. (This last constraint applies to all consultative expert systems.) To understand this latter constraint more fully, we will consider below expert system interactions as *logic programs*. To understand the procedural constraints in the light of Web-based systems, later we will consider first the added constraints presented by HTTP (the stateless protocol upon which the Web is built) and then the way in which the JXSHELL architecture meets them. Finally, we will evaluate JXSHELL in light of its practical advantages as an Expert System architecture, and we will compare it to similar professional systems.

Table of Contents

Chapter 1

Logic Programing: a brief overview

We will consider an expert system as a type of *logic program.* Logic programs, as described in Hogger [2], are typically computer programs that process a *definite program* using a resolution scheme. Definite programs are ones whose clauses are all definite, meaning that each clause in the program contains exactly one positive literal (e.g., xkb_identify(3)[1]) and zero or more negative literals (e.g., ¬prop(dorsal_crest)). Thus the following is a definite clause:

> xkb_identify(7) or ¬prop(dorsal_crest) or ¬parm(color,m,4)

as is this:

> xkb_identify(7)

For our purposes, it is important to point our that a definite clause with zero negative literals is a fact (condition), as mentioned earlier, while one with more than zero literals is a rule. The second assertion is proven by the following simple logical equivalences (here expressed in predicate logic):

> (x if (y and z)) iff (x or ¬(y and z))

and

> ¬(y and z) iff (¬y or ¬z)

[1]These examples and similar ones throughout this paper are based on the JXSHELL reference implementation. See Appendix A.

If x is does not hold, then either y or z (or both) do not hold. Thus our informal description of expert systems, as consisting of rules and sets of facts, describes a logic program, provided that expert systems also include a resolution scheme.

Resolution is an inference rule applicable to clausal form logic [2, page 73]. Intuitively, resolution is a procedure that can be applied to a set of clauses to derive another clause. The action of a logic program is to attempt to prove an assertion by deriving the null clause (conventionally, $\square$) from a definite program and the *negation* of the given assertion: $P \cup \{\neg q\}$ (where P is a definite program and q is the definite clause that is being derived). The negation of the clause to be derived is known as a *query* and is often expressed as

$$?\, q$$

The question mark is "syntactic sugar" for the logical negation symbol $\neg$. The inference of q is accomplished by deriving the *null clause* or $\square$ from $P \cup \{\neg q\}$. Intuitively, we say that we know that given a definite program P, q holds if $\neg q$ and P derives $\square$. Or in other words, that the supposition of $\neg q$ and P is inconsistent. This supposition is guaranteed by the following proof theoretic theorem:

$$P \vdash q \iff P \cup \{\neg q\} \vdash \square$$

Resolution works by composing a set of *resolution steps*, each one consisting of two clauses (or *parents*) from a set of clauses, the only constraint being where one parent contains some literal A the other parent contain the negation $\neg A$ of that literal. Here A and $\neg A$ are known as *complements*. The next step of the resolution, or the *resolvent*, consists of the disjunction of the two parent steps minus a pair of complements. Hogger [2, page 73] gives as the following simple example of a resolution step:

parents:	mother or father or ¬parent
	male or ¬father
resolvent:	mother or male or ¬parent

The last step in a successful resolution will be of the form:

parents:	¬parent
	parent
resolvent:	□

In real-world applications, the typical resolution method is SLD-resolution, for *S*election *L*inear *D*efinite resolution. The Prolog programming language [3, 1] includes SLD-resolution as its built-in inference procedure. The *Definite* in SLD-resolution simply indicates that programs subject to this procedure are definite. The *Linear* indicates that each resolution step uses as one parent the most recent resolvent (the *center clause*) and as the other parent (known as the *side clause*) a clause containing a complement to some literal in the center clause.

For our purposes the most interesting aspect of SLD-resolution is the *S*, or the *selection*, aspect. The *S* indicates that SLD-resolution utilizes a fixed selection rule for deciding which literal in the center clause is resolved upon. Thus for the following center clause:

¬parent or ¬male

the program uses some selection rule to decide whether to pick as a side clause one that resolves upon "¬parent" (i.e., contains a "parent" literal) or resolves upon "¬male" (i.e., contains a "male" literal). Whichever literal is chosen, it will be according to a consistently applied rule. Now consider the case of an expert system, in which the definite program P consists of rules of the form:

father(bob) if male(bob) and parent(bob)

As we have seen, this rule is logically equivalent to father(bob) or ¬male(bob) or ¬parent(bob)

Now we apply a simple computation rule, namely *leftmost selection.* This means that in the center clause we attempt to resolve upon the leftmost disjunct. We thus begin with the query

? father(bob)

The resolvent here is

¬male(bob) or ¬parent(bob)

Next we resolve on ¬male(bob), and so on until we reach □ or some non-resolvable clause. In the following we assume leftmost selection as our resolution selection rule.

SLD-resolution is commonly understood as a *search tree* in which each edge signifies a resolution step, and all nodes other than the root node are resolvents. By convention these trees are represented with the root node at the top. SLD-resolutions, or more simple logic program computations, either fail infinitely (i.e., loop), or they end either with a *successful computation* or a *finitely failed computation*, meaning a step in which the resolvent fails to resolve with any program clause. In tree representations we will indicate a successful computation with the null clause symbol, □, and we will indicate a finitely failed computation with a special symbol, ■.

The resolution tree represented in 1.1 is built on the query

? father(bob)

with the two rules

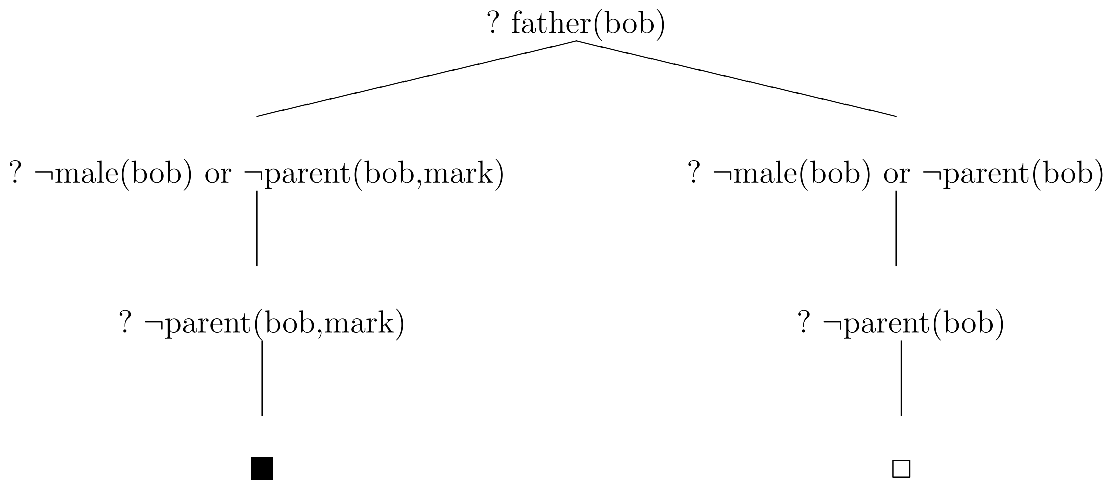
father(bob) if male(bob) and parent(bob,mark)

father(bob) if male(bob) and parent(bob)

and as input

parent(bob)

male(bob)


? father(bob)
? ¬male(bob) or ¬parent(bob,mark)
? ¬male(bob) or ¬parent(bob)
? ¬parent(bob,mark)
? ¬parent(bob)
■
□
Figure 1.1: An exceedingly simple SLD-resolution tree.

Note that every resolvent is itself subject to a next resolution step, and each resolution step, whether or not it is from the top level query, is the same type of computation. Thus every node is modeled as a query (indicated by ?).

Resolution trees are abstract representations of a logic program computation. However, they can also be understood more programatically as representations of the actual steps taken during program execution. The program attempts to reach a successful computation, or □, by running down branches of the tree going from top to bottom, and left to right.[2] Upon reaching □, the computation finishes sucessfully. Upon reaching ■, the computation *backtracks* to the last branching node and continues down the first branch (from left to right) not previously tried at that node. If there are no branches to backtrack to, the computation finitely fails. Typically

[2]This is true in the case of systems, such as the Prolog programming language [1, pages 10-11], that employ left-to-right, depth-first search. For the sake of simplicity, this is the type of search assumed throughout this paper.

systems will often include *forced backtracking*, so that upon reaching □, the user can force the system to backtrack to find more answers.[3]

Understood logically, the clauses of a logic program define *relations*. However, relations are purely logical. That is, a program understood as an expression of relations does not in itself imply any specific *computation*. To understand logic programs as *programs*, we interpret a logic program as a type of search. Specifically, the search is for an answer to a query given some computational state. Furthermore, we must extend our understanding of SLD-resolution tree processing beyond search order to the construction of the trees themselves. That is, even in our quasi-procedural representation above, we leave obscure the procedure involved in building the trees that are searched. To understand tree building, we must introduce the notion of a *search rule* for a logic program. A search rule defines the behavior of the program's inference engine over the program's clauses and program input when defining the search tree used for finding answers to queries. Note that an SLD-resolution tree implies an order for computation, namely left to right, as mentioned. SLD-resolution (with some given computation rule) gives us a means of generating *branches* in a search tree, but the search rule gives us the shape, i.e. the order of computations, for a search tree.

For instance, assume a leftmost computation rule and the following search rule:

Search Rule: select clauses in the text order shown below.

```
father(bob) if (male(bob) and parent(bob,mark))
father(bob) if (male(bob) and parent(bob))
```

```
parent(bob)
male(bob)
```

[3]JXSHELL also includes forced backtracking. Administrators enable forced backtracking for particular applications through a configuration setting; see Appendix B.2 for details.

The resulting tree will be what we have already encountered, namely 1.1.

However, with the following search rule:

Search Rule: select clauses in the *reverse* text order shown below.

father(bob) if male(bob) and parent(bob,mark)
father(bob) if male(bob) and parent(bob)

parent(bob)
male(bob)

the resulting tree 1.2 is somewhat different. Note that both trees are correct. The difference is computational, not logical.

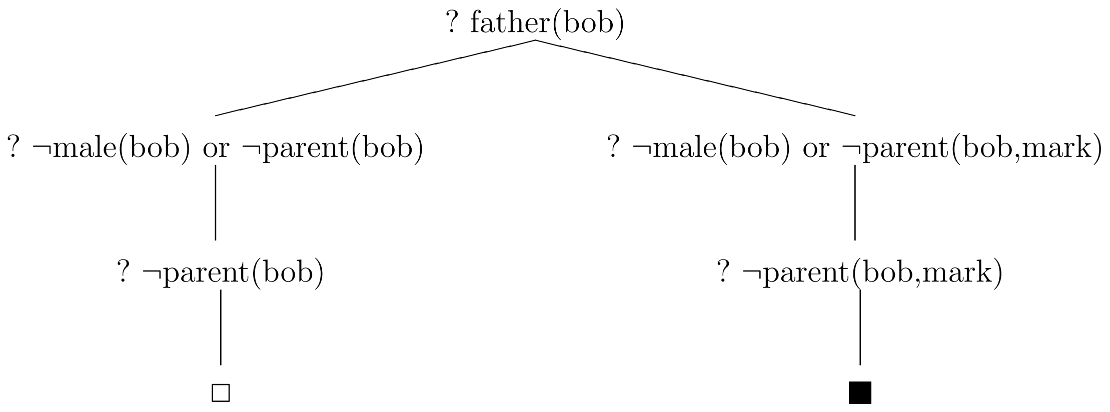

? father(bob)
? ¬male(bob) or ¬parent(bob)
? ¬male(bob) or ¬parent(bob,mark)
? ¬parent(bob)
? ¬parent(bob,mark)
□
■

Figure 1.2: An SLD-resolution tree utilizing an alternative search rule from 1.1

For a more involved set of examples (featuring variable instantiation and different computation rules), refer to [2, pages 110-112].

We can specify a computation rule by defining the rule by means of a *meta-program* interposed between the program interpreter and the *object program* (here understood as $\{P, Q\}$, where P is a given program and Q is a query [2, pages 105-108].) Typically, a logic program interpreter will impose its own fixed computation rule. For our purposes we will always assume the presence of a search computational meta-program. As it turns out, such a meta-program will prove important in our understading of JXSHELL as an appropriate platform for Web-deployment of expert systems.

In the context of logic programming we will refer to *program state*, or simply *state*, as anything that the program uses to generate a search tree. As the source from which resolvents are chosen, the clauses of a program will be part of state. In some logic programming applications, state is nothing more than the clauses of the program and user inputs. In these systems, the search rule for generating resolution trees is to apply some *lexical* criteria to the set of program clauses. For instance, in a case where there are two clauses that are equally appropriate as a branching node at a given level of the tree, a program that uses lexical criteria may choose for the leftmost node at that level the first clause as it is represented in the program. The rule would state "for constructing subtrees choose the clauses in this order," where "this order" refers to the order in which the clauses are represented in the knowledge base. Hogger [2, page 110] refers to such systems as relying on a given *text order* for clauses.

Given added constraints upon clauses, a meta-program can leverage some given lexical convention to build resolution trees. Consider the resolution tree already presented in figure 1.1.

We can consider this tree as implicitly indexed, where each branching node is assigned some number (beginning with 1), and every node on a level that branches has an a index one higher than the node immediately to its left. (See figure 1.3.)

In a system using explicit indexing, the indexes are made part of the clauses themselves, so the tree in 1.3 would be generated from the following rule base:

father(bob)$_1$ if male(bob) and parent(bob,mark)

father(bob)$_2$ if male(bob) and parent(bob) where the original input specifies that the tree be built from the first rule. In the Prolog programming language, where clauses closely resemble the representation that we use here, the rule base would appear as follows:

```
father(bob,1) :- male(bob), parent(bob,mark)
father(bob,2) :- male(bob), parent(bob)
```

For the meta-program to construct resolution trees it simply needs to start from a specified rule index to build trees where the clause for a branching node is chosen by the ordering of rules on their indexes. The lexical ordering of clauses is immaterial.

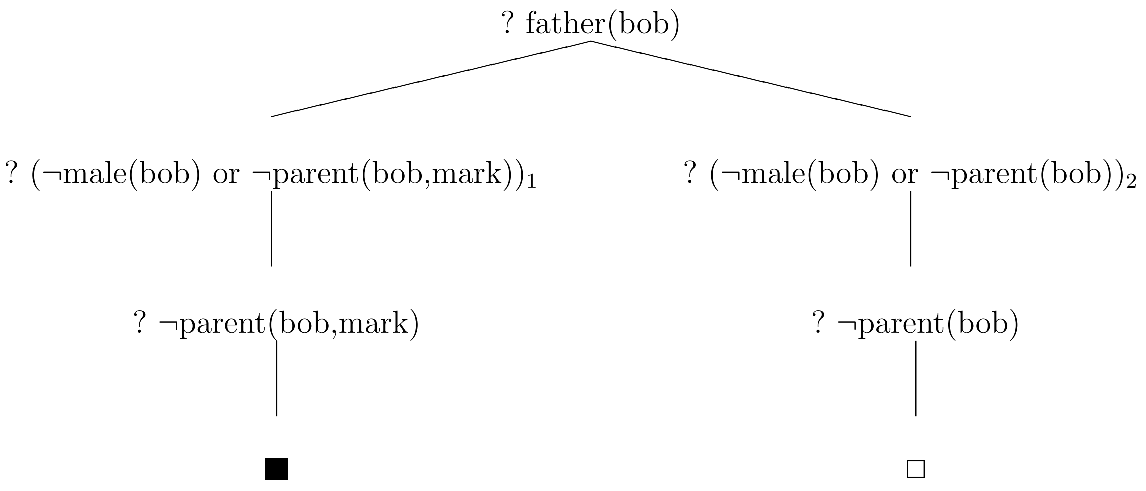

? father(bob)
? (¬male(bob) or ¬parent(bob,mark))1
? (¬male(bob) or ¬parent(bob))2
? ¬parent(bob,mark)
? ¬parent(bob)
■
□

Figure 1.3: An indexed SLD-resolution tree

Under a system employing rule indexing, we can consider the index as a parameter for the tree building meta-program. The inclusion of parameters for the meta-program is an important aspect of the JXSHELL system in so far as meta-program parameterization allows for programs that maintain only static state between user interactions, as explained more fully below.

So far, we have only considered logic program computations over programs without variables. For full generality, logic programming includes allowance for universally quantified expressions, particularly for rules. Thus instead of relying on the awkward rule

father(bob) if male(bob) and parent(bob,mark)

a real application would greatly extend the expressiveness of the program, meaning the domain which is picked out by the rule, by using variables (here denoted in Prolog-style with capital letters):

father(X) if male(X) and parent(X,Y)

The rule now reads that for *every* X and *every* Y, it is the case that X is a father if X is male and X is the parent of Y. To use a universal quantification in an SLD-resolution step, it is necessary to apply a unification, denoted θ, to the expression, meaning that variables are substituted according to a set rule. The details of unification are beyond the scope of this paper; for our purposes it is sufficient to point out that the unification employed in logic programming is such that the *refutation completeness* of SLD-resolution is preserved. [2, pages 81,97-101] This means that the system will derive $\square$ from any set of definite clauses provided that the set is unsatisfiable.

To understand logic programs as computational systems (as opposed to simply as logics), unifications are divided into two types: ingoing, denoted $\theta_{\mathbf{in}}$ and outgoing, denoted $\theta_{\mathbf{out}}$. Queries on the program are of the form $q_1 \& q_2 \& \ldots \& q_2$, where & denotes logical conjunction. Each conjunct of the query is interpreted procedurally as a call to a procedure, so that we read the query as some sequence $< C_1, C_2, \ldots, C_n >$, where each C_i is some q_j interpreted as a procedure call. The call to C_i *invokes* a clause in the programs clause set, here interpreted as a set of procedures. The general form of procedures is

C if B_1 & … & B_m

here C is a *procedure heading* and (B_1 & … & B_m) is a *procedure body.* The conditions under which a clause is invoked by a call is that C_i and C unify, and that the program scheduler determines that the clause invoked has the highest priority among all other untried clauses. Under this interpretation, ingoing unifications are those that apply to the clauses body upon a call. That is, $\theta_{\mathbf{in}}$ *passes data* from the call C_i to the invoked clause. Outgoing unifications apply to the query by means of the call, meaning that $\theta_{\mathbf{out}}$ passes data from the clause to the query (upon the call activated by the query which invokes the given clause).

In the case of a system in which a parameterized meta-program determines program execution, calls *indirectly* invoke clauses. (The indirection of calls is what is meant by calling the meta-program an interposition between the program and the interpreter.) A query *plus* other input parameters will invoke the meta-program itself, and the meta-program will construct a call that will invoke a program clause. The invocation process performed by the meta-program is the same as that outlined above. However, in the case of rule-indexed systems it is useful to introduce yet another type of unification, namely that of index unification, or $\theta_{\mathbf{index}}$. Like $\theta_{\mathbf{in}}$, $\theta_{\mathbf{index}}$ is an ingoing instantiation, but $\theta_{\mathbf{index}}$ is restricted in domain (viz. to integers greater than zero).

Chapter 2

Expert Systems as Logic Programs

2.1 A procedural interpretation for expert systems

Now we are ready for the procedural interpretation of an expert system: the expert system produces a resolution based upon a set of user inputs. The system generates a resolution by first finding a rule from the knowledge base using a fixed search rule. For instance, in an indexed rule system the meta-program finds a rule whose index matches the index passed by the user as a parameter. The system takes the body of this rule as its query and resolves against the query using the information passed by the user. If the rule fails, meaning if there is a user input that fails a condition in the rule, then the meta-program moves to some "next rule" using a fixed search rule. For instance, in an indexed system the meta-program moves to the rule with a next higher index. Upon finding a next rule, the system resolves upon the rule's body using the user-supplied inputs. If there is no input to resolve upon for some step in the resolution, the system generates a request for appropriate input (possibly including meta-program parameters) and sends the request to the user. If the rule succeeds, the system returns to the user a conclusion of the successful rule, possibly along with appropriate meta-program parameters. If the last rule fails, then the system sends to the user an indication of failure. In all cases, if the system sends to the user meta-program parameters, those parameters are returned to the system by the user upon acting upon a system-generated request, and the parameters are in turn used by the meta-program to correctly continue knowledge base processing.

Because resolution is understood as the generation and processing of a search tree, every step in a consultation is in effect the generation and processing of a search tree based upon user input.

The user inputs that provide the system with items to resolve rule bodies upon can be partial or can be complete. In the former case, the user provides some definite clause that is added to a set of definite clauses already supplied by the user and kept in a local memory store (see 2.2 below). The local memory store is maintained between user interactions. In the latter case, the user provides the entire set of definite clauses. This set will grow between user interactions, and the system will include the current set as part of the request for additional user input. This set of definite clauses, whether it is passed in full from the user or kept in a local memory store, is known as the *set of knowns* for an expert system.

2.2 Program memory for expert systems

We define *program memory* for expert systems as a (possibly empty) store that must be kept between user interactions to correctly generate the next tree. (Note that program memory is not the same as computer memory, though the implementation of a procedure that involves program memory will entail using computer memory. Computer memory will be used regardless, because in all cases the program itself, as well as whatever is needed for resolution, will require some "memory footprint.") Program memory may include meta-program tree building parameters and/or the set of knowns (less the current user input). However, program memory as we define it does not include the set of definite clauses that make up the original program, that is the program represented in source code.

We note that it is possible to construct a memory free implementation for expert systems. In this implementation, the user passes to the program both tree building parameters and the set of knowns. In all cases there is an inverse relationship between program memory and user input: the more information included in user input, the less program memory is required. The case that is minimal in regard to program memory (i.e., program memory is empty) is maximal in regard to user input (i.e., user input includes as much information as the meta-program can use to construct and compute SLD-resolution trees.)

The tree building meta-program can be parameterized in different ways. In the maximal case (i.e., that in which program memory is empty and user input is maximal), there are two different parameterization schemes that suggest themselves based on our treatment so far. In one scheme the user passes the entire program (in such and such lexical order) to the meta-program as a parameter, and the search rule for the system is simply to process the rules in lexical order. The other scheme uses rule indexing as discussed earlier, and the meta-program uses a supplied index to correctly generate a search tree. The second scheme entails a further constraint on ES knowledge bases, namely that they include an explicit index for rules, and that the meta-program is more complex in that it must use a calculation on the supplied index to generate a search tree. However, the second scheme gives us a crucial advantage over the former in that the parameters passed by the user do not grow with the size of the rule base, but only the number of knowns.

The ES modeled without program memory relies entirely upon meta-program parameterization for correctly constructing SLD-resolution trees. This means that there is a functional relationship between user input, understood as a string representing all necessary parameters, and program state. We can say that the program's input string *determines* program execution. We can also say that any correctness condition for the program is also a correctness condition for input strings. That is, we can say what a program should do under some given input string. (Note that how input strings represent input items, and how those strings are parsed, are system dependent considerations, and as such do not enter into a theorectical understanding of ESs. Here we also assume finiteness of input strings, something that can be proven for rule-indexed systems even in the case of infinite rule bases.)

2.3 A correctness criterion for expert systems

The correctness condition for an ES is that given a definite program and an input string, that string correctly determines an SLD-resolution tree. To understand this, we need to consider what *determination* means in this context. Up to now we have considered each ES interaction as corresponding to a separate resolution tree. However, we can also consider determination in a different way, namely in terms of the "original" resolution tree, i.e., that generated without a side clause and with the lowest rule index. We will do some "hand waving" here, but suffice it to say that a simple proof will show that every possible generated SLD-resolution tree for the program will be a *sub-tree* of this original tree. In other words, every generated tree will have as its root node a branching node in the original tree. Thus resolution tree determination can be thought of as determination of some branching node of the original tree. Every input string *points out* a branching node of the original tree.

Assuming left-to-right depth first search, we can say what correct determination is: the branching node (of the original tree) pointed out by the input string will be such that the input parameters represented by the string would produce a complete resolution on any branches to the *left* of the node pointed out. Stated in terms of rules, no rule will be considered for satisfaction or failure until all previous rules have either been satisfied or unsatisfied. This condition determines that no answer will be missed.[1] For a memory free, rule-indexed ES, then, the correctness condition

[1]There is a stronger correctness condition that stipulates that all leftward subtrees will *fail* given the current input parameters. However, this condition is too strong because we want to allow for multiple answer ESs, i.e., those that allow the user to search for more than one way of reaching an answer given a set of inputs.

determines that upon backtracking (i.e., upon jumping to a rightward branch of the original resolution tree), the system increments the rule index, and that the system never increments the rule index otherwise. Rule index incrementation can be forced by the user, but only upon rule satisfaction, and a correctly built ES will only automatically increment the index upon rule failure. In the case of JXSHELL, this constraint is realized programmatically.

Given the possibilities for Web-based ES deployment, the principle design decision for an ES platform is twofold:

1. How much information should be maintained in state between interactions?

2. How much information should be passed to the program from interactions?

JXSHELL follows the "maximal case" in regard to information passed from interactions and the "minimal case" in regard to program memory. This design decision is made to avoid the necessity of maintaining program memory, as explained in depth in the next section.

To sum up, JXSHELL ESs are well-formed logic programs. Each resolution step uses Prolog's inference mechanism, viz. that of leftmost resolution, depth-first search. Each search tree subject to this mechanism is generated using a well-defined meta-program, viz. that of lexical indexing. Provided that the rules of a Prolog ES obey the given constraints (viz. are lexically indexed and non-recursive), JXSHELL ESs will act according to the informal constraints discussed earlier. Furthermore, JXSHELL includes a pre-processor that analyzes given rule-bases for conformance to the specified constraints, meaning that any ES that runs under JXSHELL will necessarily be well-formed in our earlier sense.

Chapter 3

Expert System interactions as Web objects

As we have seen, ES platforms can be correct while utilizing either meta-program parameters or a memory store, and parameterized systems can include rule indexing or explicit ordering. In the case of parameterized (i.e., memory free) systems, rule indexing is clearly the better choice due to the greater brevity of call expressions. However, there is no *logical* or *semantic* reason that adjudicates between memory store and parameterized systems. The design imperative (if there is one) must be, then, procedural in the sense that some aspect of the context of procedure calls on the system will decide the issue. It is at this point that it is important to remember that our investigation concerns Web-based expert systems. The context in which the ES applications under consideration are invoked is over the Web.

In the following, we describe *addressable Web objects*, according to the working notes of Tim Berners-Lee. This investigation shows that principles of good Web design favor an ES platform that does not maintain program memory between interactions.

3.1 The Web as URI space

The Web is a space whose loci are defined by *Universal Resource Indicators*, or *URIs*. A Web object is whatever can be referenced using a URI. For instance, an HTTP address such as "http://www.expertsystems.com/index.html" is a URI referencing an object, typically an HTML document. The HTTP specification delineates how the components of an HTTP URI are interepreted; however, the importance of a URI is its relation to Web objects as a referring expression to a referent. According to Berners-Lee, this notion gives us a rough and ready definition of the Web: "An object is 'on the Web' if it has a URI." Berners-Lee also calls Web objects "First Class Objects (FCOs)," a term which in the theory of computing languages refers to objects which can be referenced or dereferenced in any context like primitive variables.

3.1.1 The "axioms" of URI space

Berners-Lee has developed a set of axioms on the URI space as part of informal notes on the Web and related topics. These axioms are not meant to be mathematically rigorous, but rather they are heuristic in that they illuminate the thinking behind and the constraints upon Web development. Because the Web is ill-defined, we have a certain amount of freedom in interpretting what it is. However, the implication behind our analysis is that the axioms are prescriptive as well as descriptive — that is, the axioms define the URI space of the future. Following the spirit of Berners-Lee's work, we leave unclear just to what extent this treatment is one or the other. We do show, however, that any URI space that is compliant with Berners-Lee's axioms will be a space which includes loci for expert system interactions.

Axioms 0 and 0a: Universality

The axiom of universality is divided into two parts:

1. *Axiom 0: Universality 1:* Any resource anywhere can be given a URI.

2. *Axiom 0a: Universality 2:* Any resource of significance should be given a URI.

Both illuminate the design and implementation constraints on the Web in regard to applicability. In terms of design, Axiom 0a specifies that objects delineated in any resonable definition of URI space will be such that they can be URI referenced. In terms of implementation, Axiom 0a specifies that the URI scheme must be flexible enough to allow for reference to many types of objects such as static Web pages, database tables and rows, and external entities such as books. Axiom 0b stands as a challenge to site developers to include within their corners of the URI space all the objects that can go there. Here the descriptor "significance" is necessarily vague. URI addressing, however, is strictly defined, meaning that site developers' work is cut out for them at least regarding the method of referencing.

Axiom 1: Global scope

The axiom of global scope reads:

> It doesn't matter to whom or where you specify that URI, it will have the same meaning.

There is no concept of "scope" for URIs. That is, there is no context within the URI space in which a URI will refer to a different object than in any other context. Note that this does not mean that all expressions for URIs are themselves fully articulated. For instance, relative URLs are elliptical expressions for full URLs, the latter being a type of URI. The context of execution determines how the elliptical URL is expanded, but the expansion, that is, the URI for the object referenced, is itself context-free.

Axioms 2a and 2b: Sameness/identity

The axiom of sameness/identity is divided into two parts:

1. *Axiom 2a: Sameness:* A URI will repeatably refer to "the same" thing.

2. *Axiom 2b: Identity:* The significance of identity for a given URI is determined by the person who owns the URI, who first determined what it points to.

Berners-Lee applies the term *idempotence* to systems of reference that conform to 2a. However, the concept of idempotence in the URI space is necessarily fuzzy given the mutable nature of objects in that space. For a simple example, again consider HTTP URLs. A URL can point to some given page from time t_1 to time t_3, but if at some intermediate point t_2 the page referenced is altered, then the objects referenced at t_1 and t_3 are different in some sense. Axiom 2b simply states that this "some sense" is defined by the originator of the URL. Furthermore, a well designed URI space architecture will provide mechanisms to the URI owner for conveying the conditions of sameness of identity of objects, e.g. by means of a time stamp.

3.1.2 The GET rule

The concept of idempotence is fuzzy, but fuzzy within well defined bounds. In particular, we can precisely delimit the relative idempotence preservation of Web based systems. Specifically, we can define *idempotence under revision (IUR)* or *soft idempotence*[1] as the conformance to Axioms 2 given that there will be a given amount of revision to all objects within the URI space. That is, if we suppose that all URI addressed objects change at t_n such that $t_n \bmod 3 = 0$, then soft idempotence mandates that at t_{n+1} and t_{n+2} a given URI will reference the same object. Here, the sameness of the objects is itself a difficulty, but one that can be resolved *in any particular instance* by means of metadata over the objects of URI space. That is, the metadata of an object will specify the criteria by which sameness of reference to the object will or will not be established. This auto-definition mechanism is consistent with Axiom 2b, and is a current topic of Web architecture research.

[1]The terminology here is mine, though the discussion does not go beyond what is implied in Berners-Lee's notes.

Clearly our definition of soft idempotence or IUR is ideal. There is in fact no mechanism which guarantees a set time step for revisions of objects residing in the URI space. Also, the definition is fuzzy in that the concept of "revision" is not defined. However, the definition does allow us to specify when IUR is broken for any given context, provided that our notion of "context" includes the revision status of the object. A specific example should provide some clarity: suppose that the object that we are interested in is a simple text file, and that revision of that file is understood – intuitively enough – to mean the editing of the file. Now suppose that the file is edited at t_5. The system under which the file is dereferenced preserves IUR if and only if the page is presented the same at t_1,t_2,t_3, and t_4, i.e., the text file presented to the user is identical byte-for-byte.

The concept of soft idempotence or IUR becomes clearer if we consider Berners-Lee's rules for the HTTP GET operation:

1. *GET rule 1:* In HTTP, GET must not have side effects.

2. *GET rule 2:* In HTTP, anything which does not have side-effects must use GET.

GET rule 2 preserves the universality axiom by disallowing POST operations from accessing first class objects. A POST operation does not generate a URI, so if a POST operation renders a first class object – i.e., a URI space object – then that operation is an instance of presenting an object without a reference, thereby betraying Axiom 0. A GET operation, on the other hand, generates a valid URI, as evidenced by what shows up in the "Location" field of a Web browser in performing such an operation. But the object rendered must be shown IUR, given that it is a first class object. Thus GET must be an operation such that future dereferencing is not affected, i.e. rule GET rule 1 must hold.

The GET rules are particularly important in designing systems that interoperate with HTTP servers to present Web objects. In such situations, the GET query string is constructed with the specific purpose of giving the system under consideration parameters by which to uniquely identify an object for presentation via the HTTP server. The IUR preservation inherent in GET sets a dual constraint on system design:

1. *The system must parse the relevant sections of the GET string in such a way that IUR is preserved.*

2. *The GET operation must generate the semantically correct strings.*

Here, "relevent sections" refers to everything after the domain specification in the URI, including everything after the `?` in an HTTP URL.

3.1.3 How ES interactions can be Web objects

Assuming that a Web-based ES operates via an HTTP server, we can develop a plan for presenting ES interactions as Web objects, bearing the GET constraints in mind. Two points should be considered:

1: ES interactions must be addressible. An ES interaction must be *evoked* by a URI. That is, a GET string must be sufficient for correctly producing a computation in the ES program. As shown earlier (Chapter 2), we can say with some precision what a correct evocation is, namely one in which a GET string invocation will point out a node in an SLD-resolution tree in accordance with our ES correctness criteria 2.3. For a GET string to be sufficient as an evocation mechanism, it must as well represent all of the information necessary for a correct call.

2: ES interactions must be IUR. A given GET string must evoke the same ES interaction at every instance. This condition is satisfied if the system in question alters neither its definite program, its meta-program, nor its program memory between computations.

In light of these considerations, it is clear that a program memory free implementation is the best choice for Web-based ESs. Such an implementation satisfies the first condition by relying exclusively upon information passed by a call for construction of a correct SLD-resolution tree. This implementation satisfies the second condition by the nature of its design; calls *only* construct SLD-resolution trees without affecting the program in other ways.

3.2 Summary

The JXSHELL design is justified by a consideration of the Web. Its memory free implementation allows ESs hosted by JXSHELL to conform fully with the requirements of good Web application design. Some benefits follow naturally from this design choice, such as the correct operation of the back and refresh buttons on a browser when running a JXSHELL ES (as detailed in 6.1.3). All that remains is to show how JXSHELL generates calls over an ES from GET strings, and how it generates interactions, realized as HTML pages, from ES output. To this point we have discussed JXSHELL in terms of theory and justification. We have seen how one particular design choice is preferrable to other candidates. In the following chapters, we will extend the discussion to show how JXSHELL actually implements this design, both in its Prolog based "backend" and its XSLT based "frontend." In answering the "how" of JXSHELL, after having answered the "why," we will provide practical pointers on implementing expert systems as JXSHELL applications.

Chapter 4

The JXSHELL implementation

The JXSHELL architecture is made up of two distinct server-side components: a Java Servlet front end and the LPA Prolog Intelligence Server (IS) backend. The servlet is configured to work with an HTTP server that passes CGI compliant input parameters to it. The parameters are processed on the servlet side, generating a call to the Intelligence Server. The IS itself is a wrapper for the LPA Prolog inference engine. The IS backend executes a Prolog program in full, passing a serialized structure as a return value by means of the IS callback mechanism. On the servlet side, the serialized structure is parsed and HTML code is generated accordingly. The code in the form of a Java string is returned to the user via the HTTP server.

Because the JXSHELL specification calls for a memory free inference operation, there is no need to maintain state between GET evocations. The only requirement is that the system correctly process GET strings formed according to the CGI specification. Therefore, the JXSHELL specification could be implemented using a “straight forward” CGI architecture that features a JXSHELL executable that loads into memory, executes, and unloads with every GET evocation. This “one shot” approach is typical of CGI applications. However, JXSHELL applications, while stateless in the sense of being program memory free, will receive series of user inputs. JXSHELL applications are also multi-user in the sense that multiple users can simultaneously run the same expert system program from the same JXSHELL site.

For this reason the Java Servlet front end is an important component of the JXSHELL architecture. The servlet lingers in memory between calls just as the

HTTP server does. This fact allows series of calls to the system while it remains in memory. While the servlet is running it also maintains a constant IS connection, meaning that the Prolog backend is likewise maintained in memory between calls. The backend Prolog programs, i.e., the XSHELL programs hosted on the JXSHELL platform, do not maintain state between calls, but the system as a whole does.

The servlet container of choice for JXSHELL is the Resin 2.0.0 servlet/HTTP engine from Cuacho Technology. In addition to supporting the latest servlet specification (2.3), Resin also supplies a powerful API for XML, DOM, and XSLT processing. Also included is a standalone HTTP server. Resin is an integral part of the JXSHELL distribution.

XML is a language for expressing information in a structured manner.[7] Unlike HTML, XML does not presuppose any specific interpretation for XML defined structures. Understood abstractly, XML is the expression of a tree in which there is a single root node (the top-level XML tag for a document), and where nodes have zero or more child nodes (which include tags that fall under a given tag, attributes for a tag, and text). XML has a more strict standard of well-formedness than HTML in that every XML tag must either have a corresponding end tag or must be a singleton tag (written as `<.../>`). XML is often used to define other languages, including XSL and XSLT, which are intended for certain types of tasks. XML parsers are used for processing XML-based languages, while other tools are used to interpret these languages according to some given specification. The XML language definition as well as the specifications for most XML-based languages are controlled by the World Wide Web Consortium.

XSLT is a language for transforming XML documents.[9] XSLT allows for the transformation of XML documents into other XML documents, though more commonly it is used for transforming XML documents into HTML documents for rendering by a browser. XSLT is a part of the larger XSL language[8], though most current applications, including JXSHELL, use XSLT exclusively.

The LPA Intelligence Server interface has no support for defining Java data types within Prolog. As a result, all communication between the Prolog and Java sides of JXSHELL applications is typed to `java.lang.String`. Thus structured data returned from the Prolog side of the interface must of necessity be serialized. Fortunately the Resin engine's XSLT API provides us with a ready made mechanism for cleanly and efficiently producing output from string serialized XML trees. For this reason the Prolog XSHELL engine generates XML as a string. XML provides an efficient and easy to understand format for serialized data as compared to a customized scheme or a clumsy looping mechanism. The XML tag set used in JXSHELL is described in 5.3.

The XML output from the Prolog engine is transformed into an HTML string using XSLT. A predefined XSLT stylesheet generates HTML output based upon a processing algorithm supplied by Resin's XSLT API. The abstraction inherent in the API allows JXSHELL's processing to remain very simple in the sense that processing details (including caching) are built into the API instead of the JXSHELL servlet. JXSHELL output processing is as bug free as the Resin API, and the latter has been extensively tested in real world, enterprise level applications [11].

An advantage of JXSHELL's output scheme is that it allows the user to push decisions regarding output to the output stylesheet. Aside from line returns (which are coded as "<BR/>") and some URLs, XSHELL output is restricted to string data and logical output. Everything from style elements to URL's are formulated using the stylesheet, meaning that JXSHELL output is as extensible as XML itself.

Standard ES programs include three elements: the interface, the knowledge base, and the inference engine [6, pages 297–8]. In the JXSHELL architecture, the details of the third element are opaque to developers, while the first and second are developed according to well defined standards.

JXSHELL knowledge bases are simple sets of rules plus utility predicates that together define a set of ES interactions. The JXSHELL KB spec is an amended version of the XSHELL specification outlined in Covington, Nute, and Vellino's *Prolog Programming in Depth*[1, pages 269-311]. The rules and facts in JXSHELL KBs are written in a "vanilla" Prolog syntax. The next section details the constructs that make up a JXSHELL KB. Predicates are referenced by name and arity, as is typical of Prolog references. For specifics on Prolog syntax refer to any basic textbook, such as the previously mentioned *Prolog Programming in Depth* [1] or Sterling and Shapiro's *The Art of Prolog*[3].

Chapter 5

A Reference Section for JXSHELL programmers

Sections 5.1 and 5.3 are references for JXSHELL knowledge base programming and stylesheet programming respectively. Section 5.2 gives an overview of the XML output from Prolog.

5.1 A JXSHELL knowledge base (KB) reference

This section includes a reference for JXSHELL knowledge bases organized by predicate. Included with examples are all the valid JXSHELL KB predicates and their arguments. The purpose is both to give a better understanding of how JXSHELL works and to provide a quick reference for programmers.

5.1.1 `xkb_intro/1`

The one place predicate `xkb_intro` provides a boiler-plate introduction to the ES. The single argument to `xkb_intro` is a list of quoted atoms where each atom represents string output separated by a line return. This predicate is optional. If `xkb_intro/1` is absent, the ES engine skips directly to rule processing when the ES is run.

The following example is from the JXSHELL reference implementation:

```
xkb_intro(
 ['',
  'CICHLID: An Expert System for Identifying Dwarf Cichlids',
  ''',
  'The cichlids are a family of tropical fish.  Many of',
  'these fish are large and can only be kept in large',
  'aquariums. Others, called ''dwarf cichlids'', rarely',
  'exceed 3 inches and can be kept in smaller aquariums.',
  '',
  'This program will help you identify many of the more',
  'familiar species of dwarf cichlid.  Identification of',
  'these fish is not always easy, and the program may offer',
  'more than one possible identification.  Even then, you',
  'should consult photographs in an authoritative source',
  'such as Staek, AMERIKANISCHE CICHLIDEN I: KLEINE',
  'BUNTBARSCHE (Melle: Tetra Verlag, 1984), or Goldstein,',
  'CICHLIDS OF THE WORLD (Neptune City, New Jersey:',
  't.f.h. Publications, 1973) for positive identification.',
  '',
  'To use the program, simply describe the fish by',
  'answering the following questions.']).
```

5.1.2 XKB_IDENTIFY/2

The two place predicate `xkb_identify` defines the knowledge component of JXSHELL ESs. Each `xkb_identify` clause is a rule that defines an answer to a set of user inputs. To best understand how `xkb_identify` works, it is useful to consider how the inference engine processes `xkb_identify` clauses. First the engine queries the clause in the standard Prolog manner. A query on a rule results in query on the clauses in that rule's body in order. Each clause represents some condition that must hold for the rule as a whole to hold. At each condition the engine checks to see if the condition is already satisfied or negated according to previous user input. If it is satisfied, the engine continues on to the next condition, while if it is negated then the rule fails and the engine *backtracks* to the next rule (if present). Once all conditions for some rule are satisfied the rule succeeds, resulting in an answer being returned to the user. If no rule can succeed given the user input, the system returns a "no remaining answers" response to the user.

The first argument to `xkb_identify/2` is an integer that represents the rule index. The index is used by the inference engine to pick out the correct rule for processing. Rule indexes must be sequential and must begin with "1," though the order of the rules in the file does not matter.

The second argument to `xkb_identify/2` is a list of atoms that the engine uses to determine output when a rule is satisfied. See the reference on `xkb_text/2` and `xkb_link/3` to see how this argument is processed.

Eight types of assertions can be modeled as `xkb_identify` conditions: `prop`, `parm`, `parmset`, and `parmrange`, as well as their negations.

A `prop` condition represents a property that does or does not hold for the given problem. The only parameter for a `prop` is a keyword. For instance, in the dwarf cichlid identification ES which is part of the JXSHELL reference implementation, the condition of having a dorsal streamer is `prop(dorsal_streamer)`.[1]

A `parm` condition represents a parameter, or a specific trait that is confined to one of at least two given possibilities. Parameters for `parm` include a keyword plus an integer that indicates which choice is intended. In the knowledge base the condition of having a spear-shaped tail-fin, as opposed to a lyre-shaped or normal tail-fin, is

[1]For more on the reference implementation, refer to Appendix A. Some of the examples that follow are not drawn directly from the reference implementation. These are used for illustration and employ a vocabulary consistent with that found in the reference implementation.

represented as parm(caudal,2) where “2” indicates the choice of “spear-shaped.” (For more on how parm choices work, see the reference for xkb_menu/4 below.)

A parmset condition represents a possible set of choices, so for instance we can represent the fact that a cichlid can have either a “lyre-shaped” or “normal” tail-fin. In the KB this condition is represented by a keyword and a set (as a list) of permissible choices, e.g. parmset(caudal,[1,3]).

Finally, a parmrange condition is similar to a parmset, but it covers a continuous range of values as opposed to a finite set of discrete values. As an example, a dwarf cichlid may be between three and five inches long. In the KB this condition may be expressed as parmrange(length,3,5). Note that this range is end inclusive.

The following is an example of xkb_identify from the reference implementation. Note the presence of different types of conditions in the rule's body:

```
xkb_identify(1,[isa,agassizii,nl,fake_hook1,nl]) :-
    parm(caudal,m,2),
    parm(body_shape,m,1),
    parm(lateral_stripe,m,1),
    prop(dorsal_streamer),
    prop(lateral_stripe_extends_into_tail).
```

Note that in principle Prolog rule processing is recursive in the sense that a given condition clause can itself be a head of another rule. However, JXSHELL strictly limits such clauses to user defined zero arity predicates whose bodies are made up of zero or more standard clauses or similar zero arity predicates. Furthermore, looping conditions are illegal. If a rule appears with a condition that breaks these restrictions (i.e., that is neither a standard clause or a non-looping zero arity predicate), the system preprocessor will fail to process the KB. The reason for these added restrictions is that side affect bearing predicates introduce undefined behavior to the system. For instance, a predicate that sends a string to standard output makes no sense in the context of JXSHELL processing because JXSHELL does not support the concept of standard output.

The zero arity predicates are supported simply for convenience. The JXSHELL preprocessor unfolds these predicates, so in effect they exist only for the developer and not for the system. Using these predicates can give rules a more compact presentation withing source files, especially when a set of conditions are repeated across rules.

Following is the rule above, rewritten with two zero arity predicates that is also defined:

```
xkb_identify(1,[isa,agassizii,nl,fake_hook1,nl]) :-
     parms,
     props.

parms:-
     parm(caudal,m,2),
     parm(body_shape,m,1),
     parm(lateral_stripe,m,1).

props:-
     prop(dorsal_streamer),
     prop(lateral_stripe_extends_into_tail).
```

5.1.3 XKB_QUESTION/4

The `xkb_question/4` predicate contains information used by the system both in formulating question output (i.e., queries for user-supplied information) as well in generating rule explanations. The first argument to `xkb_question` is an atom that the inference engine matches against the argument of a `prop` (see above) when processing an `xkb_identify/2` rule body. When the engine encounters a `prop` it first checks to see whether the information required has already been reported by the user. If not, it finds a matching `xkb_question` clause and uses the second argument to formulate a question for the user. The second argument to `xkb_question` is a list of quoted atoms that represent the question that is appropriate to the given `prop`. The third and fourth arguments are quoted atoms used in explaining rules; refer to the section "The JXSHELL explanation facility" below 5.1.8.

The following example is taken from the JXSHELL reference implementation:

```
xkb_question(dorsal_crest,
      ['Are any fin rays at the front of the dorsal fin',
       'clearly extended above the rest of the fin?'],
       'Front rays of dorsal fin are extended.',
       'Front rays of dorsal fin are not extended.').
```

Failure to supply an `xkb_question/4` predicate for every distinct `prop` condition will result in an error condition passed to the Java interface.

5.1.4 XKB_MENU/4

`xkb_menu/4` serves a purpose similar to that of `xkb_question/4`, namely to supply the inference engine with information necessary for formulating questions and explanations. The first argument to `xkb_menu` is an atom that the inference engine matches against the first argument of a `parm` (see above) when processing an `xkb_identify/2` rule body. The second argument is a list composed of quoted atoms that make up the question that matches the given condition. The question is returned to the user to elicit appropriate information. The third argument is a list of quoted atoms each of which indicates a menu item. The JXSHELL interface is responsible for presenting these items in such a way that the user can select one of them in response to the question expressed by the second argument. Note that order is important here. The meaning of a `parm` condition is based upon the order of the items in the third argument insofar as the integer argument to `parm` "picks out" a menu choice according to its position in the list. As an example, consider the condition `parm(caudal,3)`. The sense of this condition is only apparent given a corresponding `xkb_menu` predicate:

```
xkb_menu(caudal,
      ['What is the shape of the tail-fin?'],
      ['lyre-shaped',
       'spear-shaped',
      'normal, i.e, round or fan-shaped'],
      'Tail fin is ').
```

So here `parm(caudal,3)` indicates the condition of having a normal (round or fan-shaped) tail-fin. The final argument ('Tail fin is ' in our example) is part of the JXSHELL explanation facility; refer to the section "The JXSHELL explanation facility" below 5.1.8.

Failure to supply an `xkb_menu/4` predicate for every distinct `parm` or `parmset` condition will result in an error condition.

5.1.5 XKB_LINKS/4

`xkb_links/4` is a utility predicate used to map resources to `prop` and `parm` interactions. The first argument to `xkb_links` is an atom indicating type of interaction, viz. prop or parm. The second argument is a keyword (atom) indicating *which* prop or parm specifically. The third argument represents a menu choice, because in the case for resources for parm interactions, these resources will map to choices and not the the interaction as a whole. In the case of prop type xkb_links, the third argument will be disregarded by the engine. The final argument is a list of length three containing (in order) a URL for a help file, a target for the same help file, and a URL for an image. The intention is that for every yes/no question or menu choice presented to the user, there will be zero or one associated help files and/or images. The URLs can be relative or absolute, and just how they are realized in the user interface depends on the output XSLT processing.

The following two examples, one for prop and one for parm, are taken from the JXSHELL reference implementation:

```
xkb_links( prop, dorsal_crest, '',
           ['/xshell.hlp.html','10','/images/cichlid/xshell1.bmp'] ).


xkb_links( parm, caudal, 'lyre-shaped',
           ['/xshell.hlp.html','10','/images/cichlid/xshell1.bmp'] ).
```

xkb_links/4 is optional in the same sense as xkb_intro/1.

5.1.6 XKB_LINK/3

Whereas xkb_links/4 maps resources to props and parmss, xkb_link/3 indirectly maps resources to rules. The intention is that on satisfying a rule the system should present the user with resources (e.g., text information, hypertext links, etc.) that pertain to the solution. For instance, in a catalog system a satisfied rule may make reference to some item in a Web-accessible catalog. The rule-satisfaction interaction, then, would provide a link to the specified item in the relevent catalog. The first argument to `xkb_link` is an identifier. This identifier is found in the second argument to `xkb_identify/2`. Upon satisfying a rule, the inference engine processes the members of the argument in order, attempting to match them against the first argument of `xkb_link`. There is no restriction against using the same identifiers across different `xkb_identify` predicates, meaning that the same resources can be (indirectly) mapped to different rules. The second argument is an atom indicating type, where the type of the link is either `hook`, `url`, or `text`. The link type information is used by the engine to return the correct information to the Java output processor.

The third argument is a list of (possibly quoted) atoms that defines output appropriate to the link type. In the case of `text` links, the list is simple text output represented as quoted atoms. In the case of `url` links, the list is made up of two members, where the first is a quoted atom representing text output, and the second is a quoted atom represented a URL attached to that output. In the case of `hook`, the list is made up of three members, the first being text output, the second being a user defined resource identifier, and the third being extra information for linking to the resource. In the text output portions of the third argument to both `hook` and `url` type links, the developer has the option of using a link tag, expressed as `<LINK>` and `</LINK>`, to map links to specific parts of the text as opposed to the text as a whole.

The `hook` type `xkb_link` warrants more explanation. In the JXSHELL system a hook is an abstract binding between text and some resource, where the details of that binding are a matter for the user interface. Practically speaking, a hook allows the programmer to link text to a resource in the knowledge base while using the output stylesheet to specify how the link is implemented. For instance, a programmer may include the following hook:

```
   xkb_link(my_hook,
               hook,['The link is <link>here</link>.',
               'EXAMPLE','link.htm'])}
```

In the output stylesheet, 'EXAMPLE' maps to a fully qualified url, such as http://mylinks.org/app. The processor appends that url to link.htm to generate an HTML hypertext link. The HTML output would look like this:

```
The link is <a href="http://mylinks.org/app/link.htm">here</a>.
```

Defering the implementation of the link allows for greater flexibility in output without affecting the underlying knowledge base. For instance, the output can be changed by changing the mapping of `'EXAMPLE'` to `http://yourlinks.org/app`. The output can also be easily parameterized, so the specific output becomes context dependent. For instance, the link may be different depending on whether the user is using a PC or a hand-held computer. Finally, a hook need not be interpreted as a hypertext link. The stylesheet can leverage the power of Java to make database calls, embed images, generate an applet, or do any number of things. Such power and flexibility would bloat the JXSHELL KB specification. As a general principle, KB content should be separated from the system's interface logic [6]. JXSHELL hooks allow this separation without sacrificing extensibility. Hooks also allow a clean separation of labor between KB and interface designers.

The following example is taken from the JXSHELL reference implementation:

```
xkb_link(fake_hook1,hook,
          (['Check <LINK>catalog</LINK> to add 101-0001 to your cart.'],
            'CATALOG','101-0001')
          ).
```

5.1.7 `XKB_TEXT/2`

The inference engine uses information in `xkb_text/2` to generate text output based upon the second argument to `xkb_identify/2`. The first argument to `xkb_text` is an atom that matches a member of the second argument to `xkb_identify`, and the second argument is a (quoted) atom used by the system as text output. Note that this mapping works just like that of `xkb_link` to map information to rules. In fact, `xkb_text` is strictly speaking superfluous given that a `text` type `xkb_link` supports plain text output. `xkb_text` is kept because of its appearence in the original XSHELL specification [1, pages 269–288], and because it provides slightly less verbose code. Note that `xkb_text(nl,'<BR/>')` is a particularly useful clause.

The following example is taken from the JXSHELL reference implementation. It is an example of using `xkb_text` to define a newline (“nl”) atom:

```
xkb_text( nl, ['<BR/>~M~J'] ).
```

5.1.8 The JXSHELL explanation facility

The JXSHELL system supports user prompted rule explanations. Upon rule satisfaction, the user can request the explanation of how the given result was obtained. To provide this information in a human readable fashion, the system reprocesses the satisfied rule by generating an appropriate text output for each condition of the satisfied rule. To generate this output in the case of `props`, the engine uses the third and fourth arguments of the `prop`’s corresponding `xkb_question` clause. Argument three is used in case the `prop` is not negated, and argument four is used if it is negated. For instance, given the following `xkb_question` clause:

```
xkb_question(dorsal_crest,
     ['Are any fin rays at the front of the dorsal fin',
      'clearly extended above the rest of the fin?'],
      'Front rays of dorsal fin are extended.',
      'Front rays of dorsal fin are not extended.').
```

the explanation for the rule that includes `\+prop(dorsal_crest)` will include the text

```
Front rays of dorsal fin are not extended.
```

To generate explanation output in the case of a `parm` condition, the engine performs a similar analysis using an appropriate `xkb_menu`. It is easy to see how this processing works given the following examples. With the clause:

```
xkb_menu(caudal,
     ['What is the shape of the tail-fin?'],
     ['lyre-shaped',
      'spear-shaped',
     'normal, i.e, round or fan-shaped'],
     'Tail fin is ').
```

an explanation for a rule containing the condition `parm(caudal,2)` will include as part of its output

```
Tail fin is spear-shaped.
```

Similarly, an explanation for a rule containing the condition `parmset(caudal,[2,3])` will include as part of its output

```
Tail fin is one of spear-shaped, normal, i.e. round or fan-shaped.
```

5.1.9 Allowable characters within JXSHELL output strings

Various quoted atoms within a JXSHELL knowledge base are intended as output text in HTML. As such, any tag set allowed within the body element of an HTML document can appear in JXSHELL output. However, the output is more restrictive than standard HTML in two respects: singleton tags (such as line breaks) must be represented as closed tags (e.g., `<BR/>`); and, any open tag must be matched by a closed tag within the same output string (e.g., `<I>...</I>`). Failure to observe either of these restrictions will result in an error generated by the stylesheet processor. The meaning of embedded tags/tag sets is complicated by the fact that all output is processed according to a stylesheet. At the stage of the style sheet tags can be processed in any manner that the developer pleases, or they can be passed to output as-is, in which case they are interpreted as simple HTML tags. A general rule of thumb will be to stick to HTML-style tags used according to their standard meanings (e.g., `<BR/>`,`<B/>`, etc.) unless specific allowance is made for customized tags. In the latter case, an interface designer and a knowledge base designer may agree on a library of custom tags to use, in which case the output from the KB can be very expressive while remaining declarative. Note that tags passed as-is through the stylesheet are not case sensitive, but unless the stylesheet designer makes it specifically otherwise, other tags will be.

One custom tag must be defined in the interface. The `link` tag set, written `<LINK>` ... `</LINK>`, allows for embedded links in `hook` and `url` type `xkb_link` output. To correctly implement hooks and `url` hyperlinks this tag set must be defined. Refer to the file `main.xsl` in the JXSHELL reference implementation for a sample implementation.

5.1.10 A note on environment variables

The above reference does not make mention of the JXSHELL environment variable. As detailed above, the environment variable allows for segmentation by topic of the knowledge base space, thus allowing multiple ES's to be run within a single process. The environment variable is not, however, part of the JXSHELL KB specification. The JXSHELL preprocessor adds environment information by rewriting all KB predicates with the environment variable added (as a quoted atom) as the first argument. Thus, for instance,

```
xkb_intro([ ... ])
```

becomes

```
xkb_intro('ENV',[ ... ])}
```

This rewrite is opaque to the user, and specific settings are included in the JXSHELL runtime configuration file, not in the KB itself. Such environment hiding on the KB development end allows for more modular and error-free programming. See B.2 for details on JXSHELL configuration.

5.1.11 Differences between JXSHELL and XSHELL knowledge bases

It may be useful to consider the differences between the KB specification for JXSHELL and that of its predecessor XSHELL (as detailed in [1, pages 269–288]). XSHELL is intended as a console based application run within an active Prolog environment. The differences between XSHELL and JXSHELL arise primarily out of the fact that JXSHELL KB's are rewritten by a preprocessor at runtime and the fact that output is further processed instead of being sent directly to an output stream as is the case with XSHELL KB's.

Differences between the JXSHELL and XSHELL KB specification include:

1. XSHELL has no support for markup in output.

2. XSHELL KB's of necessity include a set of Prolog declarations that load the system driver and perform other tasks. In JXSHELL, dependency is handled completely by the preprocessor, so the specification includes no declarations.

3. JXSHELL includes as rule conditions only `parms`, `props`, `parmranges`, and `parmsets`, or arity 0 rules of the sort described in 5.1.2. XSHELL conditions can be any Prolog expression whatsoever.

4. JXSHELL includes two extra predicates: `xkb_link/3` and `xkb_links/4`.

5.1.12 Negation

Standard Prolog includes *negation as failure (NAF)*[1, pages 20–22]. Under NAF a query of the form

`\+` *query*

succeeds if and only if *query* fails. NAF is part of the *closed world assumption* of Prolog's semantics, by which something that is not known is assumed not to hold. JXSHELL, on the other hand, includes a semantics by which a query of the form

`\+` *query*

succeeds if and only if *query* has been determined not to hold. For example, the condition `\+ prop(dorsal_streamer)` in a JXSHELL rule holds only if the user has answered "no" when asked whether the fish has a dorsal streamer, *not* if the user has not answered the question. The syntax is the same, but the meaning of negation is distinct. JXSHELL can discard the closed world assumption because as an ES platform it provides a mechanism for remembering negative information as well as positive and for resolving failure through recourse to the user.

5.2 Prolog generated XML output

To allow flexible interoperability between the Prolog and Java elements of the JXSHELL architecture, the Prolog output routines are designed to return XML. This code is processed using the output XSLT stylesheet to generate HTML output. There is no intermediate processing stage between Prolog processing and XSLT processing. To understand how to produce output, it is suffient to understand the XML returned by the Prolog side.

5.2.1 JXSHELL XML output elements

The purpose of the Prolog generated XML encoding is not to serve as a general purpose data format but to represent the specific types of information that make up expert system interactions. The Prolog XML output includes a small set of structures repeated across all or most transactions. An advantage of using XML is that, as mentioned, the system can directly process the Prolog generated output to produce system output without intermediate processing. An advantage of using a small set of XML structures is that the output stylesheet is manageable because it does not have to be general purpose and because repeated structures allow for code reuse within the stylesheet.

Below both singleton tags (e.g., `<BR/>`) and tag sets (e.g., `<MESSAGE>` … `</MESSAGE>`) are refered to by their tag names (e.g., `BR` and `MESSAGE` respectively).

5.2.2 Top level tags

Each Prolog generated output string is a fully compliant XML document [7] without XML declaration tags or attached DTDs. Unlike in HTML, XML documents must be encompassed by a single tag set. For JXSHELL the top-level tag sets indicate the type of user interaction. For a complete reference, see 5.3 below.

5.2.3 Image and help support

JXSHELL includes support for images and contextual help in the course of consultations. In the case of `prop` interactions, the information attaches to the entire interaction. In the case of `parm` interactions, information attaches to menu items. The following is an example of two lines of XML code that represent image and help information:

```
<HELP value="/xshell.hlp.html#10"/>
<IMAGE value="/images/cichlid/xshell1.bmp"/>
```

The JXSHELL Prolog output routines generate these lines using information represented in the `xkb_links` predicate, as described in 5.1.5.

5.3 A JXSHELL XML output reference

The following is a complete reference for all XML elements output by the JXSHELL inference engine for XSLT processing. The purpose of this reference is to give stylesheet developers a clear understanding of the documents that a JXSHELL stylesheet must be able to process.

Each XML element is referenced by name, children nodes, parent nodes, and attributes. Children nodes are those which can fall under the defined node. In terms of XML output, that means which tags can fall within the given tag. The following reference also gives the special marker <`text`> as an indication of text output (i.e., characters that are not interpreted as tags.) Parent nodes are those under which the defined node can fall. Attributes, which themselves are children nodes, are the *name* and *value* pairs that are associated with a tag and are written:

<*tagname name0*=*value0 name1*=*value1* . . . >

A correctly written stylesheet will fulfill two basic conditions. One, the stylesheet output will correctly represent the XML input. Two, the stylesheet output will include some mechanism (generally, a "next" button) for proceeding with the consultation. Furthermore, the "next" step must generate a correct GET string. As a general rule, GET strings will be of the form:

> `ENV=`*environment*
>
> `&RULE_I=`*rule index*
>
> `&UNSATFACT_TYPE=`*known type*
>
> `&UNSATFACT_ID=`*known id*
>
> `&UNSATFACT_VALUE=`*known value*
>
> `&TYPE1=`*known type*
>
> `&ID1=`*known id*
>
> `&VALUE1=`*known value*
>
> ⋮
>
> `&TYPE`*n*`=`*known type*
>
> `&ID`*n*`=`*known id*
>
> `&VALUE`*n*`=`*known value*

Some interactions will not produce GET strings of this form, as explained in the reference. The `ENV` is taken straight from the XML, as is the `RULE_I` parameter, except in the case of rule satisfaction. `UNSATFACT_TYPE` and `UNSATFACT_ID` are taken from the XML, while `UNSATFACT_VALUE` is taken from the user interaction. "Unsatfacts," or "unsatisfied facts," are clauses with an unspecified value which is specified by the user interaction. In the case of both "unsatfacts" and knowns, *type* refers to the type of clause, namely `prop` or `parm`, *ID* refers to the first argument or arguments of the clause, and *value* refers to the value of the clause. In the case of `prop`'s, the value is 1 for "yes" and 0 for "no," while in the case of `parm`'s, the value is an integer indicating a menu choice, a number, or a letter, in the case of menu parms, numerical parms, or alphabetical parms respectively. The set of knowns is represented by `TYPE1` ... `TYPE`n, `ID1` ... `ID`n, and `VALUE1` ... `VALUE`n, where {`TYPE`m, `ID`m,`VALUE`m} represents the m known for every $1 \leq m \leq n$.

5.3.1 ERROR

Child Nodes: `RULE_I, ENV, MESSAGE`

Parent Nodes:

Attributes:

An incorrectly written ES knowledge base may result in a runtime error on the Prolog side. These errors are caught and represented in XML output as `ERROR` tags. Because these error conditions point to a problem with the ES itself, it is not advisable in an error interaction to give the user any options to continue with the consultation. The children nodes of `ERROR` should provide the developer with enough information for trouble-shooting. The content of `MESSAGE` is the text string that is returned from the Prolog engine.

5.3.2 YESNO

Child Nodes: `RULE_I, ENV, MESSAGE, UNSATFACT, KNOWNS, HELP, IMAGE`

Parent Nodes:

Attributes:

Yes/no interactions correspond to `prop` conditions. The intention is that a `prop` value is one that either holds or does not, so a yes/no interaction queries the user as to whether some given condition does hold ("yes") or does not ("no"). Yes/no interactions can include help information and an associated image. The `MESSAGE` child of `YESNO` includes the question that is passed from the Prolog side to the user.

5.3.3 `START`

Child Nodes: `ENV, MESSAGE`

Parent Nodes:

Attributes:

The `START` tag includes the information found at the beginning of a series of JXSHELL ES interactions. The `MESSAGE` tag includes the text string that introduces the user to the expert system. The introduction message may include markup, particularly hard returns (i.e., HTML `<BR>`), so the stylesheet developer may do well to refer to the `xkb_intro` predicate (see 5.1.1) or to capture the XML output (see 5.3.24) when designing the stylesheet. A start interaction must return to the JXSHELL system an environment variable, represented as the value of the `ENV` node mapped to the name `ENV`, and the intial rule index `1` mapped to the name `RULE_I`.

5.3.4 `EX_MULTI`

Child Nodes: `RULE_I, ENV, MESSAGE, UNSATFACT, MENU, KNOWNS`

Parent Nodes:

Attributes:

An ex_multi interaction (for "exclusive multiple choice") is one in which the user is given a series of menu items for a given query (represented in `MESSAGE`). Only one item can be chosen. An ex_multi interaction must return to the JXSHELL system an environment variable, represented as the value of the `ENV` node mapped to the name `ENV`, and the rule index mapped to the name `RULE_I`.

5.3.5 `TXTFLD-A`

Child Nodes: `RULE_I, ENV, MESSAGE, UNSATFACT, KNOWNS, HELP, IMAGE`

Parent Nodes:

Attributes:

A txtfld_a interaction gives the user the ability to enter alphabetic input. The standard way of entering input will be by means of an HTML text field input. Non-alphabetic input will generate undefined behavior on the Prolog side (though generally the Prolog side will generate an error), so the stylesheet developer should embed some Javascript code in the output to do user input validation. A txtfld_a interaction must return to the JXSHELL system an environment variable, represented as the value of the `ENV` node mapped to the name `ENV`, and the rule index mapped to the name `RULE_I`.

5.3.6 `TXTFLD-N`

Child Nodes: `RULE_I, ENV, MESSAGE, UNSATFACT, KNOWNS, HELP, IMAGE`

Parent Nodes:

Attributes:

A txtfld-n interaction is identical to a txtfld-a interaction, but the intended input is numeric instead of alphabetic. An embedded javascript validator should test user input accordingly. As with a txtfld_a interaction, a txtfld_n interaction must return to the JXSHELL system an environment variable, represented as the value of the `ENV` node mapped to the name `ENV`, and the rule index mapped to the name `RULE_I`.

5.3.7 `RULE_SATISFACTION`

Child Nodes: `RULE_I, ENV, MESSAGE, KNOWNS, HOOKS`

Parent Nodes:

Attributes:

A rule_satisfaction interaction gives the user the information mapped to a rule head. Upon satisfying a rule, which in terms of a consultation means that the user has reached a solution, the system returns to the user the solution in the form of the contents of `MESSAGE`. (Note that the contents of `MESSAGE` can include a `HOOKED_SECTION`. This latter may include hyperlinked elements that direct the user to resources relevent to the solution. See section 5.3.22).

If the relevent functionality is enabled, the user progresses from a rule satisfaction condition by continuing the consultation or by showing the rule that was satisfied. See the `HOOKS` reference (5.3.20) for details.

5.3.8 `RULE_UNSATISFACTION`

Child Nodes: `ENV, KNOWNS`

Parent Nodes:

Attributes:

A rule_unsatisfaction interaction should tell the user that there are no (further) solutions to the query. The user can continue from a rule_unsatisfaction by means of the browser's back button, but for convenience the stylesheet processor may include buttons for restarting a consultation or for returning to some start point, or exiting to some other page.

5.3.9 `RULE_DISPLAY`

Child Nodes: `MESSAGE`

Parent Nodes:

Attributes:

A rule_display interaction shows the rule by which a rule_satisfaction condition had been met. The user can continue from a displayed rule by means of the browsers back button or by means of buttons supplied by the stylesheet.

5.3.10 ENV

Child Nodes:

Parent Nodes: YESNO, START, EX_MULTI, TXTFLD-A, TXTFLD-N, RULE_SATISFACTION, RULE_UNSATISFACTION

Attributes:

The ENV node includes a string (without spaces) that indicates the system defined environment variable. Inclusion of the environment variable is an important part of most interactions.

5.3.11 RULE_I

Child Nodes: <text>

Parent Nodes: YESNO, START, EX_MULTI, TXTFLD-A, TXTFLD-N, RULE_SATISFACTION, RULE_UNSATISFACTION

Attributes:

The RULE_I node includes an integer that indicates the current rule index. Inclusion of the rule index is an important part of most interactions.

5.3.12 MESSAGE

Child Nodes: <text>, HOOKED_SECTION, BR

Parent Nodes: YESNO, START, EX_MULTI, TXTFLD-A, TXTFLD-N, RULE_SATISFACTION, RULE_UNSATISFACTION, RULE_DISPLAY

Attributes:

The `MESSAGE` node includes information relevent to the current interaction. In effect the information conveyed by rendering this node conveys information to the user in the form of text. In addition, a `HOOKED_SECTION` will include text and possibly hyperlinked resources relevent to a rule_satisfaction condition (see 5.3.22).

5.3.13 `HELP`

Child Nodes:

Parent Nodes: `YESNO, CHOICE`

Attributes: `value`

A `HELP` node includes a file path that gives the user access to a resource relevent to the current interaction. Generally, the user accesses the help resource through a hyperlink constructed in the stylesheet. The user interface can include pop-up windows for displaying help items by means of embedded javascript.

5.3.14 `IMAGE`

Child Nodes:

Parent Nodes: `YESNO, CHOICE`

Attributes: `value`

An `IMAGE` node includes a file path that gives the user access to an image relevent to the current interaction. The simplest rendering of images will be as referents of HTML `IMG` tags. However, the user interface can include pop-up windows for displaying image items by means of embedded javascript.

5.3.15 MENU

Child Nodes: `CHOICE`

Parent Nodes: `EX_MULTI`

Attributes:

A `MENU` node indicates to the processor that menu processing will take place to generate an ex_multi interaction.

5.3.16 CHOICE

Child Nodes: `MESSAGE, HELP, IMAGE`

Parent Nodes: `MENU`

Attributes: `ID`

A `CHOICE` node carries information relevent to the menu choices that make up an ex_multi interaction. Every choice will include a message, which indicates the choice to the user, and possibly child `HELP` and `IMAGE` nodes. The `ID` attribute has as its value an integer that stands as the system's internal representation of the menu choice. When a menu item is chosen by the user, the page must return to the system the integer mapped to `ID`.

5.3.17 KNOWNS

Child Nodes: KNOWN

Parent Nodes: YESNO, START, EX_MULTI, TXTFLD-A, TXTFLD-N, RULE_SATISFACTION, RULE_UNSATISFACTION

Attributes:

Because JXSHELL does not maintain session states between interactions, and because state must therefore be passed between interactions in a serialized form, the Prolog side must return an XML representation of the session state, or what may be thought of as the set of “knowns” collected during a user session. The set of knowns is represented in XML as a set of KNOWN elements under a single KNOWNS element. The hierarchy of elements makes XSLT processing simpler for information that includes an indefinite number of items. The following is an example of an XML representation of a set of knowns:

```
<KNOWNS>
 <KNOWN type="parm" id="body_shape,m" value="3">
   Body is normal fish shape.</KNOWN>
 <KNOWN type="prop" id="dorsal_streamer" value="1">
   Rear rays of dorsal fin are extended.</KNOWN>
</KNOWNS>
```

Note that the representation of the set of knowns is generated entirely by the runtime engine and is inaccessible to the KB programmer.

5.3.18 KNOWN

Child Nodes: `<text>`

Parent Nodes:

Attributes: `type, id, value`

In the JXSHELL system a known is a triple consisting of a classification (i.e., `parm`, `prop`, etc.), an attribute, and a value. In XML output, the items in this triple are represented by the `type`, `id`, and `value` attributes (respectively) of the `KNOWN` tag. In addition, attached to a `KNOWN` is a "human readable" string. In XSLT processing of the JXSHELL XML output, the attributes of the `KNOWN` tag are used to return to the system usable information, while the attached string is used to represent the known to the user.

The following two examples of `KNOWN`s are of the `parm` and `prop` types respectively:

```
<KNOWNS>
 <KNOWN type="parm" id="body_shape,m" value="3">
   Body is normal fish shape.</KNOWN>
 <KNOWN type="prop" id="dorsal_streamer" value="1">
   Rear rays of dorsal fin are extended.</KNOWN>
</KNOWNS>
```

5.3.19 UNSATFACT

Child Nodes:

Parent Nodes: `YESNO, START, EX_MULTI, TXTFLD-A, TXTFLD-N, RULE_SATISFACTION, RULE_UNSATISFACTION`

Attributes: `type, id`

In addition to the `KNOWNS`, the stylesheet must also process `UNSATFACT` information to produce a `prop` or `parm` type interaction. The `UNSATFACT` information represents the information that will be returned to the systems when a question is answered (by clicking a menu item, for instance). The term "unsatfact," short for "unsatisfied fact," is used to relate this information to individual "known" items, also known as "facts." A fact here is thought of as as information that fulfills or "unfulfills" a condition, just as the fact that a cichlid that does not have a dorsal streamer unfulfills the condition `prop(dorsal_streamer)`. The fact is a triple of type (`prop`), attribute (`dorsal_streamer`), and value (here "false"). An unsatisfied fact, then, is a fact lacking a defined value parameter. The undetermined value is subsequently determined by means of a user interaction. When the end user clicks a menu item (for instance), the action can be thought of as a determination of an undetermined value, or (in other words), the conversion of an `UNSATFACT` into a fact (or known). When JXSHELL runs the top-level Prolog processing query for interactions, it passes the Prolog side a set of knowns, plus the unsatfact information, plus the user-defined value that together with the unsatfact information is interpreted by the system as a known. The Prolog side then submits the unsatfact information plus user supplied value in a tree of known values.

Below is an example of an `UNSATFACT` in XML:

```
   <UNSATFACT type="prop" id="dorsal_crest"/>
```

This tag is rendered by the stylesheet in the reference implementation as:

```
<input type="HIDDEN" name="UNSATFACT_TYPE" value="prop">
<input type="HIDDEN" name="UNSATFACT_ID" value="dorsal_crest">
```

5.3.20 HOOKS

Child Nodes: HOOK

Parent Nodes: RULE_SATISFACTION

Attributes:

The HOOKS tag carries no attributes and contains only HOOK tags. The following is an example of HOOKS:

```
<HOOKS>
   <HOOK type="continue_consult"/>
   <HOOK type="show_rule"/>
</HOOKS>
```

5.3.21 HOOK

Child Nodes:

Parent Nodes: HOOKS

Attributes: type

A HOOK is an XML element that triggers an action on the part of the stylesheet without conveying any information that is passed on to the user. The presence of

HOOKs are set by a runtime parameter in the system configuration file. Because the Prolog side passes HOOKs, the Java side is able to pass the Prolog output in full to the stylesheet processor without amending the XML. However, the user defined knowledge base has no bearing on HOOKs.

In the reference implementation, there are two HOOKs:

```
<HOOK type="continue_consult"/>
<HOOK type="show_rule"/>
```

Upon processing the first, the processor passes to output HTML elements that allow the user to continue a consultation after an answer is reached. Upon processing the second, the processor passes to output HTML elements that allow the user to inspect the rule satisfied when an answer is reached.

5.3.22 HOOKED_SECTION

Child Nodes: `<text>, LINK`

Parent Nodes: `MESSAGE`

Attributes: `DOMAIN, TARGET`

A HOOKED_SECTION, defined by the `xkb_links/3` predicate, maps user defined hook keywords to resource keywords that indicate resources that are featured in the user interaction. To process a hooked section the stylesheet must include cases that match hooked section keywords and supply a mapping within the hooked section code of corresponding resource keywords to actions that can take place within stylesheet processing. Perhaps the simplest case of hooked section processing would be that in which a resource keyword triggers the generation of a text string to output (e.g., “the HOOKED SECTION was hooked!”). However, XSLT processing is open ended, meaning that there are no prescribed restrictions on how XML text can be processed. In addition, the JXSHELL XSLT processor can harness the power of Java to perform actions so that, for instance, a hooked section can trigger a remote database lookup via JDBC, process the return value of the lookup, and return an appropriate string to output. As an example, within an online catalog system a hooked section could be used to advise the end user of how many of a given item are available for purchase.

`HOOKED_SECTION`s include two attributes, `DOMAIN` and `TARGET`. These attributes convey information to the XSLT stylesheet that allow correct processing. For instance, a "domain" could included an SQL server name, and a "target" could be an SQL string.

Below is are two examples of `HOOKED_SECTION`s in XML:

```
<HOOKED_SECTION DOMAIN="CATALOG" TARGET="12345">
       This is fake xkb_links HOOK with implicit link.
</HOOKED_SECTION>
<HOOKED_SECTION DOMAIN="CATALOG" TARGET="12345">
      This is fake <LINK>xkb_links</LINK> HOOK.
</HOOKED_SECTION>
```

Note the use in the second example of the `LINK` element.

A note on nomenclature is in order. A `HOOK` XML element is not defined by anything in an XSHELL knowledge base, whereas a `HOOKED_SECTION` XML element is defined by the `xkb_links` predicate, specifically by `xkb_links` that include the quoted atom 'HOOK' as a first argument. The KB 'HOOK' corresponds to the XML `HOOKED_SECTION`, *not* to the XML `HOOK`. As long as we concentrate on *either* the knowledge base or the stylesheet, there is no threat of confusing `HOOK`s and `HOOKED_SECTION`s. Unfortunately, when *both* are considered the situation can be confusing. This confusion in nomenclature will be corrected in a future release of JXSHELL.

5.3.23 `LINK`

Child Nodes: `<text>`

Parent Nodes: `HOOKED_SECTION`

Attributes:

The `LINK` tag indicates where a hyperlink or some other form of linking mechanism should be inserted within a `HOOKED_SECTION`. See the `HOOKED_SECTION` reference above for details.

5.3.24 A tip for writing stylesheets

JXSHELL expert system development is envisioned as a cooperative process between stylesheet designers and knowledge base designers. If the KB author wants to include elements in the output, such as hard returns (`<BR>` in HTML) or custom tags, he will consult with the stylesheet designer so that those elements will be correctly rendered in HTML output. However, stylesheet authoring can often be aided by inspecting the "raw" XML document that the stylesheet is meant to process. To make this XML available, the stylesheet developer can *capture* the XML output from Prolog. To do so, include the following tag in the body of each XSLT template tag corresponding to each XML top level tag:

```
<xsl:comment>
  <xsl:copy-of select='\'/>
</xsl:comment>
```

This structure renders the entire XML output to HTML, but within an HTML comment section so that the XML appears only on the output source code but not on the page as seen in the browser.

5.3.25 The JXSHELL reference implementation

JXSHELL includes a simple reference implementation that illustrates the application of most of the features described here. The implementation described runs “out of the box” and includes by default a “toy” tropical fish identification knowledge base. In addition, alternative knowledge bases are included that test the JXSHELL platform in several ways. See Appendix A for details regarding the reference implementation.

Chapter 6

JXSHELL evaluation

6.1 JXSHELL advantages

An obvious advantage of the JXSHELL system is that it allows knowledge engineers to utilize the Web as an application platform. Expert systems give users access to expert knowledge that would be otherwise unavailable. This principle is extended further with a Web-based system insofar as the accessibility to expert knowledge is given to anyone with a Web browser and Internet access.

JXSHELL affords other advantages as well. Below we detail some of the advantages that make JXSHELL a powerful platform for expert system deployment.

6.1.1 Low server side load

Though a client-server system, JXSHELL carries relatively low server side load because no session state is maintained between calls to the LPA Prolog engine. State is conveyed entirely by GET strings. This means that the memory footprint of the program does not increase with an increased number of connections.

6.1.2 "Piggy-backed" servlet functionality

The servlet engine specification [10] allows for a range of useful functions that make servlets preferable to "once and out" CGI programs. Furthermore, servlet engines generally will contain support for advanced server functionality, such as load balancing, session logging, and URL redirection. These specific functions are available in the Resin engine that ships as part of JXSHELL.

6.1.3 The Back button

In compliance with Berners-Lee's axioms of Web design, JXSHELL uses only GET strings. These operations do not alter the "state of the world" of the Web, meaning that identical GET strings represent identical GET actions. A nice outcome of this Web design compliance is that the browser's back button works correctly in a JXSHELL session. The user can use the back button to return to previous interaction screens. This means that although a JXSHELL program is a custom application, it relies on a familiar interface (the Web browser) that works in the familiar way.

6.2 JXSHELL compared to LPA ProWeb and Amzi KnowledgeWright

Logic Programming Associates includes with its developer version of LPA WIN-Prolog a product called "ProWeb," which the documentation (see [4]) describes as "ProWeb = Prolog Logic + Web Control." ProWeb is a toolkit that allows developers to use LPA's WIN-Prolog application as a host environment for Web-based Prolog programs. Like JXSHELL, ProWeb uses LPA WIN-Prolog as the logic program interpreter for interactive, browser accessible programs. Also like JXSHELL, ProWeb is stateless in the sense that it processes a knowledge base from top level predicates on each user initiated call.

ProWeb developers define the user interface for interactions by means of HTML template documents containing special ProWeb tags. At runtime the system substitutes these tags with program-defined information. The developer defines tag substitution by means of a large set of ProWeb predicates present in a the Web-hosted

Prolog program. Therefore, in sharp contrast to JXSHELL knowledge bases, ProWeb knowledge must make provision for the program interface. In this way JXSHELL development affords a more complete abstraction between the interface and program logic.

The use of HTML templates allows for easy interface development, but only at the cost of flexibility as compared to using XSLT stylesheets. JXSHELL stylesheets will typically be large and complex, but they allow for functionality that goes beyond anything available in HTML templates. For instance, XSLT includes structures for looping through XML input. The stylesheet included in the JXSHELL reference implementation includes iterative processing that allows for the correct placement of multiple items in columns in output. This feature is described briefly in Appendix A. Note that this iterative processing works regardless of the size of the XML structure processed and does not entail adding any control elements to the knowledge base.

An advantage of ProWeb is that it is general purpose. In theory, any Prolog program can be rewritten with ProWeb output predicates and then Web-hosted. JXSHELL on the other hand is specifically for rule-based consulatitive ES hosting. There are many applications, then, for which ProWeb is appropriate but JXSHELL is not. However, for a large set of applications, namely rule-based consultative expert systems, JXSHELL is an effective and powerful deployment platform.

Like JXSHELL and unlike LPA ProWeb, Amzi's KnowledgeWright product is a platform specifically designed for expert system deployment.[5] Unlike JXSHELL, KnowledgeWright includes tools for the integration of the platform in a number of host environments for the development of both stand-alone software and Web-based applications. The configuration and development tools included with KnowledgeWright go beyond anything available in JXSHELL.

KnowledgeWright gives an applications developer two means of Web deployment. Under one scenario, the developer authors an interface in a language such as Java (written as an applet) or Visual Basic (written as an ActiveX control), and embeds the resulting software object in a Web page. Under the other scenario, the developer connects KnowledgeWright applications to a Web server and serves interaction screens through a CGI interface (much like JXSHELL).

Though KnowledgeWright is a powerful ES development and deployment tool, both of these options have disadvantages compared to JXSHELL. In the case of embedded objects, the application developer must rely on the user to have a browser capable of supporting object embedding, and the user must wait for the download of a large binary or byte-encoded object whenever loading an expert system application. JXSHELL interactions, on the other hand, are sent to the user as simple HTML files requiring little bandwidth and no special browser functionality beyond support for CGI (and possibly Javascript and/or VBScript support). In the case of KnowledgeWright's CGI interface, the system relies not on GET operations but on POST operations, meaning that Berners-Lee's axiom of universality is not satisfied by KnowledgeWright-based ESs using the CGI interface. Under neither scenario are ES interactions addressible Web objects in the sense described in chapter 3.

Appendix A

The JXSHELL reference implementation

JXSHELL includes a reference implementation in two files: a knowledge base `CICHLID.XSH` and a stylesheet `MAIN.XSL`. The knowledge base is for a "toy" animal identification ES. `CICHLID.XSH` contains rules for the identication of dwarf cichlids, a type of tropical fish found in home aquariums. Identification programs, in which the program helps a user identify some given object based upon that object's characteristics, are commonly programmed as rule-based ESs. `CICHLID.XSH` is a modified version of the ES of the same name found in [1, pages 289–298]. JXSHELL's `CICHLID.XSH` contains most of the features referenced in Section 5.1.

The reference implementation's stylesheet, `MAIN.XSL`, contains all the basic structures necessary for generating correct HTML output for consultations. In addition, `MAIN.XSL` contains code for producing a set of knowns summary. This summary, displayed just below user input areas for interactions, allows the user to choose specific items in the from among the current input set. In output the summary appears as a list of knowns, beside each of which is a checkbox. At any point in a consultation the user can reset the set of knowns by checking the boxes beside one or more items and clicking a submit button. This action resets the set of knowns to the given items and resets the current rule index to 1.

The set of knowns summary is generated by iterating through the `KNOWNS` node in the system's XML output. As an example of the power of XSLT processing, and of the power of XSLT's iterative structures specifically, the summary is displayed in two columns. Similarly, the menu items for ex_multi interactions are displayed in two columns.

Appendix B

JXSHELL deployment considerations

In addition to providing an explanation of JXSHELL along with background and justification for its design choices, the current paper is meant as a practical guide to those who wish to deploy it for real-world applications. To that end we supply some basic information necessary for anyone who wishes to host a JXSHELL site.

B.1 JXSHELL version numbers

In versioning JXSHELL adheres to the common convention of using a three slot numbering system of the form *major.minor.revision*. The major version number changes upon major revisions to the JXSHELL core program, including the Java interface and all Prolog output drivers. The minor version number changes upon changes of or additions to JXSHELL functionality that do not represent major rewrites. The revision number changes upon bug fixes or what developers consider minor upgrades to existing functionality. In addition, changes in the JXSHELL reference implementation constitute revisions. The decision on version changes rests with the developer responsible for issuing JXSHELL development licenses (currently, the author). The current JXSHELL version as of this writing is 1.0.1.

Note that JXSHELL allows for extensive application development on the part of JXSHELL license holders. This development is in the form of stylesheet and knowledge base authoring. Any additional development on the part of license holders, including revisions of elements of the reference implementation, do not reflect on

JXSHELL versioning. However, license holders are not precluded from using a version number scheme for their own JXSHELL-based applications.

B.2 JXSHELL installation and configuration

JXSHELL ships as a set of files in an installation directory. These files, refered to here as "installation files," contain everything necessary for setting up the JXSHELL system.

JXSHELL includes a simple installation procedure. As of the current verison, the process is not completely automated, though it is relatively simple in the case of a "standard" installation. The standard installation for JXSHELL entails the creation of two directories, `jxshell-1.0.1` and `JxshellAppFiles` on the root directory of the "D:" drive of a Windows computer. Alternatively, a similar installation on the "C:" drive is supported. Other installation options are possible, though these will involve editing some files. See the file `README-INST.txt` in the JXSHELL installation files for details.

JXSHELL configuration is implemented by means of settings in the `resin.conf` file. Although this file is a standard part of the Resin server package, it has been modified for use with JXSHELL. Specifically, the file contains settings that define the JXSHELL servlet, and it contains settings that are used by the JXSHELL driver to correctly set up JXSHELL applications. See the file `README-CONF.txt` in the JXSHELL installation files for details.

Bibliography

[1] Covington, Michael A., Donald Nute, and André Vellino. *Prolog Programming in Depth.* Prentice-Hall. Upper Saddle River, NJ. 1997.

[2] Hogger, Christopher John. *Essentials of Logic Programming.* Clarendon Press. Oxford. 1990.

[3] Sterling, Leon, and Ehud Shapiro. *The Art of Prolog.* MIT Press. Cambridge, Massachusetts. 1986.

[4] Logic Programming Associates, LLC. *ProWeb User Guide.* Logic Programming Associates, Ltd. 31 October 2001.

[5] Amzi, Inc. "KnowledgeWright Overview."

[6] Stefik, Mark. *Introduction to Knowledge Systems.* Morgan Kaufmann Publishers, Inc. San Francisco. 1995.

[7] World Wide Web Consortium. "Extensible Markup Language (XML) 1.0 (Second Edition)." 6 October 2000. <http://www.w3.org/TR/2000/REC-xml-20001006>.

[8] World Wide Web Consortium. "Extensible Stylesheet Language (XSL) Version 1.0." 15 October 2001. <http://www.w3.org/TR/xsl>.

[9] World Wide Web Consortium. "XSL Transformations (XSLT) Version 1.0." 16 November 1999. <http://www.w3.org/TR/xslt>.

[10] Coward, Danny. ”Java Servlet API Specification Version 2.3.” 17 September 2001. Sun Microsystems, Inc. <http://www.jcp.org/aboutJava/communityprocess/final/jsr053>

[11] Caucho Technology, Inc. “Resin (tm) Core.” <http://www.caucho.com/products/resin>