

## CHAPTER 5 KNOWLEDGE BASE ISSUES

### 5.1 Introduction

This chapter describes the issues surrounding the knowledge base, which forms the foundation for the intelligent agents for the proposed system. The subject areas of knowledge engineering and knowledge representation are introduced, and the issues of choosing a suitable method for acquiring knowledge as well as the choice of agent implementation language are investigated.

### 5.2 Knowledge Engineering

The term Knowledge Engineering is now commonly used to describe the process of Knowledge Based System (KBS) development. Knowledge engineering includes [Smith, 96]:

- Acquiring from experts the knowledge that is to be used in the system.
- Choosing an appropriate method for representing the knowledge
- Implementation in an appropriate computer language.

Just as with traditional computer systems development, the notion of a systems development lifecycle has been applied to the development of KBS's. The following are the main activities in the KBS development lifecycle [Bochsler, 88]:

- **Planning** - produces a formal workplan for the KBS development.
- **Knowledge definition** - defines the knowledge requirements of the KBS.
- **Knowledge design** - produces a detailed design for the KBS.
- **Code** - produces the actual code implementation of the KBS.
- **Knowledge verification** - determines the correctness, completeness and consistency of the system.

A number of KBS development methodologies have evolved such as - GEMINI (General Expert systems Methodology INitiative) [Montgomery, 88] RUDE [Bander et al., 88] and KADS (Knowledge Acquisition and Design Process) [Schreiber et al., 93]. However surveys [Smith, 96] [DTI, 92] indicate that a prototyping approach is most popular among KBS developers - perhaps because KBS development relies heavily on the involvement of the systems users (those with the appropriate expertise). In addition, prototyping is well suited to the types of problems often encountered in KBS development, where the initial requirements are typically ill-defined and the problem itself often ill-structured. The concept of intelligent agents provides an ideal basis on which to pursue a prototyping approach to the development of an intelligent assistant system. Individual agents can be rapidly prototyped, as each agent can be relatively small and in addition be an expert in just one small area of the problem domain, thus allowing a basic agent to be developed quickly and used in the testing and exploration process discussed above.

### **5.3 Knowledge Representation Systems**

Knowledge representation is the formalising and storing of knowledge in some suitable structure to allow for subsequent computer processing. To represent knowledge in a meaningful way it is important to relate facts in a formal representation scheme to facts in the real world. The formal representation will be manipulated using a computer program, with new facts concluded. Therefore the main issue in knowledge representation is to find a suitable knowledge representation language in which domains of knowledge can be described.

A knowledge representation language should allow complex facts to be represented in a clear and precise way, and in a way that easily allows the deduction of new facts from existing knowledge. The main requirements of a knowledge representation language are as follows [Ringland and Duce, 88] [Cawsey, 98]:

- **Representation adequacy** - It should allow the representation of all the knowledge that is needed to reason with.

- **Inferential adequacy** - It should allow new knowledge to be inferred from a basic set of facts.
- **Inferential efficiency** - Inferences should be made efficiently.
- **Clear syntax and semantics** - It should be known what the allowable expressions of the language are and what they mean.
- **Naturalness** - The language should be natural and easy to use (in a linguistic sense).

However, no one representation language satisfies all these requirements (perfectly). In practice the choice of representation language depends on the reasoning task - just as the choice of programming language depends on the problem. Given a particular task it will generally be necessary to choose an appropriate language for the particular requirements of the application.

The remainder of this section will briefly compare three commonly used knowledge representation schemes - Production Rules, Semantic Nets and Predicate Logic - in order to give an appreciation of the types of schemes that may be employed. Section 5.4 will then outline the main agent languages that can be used to implement knowledge representation in an agent-orientated framework.

A classical way to represent human knowledge is the use of production rules, where the satisfaction of the rule antecedents gives rise to the execution of the consequence [Ringland and Duce, 88]. Production systems represent knowledge in terms of a set of IF-THEN rules, a set of facts normally representing things that are currently held to be true, and some interpreter (inference engine) controlling the applications of the rules, given the facts. Production systems exhibit useful modularity, in that rules are independent of each other and of the rest of the system, as each rule encodes a 'chunk' of independent domain knowledge. Further, the straightforward if-then form of a rule often maps well into English for the purposes of explanation. Due to the independence of the rules from each other and from the control strategy, it is very difficult to rigorously determine the properties of the systems behavior by static analysis. Further, rules have no intrinsic structure, which makes management of large knowledge bases difficult.

Semantic Networks were first developed as a way of representing human memory and language understanding [Quillian, 68], but since then have been applied to knowledge representation. In a semantic net, knowledge is represented as a graph, where the nodes represent objects and the links represent relations between objects, where an object can be any physical item such as a book or a person. Nodes can also be used to represent concepts, events or actions. The nodes in a semantic net are interconnected by links or arcs which show the relationships between objects. Semantic nets allow knowledge to be represented about objects and relationships between objects in a simple and fairly intuitive way, with a graph notation that allows for easy knowledge organisation. However, semantic nets have certain limitations, such as the lack of link and node name standards and the combinatorial explosion of searching nodes.

Prolog is a programming language that is used for solving problems that involve objects and the relationships between objects [Clocksin and Mellish, 81] and as such is considered to be a highly suitable implementation language for semantic nets. Prolog is a declarative programming language that allows the programmer to express ‘what’ a problem is, without having to worry about ‘how’ a solution to that problem is to be obtained. Essentially a Prolog program takes the form of a database of knowledge about a particular problem domain, with a facility to query that database. Essentially Prolog can be thought of as a ‘question and answer’ environment which is continually waiting to answer queries about knowledge in its database. Prolog (and predicate logic) provides a powerful mechanism to represent and reason with knowledge. However, some things are difficult to represent using Prolog, particularly facts that involve uncertainty and beliefs.

## **5.4 Agent Representation Languages**

Various research projects have investigated languages for implementing intelligent agents in recent years [Muller et al., 99]. In the early stages, agents were local to individual projects and their languages were mostly idiosyncratic. As a result there are a large number of representation languages, each with their own particular characteristics, which do not have inter-agent communication capabilities. An obvious

solution is to have a lingua franca, where ideally all agents that implement the same lingua franca would be mutually intelligible [Huhns and Sing, 97]. However, the agent community is still a long way from attaining this goal.

The following sections discuss three commonly used agent languages that can be used to implement knowledge representation in an agent-orientated framework, in order to give an appreciation of the types of languages that may be employed.

### **5.4.1 KQML**

KQML (Knowledge Query and Manipulation Language) [Finin et al., 97] is a language and protocol for exchanging information and knowledge. It is part of a larger effort, the ARPA Knowledge Sharing Effort [Bradshaw, 97] which is aimed at developing techniques and methodology for building large-scale knowledge bases which are shareable and reusable. KQML is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML can be used as a language for an application program to interact with an intelligent system, or for two or more intelligent systems to share knowledge in support of cooperative problem solving.

Communication takes place on several levels. The content of the message is only part of the communication. Locating and engaging the attention of another agent with which an agent wishes to communicate is part of the process, packaging a message in a way that makes it clear the purpose of an agents communication is another. KQML assumes the message transport is reliable and preserves the order of messages, but does not guarantee delivery times. For this reason, the underlying paradigm of communication is asynchronous. At the application level, synchronous communication is achieved by tagging messages to relate to each other. For example, responses to queries may be linked. In this way, KQML supports some elementary interaction protocols, although more sophisticated protocols must be built externally to KQML.

When using KQML, a software agent transmits content messages, composed in a language of its own choice, wrapped inside a KQML message. The syntax of KQML is based on a balanced-parenthesis list. The initial element of the list is the performative; the remaining elements are the performatives arguments as keyword/value pairs. For example, a message representing a query about the price of a compiler might be encoded as follows:

```
(ask-one
  : content (PRICE VisualJava ?price)
  : receiver query-server
  : language standard _prolog )
```

In this message, the KQML performative is “ask-one”, the content is “(PRICE VisualJava ?price)”, the receiver of the message is to be a server identified as “query-server” and the query is written in standard form of Prolog.

KQML still suffers from poorly defined semantics and as a result many KQML implementations seem unique. [Cohen and Levesque, 97] discuss difficulties with the current specification of KQML. In particular they have identified ambiguity and vagueness in standard performatives, misidentified performatives and missing performatives. In addition, the issues of security and authentication are only beginning to be addressed by the KQML community.

KQML has been successfully used to implement a variety of information systems using different software architectures, and include projects such as: Magenta (Stanford University Logic Group), KATS (Lockheed Corporation) [Mayfield et al., 95] and COBALT [Benech and Desprats, 97] (Toulouse University and HP - an agent communication toolkit based on KQML and CORBA. COBALT is coded in both Java and C++ languages and uses HP ORBPlus / JORBPlus ORBs.

#### **5.4.2 Telescript / Odyssey**

Telescript [White, 1994] is an object-oriented mobile agent language developed by General Magic Inc. and is arguably the first commercial agent language. Telescript technology has been overtaken by the commercial success of the Internet in general and Java. This, coupled with Telescript being a proprietary language, lead General Magic to reimplement the paradigm in Java and launch a new technology, Odyssey [White et al., 97]. This system effectively implements the Telescript concepts in the form of Java classes.

There are two key concepts in Telescript / Odyssey: Places and Agents. Places are virtual locations that are occupied by Agents. Agents are providers and consumers of goods in the electronic market place. Agents are software processes and are mobile: they are able to move from one place to another, in which case their program and state are encoded and transmitted across a network to another place where execution recommences. Agents are able to communicate with one another - if they occupy different places then they can connect across a network in much the standard way. If they occupy the same location, then they can meet one another.

Four components have been developed to support Telescript / Odyssey. Firstly the Telescript language which is designed for carrying out communication tasks. Secondly the Telescript engine acts as an interpreter for the Telescript language, maintains places, schedules agent execution and provides an interface with other applications. The third component is the protocols set, which deals primarily with the encoding and decoding of agents, to support transport between places. The final component is a set of software tools (and API's) to support the development of Telescript / Odyssey applications.

Although Telescript / Odyssey is technically a very sophisticated mobile agent system it is also costly. The engine requires significant computer resources to run and financially it is an expensive item of software which inhibits its acceptance as a general mobile agent system among nearly all mobile agent developers.

### **5.4.3 JESS / CLIPS**

CLIPS (C Language Integrated Production System) [Giarratano and Riley, 94] is a multiparadigm rule based programming language, based on the OPS5 production rule language [Brownston et al., 85]. CLIPS was originally developed by NASA in 1984 for internal use, but after modifications, version 3 of CLIPS was made available to groups outside of NASA in 1986. Since then CLIPS has undergone continuous improvement. Version 5 was released in 1991, introducing two new programming paradigms: procedural programming and object-oriented programming. Version 6.0, released in the Spring of 1993, added fully integrated object/rule pattern matching and support features for rule based software engineering. Version 6.1 was released in 1998, adding C++ compatibility and functions for profiling performance. Because of its portability, extensibility, capabilities and low-cost, CLIPS has received widespread acceptance throughout industry and academia and has a user base in excess of 5,000 developers.

JESS (Java Expert System Shell) [Friedman-Hill, 99] is a clone of the core of the CLIPS expert system shell. JESS was originally developed by Sandia National Laboratories and contains the main features of CLIPS and is downward compatible with CLIPS, in that every valid JESS script is a valid CLIPS script. The primary representation methodology in both CLIPS and JESS is a forward chaining production rule language based on the Rete algorithm [Forgy, 82].

CLIPS / JESS production systems consist of both conditional statements known as rules stored in production memory and a global database known as working memory. When every condition in a rule is satisfied by matching data in working memory, the rule is placed in the conflict set. Conflict resolution is performed on the conflict set to determine which actions in the production rules are eligible to be executed. CLIPS uses a LISP-style syntax where expressions are enclosed in parentheses and use a prefix functional form. The following is an excerpt from a *deftemplate* statement which is used to define a working memory structure and a simplified example of a rule to print all Java programmers.

```
( deftemplate employee
  ( slot last_name )
  ( slot job_description )
```

```

        ( slot favoured_language
          ( default Java ) ) )

( defrule print_programmer
  ( employee
    ( last_name ?last )
    ( job_description ?job )
    ( favoured_language ?language ) )
=>
  ( printout "Likes Java:" ? last, ? job ) )

```

JESS can be used in two overlapping ways - as rule engine for an expert system and as a general purpose programming language. JESS can directly access all Java classes and libraries, and for this reason is also frequently used as a dynamic scripting or rapid application development environment [Friedman-Hill, 99]. While Java code generally must be compiled before it can be run, a line of JESS code is executed immediately upon being typed. This allows developers to experiment with Java APIs interactively and build up large programs incrementally. It is also relatively easy to extend the JESS language with new commands written in Java or in JESS itself, and so the JESS language can be customised for specific applications. JESS is therefore useful in a wide range of situations.

#### **5.4.4 Language choice**

The previous sections have outlined some of the commonly used knowledge representation schemes and agent languages which are widely available. Chapter 4 outlined the suitability of both Java and CORBA as an ideal framework on which to base the proposed assistant system. Given this, the choice of scheme / language used to implement agents should complement the chosen framework.

Currently, only a small number of Java based products for the development of expert system tools exist in the market place. [Hall, 97a] [Hall, 97b] reviews the five commercially available tools: Advisor/J (Neuron Data Inc.), Ilog Rules for Java (Ilog

Inc.), CruXpert (Crux Inc.), Selectica SRx Selection Engines (Selectica Inc.) and JESS (Sandia National Laboratories). JESS, unlike previous Java based expert system tools, is based on CLIPS which is an open, mature and portable knowledge representation language. A further pragmatic issue in the context of this research is that of cost - as tools such as Selectica and Advisor/J cost large amounts per developer license, whereas JESS is provided free for research purposes.

It is put forward that the use of production rules implemented under JESS provide an appropriate language scheme for the implementation of the agent rules for the proposed system. JESS provides a powerful knowledge representation scheme - in the form of production rules - and an open and flexible Java component which is suited for implementation in a CORBA environment. The choice of JESS may be seen as both a technical and pragmatic one.

## **5.5 Knowledge Acquisition**

Knowledge acquisition is defined as [Buchanan et al., 83]:

*“The transfer and transformation of potential problem-solving expertise from some knowledge source to a program”.*

Essentially knowledge acquisition is the process of acquiring knowledge from a human expert (or group of experts). Knowledge acquisition involves elicitation of data from the expert, interpretation of the data to deduce the underlying knowledge and the creation of a model of the experts domain knowledge in terms of the most appropriate knowledge representation mechanism. For this, knowledge engineers must familiarise themselves with the domain of the expert and represent the knowledge in a form that can be computerised.

Although there have been several moves to use software tools for knowledge elicitation, a UK survey [Smith et al., 94] revealed that 77% of KBS had used some form of unstructured interview to obtain information. Most of these started out with

informal discussions to explain the project and to gather preliminary information, followed by more formal structured interviews.

There are a number of commonly used knowledge elicitation methods as follows:

- **Printed / Electronic Sources** - The simplest form of knowledge acquisition is from printed sources. This includes searching through documents, books and other items of printed material to find the knowledge necessary to build a knowledge base.
- **Interviews** - This is the most widely used method. Informal interviews consist of the knowledge engineer asking spontaneous questions, where there has been little or no planning prior to the interview. By contrast, formal interviews use a variety of techniques exist such as: tutorial interviews, trigger-question interviews, introspective (or ‘think aloud’) interviews and retrospective interviews.
- **Questionnaires** - are a less frequently used technique and are best suited to a well understood domain.
- **Observational Studies** - are carried out at the same time as actual problem solving. That is, the knowledge engineer is present at the exact time that the experts apply their knowledge. Consequently, this allows for the observation of the experts behavior.

For the purpose of this study, the particular method of knowledge acquisition employed is not a primary concern. The chosen method should elicit a sufficient quantity of useful data which can in turn be represented by JESS in the proposed system, thereby demonstrating the usefulness of the intelligent assistant system. Further consideration should be given to the appropriateness of any given method after the demonstration of a prototype system. For this reason, a combined choice of printed / electronic sources and interviews will be used for the purposes of this study. Opinions about the use of interviews vary widely. [Kawaguchi et al., 91] consider they are “*essential in eliciting new knowledge from domain experts*”, while [Cooke and McDonald, 86] refer to them as “*a less than optimal knowledge acquisition technique*”. However the interview process remains the most frequently used method

for obtaining domain knowledge from human experts. This, coupled with printed / electronic sources, should provide an adequate mechanism to elicit knowledge for the proposed system.

## **5.6 Summary**

This chapter presented the key issues surrounding the development of the knowledge base for the proposed assistant system. In particular, three commonly used knowledge representation schemes were outlined and three commonly used agent languages were presented, which concluded with the choice of a production rule representation based on JESS. In addition, the issues of choosing a suitable method for acquiring knowledge was investigated.

Chapter 6 describes the architectural implications for the proposed intelligent assistant system in light of the architectural presentation in chapter 4 and the knowledge base issues in this chapter. An architecture for the proposed system will be described and its component modules presented.