This item is held in Loughborough University's Institutional Repository (https://dspace.lboro.ac.uk/) and was harvested from the British Library's EThOS service (http://www.ethos.bl.uk/). It is made available under the following Creative Commons Licence conditions.

COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

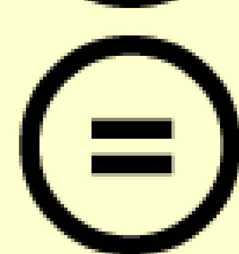
You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.



No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/

A Knowledge Representation Model to Support Concurrent Engineering Team Working

by

Jennifer Anne Harding

A Doctoral Thesis

submitted in partial fulfilment of the requirements
for the award of

The Degree of Doctor of Philosophy of the Loughborough University of Technology

January, 1996

© J A Harding, 1996.

To my mother
for the 3rd
picture

Acknowledgments

This work has been carried out at Loughborough University as part of the MOSES (Model Oriented Simultaneous Engineering Systems) research project, which has examined the role of information models to influence the next generation of computer aided engineering (CAE) software systems. The research was funded by EPSRC under grant GR/H 24273, Exploiting Product and Manufacturing Models in Simultaneous Engineering.

Very many people have directly and indirectly supported this research, and I would like to thank everyone involved with the MOSES project and other research projects in both L10 and the Penthouse Suite for sharing their lives with me through the pleasures and traumas of the last three years.

Naturally there are also some special thanks to make. Arturo Molina, for your friendship and technical knowledge. I was so pleased for you when your research was successfully completed, but I've missed your computing knowledge since you returned to Mexico! Vicente Borja, I will always be indebted to you for helping me to produce the manufacturing element of the software demonstrations, but more importantly, I value the friendship of you and your family.

Keith Popplewell, who has really made an uncertain ambition attainable. You have truly given me continuous support and understanding - thank you for always listening and hearing. Your technical experience and perception have also greatly enhanced the creativity of this work. But most importantly, its been fun working with you!

Martin Scott, for your very valuable industrial knowledge and experiences. Also for your even more valuable help proof-reading and structuring this thesis - I couldn't have coped without you! Whether I fail or succeed you and my mother are always there for me, thank you both.

Finally, I wish to thank someone whose influence on this work goes back over 20 years. I'm not sure when the ambition to write this book was actually born, but it goes back a long time, and at times has seemed highly unlikely to ever be fulfilled! It therefore seems fitting that in writing this I remember the man who actually first set me on the path to achieving it. So, thank you Mr Thomas, you cared for a 15 year old girl who really didn't want to leave school, and made sure that she did not leave mathematics as well.

Abstract

This thesis demonstrates that a knowledge representation model can provide considerable support to concurrent engineering teams, by providing a sound basis for creation of necessary software applications. This is achieved by demonstrating that use of the knowledge representation model facilitates the capture, interpretation and implementation of important aspects of the multiple, diverse types of expertise which are essential to the successful working of concurrent engineering project teams.

The varieties of expertise which can be modelled as instances of the knowledge representation model range from specialist applications, which support particular aspects of design, by assisting human designers with highly focused skills and knowledge sets, to applications which specialise in management or coordination of team activities. It is shown that both these types of expertise are essential for effective working of a concurrent engineering team.

Examination of the requirements of concurrent engineering team working indicate that no single artificial intelligence paradigm can provide a satisfactory basis for the whole range of possible solutions which may be provided by intelligent software applications. Hence techniques, architectures and environments to support design and development of hybrid software expertise are required, and the knowledge representation model introduced in this research is such an architecture. The versatility of the knowledge representation model is demonstrated through the design and implementation of a variety of software applications.

Contents

Acknowledgments

Abstract

Section

1 An Introduction

2 Aims, practices and problems related to Concurrent Engineering team working

3 Requirements from Information Technology Systems to support Concurrent Engineering team working

3.1 Concurrent Engineering team work as a form of Co-operative Working
3.2 Requirements from Future CAE Systems
3.3 Challenges to be met by Future CAE Systems
3.3.1 Distribution
3.3.2 Autonomy
3.3.3 Heterogeneity

4 Information Technology Solutions and Architectures

4.1 Related Software Systems, Environments and Architectures
4.1.1 ABE("A Better Environment") DARPA Strategic Computing Initiative
4.1.2 ARCHON Project, Architecture for Cooperative Heterogeneous ON-line systems
4.1.3 DEKLARE, Design Knowledge Acquisition and Redesign Environment
4.1.4 EUROCOOP, IT Support for Distributed Cooperative Work
4.1.5 GNOSIS (Intelligent Manufacturing Systems)
4.1.6 IDEA (ESPRIT)
4.1.7 IMAGINE (ESPRIT)
4.1.8 ITE, Southampton University
4.1.9 KIWIS (ESPRIT)
4.1.10 Knowledge-Based Engineering Systems Research Laboratory
4.1.11 MFK (Design for X)
4.1.12 PACT (The Palo Alto Collaborative Testbed)
4.1.13 PECOS (ESPRIT)
4.1.14 SCHEMEBUILDER & IDEAS
4.1.15 SHARE

4.1.16 STRETCH (ESPRIT)
4.1.17 TEMPORA (ESPRIT)
4.2 The MOSES Architecture for Future CAE Systems
4.2.1 The Product Model
4.2.2 The Manufacturing Model
4.2.3 Strategist Applications
4.2.4 Integration Environment
4.2.5 Engineering Moderator

5 The Knowledge Representation Model Concept
5.1 Modelling Diverse Types of Software Expertise
5.2 The Production System Metaphor
5.3 Instantiation of the Knowledge Representation Model

6 Design and Implementation of Instances of the Knowledge Representation Model
6.1 Implementation of the Knowledge Representation Model
6.2 Design and Implementation of Instances of the Knowledge Representation Model
6.2.1 A Shaft Design for Function Expert
6.2.2 An Engineering Moderator
6.2.2.1 Knowledge Acquisition Module
6.2.2.2 Design Moderation Module
6.2.2.3 Design Agent Modules
6.3 Extensions of the Knowledge Representation Model

7 Conclusions

References

Glossary of Terms

Figures

2.1 Relative strengths and weaknesses of humans and computers (Oh, 1993)

3.1 Synchronous and Asynchronous Decision Making

3.2 Members of a Project Team are often also members of Functional Teams. Each Team can benefit from an individual's membership of the other.

3.3 Speed of transition depends upon 'New Work' required and the rate of exchange of information. Models, Infrastructure and Culture (Scott 1994)

3.4 Support Requirement Matrix

3.5 Project Team Members need to be aware of other Team Members' Views and Knowledge

3.6 Addition of a new agent to the system requires less changes if a Managing Agent stores knowledge of the expertise existing within the CE team rather than all existing agents storing their own mental models

4.1 MOSES Research Concept

5.1 A Representation of Investigated types of Software Expertise using Booch Object Oriented Design Graphical Notation

5.2 A Representation of Software Expertise Using Booch Object Oriented Design Graphical Notation

5.3 A Representation of Knowledge captured within Rule and Ruleset Classes, using Booch Object Oriented Design Graphical Notation

5.4 A Representation of the Expression Class, which is the parent class for a wide range of similarly behaving object classes, using Booch Object Oriented Graphical Notation

5.5 A Representation of the Simple_Action Class, which is the parent class for a wide range of similarly behaving object classes, using Booch Object Oriented Graphical Notation

5.6 Three types of information can be stored within working memory objects.

6.1 Database Browser showing 5 subclasses (derived types) of the parent class, working memory

6.2 A rule object needs to communicate with an associated condition object and an associated resulting action object. The details of the types and structure of these associated objects may remain hidden from the rule object.

6.3 Database Browser showing one to one, bi-directional associations between objects of rule class and objects of both condition and resulting action classes

6.4 A compound condition consists of a simple condition and a condition

6.5a Database Browser showing an implementation of the Condition class and Simple Condition class

6.5b Database Browser showing an implementation of the Condition class and Compound Condition class

6.6 Database Browser showing a selection of the subclasses of simple resulting action class.

6.7 Instances of the Product Model, Manufacturing Model and Knowledge Representation Model can exist within the same federated object oriented database.

6.8 Database Browser showing sections of the Shaft Design for Function Expert's working memory, which was implemented as a subclass of working memory class.

6.9 A Representation of Engineering Moderator Expertise Using Booch Object Oriented Design Graphical Notation

6.10 A Representation of EM's Knowledge Acquisition Module Expertise Using Booch Object Oriented Design Graphical Notation

6.11 Database Browser showing the Moderator Working Memory database, which contains the Working Memory objects for all the Expert Module Components of the Engineering Moderator.

6.12 **A Representation of the knowledge stored by the EM relating to design agent expertise existing within or interacting with the CAE system.**

6.13 **Database Browser showing the relationships between the EM's Design Expert element of the Design Moderation working memory and Mod Agent Values working memories.**

6.14 **Database Browser showing the instance of a Mod Agent Values Working Memory containing knowledge of the Manufacturing Strategist design agent.**

Appendices

I **Data Definitions for Knowledge Representation Model as implemented in DecObject DB**

II **Tables showing implemented Sub-Classes of Expression and Simple Resulting Action**

1. An Introduction

The purpose of this thesis is to demonstrate that a knowledge representation model (KRM) can provide considerable support to concurrent engineering (CE) teams, by providing a sound basis for creation of necessary software applications. This is achieved by demonstrating that use of the KRM facilitates the capture, interpretation and implementation of important aspects of the multiple, diverse types of expertise which are essential to the successful working of CE project teams. Varieties of expertise which can be modelled as instances of the KRM range from specialist applications which support human designers with highly focused skills and knowledge sets, to applications which specialise in management or coordination of team activities. Thus it will be shown through exploration of the KRM concept and structure, consolidated by several examples of implementations of software instances of the KRM, that the KRM can successfully support CE project teams by facilitating the capture and implementation of the range of software expertise which is essential for effective working of a CE team.

In most situations, the problem must be examined before a solution may be obtained, therefore, initially an examination is made of the team activities which must be supported. In order to do this, consideration is first given to the aims, practices and problems related to CE team working. These issues are explored in chapter 2. Then in chapter 3, ways in which information technology based systems can provide solutions, and thereby support CE team working are considered.

Analysis of the findings of chapters 2 and 3 indicates that various types of intelligent support are required within Computer Aided Engineering (CAE) systems in order to provide comprehensive support for CE team working. Existing and proposed software solutions and architectures which may provide some support for CE team

working are considered in chapter 4. The strengths and weaknesses of these solutions are discussed.

Examination of the identified requirements and proposed solutions indicate that there are various areas in which intelligent software solutions may support and empower the CE team, without restricting their modes of working. These include design strategists or design suggesters, which can work with the user and offer support to him through innovative aspects of the design. More self-reliant, automated, artificial intelligence solutions may be appropriate for repetitive, well-defined aspects of the design. Intelligent support may also be provided for coordination, integration and communication activities. The depth and breadth of these examples indicate that no single artificial intelligence paradigm can provide a satisfactory basis for such wide-ranging requirements. Hence techniques, architectures and environments to support design and development of hybrid software expertise are required, and the KRM introduced in this research is such an architecture. The KRM which facilitates the design and implementation of hybrid software experts is proposed in chapter 5.

In chapter 6, examples of diverse types of software expertise which have been designed and implemented as instances of the KRM, using case studies, are given. The diversity of knowledge structures and expert functions within these examples demonstrates the value and flexibility of the KRM. This has been proved further through case study demonstrations involving implementations of the example software experts. Further details of the software implementations are given in appendix I, and appendix II.

The meaning of particular terms can vary in different papers, therefore, a glossary of terms has been included at the end of this publication, to clarify the meaning of key

words and phrases which are of importance throughout this thesis. For example, throughout this work the terms data, information and knowledge should be considered to have distinct meanings. Data relates simply to words or numbers, the meaning of which may vary and is dependent upon the context in which the data is used. Information is data which is structured or titled in some way so that it has a particular meaning. Knowledge is information with added detail relating to how it may be used or applied. Thus, in terms of a value line, data is at one end (being least valuable), and knowledge at the other (being most valuable), with information somewhere between.

2. Aims, practices and problems related to Concurrent Engineering team working

Most manufacturing industries work under great pressures to produce their products more efficiently and cheaply in order to perform effectively in a highly competitive market place. In recent years an increasing number of manufacturing organisations have introduced policies and working methods intended to promote the adoption of Concurrent Engineering (CE) principles. The philosophy of CE advocates the design of the 'right' product in the first instance by consideration, at the design stage, of all aspects of the product life cycle, from conception through disposal. The main objectives behind these policies are the reduction of product life-cycle costs and product development time scales whilst improving product quality. (Winner et al, 1988), (Nevins & Whitney, 1991), (Parsaei & Sullivan, 1993). Thus CE means that concerns from downstream of the product life-cycle, like manufacturing, are taken into account much earlier on at the conceptual design stage, and CE also implies that a multi-disciplinary approach to design is adopted (Oh, 1993).

Organisations have approached the implementation of CE philosophy in many ways. Issues, such as organisation and work force structure, team working, supplier status, communication and cooperation with customers, etc., have all been tackled. Process improvement techniques, such as statistical process control, Kanban systems, and total quality management strategies, which aim at continuous improvement within processes, are often adopted. Indeed CE is a multi-faceted philosophy for which no single approach can produce all the promised benefits. Many of the approaches which have been advanced for attainment of CE are complimentary in nature, and a thorough comparison of approaches to CE can be found in Dowlatshahi, 1994. CE is a philosophy, not a technology, and some of the

principles it advocates have been practised by different organisations, in different ways, for many years (Jo et al, 1993)

This chapter examines key elements of CE which need to be supported by Computer Aided Engineering (CAE) Systems, and ways in which this may be achieved will be discussed in chapter 3. Essentially, CE is a creative process for the design and development of new products. A creative process can only thrive within an organisation if the organizational culture supports and nurtures it. (Majaro, 1992). Organizational culture has been defined by Schein (1984) as 'the pattern of basic assumptions that a given group has invented, discovered or developed in learning to cope with its problems of external adaptation and internal integration and that have worked well enough to be considered valid and therefore to be taught to new members as the correct way to perceive, think and feel in relation to those problems.'

CE enables creative solutions to problems to be achieved through a clear articulation of the problem and/or goal, using a bank of relevant information. The steps of the process which require experience or expertise, and new thinking, are essentially human processes, and are a manifestation of an individual personality and behavioural responses. These personal traits are influenced strongly by the context of operation (team, management structure and organisational systems), but have a dependency upon data and the way in which information is presented. Thus communication is an important, if not critical ingredient for project success (Pinto & Slevin, 1987). Communication is required to reduce uncertainty. Task uncertainty has been defined by Galbraith (1973) as the difference between (1) the amount of information that must be processed in order to accomplish a task and (2) the amount of information the system (e.g. the project team) already possesses. The greater the uncertainty, the greater the information processing needs of the

group. Communication is dependent upon the way information is presented, and two key aspects of this are media richness and media selection (Stork & Sapienza, 1992). Routine problems with a low level of ambiguity may be handled using lean media, such as memoranda, and group email messages, which are impersonal, and allow only limited feedback. Non-routine problems, which have a high level of ambiguity need to be handled using rich media, such as face-to-face communication, video conferencing or telephone conversations, as these provide immediate feedback, have multiple cues about meaning and allow the message to have a personal focus. Hence CE, which is the outcome of projects for which CE processes have been used is a strongly interdependent mixture of hard and soft issues.

The main factors relating to CE may be categorised into three areas, firstly there are the hard issues, the environments and systems within which people work i.e. the organisational and management structures, processes and techniques introduced to facilitate the introduction and maintenance of CE principles. Secondly there are the soft issues, the factors relating to how people work, to their individual behaviour and the interactions between CE team members. Organizational culture, as defined above, provides the relationship between these hard and soft issues. (Ekvall, 1991) Finally there are information technology issues, which relate to tools and support systems which can be provided to satisfy requirements identified though consideration of both of the above mentioned factors. Good information technology provision establishes a infra-structure within which both hard and soft issued may coexist and when necessary be resolved.

Accurate, easily accessible information and knowledge are arguably the most valuable asset of any business. Indeed they are fundamental to the attainment of CE objectives, since consideration can only be given to all aspects of the product life-

cycle during design, if both suitable information, and the knowledge of how to use that information, exist to support that activity. Information technology provides the mechanism by which the clear articulation of problem and goal may be achieved between individuals, team and organisation. To fully facilitate CE team working a CAE system of the future, as part of a company's information technology system, must provide support for both the hard and soft elements of the CE principles.

One key area in the successful implementation of CE principles is that of Information Management. Accurate, up to date information must be readily available to product design team members to enable them to make correct decisions. One view of 'concurrent design' is literally simultaneous design in the sense of many designers working on the same design at the same time (Londono et al 1989), which requires many designers to access and modify the same design database (Whitney, 1990). Effective use of information technology becomes increasingly important as design teams which are distributed over multiple sites become more common, and this in turn places increased demands upon CAE systems which are required to support such design teams. The requirements of CE team members are very varied, for example, they may simply require quick, simple means of communication between themselves, or alternatively, they may require access to complex analysis software systems. The creative process of design must also be progressed in some manner, i.e. the concurrency aspect of design must be managed or even driven, to ensure positive coordination of the available expertise within the CE team is achieved. The CE philosophy enables creative solutions to be produced and information technology can provide the infra-structure through which access to the necessary bank of knowledge and information is gained.

Innovative design is essentially a human process, and therefore no single automated system will solve all design problems, but integrated, cooperating systems may

serve to keep human designers informed, involved and committed to each others' activities as these bear on their own primary fields of interest.

	HUMAN	COMPUTER
Strength	• flexible • possess common sense • creative and intuitive • bouts of inspiration	• permanent external memory support • consistency maintenance • information structuring • powerful retrieval capabilities • visualisation support • simulation • modelling
Weakness	• mental and memory capacity constraint • slips of skills and attention • memory lapses • succumb to work pressure • knowledge decay • insufficient knowledge with limited discipline breadth, i.e. unable to know it all	• lack common sense • not creative • does not know when to break the rules

Figure 2.1 Relative Strengths and Weaknesses of Humans and Computers (Oh, 1993)

In an automated system approach, problem-solving is almost completely performed by the computer, i.e. the computer assumes control of the design activity proper,

whereas if a cooperative problem solving approach is adopted, the computer and human share the problem-solving tasks in a kind of synergistic relationship. In order to determine delegation of responsibilities of both the computer and the human designer, it has been considered helpful to list out the strengths and weaknesses of each, see figure 2.1 (Oh, 1993). The belief that the appropriate approach to computer-based design tools is for the computer to provide decision support and allow the human designer to apply the judgement, is shared by Bracewell et al (1994).

Many types of expertise will be involved at different stages of the product design and development, and historically the information and knowledge used and generated has been stored in various forms and in different locations. This distribution of information restricts information sharing and leads to problems of information duplication and information inconsistency. Thus an information-integrated system is required, i.e. one in which the primary mechanism for achieving system integration is information (Mayer & Painter, 1991).

Significant challenges must be met by management and workforce to initiate and maintain effective CE team working in any environment. It is commonly felt that the multi-discipline project team should be formed at the start of the design process (Corbett et al, 1991), and it is helpful if teams can remain together and develop throughout the project. Unfortunately this is just not practical when any individual team member may be a member of several different teams (Scott, 1994). Also the complexity and magnitude of these challenges increases enormously when team members are widely distributed both geographically and culturally. Therefore, in this research consideration will be given to the requirements and challenges to be satisfied by future CAE systems in order to support design teams which use CE

techniques in the context of modern, changing, multi-site or global enterprises. In this context, the author defines CE as being:

> **'An holistic methodology for the coordination of distributed, heterogeneous expertise to achieve cost-effective, market-driven products in minimum time scales'.**

To effectively work within such a context future CAE systems require elements beyond those available in existing systems. They particularly require elements to actively promote concurrent working. The need for human integrators to liaise between functional groups or program managers (mediators) to coordinate the activities of cross-functional teams has long been recognised as crucial in achieving reductions in product design time and costs (Dean and Susman, 1989). The duties, responsibilities and interdependencies of these individuals become much more complex when the CE team is widely distributed and particular team members may be members of several CE teams. It is therefore proposed that future CAE systems must also provide support for these integrating and coordinating activities, and that the support required goes beyond the provision of an integration environment to enable different software packages to work together. These support requirements will be examined in detail in chapter 3.

The discussion in this chapter highlights the multi-facetted aspects of CE. Various types of expertise must be combined and work together in a successful implementation of the CE philosophy. Such expertise falls into two categories, specialists with highly focused skills and managers whose role is to co-ordinate the activities of specialists and promote concurrency in working. Ways of supporting both these categories of expertise will be explored throughout this thesis.

3. Requirements from Information Technology Systems to support Concurrent Engineering team working

3.1 Concurrent Engineering team work as a form of co-operative working

Chapter 2 highlighted the important role which future CAE systems, as part of industry's information technology networks, have in promoting and enabling CE philosophy to fulfill its promised potential in improving product design and development. The primary function of the CAE system should be to support the human designer, not to attempt to replace him with automated artificial intelligence systems. Part of the role, i.e. through maintaining information and knowledge bases, and facilitating the sharing of a common consistent copy of these, can be clearly identified. CAE systems must however do more if they are to actively facilitate CE team working. They must provide support for the diverse types of expertise which enable the various, difficult issues relating to both the hard and soft aspects of CE team working to be handled and controlled. Historically such expertise has existed in the guise of human experts, both specialists and managers, most likely supported by pieces of application software, taking various forms (Moynihan 1993) which can range from applications supporting simple data processing activities, through decision support systems, using complex algorithms, to expert systems. In this work an expert system is considered to be any computer system which demonstrates expert performance in a given domain.

Thus human experts with knowledge of the specific problem domain may be supported by software expertise, and there is a requirement from future CAE systems that they must easily integrate with many types of software expertise, so

that the user should not be unduly restricted in his methods of working. A human expert, interacting with the CAE system, by means of a software application which may demonstrate an associated form of artificial expertise, will be referred to in this work as an agent. The purpose of collaborative human/computer problem solving systems is to attain a level of decision quality superior to the level attainable by either the human or computer alone. (Hale et al, 1991)

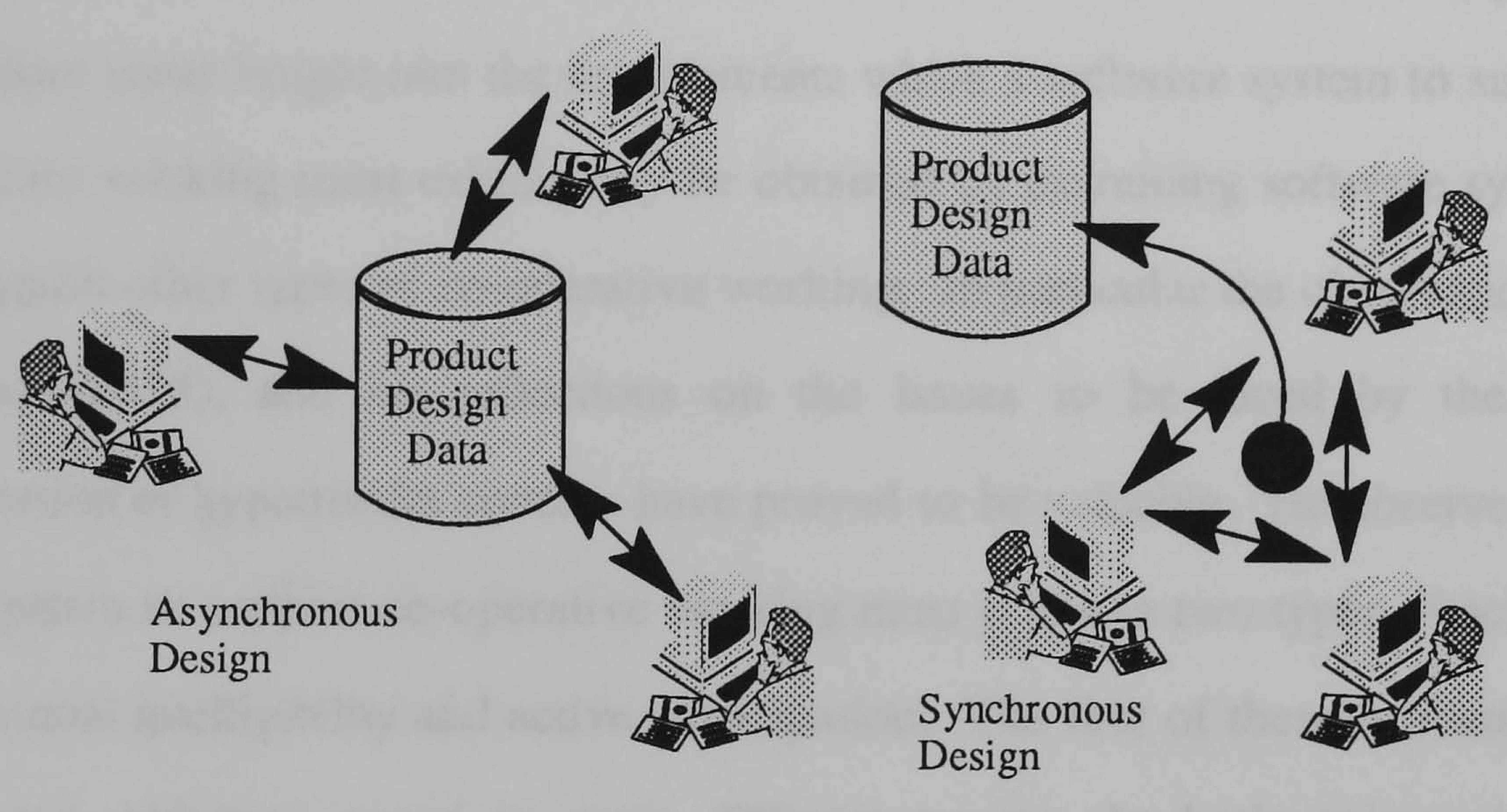


Figure 3.1: Synchronous and Asynchronous Decision Making

In this chapter, several types of expertise which are essential for effective CE team working are identified, and these will be considered in detail below. When these are read, it should be remembered that CE team members have 2 basic modes of working, either asynchronously, when individual members can continue with their own area of design, and safely make decisions without reference to agents with different types of design expertise, or synchronously, when some

negotiation is required between design agents, as a current aspect of the design may impinge upon the domains of several design experts, (figure 3.1). The real difficulty lies in deciding when the changes between synchronous and asynchronous activity should be made, particularly when the design team is highly distributed. This will be discussed further later in this chapter, and in chapter 4.

CE team working is a form of collaborative or co-operative working, and therefore some insight into the requirements which a software system to support CE team working must exhibit may be obtained by examining software systems to support other types of co-operative working. In particular the observations of Halasz (1988), and his reflections on the issues to be faced by the next generation of hypermedia systems have proved to be valuable. He observes that any system to support co-operative working must promote two types of activity, i.e. mutual intelligibility and active participation. The first of these is commonly accepted, although stated in many different terms. In basic terms, mutual intelligibility is taken to mean the existence of a common understanding. If two agents are to successfully communicate together, they must be able to understand each other. This may be achieved in many ways, e.g. by a common language and/or culture, effective translations, or use of common sources of information which may be individually accessed and comprehended by each. The task of actively promoting participation of all team members is an important part of the duties of an effective project manager, and these duties become more difficult to carry out effectively when the team is widely distributed. Yet the need for the software system to actively promote participation of human

designers is not commonly considered, or explicitly dealt with within proposed software system architectures. Individual agents must be actively encouraged to interact with other team members whenever this is most effective for the team activities and attainment of team goals. Often individual agents are very willing to make such switches between asynchronous and synchronous working, but may be unaware when such a switch should be made, particularly when team members seldom, if ever meet, and their disciplines are very different. Therefore some sort of management activity may be required to actively promote participation in a change of working mode.

Bobrow (1991) also considers issues of interactions. He examines systems consisting of active agents, which may be human or programmed machines, communicating among themselves and interacting with the world to solve multiple goals. He claims there are three dimensions of interaction which must be considered. The first dimension is communication i.e. there must be some common ground of mutual understanding. The second dimension is coordination, which is necessary both to share resources, and to jointly commit to future action. The third dimension is integration i.e. to be useful, agents must fit in with the current work practices of both people and other computer systems. These views, although expressed in different terms, relate closely to the previous observations made in this thesis. Mutual intelligibility and mutual understanding are clearly similar requirements, and the dimension of integration appears to be closely allied to the view that creative process can only thrive if it fits into an organizational culture which supports and nurtures it.

3.2 Requirements from Future CAE Systems

The distribution of CE team members is considered to be critical when examining the support which must be provided by CAE Systems. Each individual team member must be helped to apply his particular expertise to design the best possible product but he also needs to be aware of when or how design decisions he makes could affect the views or aims of other members of the CE team, or of how his decisions may be affected by the overall product strategy, and the work of other product CE teams within the organisation. These are very taxing demands to place on an individual whose working location is remote from parts of the organisation, and may even be in a different time zone from other members of his CE team. Each team member therefore requires the best possible support which can be provided from computer-based design tools. The support required from the CAE system is multi-faceted, and for this reason examination has been made of the types of support required, at different levels, and dimensions, in an attempt to simplify the necessary analysis. Clearly the breakdown can be made in a variety of ways, but the following has proved to be useful in better understanding the requirements from future CAE systems.

The author believes that CAE systems of the future must be able to provide support on at least three levels: at an organisational level, at a team level and at an individual level. Support at the organisational level covers the satisfaction of the requirements of the organisation within which the team operates. This could include interactions and information exchange between different design teams, or between individual team members and other members of their particular discipline group (figure 3.2). It could also include provision of information to support senior management strategic decision making.

Support at the team level covers the satisfaction of any requirements imposed by team working methods, for example, any activities which will assist the team to work as a single, effective, efficient entity. This could include promotion and maintenance of a common view of the team's objectives, and encouragement of exchange of knowledge and comprehension between team members.

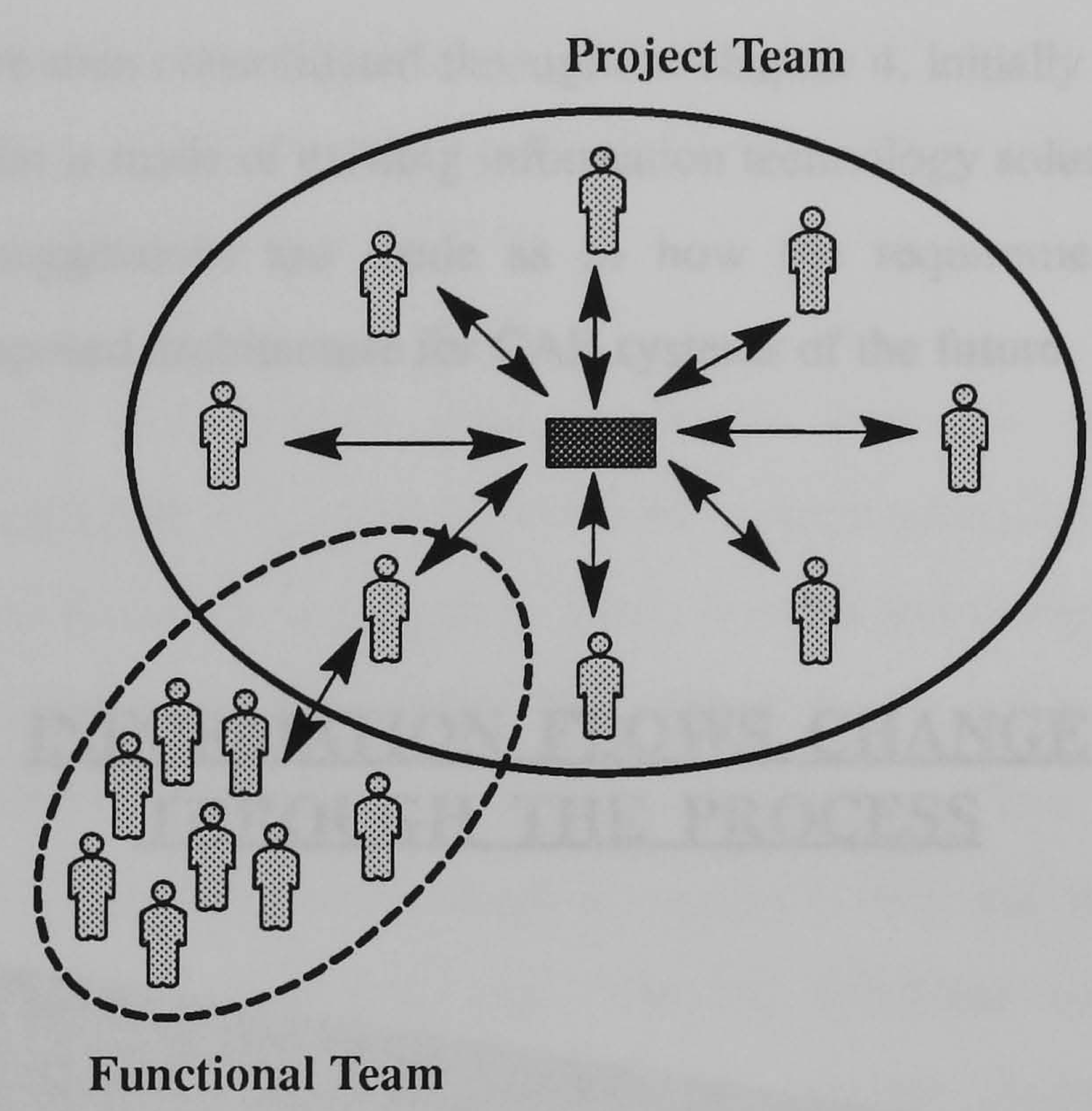


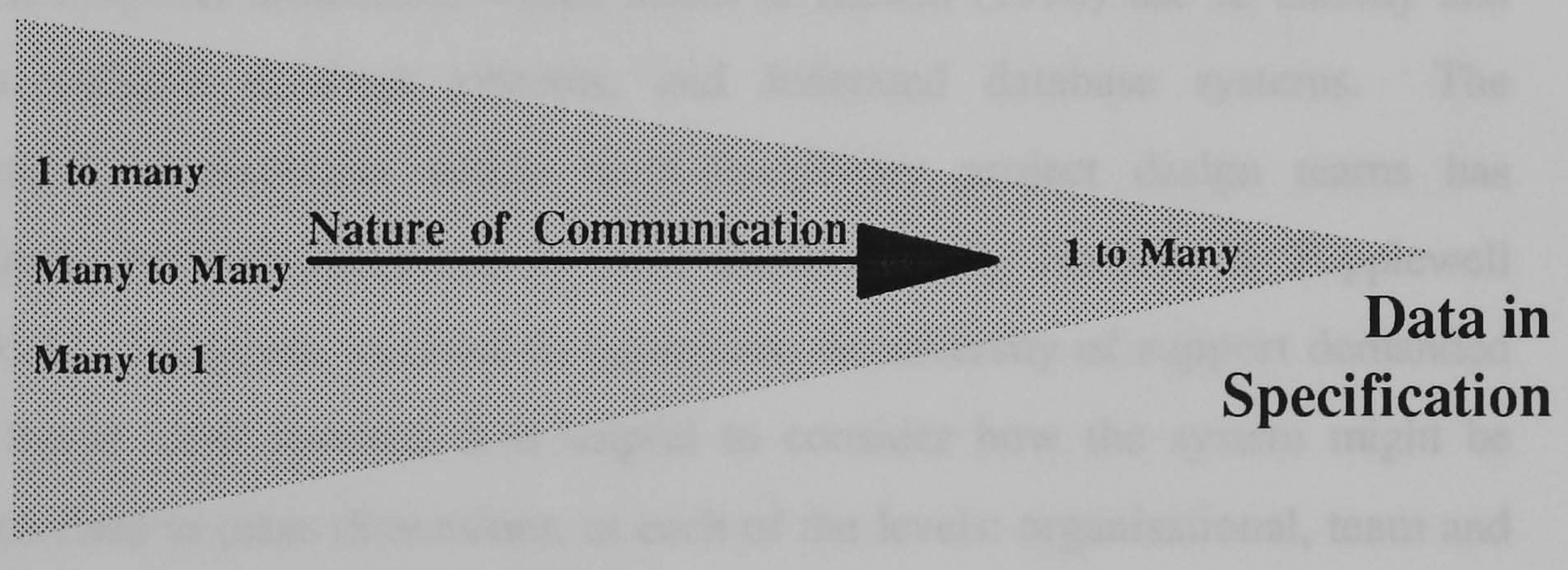
Figure 3.2: Members of a Project Team are often also members of Functional Teams. Each Team can benefit from an individual's membership of the other.

Support at the individual level requires the CAE system to be sufficiently flexible to provide assistance for all members of the design team, irrespective of their role within the team. The CE product design and process development team will

be multi-disciplinary, its members representing many functions, including design, manufacturing, marketing, finance, production and senior management. The CAE system should therefore contain applications and software experts to support the diverse disciplines of each individual member of the team.

Satisfaction of requirements at these three levels places diverse and even contradictory demands upon the CAE system. The remainder of this chapter explores a range of these requirements, and considers the problems they impose. These findings are then consolidated throughout chapter 4, initially in section 4.1 where examination is made of existing information technology solutions and then in section 4.2 suggestions are made as to how the requirements might be satisfied by a proposed architecture for CAE systems of the future.

INFORMATION FLOWS CHANGE THROUGH THE PROCESS



IDEAS

Figure 3.3: Speed of transition depends upon 'New Work' required and the rate of exchange of information. Models, Infrastructure and Culture

(Scott, 1994)

Why is it necessary to examine the environment within which the CAE system operates? Why are the demands placed upon the CAE system so diverse? An enterprise which has adopted the CE philosophy is clearly a multi-agent system, (Sycara, 1990) (Bobrow, 1991) and considered from the three levels described above, can in fact be considered as several different types of multi-agent system. Problems may occur in the design since individual agents may have different mental models of the design, and they may not 'speak the same language'. (Sycara, 1990). So, during the design process, many views of the product must evolve into a common view or vision, through the passage and exchange of information, see figure 3.3, (Scott, 1994).

The multiple multi-agent systems could be broken down as multiple design teams, or as multiple functional groups within the organisation, or as individuals, each with a different design perspective within a particular design team. Bird (1993) believes that multi-agent systems should be characterised in at least three dimensions, i.e. distribution, heterogeneity, and autonomy. These are the same three orthogonal dimensions which Sheth & Larson (1990) use to classify and define multiple database systems, and federated database systems. The integration of expertise within multi-disciplinary project design teams has previously been characterised in these dimensions by Harding & Popplewell (1994)(1). However, in order to appreciate the diversity of support demanded from future CAE systems, it is helpful to consider how the system might be characterised in these dimensions, at each of the levels: organisational, team and individual. A breakdown can be achieved by considering the three levels and the three dimensions as the rows and columns of a three by three matrix, (figure 3.4). The breakdown in this matrix is not exhaustive, but it does highlight problems which must be tackled by the CAE system, and help to identify

priorities in the requirements which the CAE architecture must satisfy. An example breakdown is given in the following section.

Levels

Dimensions	Organisational	Team	Individual
Distribution	Move information between multiple sites	Reduce remoteness and promote exchange of information between team members at different physical locations	Make information available to individuals
Heterogeneity	Support Organisations to achieve different missions	Support Project Teams to achieve different goals	Support Individuals to perform different jobs
Autonomy	Discourage multiple individual stores of information	Support team members to work as individuals, or as a group, and transitions between these two types of working	Support individual's preferred manner of working

Figure 3.4: Support Requirement Matrix

3.3 Challenges to be met by Future CAE Systems

CAE systems should support human designers to make the best, innovative use of their expert skills possible. This section explores the challenges which need to be met by future CAE systems in order for them to support distributed CE team working, viewed in the context of individual, team and organisational levels described above. The section has been included to provide a requirements context against which the proposed architectures, described in chapter 4, can be evaluated. No existing CAE architecture has been identified which can meet all the challenges identified, especially relating to the requirements of coordination and promoting concurrent working. A thorough review of the capabilities of current and proposed computer aided simultaneous engineering systems and architectures can be found in Molina et al, 1995.

3.3.1 Distribution

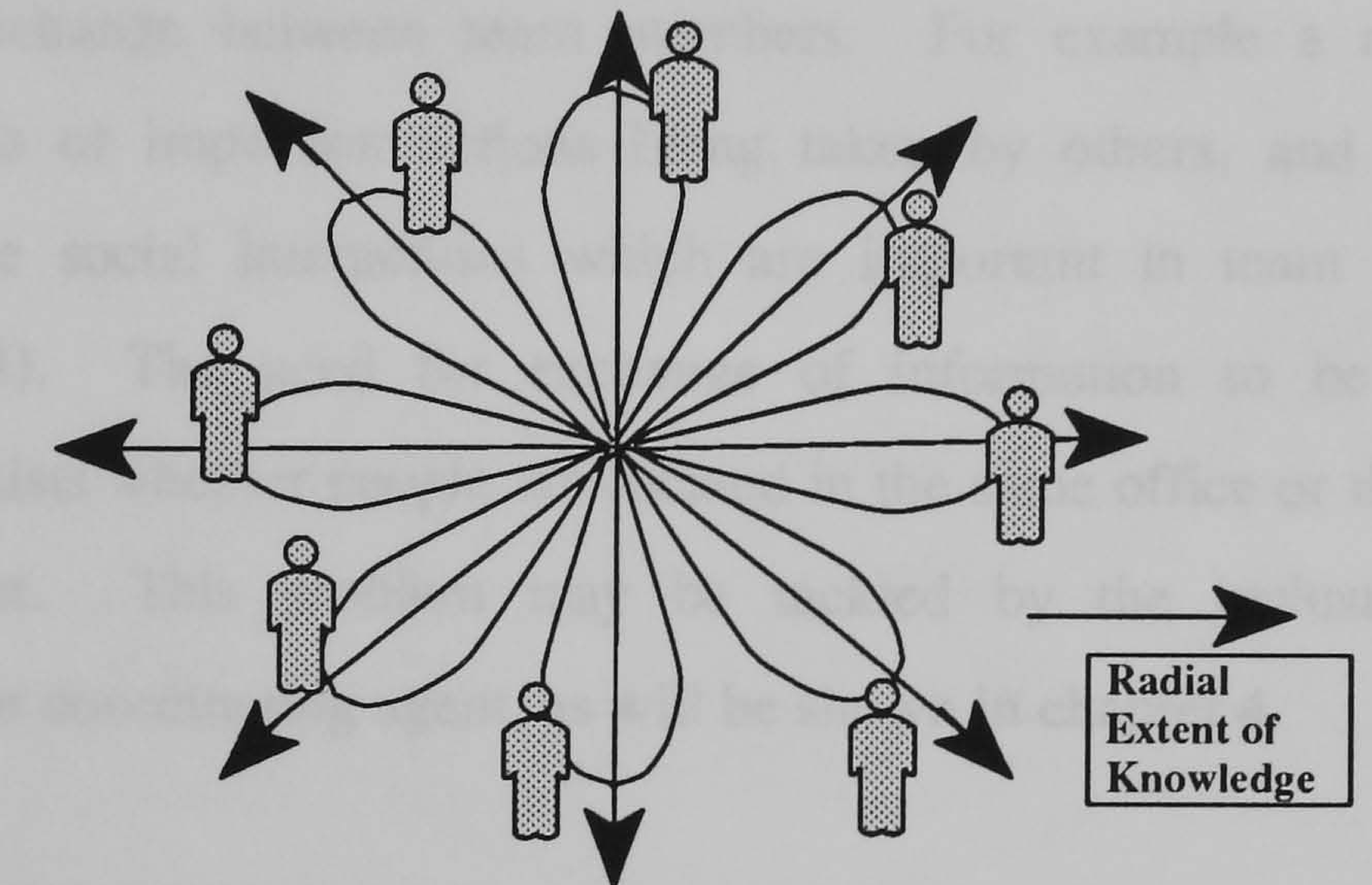
Consideration must be given to distribution relating to both human and computing issues. Discussion on distributed computing generally places emphasis at a level that is closely related to physical connection of different processors, secure transmission of data among them and the corresponding operating system problems of scheduling different processors. Yet distributed problem solving, i.e. the decomposition and coordination of computation in a distributed system are better viewed at a higher level of abstraction, and Chandrasekaran (1981) identifies good reasons for distribution, including controlling the complexity of computation, changes are easier to make to modular systems, and the fact that it is good research strategy to look for decompositions of a complex problem.

If distribution is considered at the three levels defined in the previous section, the following points must be examined. The enterprise, and indeed the

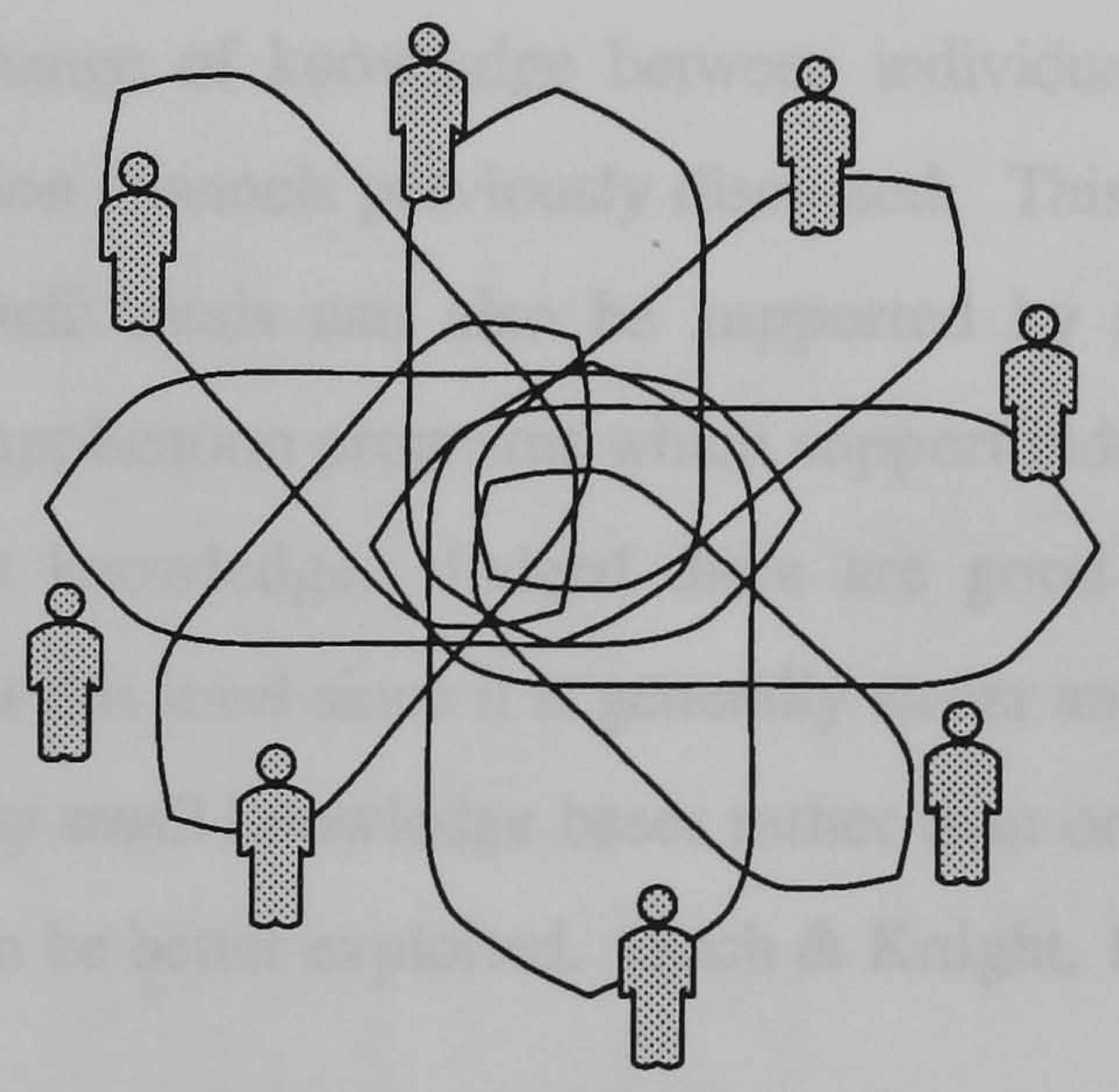
project team may be distributed over many sites, even across many countries. For example, software modelling experts may be located in Britain, electronics experts sited in the United States and the production facilities established in the Far East. At the organisational level, there are many reasons why such distribution may be advantageous: for example, costs related to production in different areas may vary considerably, or particular centres of excellence may have developed over time. The enterprise will therefore require information of many forms to be exchanged between different discipline groups, or project teams located at multiple sites. The sites will need to be linked by networking of various types, and the CAE architecture must include an integration environment to support communications between sites. Thus the CAE system must run across networks, probably with different applications running on different software platforms and working in different software environments.

At the team level, distribution causes different problems. The team members must feel they are working as a single unit, with a common goal, even though they may be widely distributed. Regular team meetings and discussions are an accepted way of promoting exchanges of views and ideas. Within large scale, widely distributed teams such regular face to face meetings become very expensive both in terms of time and money. Electronic methods of promoting communication and cooperation between team members should therefore be explored in the search for viable alternatives, and these methods should be harnessed to the CAE system wherever possible. If co-location of team members is impossible on a regular basis, virtual co-location (Douglas & Brown, 1993) may be an acceptable alternative solution. Indeed it is arguable that virtual co-location provides a better solution, since it facilitates use of

extended knowledge networks (Scott, 1994), see figure 3.2, through functional teams.



a) Traditional Spread of Expertise – Little shared Inter–disciplinary expertise



b) Required Spread of Expertise – Greater shared understanding within multi–discipline team

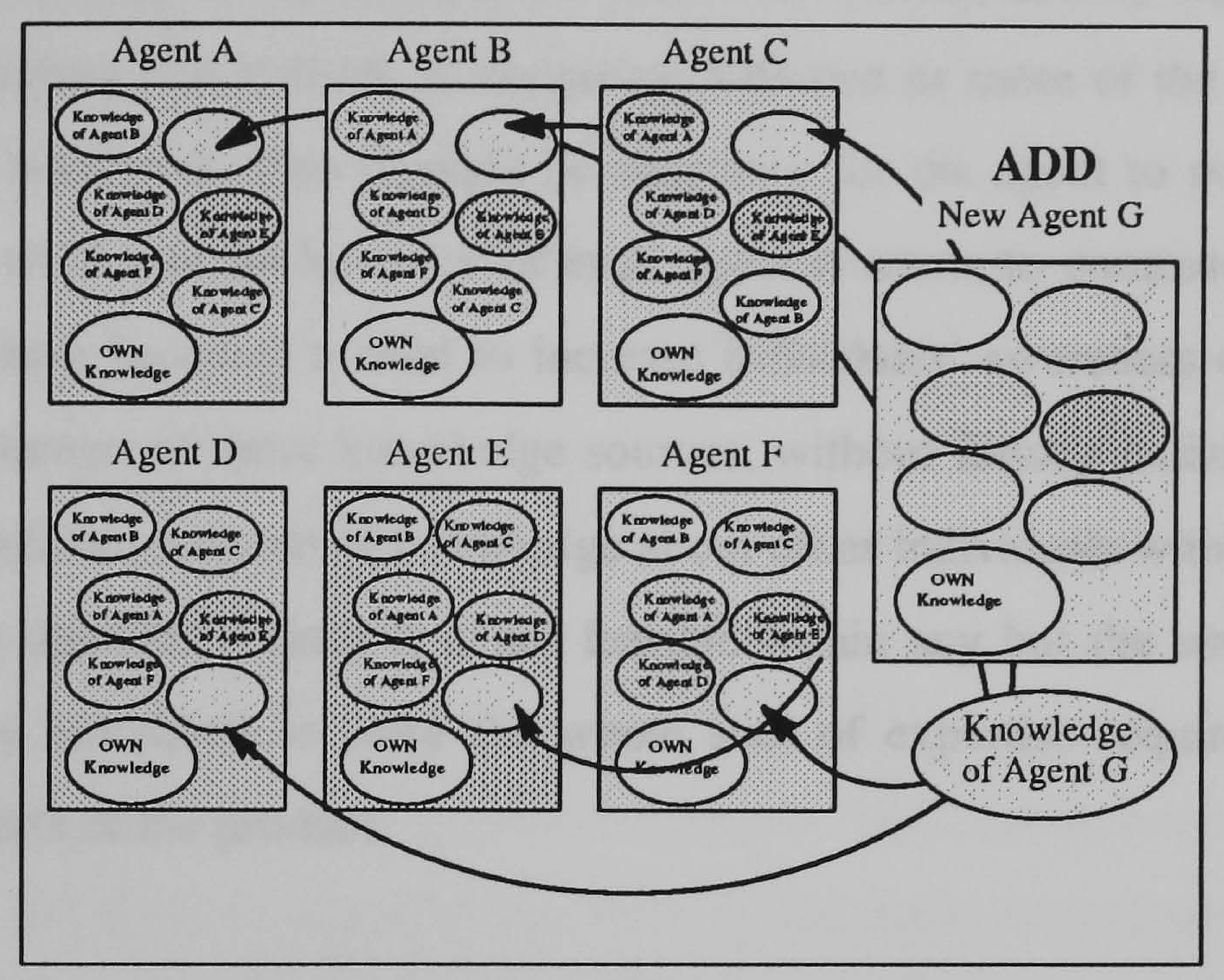
Figure 3.5: Project Team Members need to be aware of other Team Members' Views and Knowledge

Hypermedia/Multimedia systems can meet many of the challenges faced in establishing a virtual working environment, and they provide valuable support

by reducing feelings of remoteness and isolation between co-operating team members who are working at a distance (Ishii & Miyake, 1991). However, additional elements are required to actively promote information and knowledge exchange between team members. For example a means of notifying users of important actions being taken by others, and ways of supporting the social interactions which are important in team working, (Halasz, 1988). The need for exchange of information to be actively encouraged exists whether people are located in the same office or thousands of miles apart. This problem may be tackled by the inclusion of a management or coordinating agent, as will be shown in chapter 4.

Human knowledge is inherently distributed, (i.e. each person has their own knowledge store), so at the individual level, problems of distribution can be partially satisfied by exchange of knowledge between individuals using the network and communication channels previously discussed. This distribution of knowledge between individuals can also be supported by provision of expert systems and other application programs which support individuals with particular types of expert knowledge. Indeed there are good reasons for encouraging distribution at this level since it is generally easier and cheaper to develop and maintain many small knowledge bases rather than one large one, and parallel processing can be better exploited. (Rich & Knight, 1991).

This identifies an apparent contradiction in the requirements from the CAE at the team level, and at the individual level, since at a team level, there is a requirement for each team member to have an awareness of other members of the team, figure 3.5, yet at the individual level, small distributed sources of knowledge are encouraged.



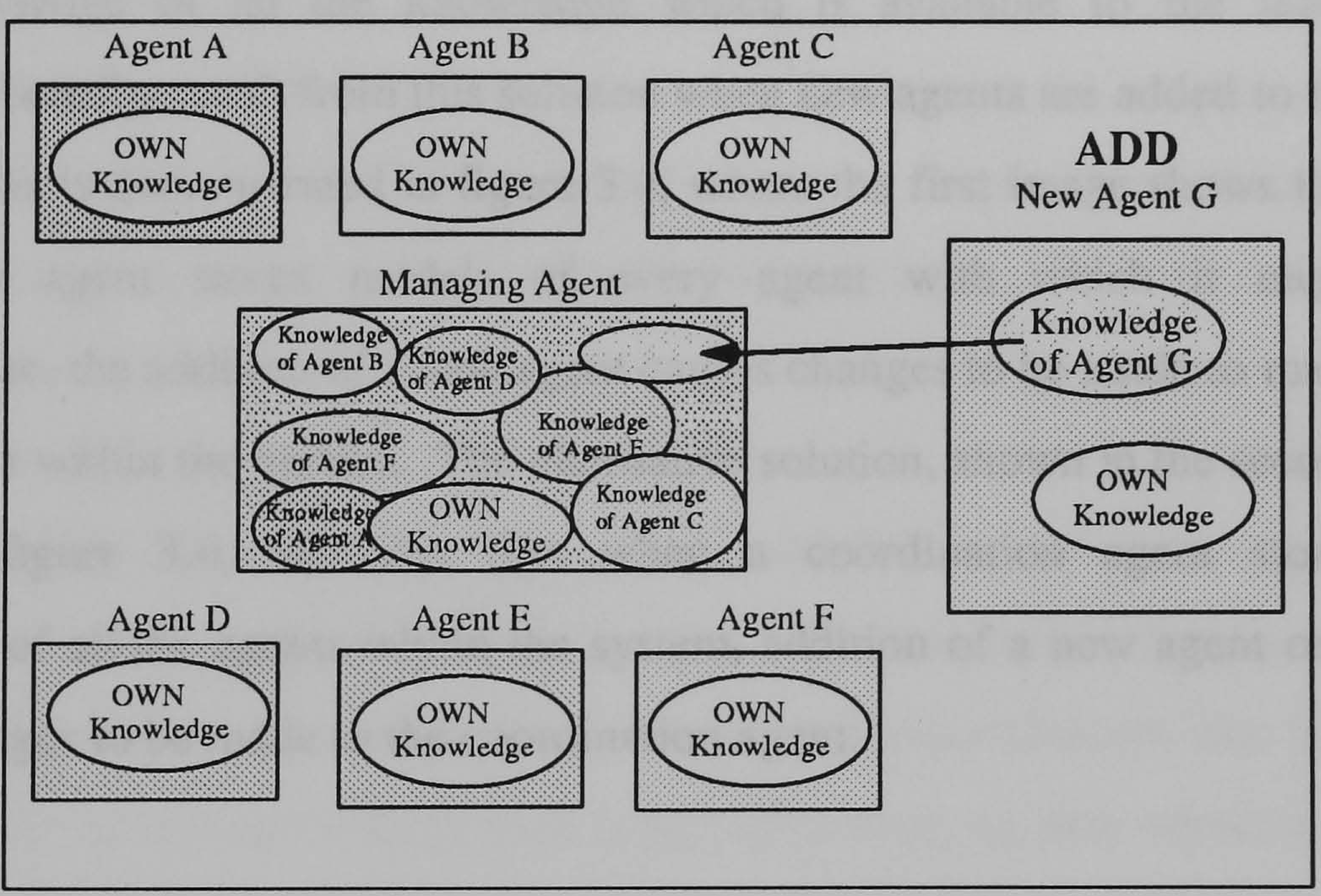


Figure 3.6: Addition of a new agent to the system requires less changes if a Managing Agent stores Knowledge of the expertise existing within the CE team rather than all existing agents storing their own mental models

Adler et al (1992) also identified this requirement for individual elements of the system to be aware of other expertise within the system, as they state, 'if an agent determines that it needs to cooperate with one or more of the other agents in the framework, then it might be necessary for the agent to possess and employ a model of the behavior of each agent it wants to communicate with'. Thus there becomes a need to increase individuals' awareness of the location and content of these knowledge sources, without forcing individuals to store an increasing amount of knowledge about other individuals within the team. This is significant, since it is not feasible within any but the smallest system for any individual to store the whole sum of expertise required to design all aspects of the product.

It may however be possible for a management or co-ordination expert to store knowledge of all the knowledge which is available to the team. Additional benefits result from this solution when new agents are added to the system. This is demonstrated in figure 3.6, where the first image shows that when each agent stores models of every agent with which it might communicate, the addition of a new agent causes changes to be made to many other agents within the system. The alternative solution, shown in the second image of figure 3.6, indicates that when a coordination agent stores knowledge of all the agents within the system, addition of a new agent only causes changes to be made to the coordination agent.

3.3.2 Autonomy

At the organisational level, some level of autonomy will be required due to legal and financial requirements of particular countries. It is believed that generally autonomy should be restricted when it comes to information availability and exchange since many problems can arise when the same

information is not available to all interested parties (naturally it is accepted that politically sensitive information may need restricted access). Duplication of information in separate 'empires' should be discouraged.

At the team level there remains the requirement to reduce individuals' private stores of information, as there is a high risk of inconsistency within multiple stores of information, and this can be achieved by use of common information models. However, there is an additional type of autonomy which must be allowed and supported, team members must be allowed to work separately, or as a team, whichever best suits the current stage of product design. Any CAE system to support CE design must support both synchronous and asynchronous decision making within the team, and most importantly, the switching between the two modes of working. This is a very complex requirement to satisfy, and aspects related to this will be considered further when examining the architecture proposed in section 4.2, since this architecture includes proposals relating to a managing or coordinating agent, which directly supports transition between asynchronous and synchronous working.

At the individual level, team members should also be able to use whatever individual software applications they believe will best assist their work. Inevitably individual's views of which packages or applications are best able to satisfy their needs may change over a period of time, as new versions or software become available. Thus, the CAE support system can therefore never be 'complete', there must always be the potential for additional, probably third party, software applications and experts to be added. This requirement naturally adds complexity to any 'manager' or 'co-ordination' elements of the CAE system

Individual areas of expertise, which may include both human experts and their related software systems, will have their preferred software environments, and any CAE system which truly supports concurrent engineering must enable such environments to be linked and permit communications across the platforms. This is a further challenge to be met by the integration environment part of a CAE architecture.

The separation of organisational, team and individual levels for autonomy can be considered in the following terms. At each level there will be a different goal, and goal congruency (Feltham & Xie, 1994) (Roth & Ricks, 19994) (Schoemaker, 1993) (Vancouver & Schmitt, 1991) is a driving requirement, so autonomy must be restricted to a level which allows this to be achieved. For example at the organisation level the quest for goal congruency will take into account resource scheduling, since resources must be shared between different teams, products or groups. So autonomy must be restricted to the extent that no individual group controls usage of a particular resource to the detriment of all others. At the team level the output from the team must satisfy the perspectives of all members of the team, so individuals must not be allowed to 'do their own thing' to the extent that information is hidden from other team members. However, at the individual level team members must be allowed to adopt a pure approach to the design in the sense that the individuals consider only aspects of the design of direct interest to themselves. This permits the greatest exploitation of individual expertise, but must be supported by a mechanism for identifying potential conflicts between the interests of different individuals.

3.3.3 Heterogeneity

Heterogeneity can be considered at the different levels in the following terms: organisations have particular missions, teams have particular goals and individuals have particular specialities. The CAE system must support (even encourage) differences in approach and style to the extent that jobs, goals and missions can be successfully executed. Once again there may be conflict between the requirements at the different levels, since an organisation's mission may require its project teams to achieve overall wider market share, not cannibalizing each other's market. Whereas at team level, the primary requirement may be to design and produce a product that can stand up well in the market against any competition.

Individual heterogeneity can be supported by Design for 'X' applications (Meyer, 1993), which can provide support for individual design perspectives, and these may be specialist or expert software applications. The writer also believes it is important to maintain purity of design perspectives to prevent trade-offs being made too soon and resulting in sub-optimal solutions.

At the team level, the differences between individual perspectives remains important, but individuals need to be aware of each other's different views and knowledge see figure 3.5, and where this different knowledge can be accessed. In a true team, there is a high degree of overlap of knowledge and learnt competence, and the CAE system should support and facilitate this learning/sharing process. The CAE system can support this by prompting users when someone else might be interested in the addition or change they have made to a design.

The CAE system should in no way restrict methods of working, as to do so may stifle innovation. This suggests that a high level of autonomy (and indeed heterogeneity) should be granted to the human and software agents concerned. However, the most current details of the design and the views and knowledge of individual team members should ideally be available to all team members. A common source of consistent, accurate information is therefore required since concealment, misunderstanding or simply lack of availability of information can lead to sub-optimal team performance, duplication of effort, and faults in the design. This issue may be addressed by the inclusion of information models, to provide data for applications working within the system. The importance of product and other information models is becoming widely accepted (Krause, 1993). It is believed that exchange of information and knowledge can only be achieved if some form of common language, which implies a common understanding of terms, exists between the participating agents (Bobrow, 1991). Since negotiation requires a common language in which the negotiations can be couched (Adler et al 1992). However, the shared or common language may be unique to the team (Scott, 1994). The need for individual agents to co-operate in this way will thus require the autonomy of the individuals to be restricted. Hence, there is an apparent conflict in the level of autonomy which should be allowed both to team members using the CAE system and software components existing within the system. Such conflict is inevitable, yet it is considered possible for the CAE system to balance the permitted levels of autonomy whilst still providing high levels of support, as will be shown in section 4.2.

The heterogeneity of both human team members and software components of the system is an important and powerful aspect of the CAE system. Representatives of the different functions which contribute members to the

project team will each have different perspectives on the emerging product design. Creating a design which satisfies the criteria of any one of these perspectives may well be a complicated process requiring compromises to be made in the satisfaction of multiple design criteria. Thus, satisfying the criteria of multiple design perspectives will inevitably require many 'trade-offs' to be made. Great care must be taken in deciding when the compromises should be made. A predefined path for problem solutions should not be imposed upon designers by the system. If the compromises are made too early, for example, at the component design stage, a less good design may be achieved at the assembly level.

The importance of heterogeneity therefore cannot be over emphasised. The software expertise to assist a particular design perspective should be kept as pure as possible, and be unadulterated by the design values of different perspectives. This can be achieved by supporting individual areas of expertise with specialist strategist applications whose design criteria are concentrated on one specific area of design. However, satisfaction of this requirement does not mean that designers should work in 'splendid isolation', oblivious to all but one view of the product. This is clearly unacceptable, and totally against the advocated CE philosophy. It would inevitably result in problems with the design not being detected until late in the product life cycle by which time they are expensive or even impossible to correct. This requirement for highly focused design perspectives, without isolationism, places increased pressure on any co-ordination agent element of the CAE support system, requiring it to promote communication and co-operation within the team, *at appropriate times*. Elements of the CAE system should not restrict the innovation of human designers by leading them down a predetermined design path and dictating solutions to them. As previously stated, the CAE system is there to

support the human designer, not to attempt to replace him by an automated design process. The designer should rather be offered possible or alternative solutions, and given advance warnings of potential design problems whenever possible. There is, therefore, a need for a flexible, versatile means of detecting when the requirements of one design perspective are being infringed by decisions made from a different design perspective. However, the detection of possible problems is not adequate in itself, the system must also be flexible in how and when such conflicts of interest are reconciled. That is, some design compromises may need to be made as soon as conflicting interests are detected, whereas in other cases, it may be better to flag the problem for reconciliation some time in the future.

4. Information Technology Solutions and Architectures

4.1 Related Software Systems, Environments and Architectures

Various software system, architectures or environments relevant and of value for CAE systems have been proposed. The following discussions will show that whilst several of these make valuable contributions towards satisfying some of the requirements demanded of future CAE systems, none of them fully satisfy the requirements established in chapter 3. Indeed, it has been claimed that although intelligent computer support addresses many different tasks, all of these tasks belong to that part of the design space in which the design (the product) is known, as a concept, as a possible list of standard components, or in more detail (Blessing, 1991).

In the following sections, each architecture or system is considered individually so their particular strengths or weaknesses, in the current context, may best be explored

4.1.1 ABE ("A Better Environment") DARPA Strategic Computing Initiative

This research is not into a CAE system directly, but it does include important integration environment issues. The ABE software system provides an environment for combinable frameworks and associated analysis tools for building intelligent systems. Hayes-Roth et al (1991) state that they sought ways to modularize and standardize knowledge-processing components so that system integrators could access and exploit them. Thus their goal was to create technologies and methodologies for building cooperative, intelligent systems with modular heterogeneous components. Their work has focused on ways to import existing modules, regardless of implementation, and to

treat them as if they were native to the ABE environment. This is achieved by 'wrapping' each module with interface code to make it appear to have been created originally as an ABE module. An aim of this research was to shield software developers and designers from platform changes, and they achieved this by adopting the concept of a virtual machine, whose key properties can be described independent of platform. These key properties are basically the services of a distributed operating system, whose principal services include process creation, initialization, execution, termination and message passing between processes.

As previously implied, this research is of value in the current context solely for integration, and communication issues related to automated systems, as it provides a mechanism whereby a variety of separate, heterogeneous software elements may be linked.

4.1.2 ARCHON PROJECT (ESPRIT P-2256)
Architecture for Cooperative Heterogeneous ON-line systems

This research examines a general purpose architecture which can be used to facilitate cooperative problem solving in industrial applications, using multi-agent systems. (Wittig et al, 1995) (Jennings, 1995). Agents in the ARCHON project appear to be purely software experts, particularly pre-existing expert systems dealing with different aspects of decision making of a given complex environment.

This research is of value in the current context solely for integration, and communication issues, mainly related to automated systems and legacy systems. It provides an architecture within which pre-existing software solutions may be loosely coupled and cooperate in a mutually beneficial way.

The human operator is also treated as an active problem solving member of the system, but the main focus of the research appears to be on the software expert elements.

4.1.3 DEKLARE (ESPRIT)
Design Knowledge Acquisition and Redesign Environment

This research attempts to define a system which can encapsulate the design guidelines and standard of a company. The developed framework will allow existing CAD tools and inference engines with design databases to be combined in order to provide an interactive design advisory system for interactive redesign. In many aspects this systems is the antithesis of a system to satisfy the requirements identified in the previous chapters, since it apparently tackles the issues of ensuring cooperation by allowing definition of a pre-defined design project path.

4.1.4 EUROCOOP (ESPRIT)
IT Support for Distributed Cooperative Work

The application area for this project is essentially that of project management for bridge and tunnel construction industries. However, the research does address many of the issues related to the requirements of concurrent engineering team working as identified in the earlier chapters. The approach allows for the integration of existing computing components and new tools, with the intention that they should be able to interoperate with each other. The need to share information is accepted, and this appears to be achieved and promoted through using hypermedia to link documents and databases. Significant use is made of hypermedia, particularly as a means of facilitating both individual and group working. The research also addresses the need to support both synchronous and asynchronous working, and implicitly touches on the switching from one type of activity to the other. This is done by

notifying users of deadlines, expected actions, and warning them of emerging difficulties. Regrettably, it has not yet been possible to obtain details of how the 'emerging difficulties' are identified.

4.1.5 GNOSIS (Intelligent Manufacturing Systems)

The application area for this research was enabling technologies for design and manufacture, with a main joint demonstration working from functional design to STEP based manufacturing. A commitment to information sharing and product modelling (Gu & Chan, 1995) is implicit in the research through the use of standards such as STEP (Wu et al, 1992).

Areas of software expertise to support designers were also considered, for example MCOES (Manufacturing Cell Operator's Expert System) which aims to shorten the production lead time and improve the quality of design and manufacturing of one of a kind and small batch products. Integration and coordination issues were researched through the Mediator element of the project. The Mediator architecture enables users to collaborate synchronously or asynchronously through processes running anywhere on the network (Gaines & Norrie). Many of the software systems may have been developed separately, without any coordination facilities, and the Mediator architecture enables them to interact, using a range of generic and proprietary knowledge and data interchange formats. Thus the Mediator software needs to know a lot about the applications, whilst the applications need to know virtually nothing about Mediator. The visual language used in the system may be used as a 'wrapper' to existing applications, or as an embedded component for other, new applications.

This research clearly addresses most of the issues and requirements identified earlier in this thesis, but does not include elements to prompt or support the user through changes from asynchronous to synchronous working, although both of these modes of working are supported individually.

4.1.6 IDEA (ESPRIT)

Research into an intelligent object oriented database system, tested in the application area of biochemical structures and managing system. Basically a database system with explanation facilities which can support multiple language paradigms and parallelism. This research is of value in the current context solely for information exchange and sharing issues.

4.1.7 IMAGINE (ESPRIT)

This research aims to provide a sophisticated environment to support interaction and cooperation within a multi-agent system. In this research the definition of agent is very similar to the definition used in this thesis. The application area of this work was urban traffic control and airport catering and workflow management. Exchange of information and knowledge is implicit in the research, and software expertise to support human experts are clearly considered. The main interest of this research in the current context is for issues relating to coordination and integration. Human agents may be supported by this environment through different modes of working, but there is no support provided to the user to initiate changes between different modes of working.

4.1.8 ITE (EPSRC Grant GR/H 43038) Southampton University

This research explored the potential of an open model for hypermedia as an operational interface with an advanced manufacturing environment. It has

provided an environment for the implementation of large-scale hypermedia information systems for industrial applications. The main areas considered in the research were maintenance of machinery, fault finding activities and operator training. Integration of a knowledge based system (KBS) with the hypermedia system enables the user to input symptoms of the fault, which are then evaluated so the user is provided with details of possible causes. The KBS is loosely coupled with the hypermedia system, and communications are achieved by message passing. (Heath et al, 1994) (Crowder et al, 1995).

This research has clearly not produced a design tool or environment as such, since that was not its objective, but the techniques adopted are considered to be valuable and transferable, for example when considering methods of heightening individual's awareness of other design perspectives. It therefore provides valuable mechanisms for transfer or sharing of information and possibly knowledge (since it is arguable that a video clip showing how something is done is transferring knowledge rather than information). However, the motivation for agent interaction with the system is appears to be their own desires or requirements, i.e. there is no coordination or management activity to promote individual agents' active participation with the system.

4.1.9 KIWIS (ESPRIT)

This research provides an integrated knowledge-representation language and programming environment for distributed databases and knowledge bases. Thus, in the present context the research is relevant for integration issues related to distributed databases.

4.1.10 KNOWLEDGE-BASED ENGINEERING SYSTEMS RESEARCH LAB

The research from this laboratory covers a wide range of projects covering technologies for the next generation of computer-aided engineering systems and computer tools to improve engineering practice. The report from the laboratory (Lu, 1992), identifies one of the fundamental challenges of CE as being that product development practices have changed from being centralized to being distributed. Within this research they see that CE requires the four challenges of *integration* (of complementary engineering expertise), *cooperation* (of multiple competing perspective), *communication* (of upstream and downstream concerns) and *coordination* (of group problem-solving activities) to be simultaneously satisfied.

Individual projects from this laboratory which are particularly relevant are IDEEA, which provides a way of integrating AI techniques (frame-based representation, constraint-based language, rule-based reasoning, truth-maintenance systems and object oriented composite values) through a blackboard structure. This would enable software expertise to be developed to support the human designers. INDEED which uses object oriented database technologies to provides consistent and persistent storage of information, for use by multiple designers. It effectively supports information sharing and both synchronous and asynchronous modes of working, but does not provide support to initiate or actively promote changes in mode of working.

The SWIFT (System Workbench for Integrating and Facilitating Teams) project provides the integration environment, covering three types of integration functions, namely, knowledge, tool and team integration.

Knowledge integration is achieved by supporting combinations of AI representations of knowledge (as in IDEEA). Tool integration permits users to access different computer tools and to handle the heterogeneous data/knowledge required and generated by these tools. Lawley (1992) claims SWIFT achieves team integration by facilitating group communication and by coordinating team activities, however, it is not clear that group interactions are actively promoted as there is no mention of either a management or coordination agent, or of heightening individual agents awareness of other users of the system.

4.1.11 MFK (Design for X)

This research appears to be at an early stage, although it is claimed that a partly realized prototype has been produced (Meerkamm, 1994). The system contains a product model (component model) which facilitates information sharing within the system. Support for individual aspects of the design is provided through design modules (software experts) which provide support for different design for X activities, for example designing for stress, designing for production, design for environment/recycling. However, all the knowledge for these different activities appears to be captured in one structured knowledge base, and this would limit the use of different artificial intelligence paradigms, and most likely prohibit inclusion of existing systems of software expertise. Also, the designer has the full responsibility for deciding which of these tools to use at any stage of the design, thus there is apparently no mechanism for coordinating team activities.

4.1.12 PACT (The Palo Alto Collaborative Testbed)

This research examines a concurrent engineering infrastructure which encompasses multiple sites, subsystems and discipline. It served as a testbed

for emerging data-exchange standards such as PDES/Step (Product Data Exchange Using Step/Standard for the Exchange of Product Model Data). They consider information sharing by means of a design model to represent the evolving design, and this makes use of shared design-domain ontologies (that is, sets of agreed-upon terms and formally described meanings). Literature on this project (Cutkosky et al, 1993) claims that the design model forms a basis for knowledge sharing among diverse systems, but it is not clear that their research makes the same distinction between knowledge and information which has been defined in this thesis. Also, in their work, agents are defined as programs that encapsulate engineering tools.

Interaction and integration of (pact) agents is achieved through facilitators which translate tool specific knowledge into and out of a standard knowledge interchange language (KIF). Thus interactions are between facilitators and agents, or between pairs of facilitators, but not directly between agents. However, there are significant problems associated with attempting to define a standard knowledge interchange language at present, and these are effectively discussed by Ginsberg, (1991).

PACT covers many of the requirements from future CAE systems well, both in terms of integration and support of individual areas of expertise, through inclusion of software expertise, in the guise of systems such as Next-Cut, which supports product and process design of mechanical assemblies. However the PACT system does not appear to include any element to stimulate active participation of human designers in the design process.

4.1.13 PECOS (ESPRIT)

Basically an investigation of models for Computer Supported Cooperative Work (CSCW), and as such it is of interest since CSCW greatly supports cooperative working (of which CE is a form) by making it easier for team members to work together even though they may be many miles apart. It can also be argued that as working together is easy, people are more likely to want to do it - hence it promotes collaborative working. However, this is a passive rather than active promotion of group participation.

4.1.14 SCHEMEBUILDER & IDEAS (Lancaster Engineering Design Centre)

A package of software tools supporting the conceptual and embodiment stages of mechatronic and mechanical systems design. IDEAS stands for Intelligent DEsign-Assistant Systems, which are aimed at providing better support for augmenting and empowering the designer rather than replacing him. Since this research focuses on systems where the computer provides augmentative support to the human designer, rather than the 'expert systems' approach where the computer tries to perform the design with as little human intervention as possible (Oh, 1993), these systems are apparently conceptually near to the beliefs stated earlier in this thesis, i.e. that innovative design is essentially a human process and therefore best supported by integrated, cooperating systems, rather than fully automated systems.

The combined systems developed in this research satisfy many of the requirements identified in earlier chapters of this thesis, including support of human design expertise and integration of activities. They include software design agents of various types, each of which exhibits particular characteristics, as suggested by their names. For example Design Experts can fully automate tedious, routine or repetitive parts of the design, whilst Design

Suggesters offer the human design agent solutions to parts of the design and Design Critics may evaluate parts of the design from particular design perspectives. Integration of activities is achieved through use of shared databases, and Design facilitators which may be used to view the design-in-the-large situation, i.e. from multiple different perspectives, and to help with the data translation between one tool and another - the mapping mechanism for translation from one schema definition to another may be achieved by object 'wrappers' (Bracewell et al, 1994). In the Schemebuilder environment, cross-disciplinary component descriptions in the form of bond graphs are used to satisfy the requirement of a common language.

Once again, these systems potentially provide excellent support for designers working alone, or consciously looking at alternative design perspectives, but there does not appear to be any support to actively promote changes in working mode, when necessary.

4.1.15 STRETCH (ESPRIT)

This research examines the design and implementation of a system to support the representation and manipulation of large knowledge bases. Their database approach supports non-traditional data structures and provides a multi-paradigm programming environment including rule-based language and object oriented language. Thus it provides an environment which could support a diverse range of software expertise, but does not examine specific examples of such expertise.

4.1.16 SHARE

Described by Toye et al, the top level architecture of SHARE is a set of agents interacting as peers over the Internet, where each agent can represent

one or more of the following: a designer, his personal CAD tools, a database or other information service, a computational service that supports engineering, or the engineering process. This project potentially makes a significant contribution to supporting the soft elements of team interactions, trying to help the team reach the 'shared understanding' of the domain, the requirements, the artifact, the design process itself and the commitments it entails, i.e. the view is similar to that expressed by Scott (1994) in figure 3.3. The project promotes computer use in all communication-documentation activities, and encourages as much information as possible to be captured electronically. Such information ranges from email messages, to movie mail, to output from processing programs. This research also makes use of the knowledge sharing interfaces developed in the PACT research discussed above.

4.1.17 TEMPORA (ESPRIT)

This research combines a relational database with rule-based reasoning. The main interest of this work in the present context is that they add a temporal dimension to the relational model, making clear that two types of temporal information must be recorded, i.e. event time, which is the time over which we know (or think) a piece of information holds in the universe of discourse, and transaction time, which is the time over which the information holds in the information system. (McBrien et al, 1992). This research is therefore primarily of value when considering the information and knowledge content which must be exchanged or shared within the system, specifically when considering the changing value of particular knowledge sets at different stages of the product design.

The systems discussed in this section all satisfy some of the identified needs of CE, but none adequately address the issue of promoting concurrency in working. Greater support is needed to raise awareness of when design decisions may affect other team members' activities, and of when a change between synchronous and asynchronous working should be made. Elements of the MOSES system, which is described in the next section, do address this issue. This research contributes to the MOSES research.

4.2 The MOSES Architecture for Future CAE Systems

The MOSES architecture (figure 4.1) is based on the use of 2 information models, a Product Model and a Manufacturing Model, which can be accessed by an open set of application programs, via an integration environment.

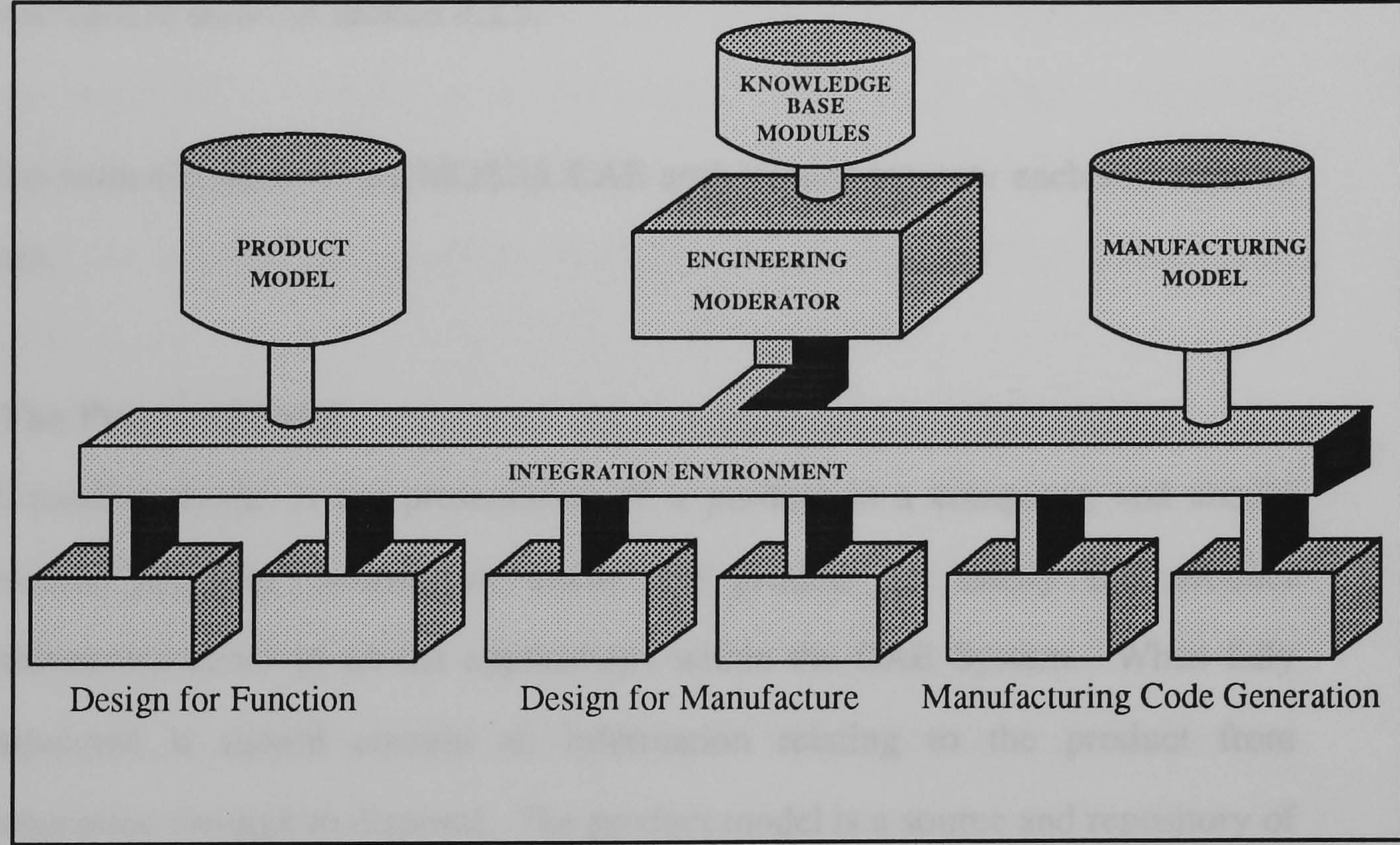


Figure 4.1: MOSES Research Concept

The information models are implemented as object oriented databases. The open set of application programs may contain any application program which may be used by a CE team member during the course of the product design. Such applications may include CAD (computer aided design) and FEA (finite element

analysis) packages, mathematical modellers, expert systems and any other application program, any or all of which may interrogate, add, modify or delete information within the product model database. Interactions between the information models and the applications are enabled through the integration environment. Any modifications to the product model are monitored by a specialist coordinator application called the Engineering Moderator, which is described in detail in section 4.2.5.

The main elements of the MOSES CAE architecture are now each examined in turn.

4.2.1 The Product Model

A product model is a representation of a product in a computer, and should contain adequate information about the product to satisfy the product information needs of all the applications within the CAE System. When fully populated it should contain all information relating to the product from conception through to disposal. The product model is a source and repository of information for many applications, and as such allows information to be shared between the many users and software components of the CAE System. It therefore helps to promote a common understanding of the product design, whilst not restricting the individuality of the human or software agents which are involved in the design process.

The structure of information which may be stored within a product model is called a product data model. Thus a populated product model, for a particular product, is an instance of a product data model.

The importance of product models is increasingly accepted by research groups and industry (Krause, 1993). The product model adopted by the MOSES project is based on considerable research experience gained through this and previous project work at Leeds University. (Shaw et al, 1989). Significant recent work has been in the areas of specification modelling (McKay, 1993), (Erens et al, 1993) and assembly modelling (Henson et al, 1993), (Baxter et al, 1994).

Autonomy is not actively restricted by the use of information models but in order to use the CAE System to its best advantage all agents must actively participate in information sharing, by utilising the common product model database. If agents create their own individual databases of information relating to aspects of the product, problems may be stored up for other agents who have been unaware of decisions taken. Thus any additions or changes to information relating to the product, as made by any application from the open set of application programs described above, should be stored immediately in the product model rather than being developed in private databases or files, since all agents should be allowed the opportunity to access the most recent product information available.

4.2.2 The Manufacturing Model

A manufacturing model is used to describe available manufacturing processes, resources and strategies. Its purpose is to provide a consistent source of manufacturing information for applications. This model has the potential to contain information which is valuable to many different parts of the enterprise as a whole as well as to individual project team members. Thus it may be accessed by many different types of application, with purposes ranging from the formulation of improved business strategies to real time production control. The model developed during the MOSES project has four levels based on a de-facto standard (i.e. Factory, Shop, Cell, Station). By acting as a single source of information on available manufacturing capabilities and status it helps to promote a common understanding of the manufacturing enterprise without placing undue restrictions on the autonomy and heterogeneity of the agents who wish to use this information. (Ellis et al, 1993), (Molina et al, 1994), (Al-Ashaab & Young, 1992), (Al-Ashaab & Young, 1994).

4.2.3 Strategist Applications

Strategist applications are specialist expert applications which assist users of the CAE system to evaluate, modify and extend the product design using criteria which are closely allied to particular design perspectives, i.e. they form part of the open set of applications described at the beginning of this section. Ideally they offer the designer more than Computer-Aided Design (CAD) tools, which are valuable design aides, but which are generally used long after the major design decisions are settled. CAD tools do not generally support the engineer at a much earlier stage in the design process, i.e. at the conceptual design stage,

when engineers make the major and more expensive decisions (Oh 1993). A design agent, as referred to in this work, is most commonly a strategist application, in combination with the human designer.

It is necessary to consider many different perspectives on the product in order to achieve a good product design which satisfies all the requirements demanded from the product throughout its entire life cycle. Each design perspective has its own design criteria, rules and heuristics to which the product design should conform. Examples of design perspectives are: design for manufacture, design for assembly (Boothroyd & Dewhurst, 1987), design for disposal, design for human factors (Tayyari, 1993). Such perspectives will be termed 'Design for X' or 'DFX' hereafter since the list is endless (Meyer, 1993).

Within the MOSES project, effort is focused on two example DFX perspectives, these being design for manufacture and design for function, and various applications have been researched in these areas. A manufacturing strategist, under development, may assist users in design for manufacture activities, using information from the manufacturing model, and product model. Additionally, work has been carried out into how these models can support strategic policy making at the enterprise level (Molina et al, 1994). Consideration has been given to design for manufacture specifically relating to injection moulding by Al-Ashaab (1993) and Lee (1995). The capture of information and knowledge through reverse engineering of components is currently under examination by Borja (1995). An application to support cost and delivery estimation for cranes has also been designed and implemented (McKay et al, 1995(2)).

Work has also been carried out on a design for function expert, specifically looking at shaft design for electric motors and generators (Sadler, 1994). This research resulted in the implementation of a rule-based expert which uses and produces product model information. The resulting expert is examined in detail in section 6.2.1, as this has been used as an example instance of a software expert, and was designed and instantiated using the knowledge representation model which is the focus of this thesis.

As explained earlier, care must be taken that the design criteria knowledge incorporated in such design environments is kept as pure, in the sense that it must provide expert support for a particular design perspective, as possible with respect to the requirements of the particular perspective. This is necessary in order that the best possible support may be provided to the designer for their area of expertise. The requirements for coordination of expertise and raising awareness of other design perspectives are separate issues and must be supported by other elements of the CAE architecture. The author does not believe it is appropriate (or possibly even feasible) to achieve a fully automated system for the resolution of design conflicts which may arise when the multitude of Design for 'X' perspectives are considered simultaneously. This view is also shared by other authors (Bahler et al, 1994). A better solution is to provide designers with CAE support through partially automated systems, which can also raise their awareness of other decisions being taken relating to the design, and thereby empower the human designers to anticipate and resolve design problems as soon as they appear. For this reason information generated by Design for 'X'

applications should be added to the product model as soon as possible, to ensure that the strategists are actively sharing product information.

4.2.4 Integration Environment

A MOSES CAE system consists of many elements, including models and applications. An integration environment is required to enable these elements to work together even though they may be distributed over many computing platforms, probably located at several sites.

The integration environment must satisfy the requirements of each individual element, so that the models may store and maintain the information accurately, and the applications may each perform their particular functions, and access information as required. It must also provide support for interactions and communication between applications. This may require the provision and support of translators or wrappers to enable communications as necessary between agents in the system and to allow information to be exchanged. As previously stated, the CAE system can never be considered to be complete, since there is always the possibility that additional, possibly third party software may be needed to provide particular support for certain users of the system.

Commercial systems, such as Digital's Object Broker, are already available to support integration by assisting in the exchange of information between applications. There is also considerable research effort currently focusing on integration issues, as can be seen by the review of related software systems and environments given in section 4.1.

The need for a common language, or understanding of terms to facilitate the exchange of knowledge and to resolve conflicts of views, has already been mentioned, when considering Halasz's identified requirement for mutual intelligibility. However, when considering the integration environment, problems exist in two main areas. Firstly, elements of the software systems and networks must be able to communicate together, and secondly, the network user (human) must understand the systems particular interfaces. According to Manola (1995) *interoperability* is the ambitious goal which needs to be attained by software systems and networks. He states that two or more systems are interoperable if they can interact to execute tasks jointly, and intelligent interoperability requires interaction among information systems, some of which may be intelligent and capable of functioning as intelligent agents. Such systems may be called knowledge-based integrated information systems (KBIIS) and they involve integrating any heterogeneous information sources, including heterogeneous distributed databases, knowledge-based systems involving heterogeneous knowledge representations and conventional application programs and their associated processors. Manola also states that the full range of KBIIS requirements are only beginning to be addressed by researchers.

It has also been acknowledged that the achievement of agreement of common meaning between experts from different disciplines, can be very difficult to obtain (Cutkosky et al, 1993). Regrettably there would seem to be no easy route to obtaining this common understanding of terms between experts of different disciplines, but the shared language does only need to cover an

intersection of expert interests. Providing agreement on terms or translation of terms can be achieved between experts working with specific product types, or in particular industries, advances can be made, as in principle the language can evolve from a few core concepts (Gruber et al, 1992). Information sharing, through common, integrated information models can significantly reduce translation requirements.

4.2.5 Engineering Moderator

The necessity to stimulate active participation of all team members, and the need to raise the awareness of each individual team member to the concerns and aims of other experts from different disciplines have already been mentioned many times in this work. These activities have also been identified as being part of the duties of a management of coordinating agent. Also, the review of related software systems in section 4.1 indicated that existing and proposed software solutions do not currently provide support for these activities.

Within the MOSES architecture, the Engineering Moderator (EM) is a specialist manager or coordinating program whose role is to drive concurrency within the MOSES system. The previously described elements can provide excellent support for individual team members or groups working from particular design perspectives. The task undertaken by the EM is however rather different: it has been included specifically to promote communication and negotiation by the active exchange of information and knowledge between team members with different areas of expertise. However, it is NOT an engineering arbitrator, as it is not included to automatically generate compromise solutions to design problems. It is included to raise the awareness of human designers within the CE

team of how their decisions may affect, or be affected by actions of other team members. In this way, it stimulates communications between CE team members and thereby supports and empowers the human designer. As previously shown, this is a vital element of any CAE system in a CE environment, indeed it has been claimed that communication is a critical ingredient for project success (Bobrow, 1991).

The rationale behind the inclusion of the EM has been indicated in this thesis by examination of the need for a management/coordination agent, further treatment of this subject may be found in Harding & Popplewell (1994)(1). The EM's role is as a driver of concurrency within the CAE system which it does by using knowledge of the expertise which exists in the form of agents which may interact with the CAE system. Clearly this knowledge must vary over time, since the expertise which is available to, or relevant to, a product design will vary as the design evolves. Also the structuring of the knowledge within the EM is critical as the larger the team is, and the more diverse the expertise exhibited by team members, the more complex the knowledge which the EM needs to store and use becomes. However it must be stressed that the role of the EM is **not** to solve the design problems itself, its mission is rather to raise the awareness of individual team members when a particular problem may exist which should be resolved.

Details of the design and implemented structure of the EM are given in section 6.2.2, since it has been used as an example of a sophisticated form of software expertise which can be modelled using the KRM. The complexity of the EM has provided an excellent test of the flexibility of the KRM.

5. The Knowledge Representation Model Concept

5.1. Modelling Diverse Types of Software Expertise

The expertise within the concurrent engineering project environment comes from many disciplines, and it has been shown in the previous chapters that the skills required to meld the individual specialists into an effective proactive CE design team are very diverse. To really provide support for such CE team working, a CAE system must provide support for all elements of these skill sets, and therefore address all the requirements identified in chapter 3. It is believed that this may be achieved by basing future CAE systems on the MOSES architecture proposed in section 4.2. However, this only partially addresses the problem, since ways must still be found of implementing instances of CAE systems based on this architecture, and this includes the implementation of many diverse types of software expertise, such as the strategist applications described in section 4.2.3. The knowledge required by an expert should be captured in whatever way best suits the expert's requirements, since the best approach depends on specifics of the problem (Knaus & Jay, 1990). Also, there are deep differences among the approaches taken to knowledge representation, which lead to the belief that attempts to automatically translate between one knowledge representation scheme and another are premature, and that attempts in this direction will inevitably constrain future knowledge representation efforts (Ginsberg, 1991). Therefore care has been taken to identify techniques for modelling knowledge which do not restrict the ways in which knowledge is captured, and represented. An approach to doing this, based on a knowledge representation model, KRM, will now be proposed.

Software experts may be developed to support many different aspects of human expertise, and these range from applications to support highly focused, specialist, computational work, to applications which specialise in the management or coordination of team activities. During the course of this research several different types of software expertise have been examined, designed and developed, these include: a Shaft Design for Function Expert (Harding & Popplewell, 1995), a Manufacturing Strategist (Ellis et al, 1994), a Cost and Delivery Expert (McKay et al, 1995), and an Engineering Moderator (Harding & Popplewell, 1994(1) & 1994(2)). Examination of these particular, individual, and diverse types of expertise has led to an understanding of the fundamental similarities of their concepts, i.e. the generic aspects of how the requirements from such expert applications may be satisfied and modelled have emerged.

A comprehension of the similarities between the studied forms of software expertise has been fundamental to the modelling of knowledge to be applied and stored within the CAE system. Identification of a method of modelling the expert knowledge is considered to be of value in the creation of software expertise, since experience has shown that it is much easier to write well-structured and re-usable code to solve well defined problems than to solve experimental, research problems. When dealing with complex systems of a type not worked on before, a natural and productive approach to solving the problem is to make a quick model of the system, analyse it, and then refine the solution based on the ever-increasing understanding to the problem which is being gained. This approach has been called the round-trip gestalt design method (Rational Rose, 1993). Initially, adoption of this approach enabled a model for the capture of software expertise to evolve through progressive experimentation with views

of a diverse range of knowledge within the concurrent engineering project environment.

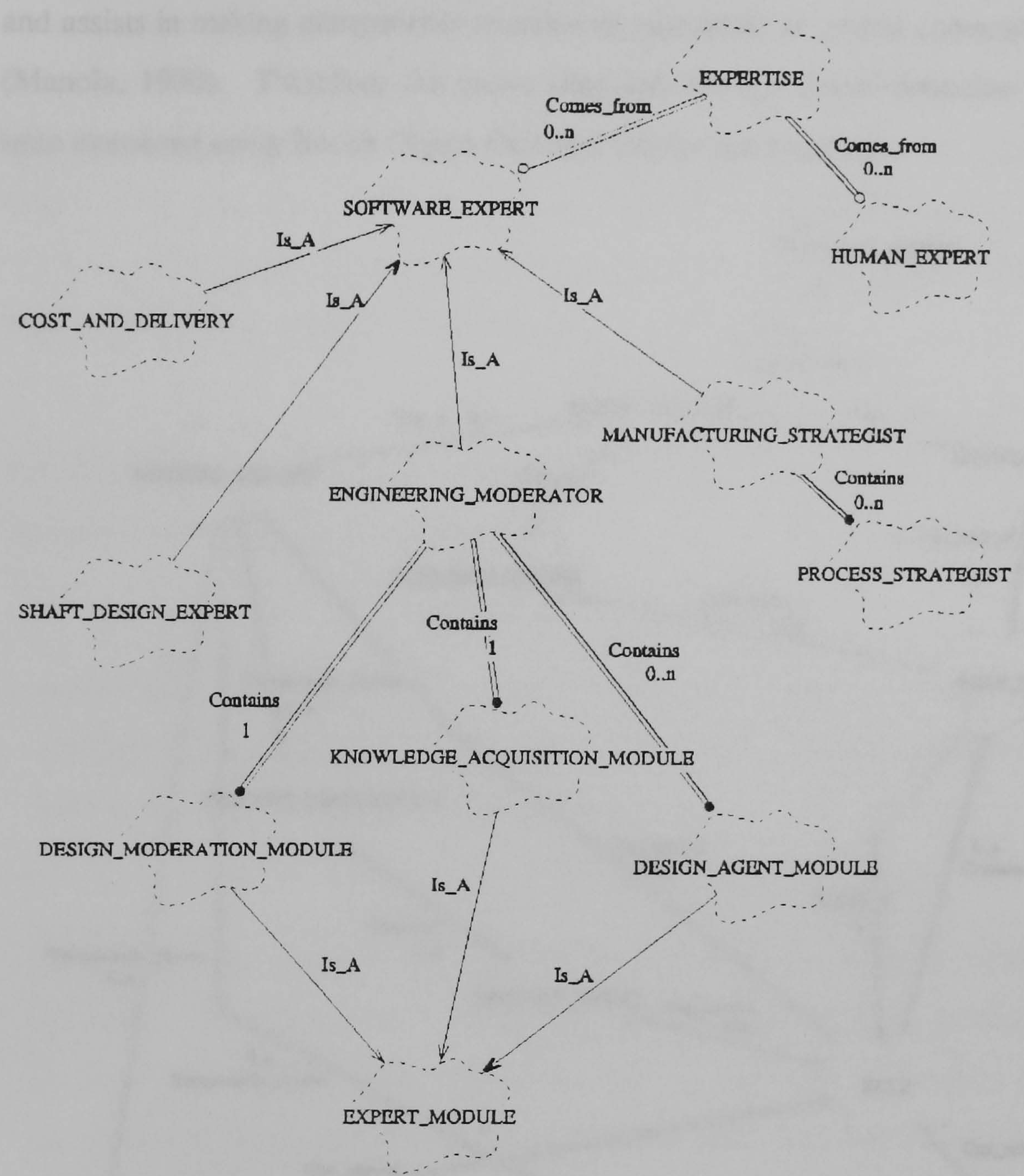


Figure 5.1: A Representation of Investigated types of software expertise using Booch Object Oriented Design Graphical Notation.

An object oriented approach has been taken for the modelling since 'defining a system in terms of objects facilitates the construction of software components that closely parallel the application domain, thus assisting in system design and

understandability' and 'using classes and inheritance provides a simple and expressive model for the relationship of various parts of the system's definition and assists in making components reusable or extensible in system construction' (Manola, 1990). Therefore the views obtained through experimentation have been examined using Booch Object Oriented Design methodology.

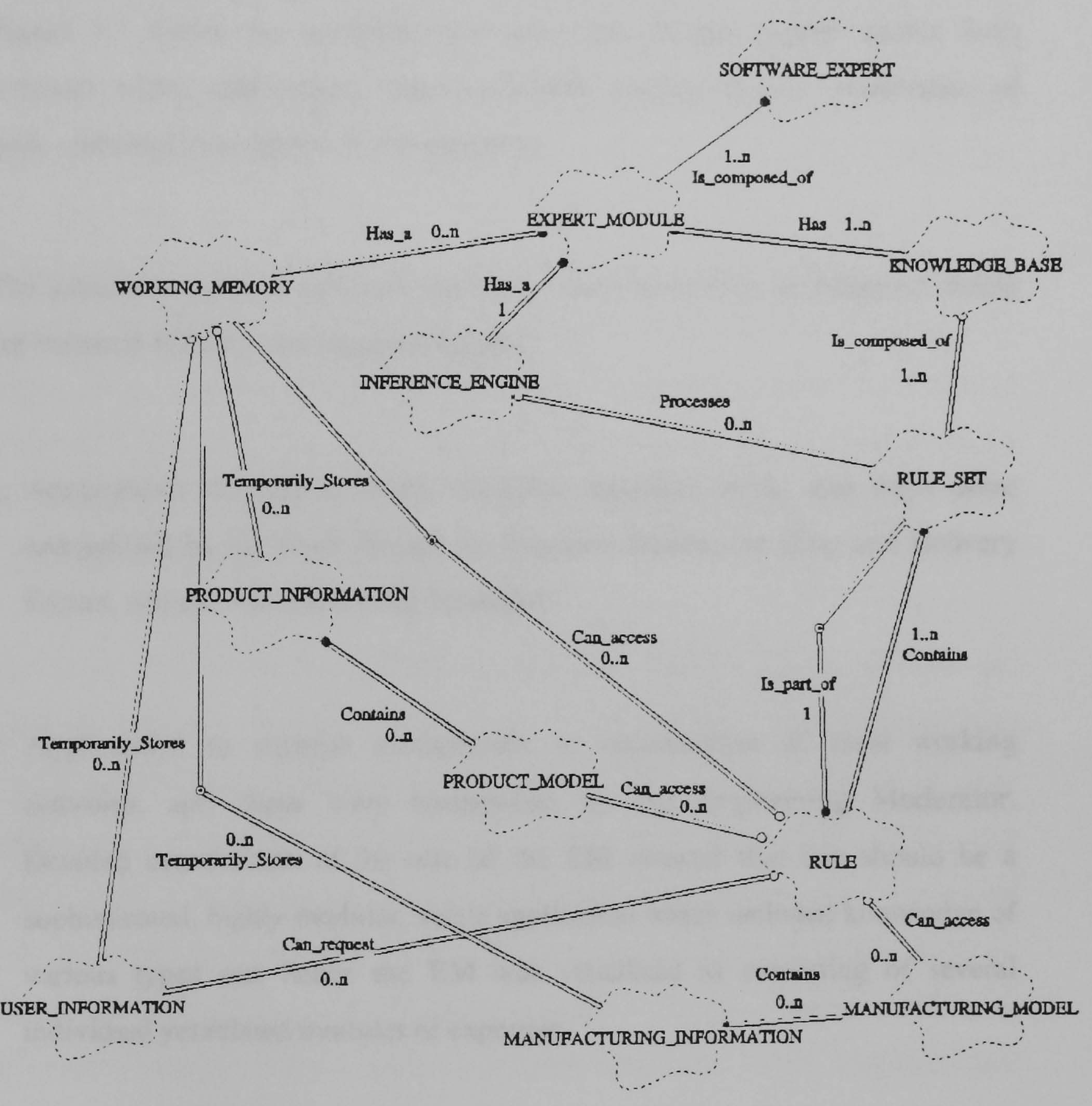


Figure 5.2: A Representation of Software Expertise Using Booch Object Oriented Design Graphical Notation

Using the Booch approach, a representation of software expertise was developed and this view is expressed in figures 5.1 and 5.2, which show the relationship of classes, using Booch notation, so that the clouds are object classes and the lines are relationships between classes, where single lines with arrows illustrate inheritance, and the double lines illustrate other associations between classes. Figure 5.1 shows that expertise can come from human experts and/or from software expert applications (most commonly coming from a combination of both - referred to as agents in this research).

The particular types of software expertise which have been investigated during the research fell into two broad categories,

1. Applications to support highly focused, specialist work, and these were exemplified by the Shaft Design for Function Expert, the Cost and Delivery Expert, and the Manufacturing Strategist.

2. Applications to support management or coordination of team working activities, and these were exemplified by the Engineering Moderator. Detailed examination of the role of the EM showed that this should be a sophisticated, highly modular, single application which included knowledge of various types and hence the EM was visualised as consisting of several individual yet related modules of expertise.

Details of how these particular forms of software expertise were eventually instantiated using the KRM are given in chapter 6. The remainder of this chapter

is devoted to defining the KRM concept itself which emerged through experimentation with such design agents.

5.2. The Production System Metaphor

The KRM concept enables software expertise to be represented by one or more expert modules. This concept is captured in Figure 5.2, which represents the highest level representation of the KRM, modelled using Booch's graphical notation. In this figure it can be seen that each object of the expert_module class is associated with objects of three other classes, i.e. one or more knowledge_base objects, one or more working_memory objects, and an inference engine object. In performing an object class breakdown for the class expert_module, the terminology of standard expert systems, in particular of production systems, (Jackson, 1990) has been adopted. This has been done for good reason, but with some caution, since misunderstanding of the metaphor may result in the model being undervalued.

The names of the three related classes, *working memory*, *knowledge base* and *inference engine*, have been used as it is believed they are sufficiently well known terms for some level of understanding of the primary functions of these objects to be immediately gained by any casual reader who has had some prior contact with expert systems. For example part of the behaviour of a working memory object enables it to hold the data and intermediate results that make up the current state of the problem, and this agrees well with a standard definition of working memory within a production system (Jackson, 1990). Similarly, part of the behaviour of objects belonging to the knowledge base class enables knowledge to be added, changed or removed from the knowledge base without any changes being made to objects of the other two classes. However, it must be remembered that in this work the names relate to classes of objects, so

instances of these classes have state, behaviour and identity (Booch, 1991), and therefore they provide all the flexibility and power of object oriented systems. By making use of inheritance structures and polymorphism, it has been possible for the similarity of behaviour of certain classes of objects to be exploited even though the implementations of particular classes of objects is significantly different. This point is central to the concepts of this work.

Each expert module is associated with one or more knowledge base objects, an inference engine object and one or more working memory objects. Details of actual instances of these objects, as created to support instances of individual software experts which have been designed and implemented as examples of the KRM, are given in chapter 6. However the current working definitions of these classes of objects are as follow:-

Knowledge base object - contains knowledge of a particular type, or related to a specific type of expertise or domain. In embodiments of the KRM, instances of knowledge base objects are normally object oriented databases within the same federated database. Knowledge within a knowledge base object database is captured in a miscellany of associated objects, and through their interactive behaviour, as described below. The class hierarchies for these objects are also based on production rules, as can be seen by the rule_set and rule classes in figure 5.2. However, knowledge expressed in other artificial intelligence paradigms (e.g. neural networks) may be embedded within the objects in such an object oriented database. Thus the production system metaphor does not constrain the ways in which expertise is represented.

Inference engine object - carries out the processing of knowledge from one or more knowledge base objects. The inference engine object provides the

software expertise with the ability to apply and use its knowledge. In the initial implementations of the KRM this type of object has been represented by program code or function code rather than as true objects. However, this need not be the case, as the functionality required from their processing could be captured within the behaviour of object classes.

Working memory object - this is a store of variable information, which is possibly only of temporary value, to be used in association with the expert's domain knowledge, possibly to facilitate the processing of that knowledge. In particular, as demonstrated in figure 5.2, such information could originate from product or manufacturing models, be input by the user, or be generated through the expert's activity. The working_memory class has been implemented as a parent class for a hierarchy of domain specific working memory objects.

The production system metaphor is continued to allow storage of knowledge associated with any particular knowledge base object, through the definition of ruleset and rule objects. Figure 5.3 shows the breakdown of these classes leading eventually to two key classes, expression (each instance of which is associated with a simple condition object) and simple_action. These are parent classes for a wide variety of objects, see figures 5.4 and 5.5, which are similar to the extent that they all demonstrate a specific type or aspect of behaviour.

Any expression object must be able to pass messages to other objects (particularly objects of class simple_condition) to state whether the expression is currently in a true or false state. So, for example, an object of the class user_input_response_expression is in a true state if a user has answered yes to a particular question, and is in a false state if the user's answer was no. Alternatively an object of the class object_exists_in_pm_expression is in a true

state if an object of a specified type can be found in the product model, and is in a false state is no such object exists in the product model.

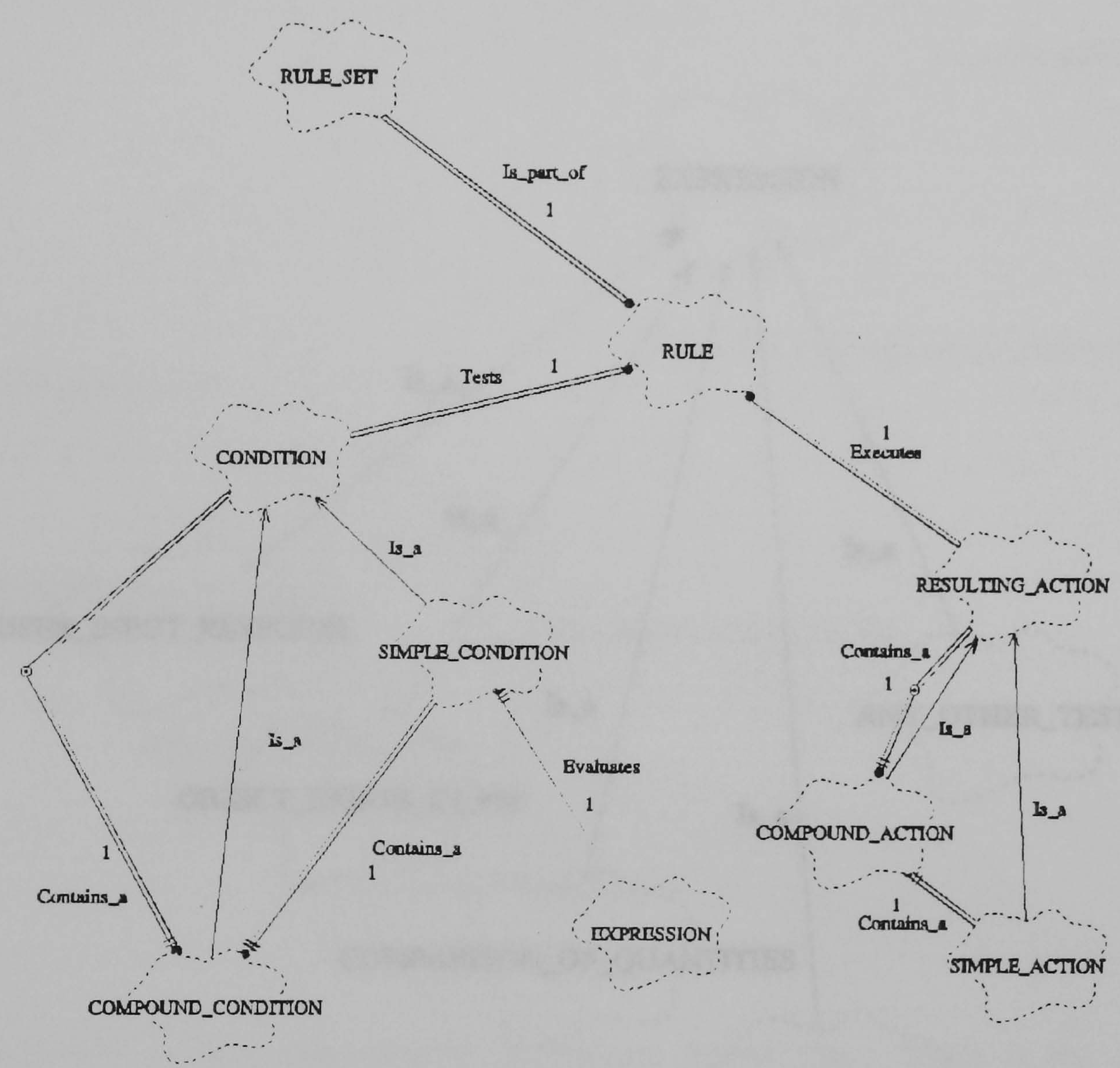


Figure 5.3: A Representation of Knowledge, captured within Rule and Ruleset Classes, using Booch Object Oriented Design Graphical Notation

Similarly, any simple_action object must be able to execute, in order to carry out some task or tasks. For example, an object of the send_message_to_user class can display a predefined message in a window on the user's computer screen. Alternatively, an object of the class fire_a_rule_set, can initiate the processing of

a further set of knowledge. The diversity of activities which can be incorporated into valid instances of these classes demonstrates the strength, flexibility and potential of the KRM.

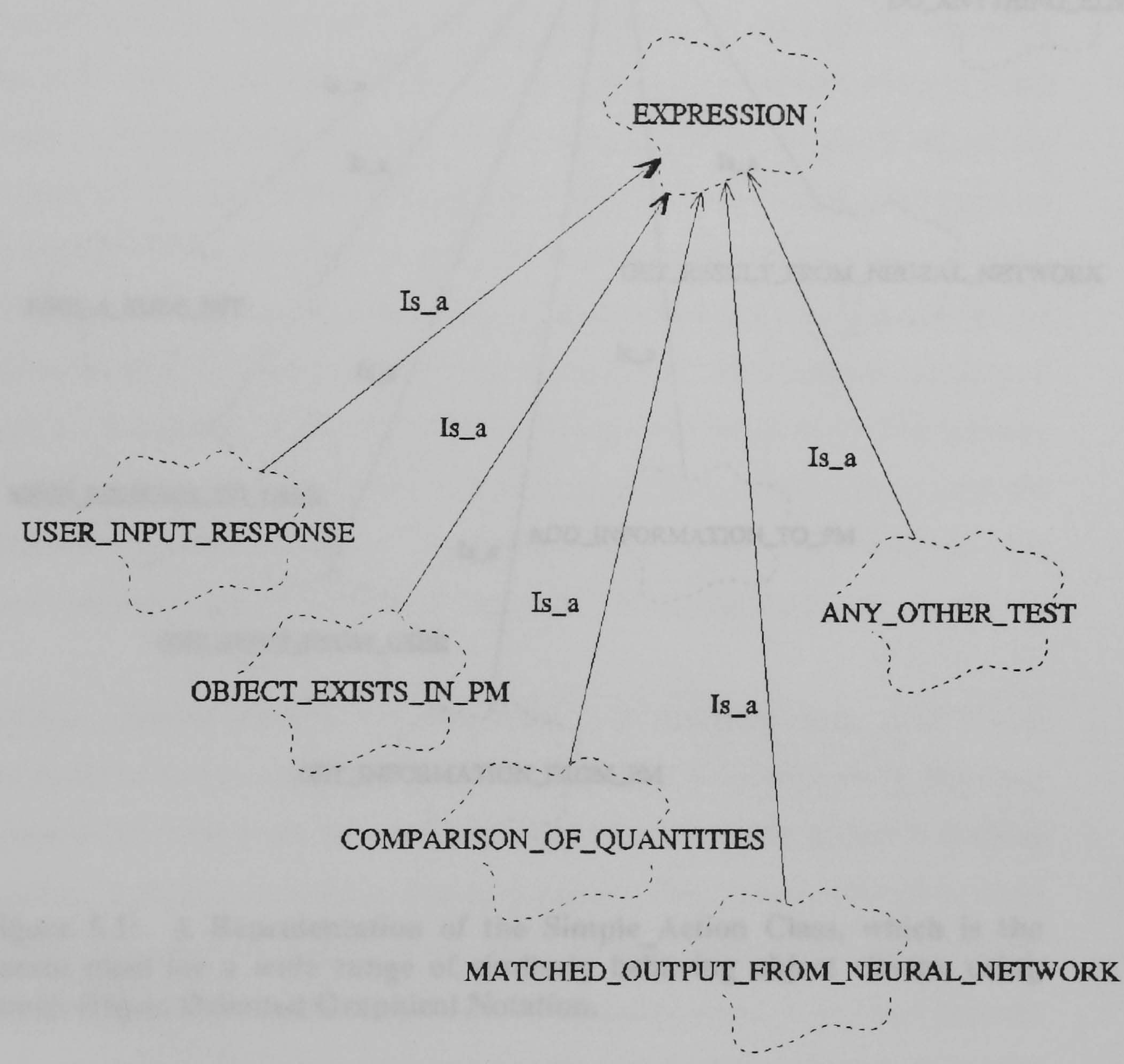


Figure 5.4: A Representation of the Expression Class, which is a parent class for a wide range of similarly behaving object classes, using Booch Object Oriented Notation

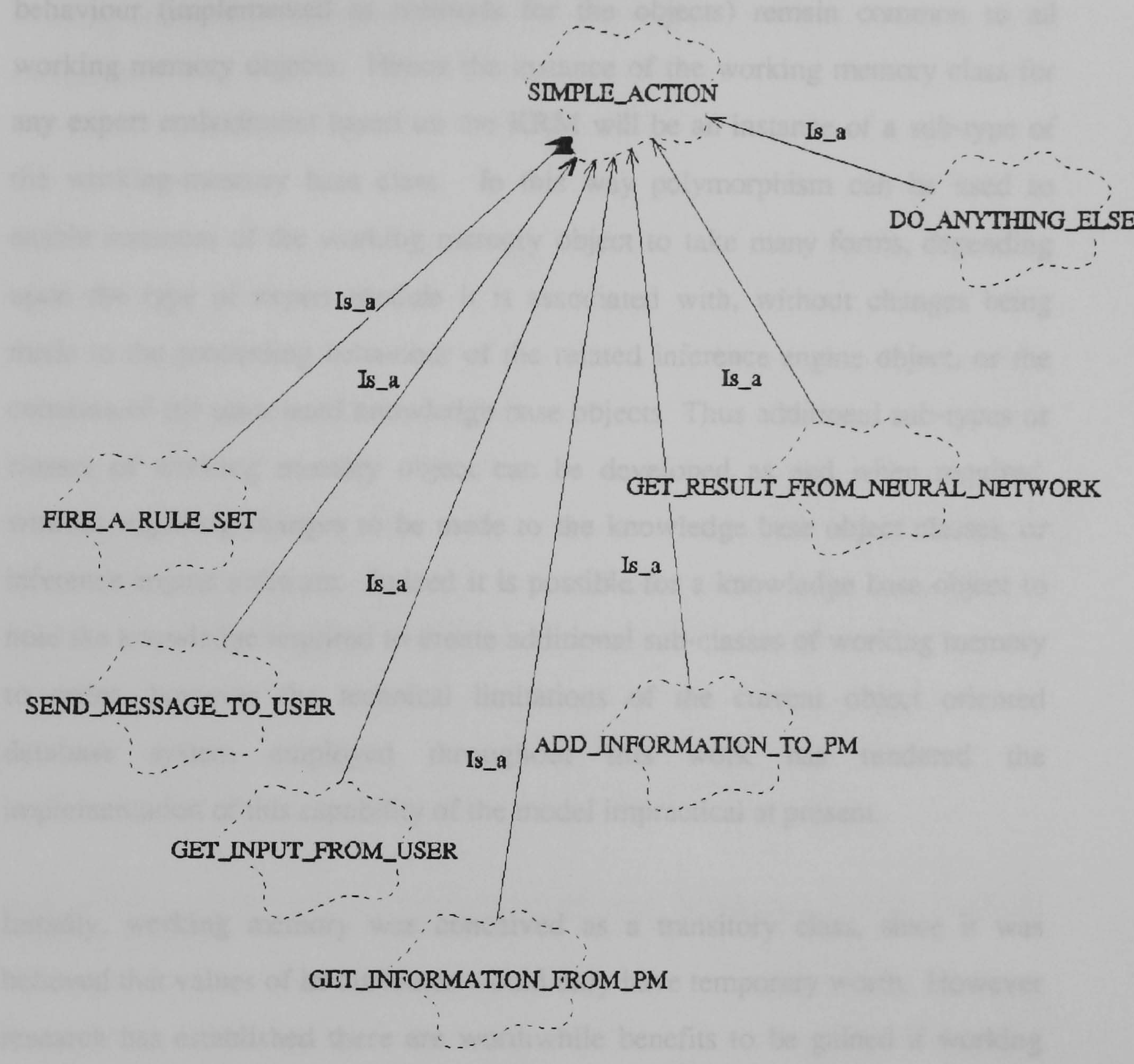


Figure 5.5: A Representation of the Simple_Action Class, which is the parent class for a wide range of similarly behaving object classes, using Booch Object Oriented Graphical Notation.

Working memory objects all exhibit behaviour which enables them to be accessed or updated by the inference engine objects. The variable contents (or attributes) of a particular working memory object class will depend upon the type of expertise which is being modelled, but the fundamental aspects of

behaviour (implemented as methods for the objects) remain common to all working memory objects. Hence the instance of the working memory class for any expert embodiment based on the KRM will be an instance of a sub-type of the working-memory base class. In this way polymorphism can be used to enable instances of the working memory object to take many forms, depending upon the type of expert module it is associated with, without changes being made to the processing behaviour of the related inference engine object, or the contents of the associated knowledge base objects Thus additional sub-types or classes of working memory object can be developed as and when required, without requiring changes to be made to the knowledge base object classes, or inference engine software. Indeed it is possible for a knowledge base object to hold the knowledge required to create additional sub-classes of working memory to order, however the technical limitations of the current object oriented database system employed throughout this work has rendered the implementation of this capability of the model impractical at present.

Initially, working memory was conceived as a transitory class, since it was believed that values of its attributes would only have temporary worth. However research has established there are worthwhile benefits to be gained if working memory is modelled as a set of persistent classes. The concretization of working memory as persistent objects in the object oriented database has enabled both temporary information and longer-term information, which is of value between different runs of the expert application to be stored. This latter type of expert information is especially valuable in implementations of sophisticated forms of expertise, such as the Engineering Moderator, the concept of which was described in section 4.2.5, and the implementation of which will be described in section 6.2.2. Figure 5.6 shows that the information which can be stored in

working memory, for example the working memory of the EM, falls into three categories.

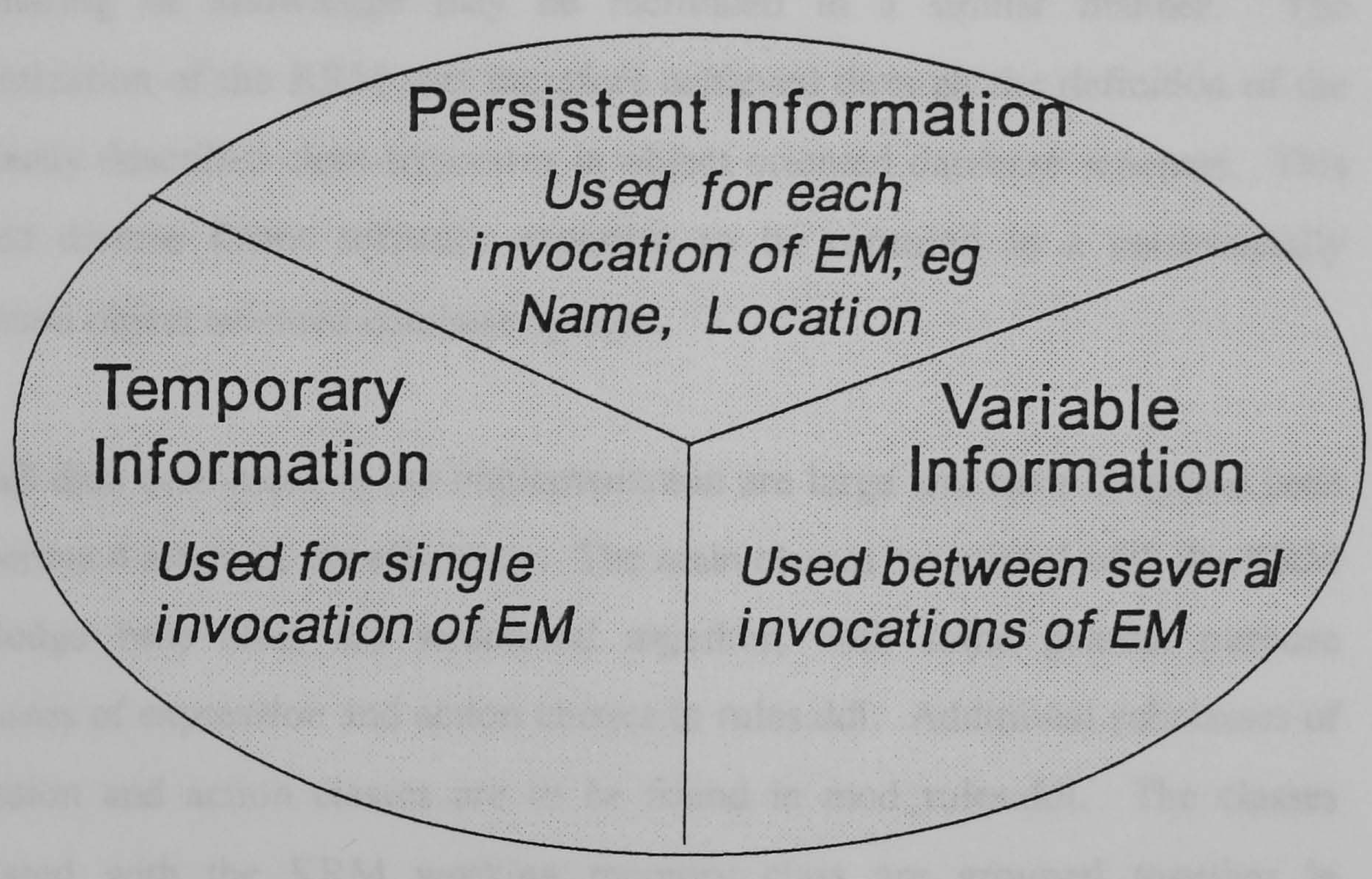


Figure 5.6: Three types of information can be stored within working memory objects

5.3. Instantiation of the Knowledge Representation Model

Having established the conceptual basis upon which a variety of types of software expertise might be designed, it became necessary to instantiate the KRM to enable particular instances of software expertise to be implemented.

The requirement, and indeed necessity, for information and knowledge to be freely available and shared between all team members was clearly established in chapters 2 and 3. Hence it is believed that any instance of the KRM should facilitate the sharing of captured knowledge between team members. As

explained in section 4.2, the MOSES concept of future CAE systems envisages that information may be shared through the use of product and manufacturing models, which are instantiated as object oriented databases. It is believed that the sharing of knowledge may be facilitated in a similar manner. The concretization of the KRM was therefore achieved through the definition of the previously described class structures in object oriented database schemas. This enabled diverse forms software expertise to be captured in a commercially supported object oriented database system.

The full data definitions in the implementation are large and have therefore been split across 4 schemas for efficiency. The main classes associated with the KRM knowledge base class are structured together, with some general purpose subclasses of expression and action classes in rules.ddl. Additional subclasses of expression and action classes are to be found in mod_rules.ddl. The classes associated with the KRM working memory class are grouped together in working_memory.ddl, and mod_working_memory.ddl. Details of all these data definition schemas may be found in appendix I of this thesis.

The varied knowledge which is utilised by the software experts which have been implemented as instances of the KRM, has been captured using many different sub-classes of expression and action classes. Details of some 25 examples of expressions, and some 30 examples of actions, which have been implemented during this research may be found in the tables included in appendix II.

Achieving a working version of the KRM has required substantial amounts of software development, both for the data definitions mentioned above and for associated implementations of class methods and application code. This is partially due to the fact that there is no SQL for the object oriented database

system used, hence all code had to be purpose written to enable objects to be created, queried, modified and deleted. This was achieved by implementing several methods for each of the classes whose definitions are included in appendix I. Also, in addition to the implementation of the software experts described in chapter 6, support applications to populate and modify the database with both product model information and knowledge for software experts have also been written. Without these methods and support applications the example software experts detailed in the next chapter could not have been implemented.

In the implementation of the KRM, the essential associations and relationships between classes were captured using inheritance and class attributes. The essential behaviour of objects of particular classes was captured through the class methods. In this way the knowledge for any software expert could be captured by creating instances of these persistent object classes in object oriented databases, within the same federated database. Additional subclasses of working_memory, expression and simple_action classes may be implemented at any time without affecting the structure or overall operation of the implemented KRM. The KRM has been tested through the instantiation of several software experts in this way, as will now be shown.

6. Design and Implementation of Instances of the Knowledge Representation Model

Expertise may be founded on a wide diversity of knowledge, originating from many different disciplines, it was therefore considered essential that the results of the research project did not restrict the user in the types of support systems, applications or artificial intelligence techniques which may be utilised. These views naturally influenced the conceptual design of the KRM. For example, production rule techniques, neural networks, genetic algorithms, any other knowledge support, or any numerical analysis software should be available to experts using a CAE system utilising the KRM, if they so wish. Since it is believed that an information technology system should support the human user, not coerce or restrict his desired or optimum method of working. The flexibility of the KRM permits the representation and utilisation of knowledge which exists in a variety of forms. The use and reuse of such knowledge by many different applications, and hence the sharing of knowledge and information between members of the CE team, is further simplified by the storage of knowledge as persistent objects within an object oriented database, which may be freely accessed by any number of varied applications.

6.1 Implementation of the Knowledge Representation Model.

Several instantiations of the KRM have been implemented using purpose written C++ code and DecObject DB, which is an object oriented database system, (a version of Objectivity). The essential information content and structure of classes of objects, and the relationships or associations between them, (excluding inheritance, which is discussed separately here as it is dealt with explicitly during data definition), have been captured through the attributes for object classes. The Knowledge base objects have been implemented as database objects, which

contain ruleset, rule, condition, expression and action objects. Both condition and action objects can be either simple or compound, as described below.

A hierarchy of working memory objects has been implemented, where the working memory for any particular expert module is captured either as a single subclass of the working memory base class, or more commonly as a set of associated subclasses of the working memory class. Figure 6.1 shows a screen dump of a window generated by the ooToolManager application (one of the software tools provided with DecObject DB), which allows the database to be browsed, to show either the types (class structures) or the instance data within the database. In this image, the type (class) working_memory is being examined. The derived types in the box on the right hand side are sub-classes of the working memory class. Some of these sub-classes will be described in further detail later in this chapter, as they relate to particular software experts which have been instantiated as instances of the KRM. Future implementations of the KRM should enable a software expert to be designed and implemented with the knowledge to be able to extend this hierarchy at will, by creating additional subclasses of working memory. This would facilitate creation of additional software expertise within the CAE system whenever required.

As previously mentioned, in the current implementations, the inference engine objects can be identified as particular functions in the C++ code. However, in future versions of the KRM, inference engine classes should be implemented, their behaviour being modelled on the functionality of existing experimental code. The limitations and complexities of the current third party software systems employed have restricted the software implementations to some extent, since any additions or modifications to the database schema require significant

amounts of extra source code to be written, leading to considerable re-compilation and linking of code.

The use of an object oriented database system to capture instances of the KRM has proved to be very successful and powerful, although implementations have not been without problems. The technology adopted was immature, and consequently considerable artifice had to be employed during software design, and large quantities of subsidiary code written, in order to satisfactorily test the KRM concept, and produce the required results. As the technology matures and the problems experienced with the database software during this research are cured, the full potential of the KRM should be attainable based on object oriented database systems.

However, considerable advances have been made whilst implementing demonstrable examples of the KRM, these include, for example, successfully capturing the behaviour of the various classes of objects which are fundamental to achieving a valid implementation of the KRM concept. This has been achieved by using the object oriented database system to store persistent instances of the objects thus enabling their behaviour to be held within class methods. For example, a basic aspect of the behaviour of a condition object is that it must know how to test whether their current state is true or false, and how to pass this information on to other associated, interested objects (e.g. to a rule object).

There are many different types of conditions, which can be implemented as either simple or compound conditions, each of which is associated with any of a variety of instances of sub-classes of the expression class (25 implemented versions of

expression are detailed in table 1, appendix II). But however different any two conditions may appear they must exhibit this same aspect of behaviour.

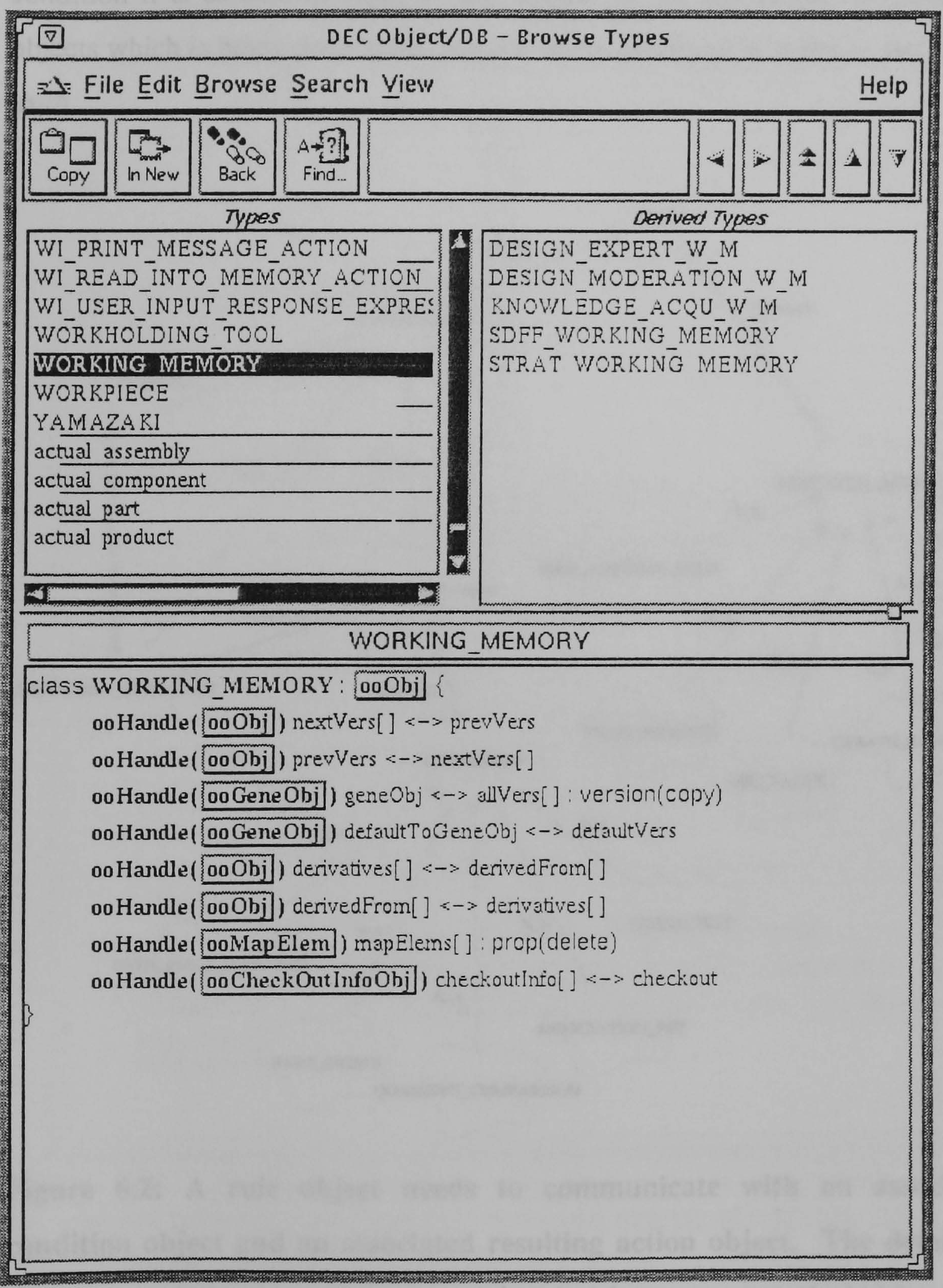


Figure 6.1: Database Browser showing 5 subclasses (derived types) of parent class working memory

When this is achieved a rule object only needs to know it is communicating with a condition object - it does not need to know anything about what type of condition it is associated with, or any details of the processing of expression objects which is being done at the request of the condition in order to test its self truth.

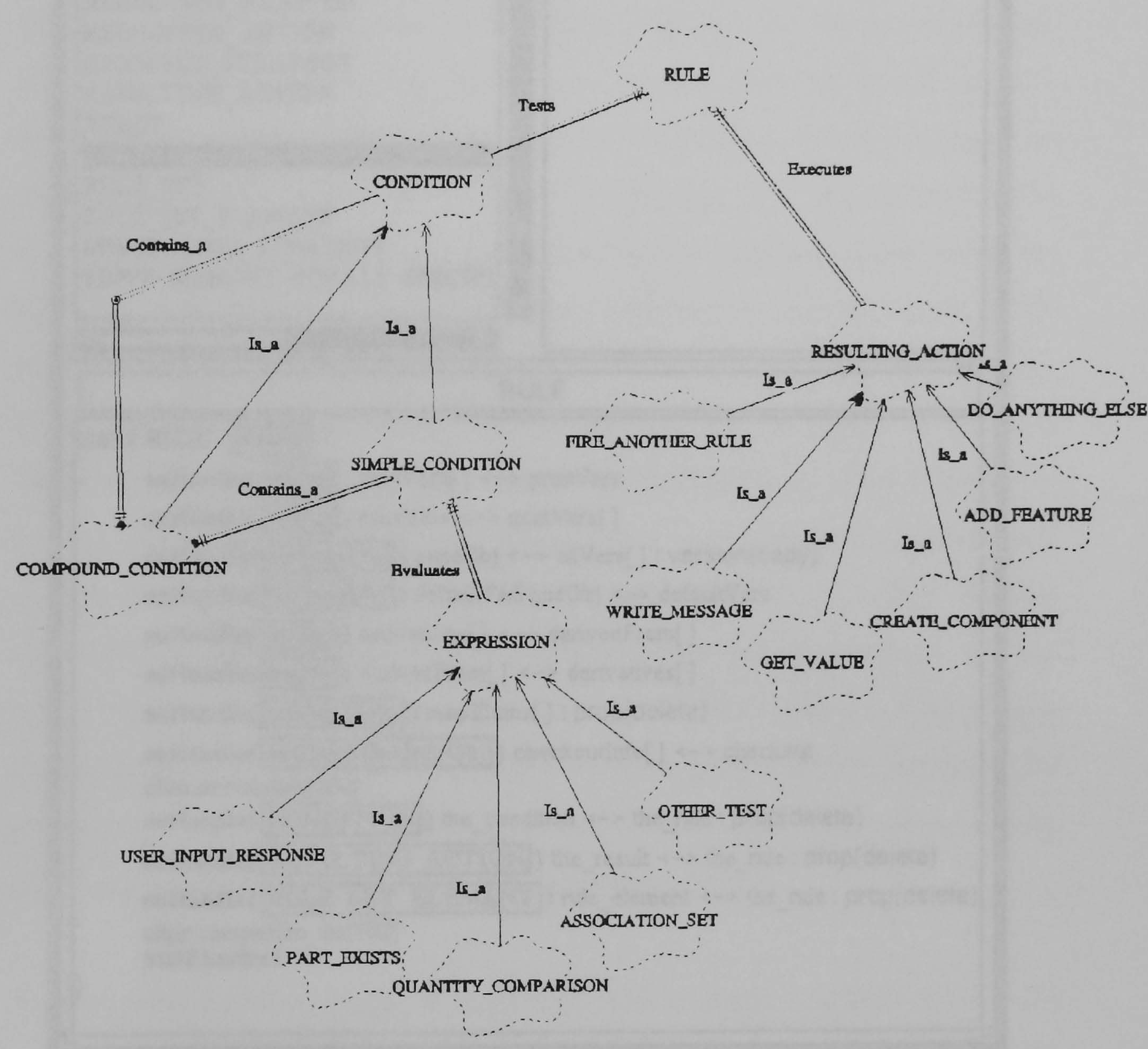


Figure 6.2: A rule object needs to communicate with an associated condition object and an associated resulting action object. The details of the types and structure of these associated objects may remain hidden from the rule object.

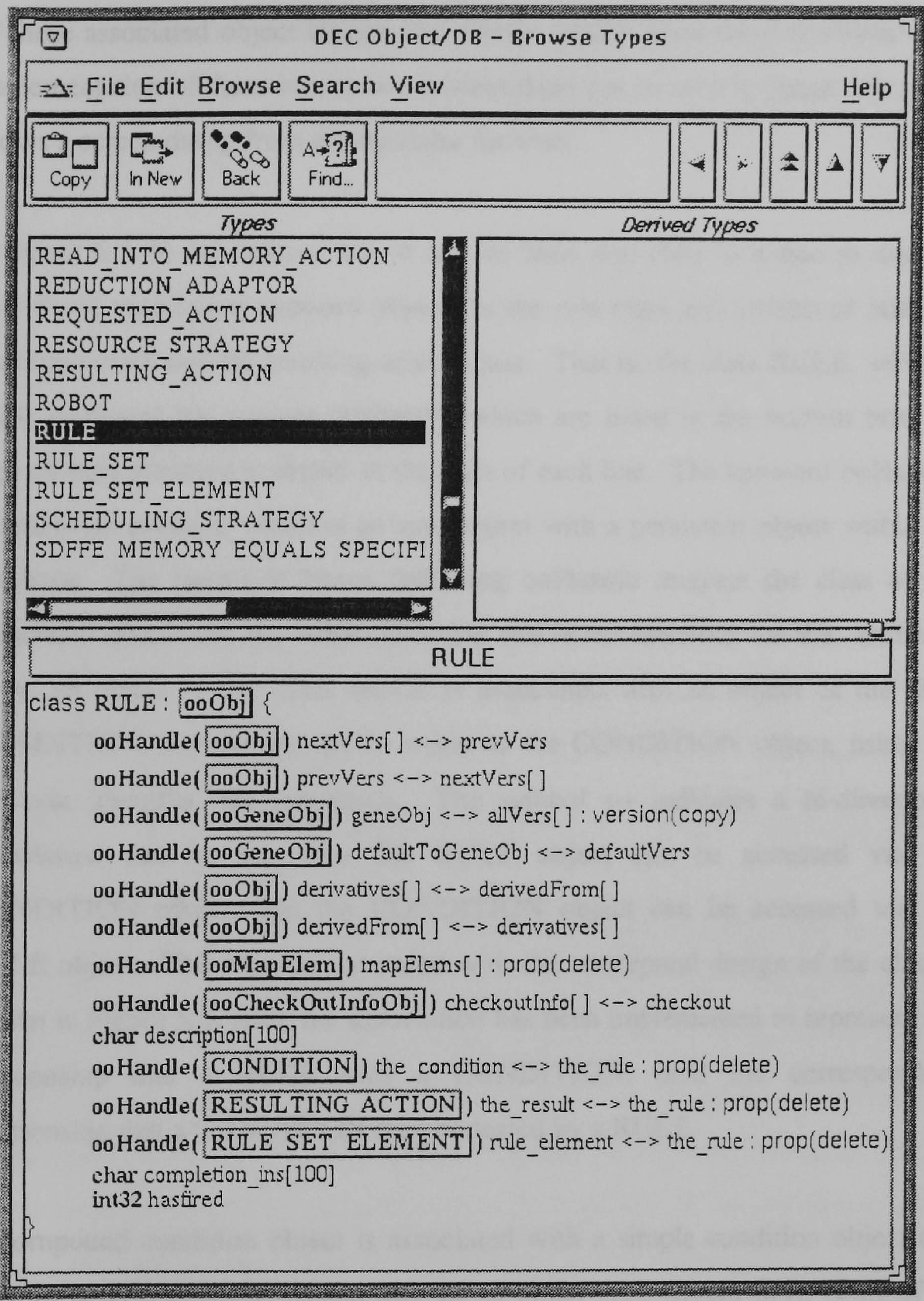


Figure 6.3: Database Browser, showing one to one, bidirectional associations between objects of rule class and objects of both condition and resulting action classes.

Figure 6.2 shows that a rule object simply needs to communicate with a condition object and a resulting action object, details of the types and structure

of these associated object classes can remain hidden from the rule object. The implementation of these one to one relationships can be seen in figure 6.3, which shows a screen dump from the database browser.

In figure 6.3, in the bottom box it can be seen that there is a one to one, bi-directional association between objects of the rule class and objects of both the condition class and the resulting action class. That is, the class RULE, which is being displayed has various attributes, which are listed in the bottom box, the type of each attribute is shown at the start of each line. The keyword ooHandle, indicates an attribute which is an association with a persistent object within the database. The names in boxes following ooHandle indicate the class of the associated object and the name following this is the identifier for the attribute. Thus, an object of the class RULE is associated with an object of the class CONDITION, and access can be made to the CONDITION object, using the attribute identifier, the_condition. The symbol $\leftrightarrow$ indicates a bi-directional association can be set, thus the RULE object can be accessed via the CONDITION object, and the CONDITION object can be accessed via the RULE object. This may be compared with the conceptual design of the classes shown in Figure 5.3, since the association has been implemented to represent the relationship that a RULE tests a CONDITION, (and the corresponding relationship that a CONDITION may be tested by a RULE.

A compound condition object is associated with a simple condition object and another condition object, which may be of either type (figure 6.4). Part of its behaviour therefore includes the capability of determining its self truth value, based on combining the truth values of these associated conditions, using either AND or OR. Once again, this Booch representation of the conceptual class structure design may be compared with the implementation of the associations,

and inheritance within these class structures as seen in the screen dumps of the database browser shown in figures 6.5a and 6.5b.

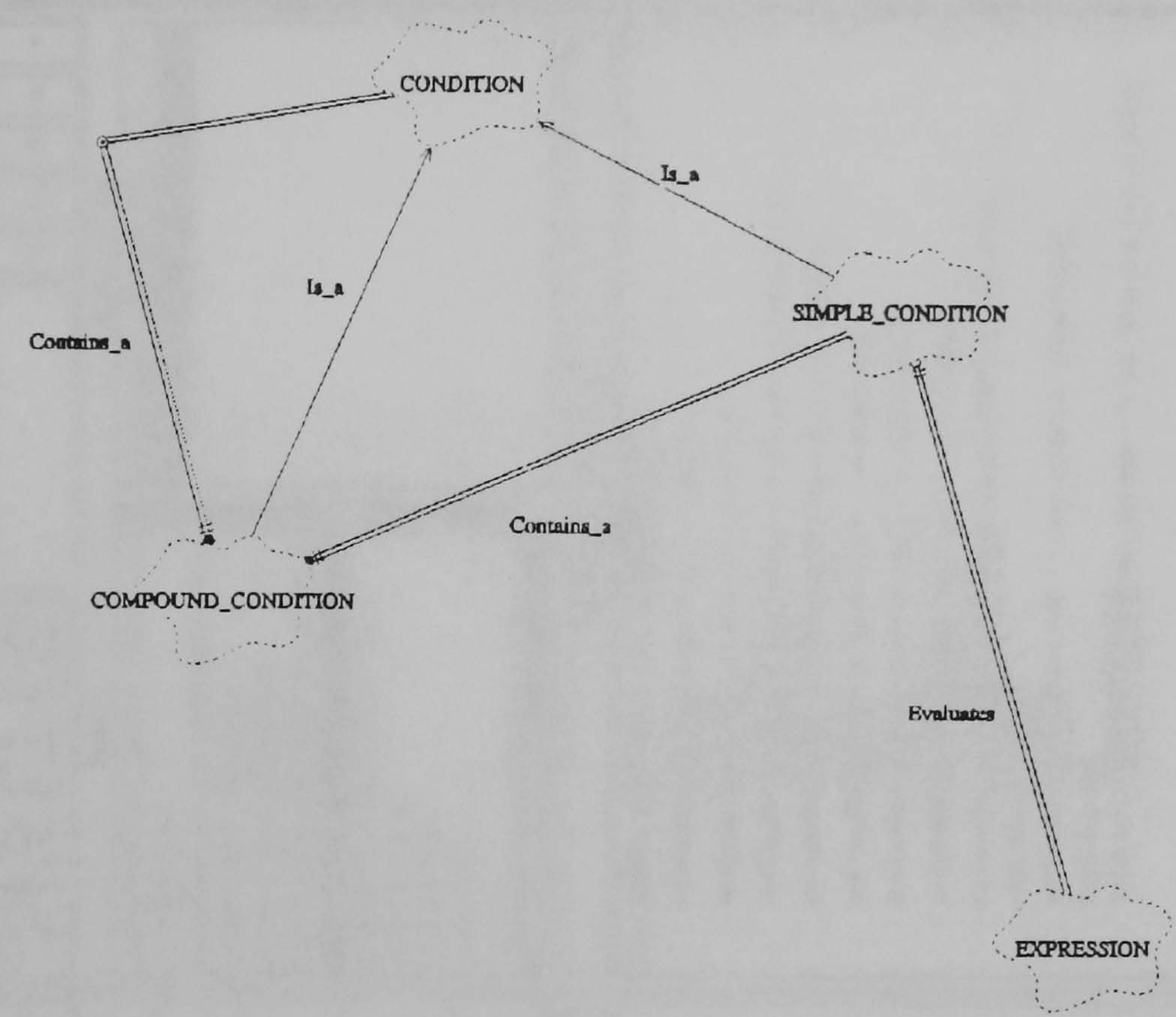


Figure 6.4: A compound condition consists of a simple condition and a condition

The behaviour of an object of the simple resulting action class includes the ability to execute, i.e. to carry out a set of instructions or duties. There are many different types of actions, which range between something as simple as printing out a message to the user, to performing complex sets of calculations related to the design of a specific product. Details of the 30 implemented subclasses of simple resulting action can be found in table 2, Appendix II. A compound action is similarly executed by sequentially performing the actions of its associated simple action and another action. Thus, the order in which the associations were originally made may well be significant, since actions are not necessarily commutative, so this had to be taken into account when implementing

constructors for the classes. Figure 6.6 shows a variety of the sub-classes of the simple_resulting_action class which have been captured in the database.

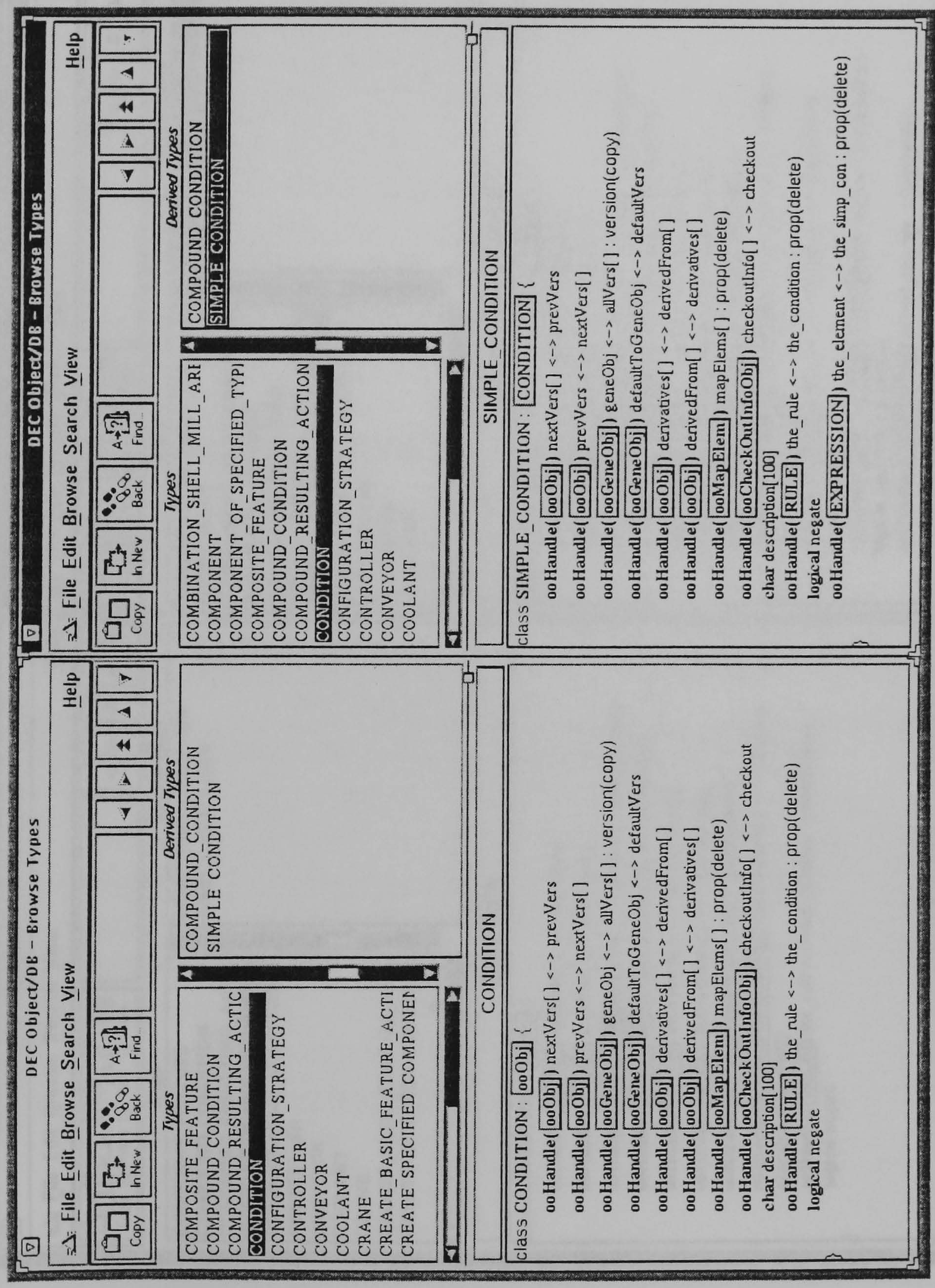


Figure 6.5a: Database Browser showing an implementation of the Condition class & Simple Condition class

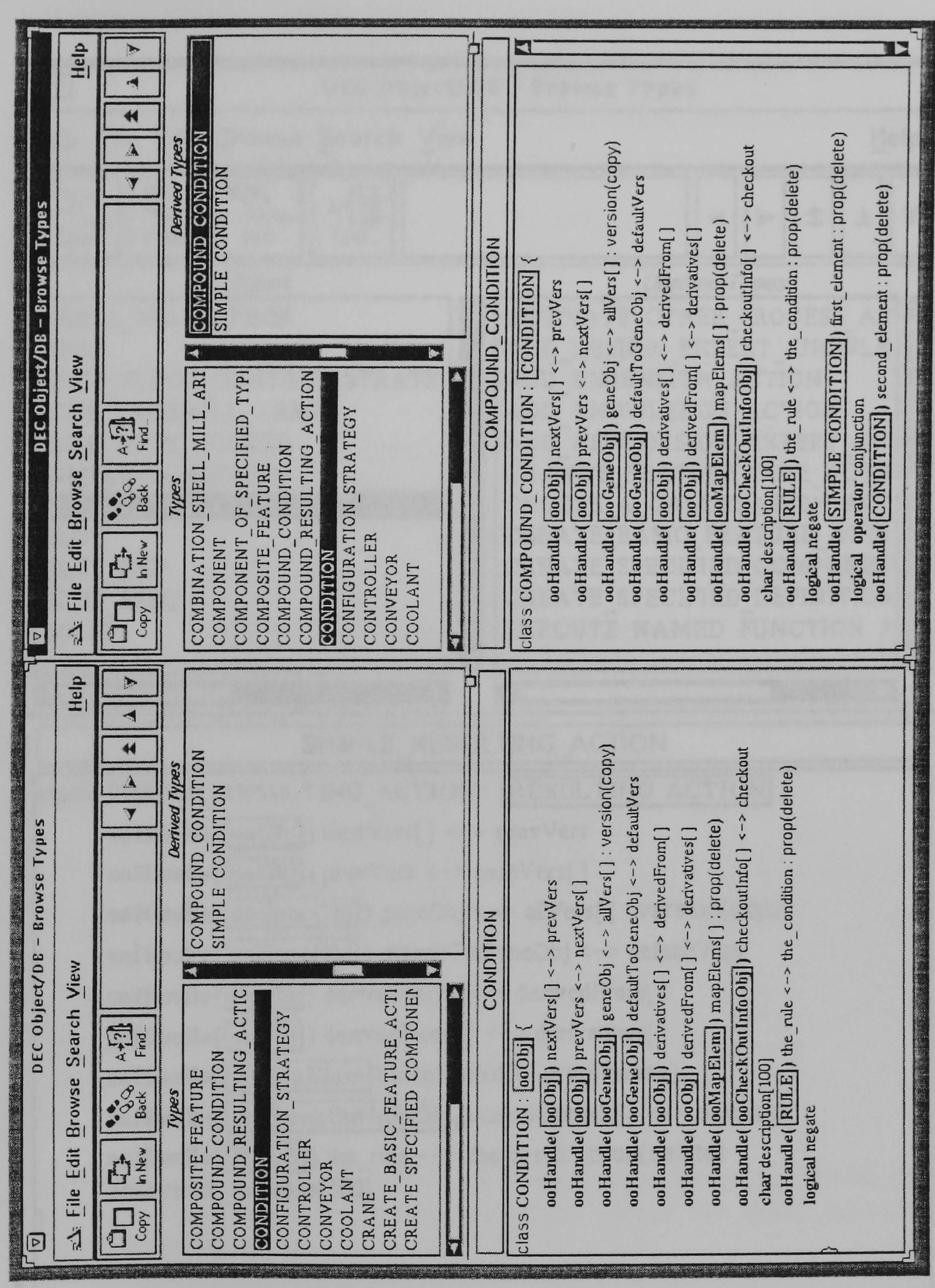


Figure 6.5b: Database Browser showing an implementation of Condition Class and Compound Condition class

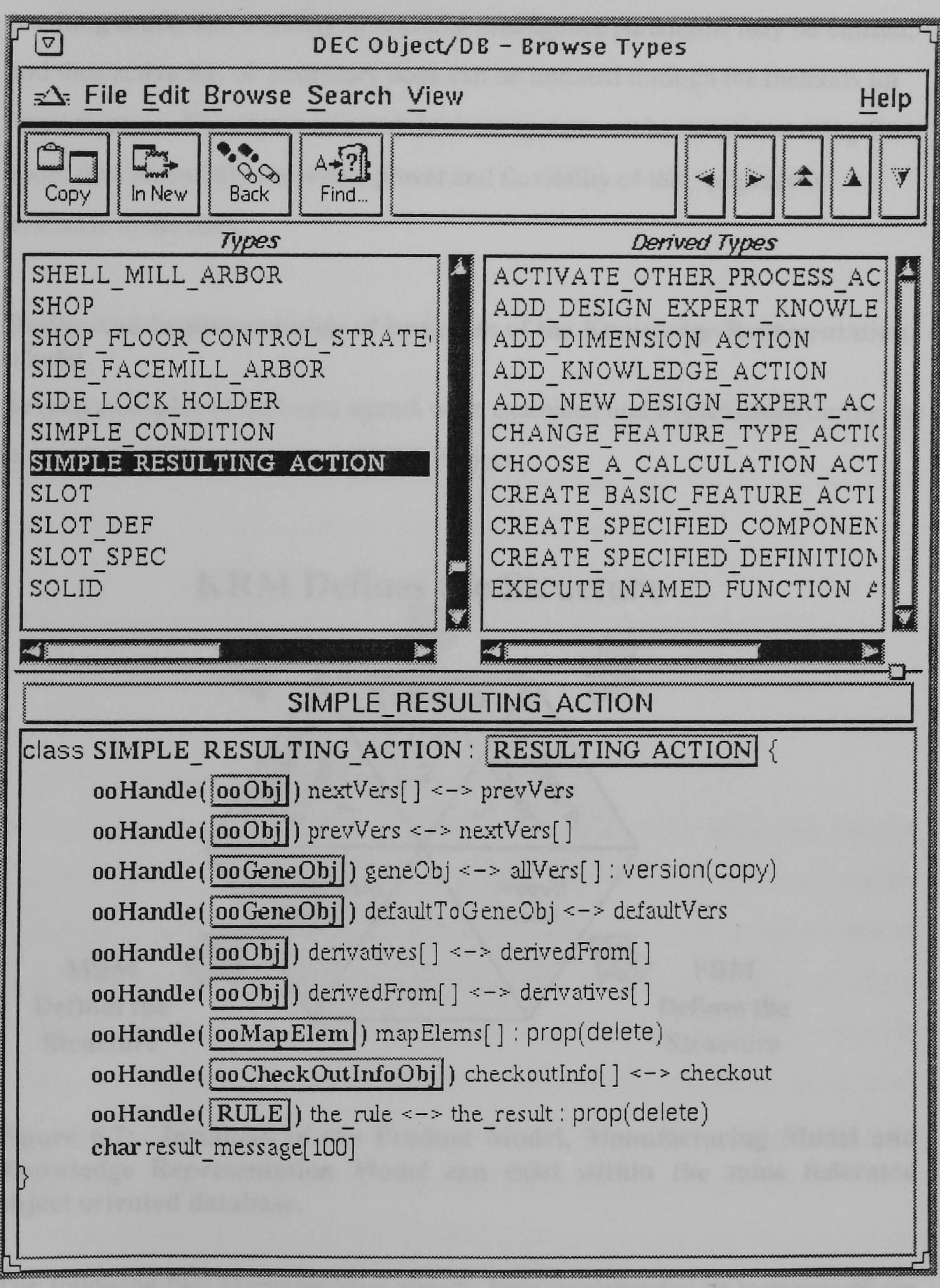


Figure 6.6: Database Browser showing a selection of sub-classes of simple resulting action class

It is during the evaluation of expression truth status or execution of a simple resulting action that a variety of artificial intelligence paradigms may be utilised, and thus activation of necessary code can be initiated through the methods for these classes. The object oriented database system can be interfaced using C++ code, thus potentially the whole power and flexibility of that language is available to the user.

6.2 Design and Implementation of Instances of the Knowledge Representation Model

Several examples of software agents were examined and implemented during the research associated with the MOSES project.

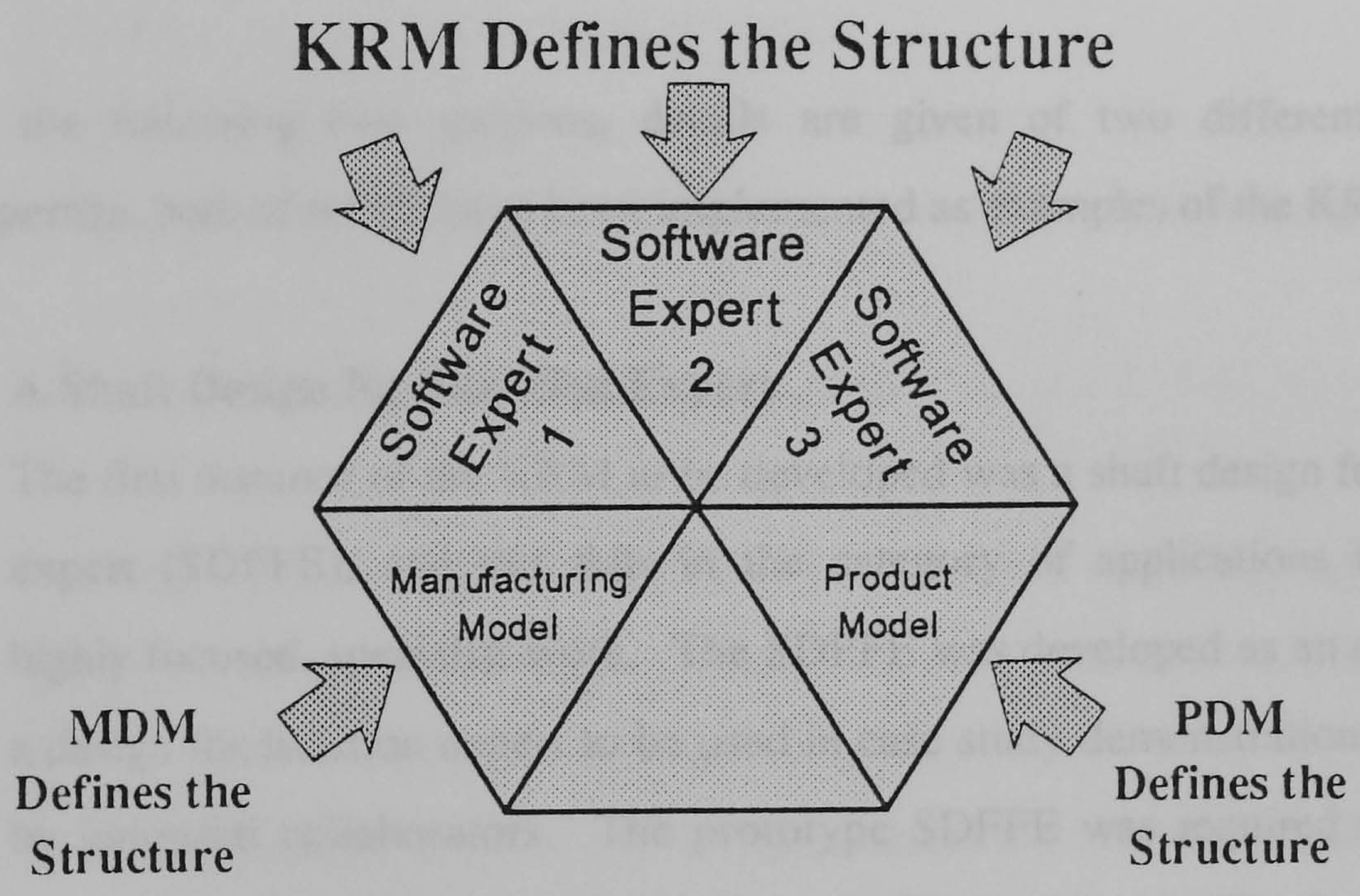


Figure 6.7: Instances of the Product Model, Manufacturing Model and Knowledge Representation Model can exist within the same federated object oriented database.

The following two examples have already been implemented as instances of the KRM, and details of their implementations have been chosen for inclusion here

as they represent very different types of software expertise and hence they have provided significant tests of the KRM.

The inclusion of details of these software agents demonstrates that implementation of specific software experts is facilitated by their embodiment as instances of the KRM. Thus the KRM is truly a model, in the same sense as a product data model, which may be instanced as product models for particular products (i.e. computer representations of specific products) (McKay et al 1995(1)) and a manufacturing model, instances of which store information relating to resources, processes and strategies of a manufacturing company (Molina et al 1994). Indeed instances of these information models and the KRM may be stored in the same federated data base, figure 6.7.

In the following two sections, details are given of two different types of expertise, both of which have been implemented as examples of the KRM.

6.2.1 A Shaft Design for Function Expert

The first instance of the KRM to be developed was a shaft design for function expert (SDFFE), and this falls in the category of applications to support highly focused, specialist work. The SDFFE was developed as an example of a design for function expert to be used in case study demonstrations provided by industrial collaborators. The prototype SDFFE was required to provide advice to support design for function activities, its specific domain being spinning shafts which are intended to be components for mechanical and electrical machines. Since it was intended to be a prototype example of a design for function expert, the SDFFE's design perspective was kept pure, focusing only on designing shafts with respect to their function, and taking no account of cost or manufacturing issues.

The design of shafts was primarily considered to be a form of variant design, and the initial implementation of the expert concentrated on taking a design from the conceptual stage through the embodiment stage. The knowledge was developed during a MSc project, and was based on a combination of the researchers' own experience, the content of various design text books, and advice from experienced design engineers at a collaborating company (Sadler, 1994). The approach taken to defining the knowledge content was initially to establish the general functionality of shafts and the role their constituent geometric features play. Using this approach, once the need for a shaft was established, the shaft was considered to have four basic requirements:

- It must have the ability to rotate freely
- It must locate in space both axially and radially
- It must transmit energy and therefore include areas for both energy input and energy output
- It must not fail as a result of its usual use

The knowledge required by the SDFFE to enable it to provide advice to the designer, related to these requirements, was developed as a set of production rules. These rules provided recommendations for a set of functional, geometric features which could be included on the shaft to satisfy the above functional requirements. Rules were also developed to record the designer's decisions within a product model database. The designer may wish to base his design decisions on information contained within the product specification. He may also wish to make an initial selection of material for the shaft, and perform basic calculations based on physical values related to his

chosen material. Thus, initially knowledge was acquired for the expert in the form of rules of a type similar to the following example:-

If the duty required from the shaft is not heavy, **then** a keyway is recommended as the functional feature for transmitting torque to or from the shaft.

The rules were structured into rulesets related to different stages of the design, and different sections of the shaft. Details of the specification requirements, available materials and of the geometry of the shaft were stored in a product model database. Hence sub-classes of expression were implemented to enable the truth of conditions relating to current contents of the product model database to be tested, and sub-classes of simple resulting action were implemented to enable details of the created design to be added to the database, or subsequently changed.

The SDFFE was then successfully implemented using a single expert module, whose single knowledge base (an object oriented database) held approximately 250 rule objects and their associated condition, expression and action objects. The SDFFE's working memory was comprised of several sections, relating to the main section of the shaft, two bearing journals, a shaft extension section, specification requirements and details of the material chosen for the shaft construction. Hence the SDFFE's memory was implemented as a sub-class of the working memory base class, with associated classes relating to the various functional sections of the shaft. The implemented solution for this class structure hierarchy within the database can be seen in figure 6.8. Further details of the functionality and implementation of the SDFFE may be found in Harding & Popplewell, 1995.

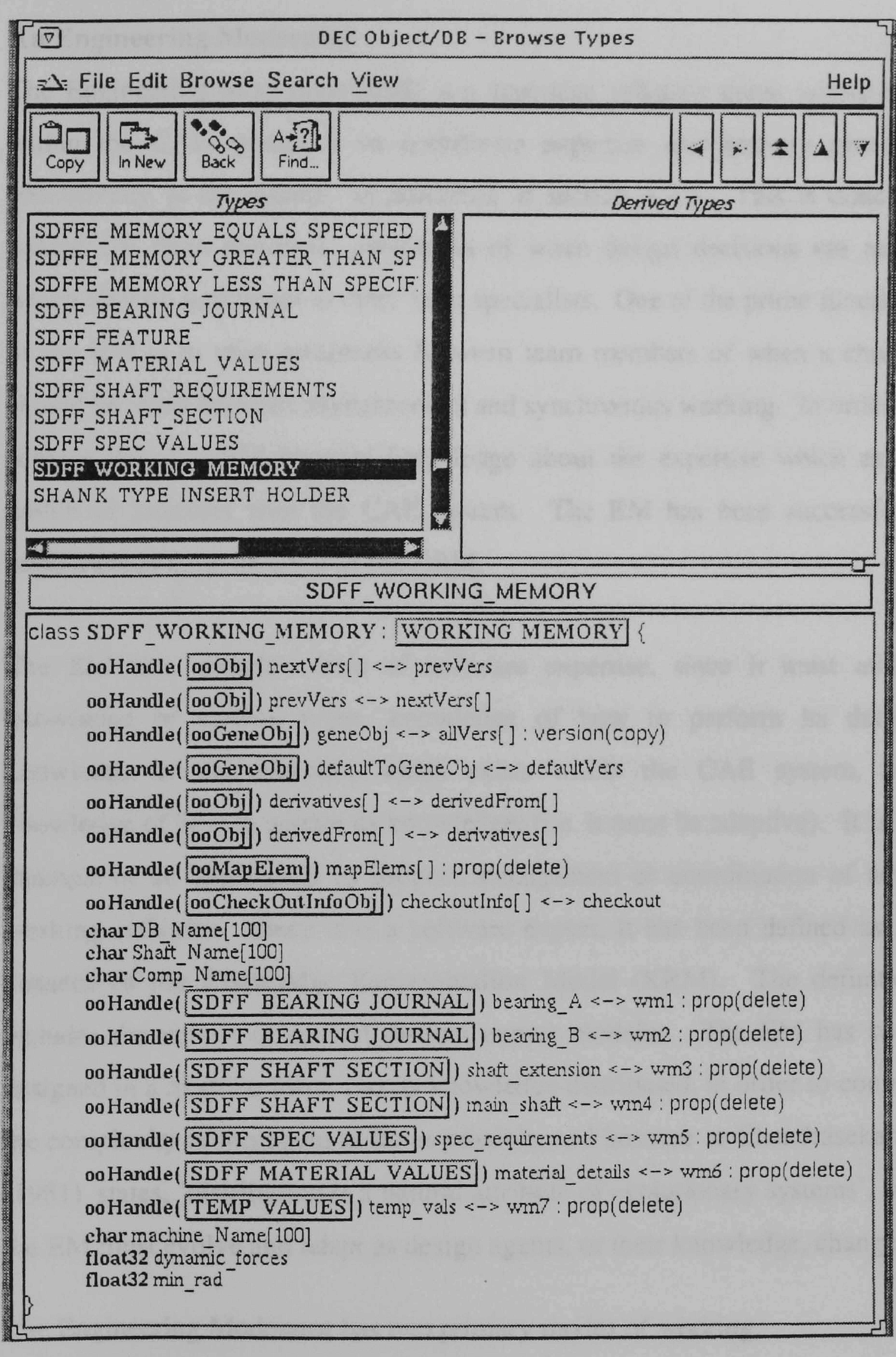


Figure 6.8: Database Browser showing sections of the Shaft Design for Function Expert's Working Memory class, which was implemented as a subclass of Working Memory base class.

6.2.2 An Engineering Moderator

The Engineering Moderator (EM) is a specialist software agent whose role within the CAE system is to coordinate expertise and actively promote concurrency in the system, as described in section 4.2.5. This is done by raising CE team members' awareness of when design decisions are taken which may be significant to other team specialists. One of the prime functions of the EM is to raise awareness between team members of when a change should be made between asynchronous and synchronous working. In order to achieve this, the EM requires knowledge about the expertise which exists within or interacts with the CAE system. The EM has been successfully implemented as an instance of the KRM.

The EM is a complex form of software expertise, since it must utilise knowledge of various types, knowledge of how to perform its duties, knowledge of the expertise which exists within the CAE system, and knowledge of how to update its knowledge, (i.e. it must be adaptive). It is an example of an application to support management or coordination of team working activities. Since it is a software expert, it has been defined as an instance of the Knowledge Representation Model (KRM). The definition includes the details of several distinct expert modules. The EM has been designed in a modular form, and its knowledge distributed, in order to control the complexity of this agent as far as possible, and because as Chandrasekaran (1981) states, 'distribution is a natural attribute of evolutionary systems', and the EM must evolve and adapt as design agents, or their knowledge, change.

The Engineering Moderator has two primary modes of working,

- **Knowledge Acquisition Mode** - this is the working mode in which the engineering moderator can update its knowledge relating to particular design

agents which may operate within the CAE system, i.e. actual software experts such as the SDFFE, or human expertise as considered during a case study involving cost and delivery knowledge (McKay, 1995(b)). Details of additional or new design expertise can also be added to the EM when it is operating in this mode. The EM may be run in this mode, by choice, at any time.

- **Design Moderation Mode** - this is the EM's normal mode of operation, as it runs in this mode in the background throughout the design of a product, from specification onwards

Operation in either of these modes requires use of one or more types of expertise, each of which has been defined as an expert module based on the KRM. Hence, the Engineering Moderator is made up of a collection of expert modules, each of which can be thought of as an 'Expert' in its own right. See figure 6.9, which is a Booch representation of the EM's expertise, showing that it comprises of three types of expertise, represented by knowledge acquisition module, design moderation module and design agent modules, the purpose and implementations of each are described below.

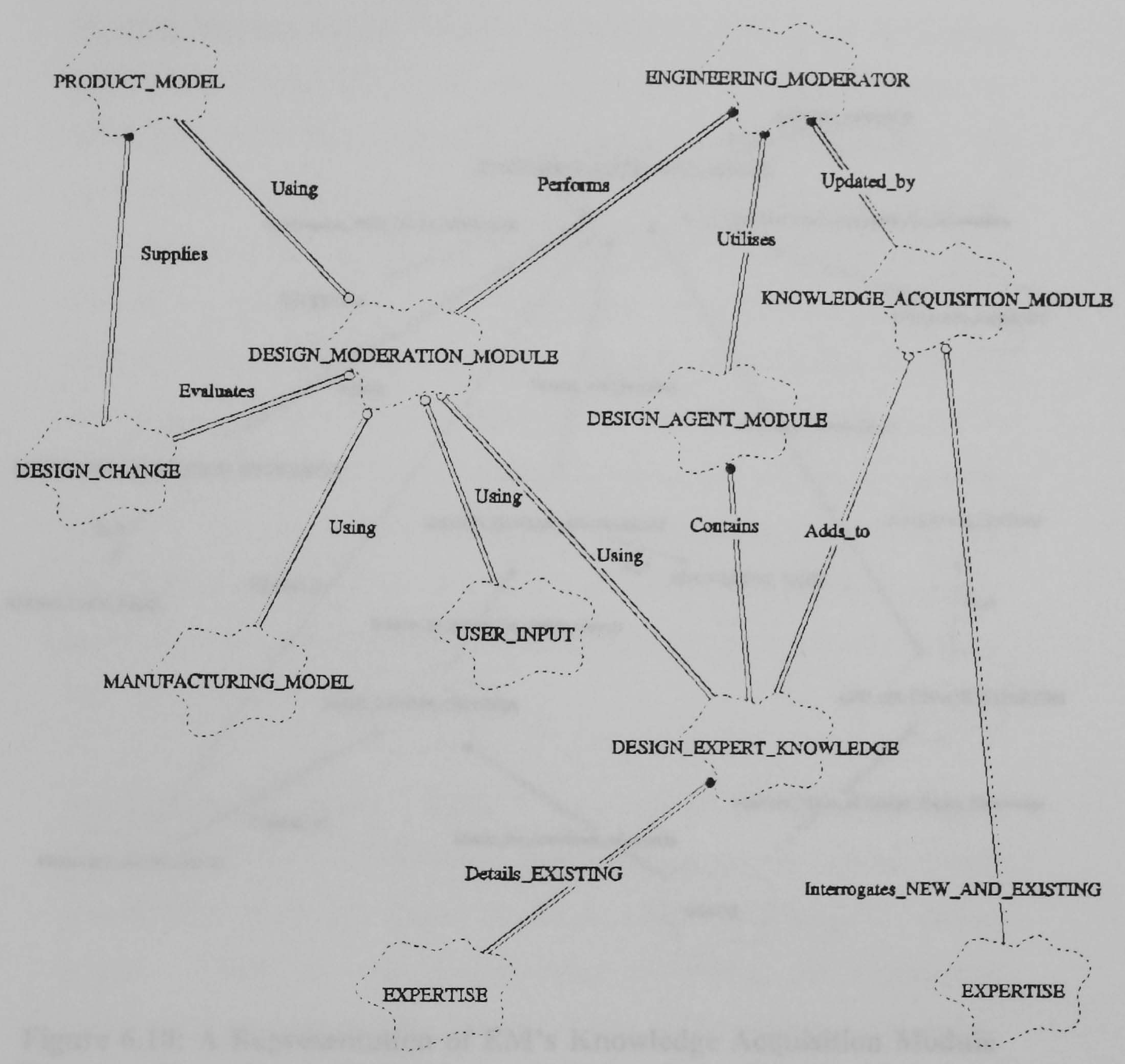


Figure 6.9: A Representation of Engineering Moderator Expertise Using Booch Object Oriented Design Graphical Notation

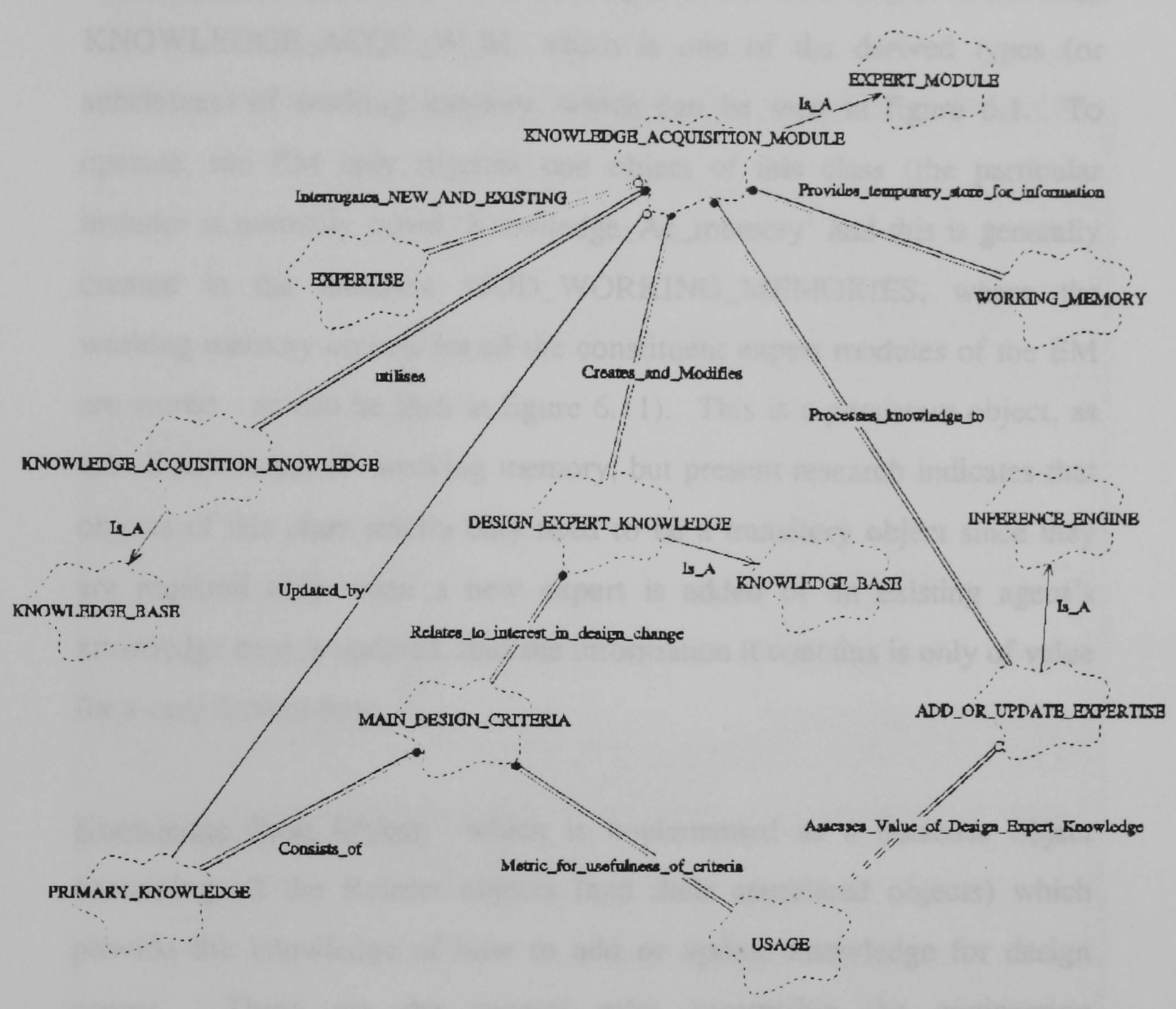


Figure 6.10: A Representation of EM's Knowledge Acquisition Module Expertise Using Booch Object Oriented Design Graphical Notation

6.2.2.1 Knowledge Acquisition Module

This aspect of the EM's expertise enables the EM to update its knowledge of the expertise which exists within and interacts with the CAE system. This is an expert module, and its structure as an instance of the KRM is modelled in figure 6.10, where it is shown to be associated with the following objects:-

Working Memory Object: which is implemented as an object of the class KNOWLEDGE_ACQU_W_M, which is one of the derived types (or subclasses) of working memory, which can be seen in figure 6.1. To operate, the EM only requires one object of this class (the particular instance is normally called 'Knowledge_Ac_memory' and this is generally created in the database MOD_WORKING_MEMORIES, where the working memory objects for all the constituent expert modules of the EM are stored - as can be seen in figure 6.11). This is a persistent object, as are all subclasses of working memory, but present research indicates that objects of this class strictly only need to be a transitory object since they are required only when a new expert is added or an existing agent's knowledge base is updated, thus the information it contains is only of value for a very limited time.

Knowledge Base Object: which is implemented as a database object containing all the Ruleset objects (and their associated objects) which provide the knowledge of how to add or update knowledge for design agents. These are the general rules comprising the engineering moderator's knowledge about how to update its knowledge about design agents and their knowledge (the engineering moderator's mental models of the design agents). This knowledge base object is normally called 'KNOWLEDGE_ACQ_RULES'.

Inference Engine Object: which is currently implemented as a function within the main EM program, Mod_mk4.C. (It is the function know_ac_rules_fire_all). This could just as easily be implemented as an object within the object oriented database, where the functionality of the behaviour of the object matches the functionality of the function

know_ac_rules_fire_all. The functionality enables the knowledge information contained within the knowledge base to be used (processed).

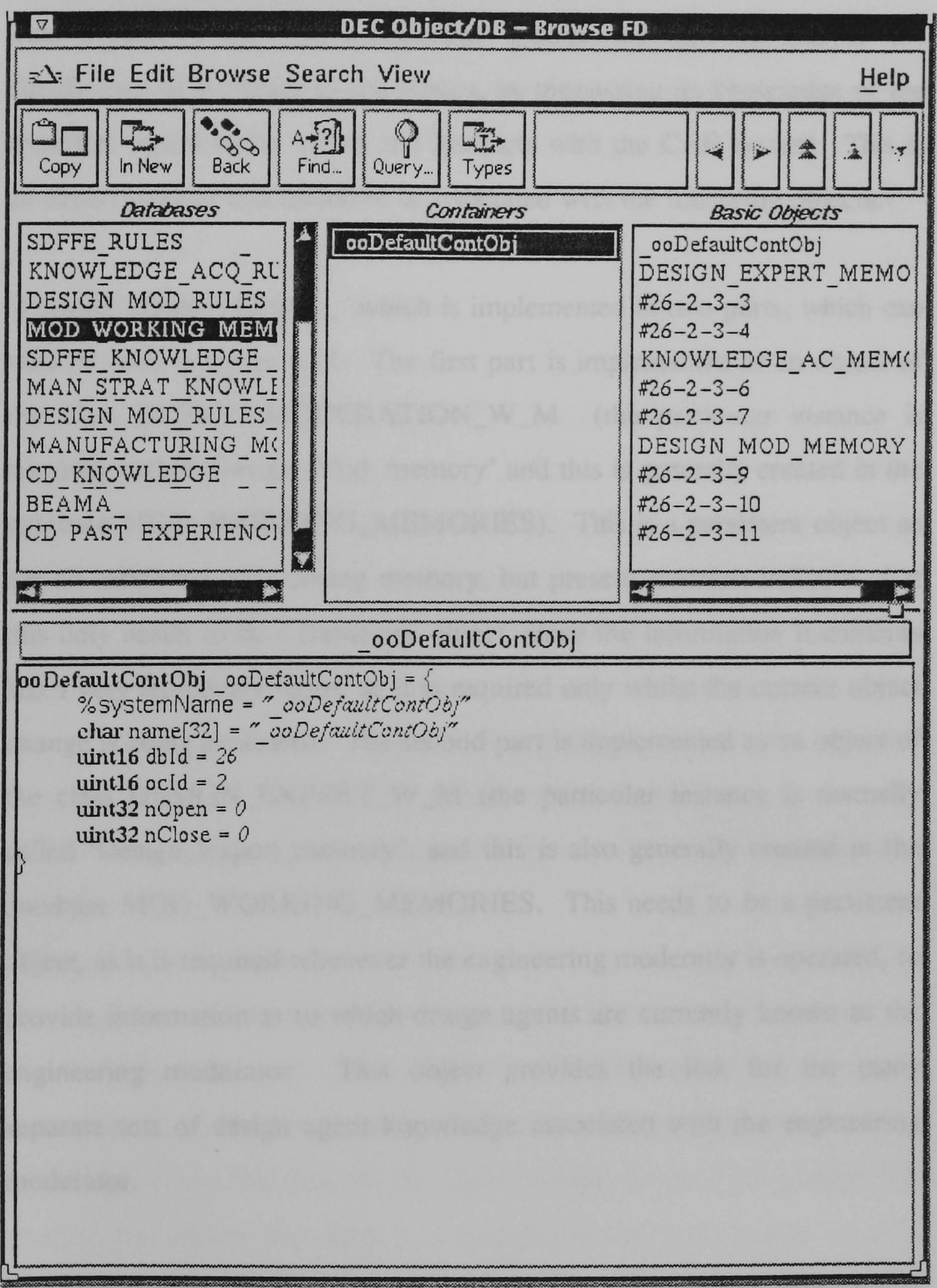


Figure 6.11: Database Browser showing the Moderator Working Memory database, which contains the Working Memory objects for all the Expert Module Components of the Engineering Moderator.

6.2.2.2 Design Moderation Module

This aspect of the EM's expertise enables the EM to analyse the significance of a change to the design, by processing its knowledge of the expertise which exists within and interacts with the CAE system. This is an expert module and therefore is associated with the following objects:-

Working Memory Object: which is implemented in two parts, which can both be seen in figure 6.11. The first part is implemented as an object of the class DESIGN_MODERATION_W_M (the particular instance is normally called 'Design_Mod_memory' and this is generally created in the database MOD_WORKING_MEMORIES). This is a persistent object as are all subclasses of working memory, but present research indicates that this only needs to be a transitory object, since the information it contains has a very temporary value, as it is required only whilst the current object change is being evaluated. The second part is implemented as an object of the class DESIGN_EXPERT_W_M (the particular instance is normally called 'Design_Expert_memory', and this is also generally created in the database MOD_WORKING_MEMORIES. This needs to be a persistent object, as it is required whenever the engineering moderator is operated, to provide information as to which design agents are currently known to the engineering moderator. This object provides the link for the many separate sets of design agent knowledge associated with the engineering moderator.

Knowledge Base Objects: there are currently 3 knowledge base objects associated with this expert module, since this module uses three types of expert knowledge. Firstly it requires knowledge of how to obtain and use

information related to the object which has been changed, i.e. product model objects. Secondly it requires knowledge of how to use the Engineering Moderator's knowledge of design agent(s)'s expertise. Thirdly it requires knowledge of how to communicate the results of its moderation activities. Currently these knowledge base objects are implemented as three database objects called DESIGN_MOD_RULES_GEN, DESIGN_MOD_RULES_AGENT and DESIGN_MOD_RULES_REPORT, each of which contains all the rulesets which comprise the knowledge related to its specific type of expertise applied to evaluating the current design change.

Inference Engine Object: which is currently implemented as a function within the Mod_mk4.C program. (It is the function moderate_rules_fire_all). This could just as easily be implemented as an object within the object oriented database, where the functionality of the behaviour of the object matches the functionality of the function moderate_rules_fire_all. The functionality enables the knowledge information contained within the knowledge bases to be used (processed).

6.2.2.3 Design Agent Modules

This aspect of the EM's expertise enables the EM to utilise knowledge of the expertise of particular design agents. The EM requires knowledge of the expertise which exists within, or interacts with the CAE system. This knowledge is not the same as the actual design expert's knowledge, but research has shown that there is a mapping between the design expert's knowledge, and the EM's knowledge of the design expert's knowledge. The EM's knowledge of design agents is stored as several expert modules, i.e. one for each design agent. A Booch representation of the design

expert knowledge stored as part of a design agent module, is shown in Figure 6.12.

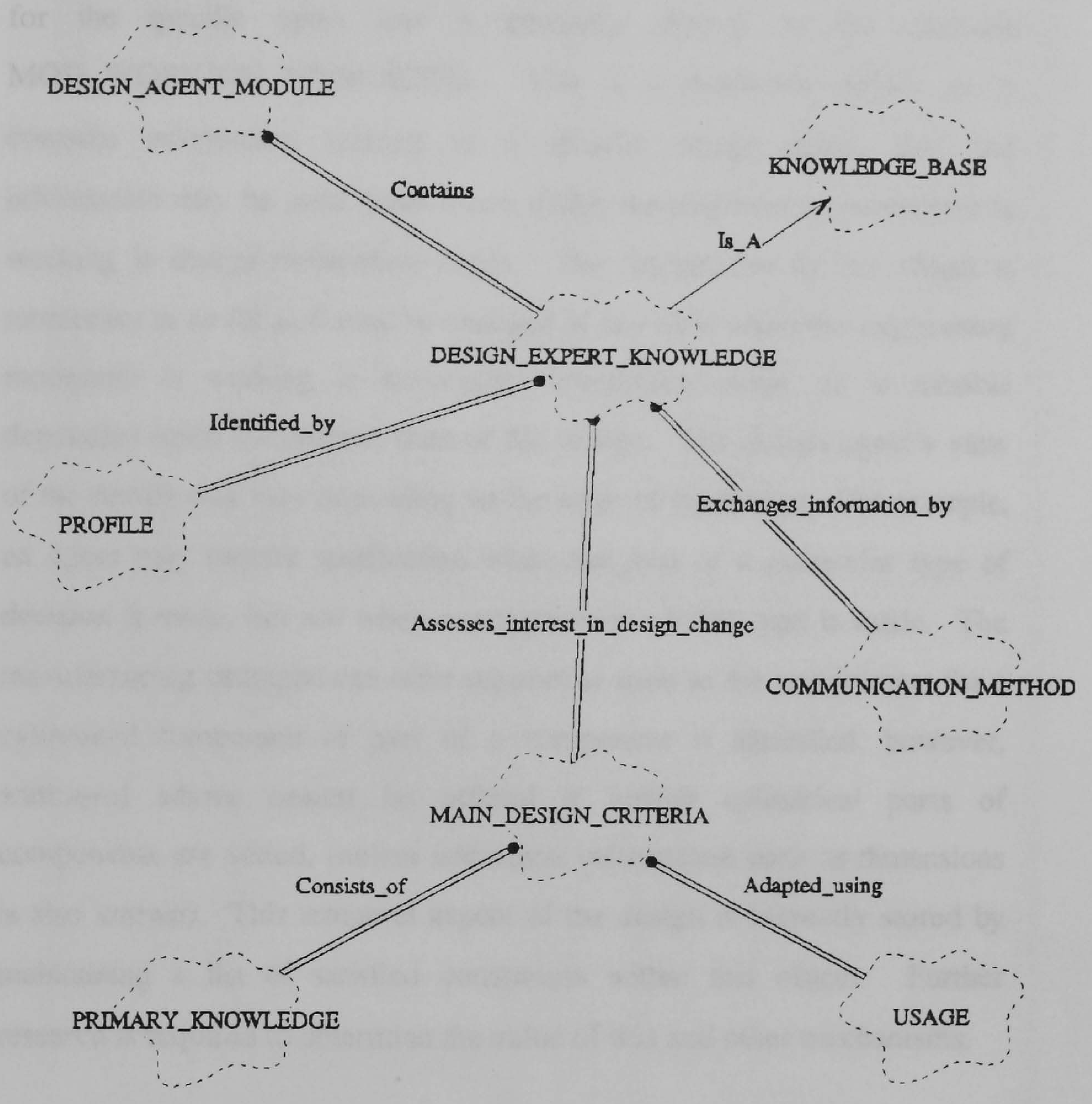


Figure 6.12: A Representation of the knowledge stored by the EM relating to design agent expertise existing within or interacting with the CAE system.

Each design agent module is therefore associated with the following objects:-.

Working Memory Object: which is implemented as an object of the class MOD_AGENT_VALUES (the particular instance is named appropriately for the specific agent and is generally created in the database MOD_WORKING_MEMORIES). This is a persistent object, as it contains information relating to a specific design agent, and this information may be used many times whilst the engineering moderator is working in design moderation mode. The information in this object is temporary in so far as it may be changed at any time when the engineering moderator is working in knowledge acquisition mode, or is variable dependent upon the current state of the design. The design agent's view of the design may vary depending on the stage of the design. For example, an agent may require notification when the *first* of a particular type of decision is made, but *not* when *every* decision of that type is made. The manufacturing strategist can offer support as soon as the requirement for a cylindrical component or part of a component is identified, however, additional advice cannot be offered if further cylindrical parts of components are added, (unless additional information such as dimensions is also known). This temporal aspect of the design is currently stored by maintaining a list of satisfied constraints within this object. Further research is required to determine the value of this and other mechanisms.

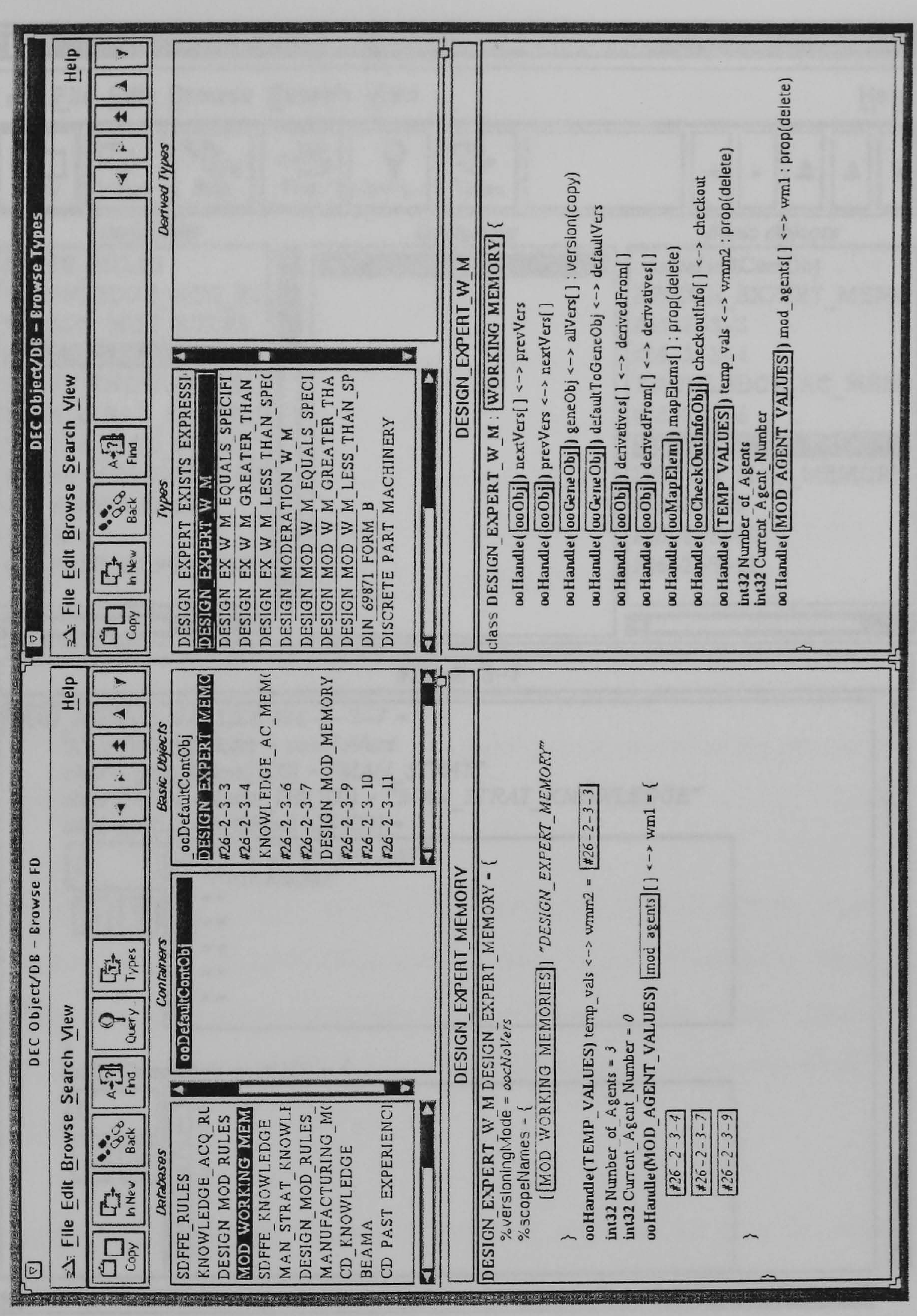


Figure 6.13: Database Browser showing the relationships between the EM's Design Expert element of the Design Moderation working memory and instances of Mod Agent Values working memories (for an implemented example of the EM).

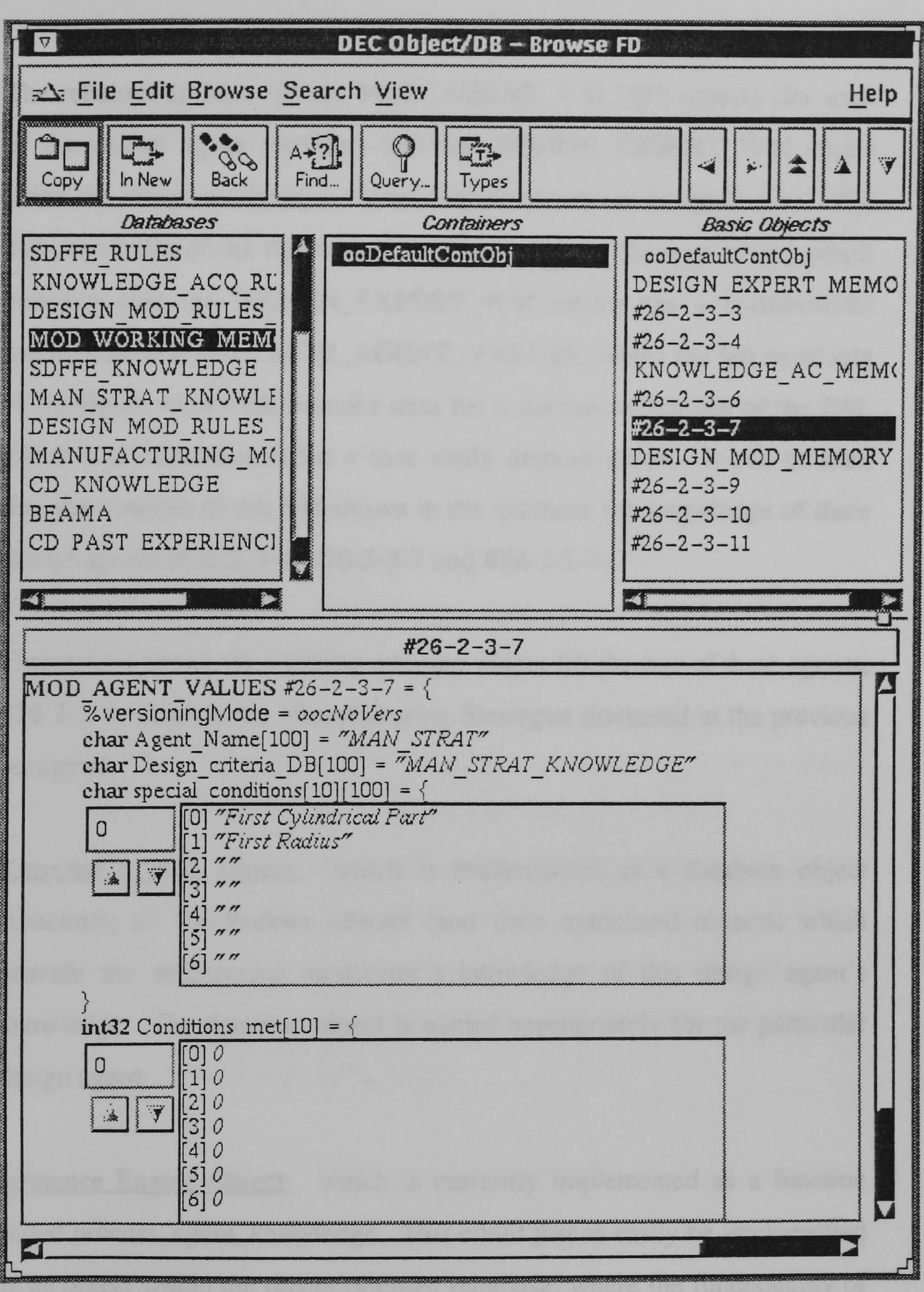


Figure 6.14: Database Browser showing the instance of a Mod Agent Values Working Memory containing knowledge of the Manufacturing Strategist design agent.

The relationship between the MOD_AGENT_VALUES objects (for each of the design agent modules) and the DESIGN_EXPERT_WM object (from the design moderation module) is demonstrated in figure 6.13. The right hand side of the figure shows the structure of the type (class) which indicates that the DESIGN_EXPERT_WM object has a bi-directional association with many MOD_AGENT_VALUES, whilst the left hand side of the figure shows the instance data for a particular example of the EM, which was implemented for a case study demonstration. It can be seen that the example of the EM shown in the database has knowledge of three design agents (#26-2-3-4, #26-2-3-7 and #26-2-3-9).

Figure 6.14 shows the working memory object for the one of these agents, #26-2-3-7, which is the Manufacturing Strategist discussed in the previous paragraph.

<u>Knowledge Base Object:</u> which is implemented as a database object containing all the Ruleset objects (and their associated objects) which provide the engineering moderator's knowledge of this design agent's knowledge. The database object is named appropriately for the particular design agent.

<u>Inference Engine Object:</u> which is currently implemented as a function called process_agent_knowledge. This could just as easily be implemented as an object within the object oriented database, where the functionality of the behaviour of the object matches the functionality of the function process_agent_knowledge. The functionality enables the knowledge information contained within the knowledge bases to be used (processed).

6.3 Extensions of the Knowledge Representation Model

The KRM has been tested through case study work, and its value established. Inevitably this has led to the identification of aspects of the research which can be extended and improved. In the next version of the KRM, a class structure for inference engine objects will be implemented and tested. Also, whilst the flexible hybrid structure of the KRM has already been tested through embedding neural network simulation code (Harding & Popplewell, 1995), it is hoped that further experimentation of this nature can be undertaken using Genetic Algorithm applications. Indeed, it should be possible to extend the KRM to support any artificial intelligence paradigm which can be simulated or captured in program code.

More flexible methods for extending the working memory class hierarchy should also be implemented, since this will facilitate the creation of additional software experts to integrate with the existing CAE system. Further effort should also be expended in determining ways in which the temporal aspects of design knowledge may best be captured using the KRM, since it is clear from the existing research that the value of particular pieces of knowledge changes over time, i.e. at different stages of product design.

7. Conclusions

This thesis has shown that the Knowledge Representation Model can satisfy the requirements for supporting CE teams. The KRM facilitates the design and implementation of the multiple, diverse forms of software expertise, which should exist as part of the support for CE team working provided by future CAE systems.

This has been achieved by initially identifying the range of support which should be provided by CAE systems. The KRM is specifically of value in creating intelligent software systems, so aspects of support in which artificial intelligence could be usefully employed were then analysed. Two broad categories of essential software expertise were identified. Firstly, intelligent support may be provided for highly focused, specialist work. The requirement for this type of application is well established, and a variety of examples of such software expertise can be identified in many of the software systems discussed in chapter 4. The second form of essential software expertise identified is required to provide intelligent support for coordination of team-working activities. More than an integration environment or a CSCW system to support synchronous team working activities is generally needed. In addition, concurrent team working must be actively encouraged, therefore there is the need for communication between team members to be actively promoted, or even driven when a change from asynchronous to synchronous working methods is required. This requirement is not adequately addressed by most of the identified software systems.

The KRM has been proved to effectively support design and implementation of both of the above types of software expertise. This has been done through production of significant software demonstrations which are described in detail in chapters 5 and 6. The concepts have been well tested since software has been written to instantiate

the KRM itself, and to instantiate different examples of software expertise which have been modelled using the KRM. Hence it is believed that the principle aim of this work has been fully satisfied.

The main strengths of the KRM approach are as follow:-

- It is flexible. It enables knowledge to be captured within any commercially supported, object oriented database system.

- It is versatile. It supports design and implementation of a wide range of diverse software experts - as demonstrated in chapter 6

- It is hybrid in nature and enables knowledge to be captured in a variety of forms, e.g. production rules, neural networks, etc. Indeed, any simulation or activity which can be captured as program code can be activated through the KRM.

- It supports knowledge sharing, since knowledge is available in a normal database system.

In its current form, the main limitations of the KRM are as follow:-

- It does not effectively support the changing value of knowledge over time. That is, there are currently no in-built metrics for evaluating worth of knowledge, and prioritising its use.

- It does not support conflict resolution - implementation of fully automated software systems has not been attempted, since they have not been considered valuable in the context of the research so far. The KRM concept may need to be extended to enable their implementation.

However, these limitations are not considered to be insurmountable. They are believed to be limits of the current research rather than serious restrictions on the KRM Concept.

Hence, the purpose of the thesis has been satisfied, since the KRM meets the requirements which have been identified as necessary to support CE teams. This has been achieved since the KRM provides a sound basis for the creation of necessary software applications.

REFERENCES

Adler M, Durfee E, Huhns M, Punch W and Simoudis E, (1992) AAAI Workshop on Cooperation Among Heterogeneous Intelligent Agents, *AI Magazine*, Summer 1992.

Bird S D (1993) Toward a taxonomy of multi-agent systems. *International Journal of Man-Machine Studies,* 39, 689-709.

Blessing Lucienne (1991) Engineering design and artificial intelligence: a promising marriage? *Research in design thinking* proceedings of a workshop meeting held at Faculty of Industrial Design Engineering, Delft University of Technology, The Netherlands, May 29-31, 1991. eds N Cross, K Dorst and N Roozenburg.

Booch Grady (1991) *Object Oriented Design with Applications* The Benjamin/Cummings Publishing Company, Inc.

Bobrow D G (1991) AAAI-90 Presidential Address, Dimensions of Interaction, in *AI Magazine,* Fall 1991 64-80.

Borja Vicente (1995) *Product and Manufacturing Models applied to reverse engineering,* Annual PhD report, March, 1995.

Bracewell R H, Bradley D A, Chaplin R V, Langdon P M and Sharpe J E E (1993) Schemebuilder, A Design Aid for the Conceptual Stages of Product Design. *ICED'93 International conference on engineering design,* The Hague, August 17-19, 1993.

Bracewell R H, Chaplin P M, Landon P M, Li M, Oh V K, Sharpe J E E and Yan X T (1994). Integrated Computer Support for Inter-disciplinary System Design, *Engineering Design Centre Internal Report,* January 1994.

Chandrasekaran B, (1981) Natural and Social System Metaphors for Distributed Problem Solving: Introduction to the Issue. *IEEE Transactions on Systems, Man and Cybernetics,* Vol SMC-11 No:1 January 1981.

Corbett J et al (1991) *Design for Manufacture: Strategies, principles and techniques* Addison-Wesley Publishing Company, Wokingham.

Crowder R, Hall W, Heath I and Bernard R (1995) The Application of Large-Scale Hypermedia Information Systems to Training, *IETI*, 32, 3, 245-255.

Cutkosky M R, Engelmore R S, Fikes R E, Genersereth M R , Gruber T R, Mark W S, Tenenbaum J M and Weber J C, (1993) PACT: An Experiment in Integrating Concurrent Engineering Systems. *Computer* Vol 26 #1.

Dean J W & Susman G I (1989) Organizing for Manufacturable Design, in *Harvard Business Review,* Jan-Feb

Douglas R E & Brown (1993) Concurrent accumulation of knowledge: a view of concurrent engineering, in *Concurrent Engineering: Contemporary Issues and Modern Design Tools,* eds H R Parsaei & W G Sullivan, Chapman & Hall

Dowlatshahi S (1994) A Comparison of Approaches to Concurrent Engineering, in *The International Journal of Advanced Manufacturing Technology* 9; 106-113

Ekvall G (1991) The Organizational culture of idea-management: a creative climate for the management of ideas. *Managing Innovation* eds Jane Henry & David Walker, Sage Publications. ISBN 0-8039-8506-1.

Ellis T I A, Young R I M, Bell R (1993) Modelling Manufacturing Process Information to Support Simultaneous Engineering, published in *Proceedings of International Conference on Engineering Design, (ICED '93),* The Hague, August 17-19, 1993.

Ellis T I A, Molina A, Young R I M, Bell R (1994) The Development of an Information Sharing Platform for Concurrent Engineering, presented at the International Manufacturing Systems Engineering Workshop, December 12-14, 1994, Grenoble, France.

Erens F, McKay A, Bloor M S (1995) Product Modelling using Multiple Levels of Abstraction, Instances as Types, *Computers in Industry*, 24(1) 17-28.

Feltham G A & Xie J (1994) Performance-Measure Congruity and Diversity in Multitask Principal-Agent Relations. *Accounting Review,* V69 N3, P429-453.

Gaines B R & Norrie D H. Mediator: Information and Knowledge Management for the Virtual Factory. Knowledge Science Institute & Division of Manufacturing Engineering, University of Calgary, Calgary, Alberta, Canada. T2N 1NA.

Galbraith, J (1973) *Designing Complex Organizations,* Reading M A: Addison-Wesley.

Gannon, S (1995) *The Embedding of a neural network into a knowledge based system.* Final year project submitted to Dept. of Manufacturing Engineering, Loughborough University of Technology, May, 1995.

Ginsberg, M L (1991) Knowledge Interchange Format: The KIF of Death. *AI Magazine,* Fall 1991.

Gu P and Chan Kam, (1995) Product Modelling Using STEP. *Computer-Aided Design*, Vol 27, No3, pp163-179

Halasz F G (1988) Reflections on Notecards: Seven Issues for the next generation of hypermedia systems. *Communications of the ACM* Vol 31 (7)

Hale D P, Hurd J E, Kasper G M (1991) A Knowledge Exchange Architecture for Collaborative Human-Computer Communication. *IEEE Transactions on Systems, Man and Cybernetics* Vol 21, No 3, May/June, 1991.

Harding, J A & Popplewell, K (1994) (1) The Rationale for an Engineering Moderator. MOSES report series 35, November 1994.

Harding, J A & Popplewell, K (1994) (2) Engineering Moderation Within A Concurrent Engineering Environment. MOSES report series 36, November 1994.

Harding, J A & Popplewell, K (1995) A Shaft Design for Function Expert with Object Oriented Database Knowledge. MOSES report series 40, May 1995.

Hayes-Roth F, Davidson J E, Erman L D, and Lark J S (1991) Frameworks for Developing Intelligent Systems, *IEEE Expert.* June, 1991.

Heath I, Hall W, Crowder R M, Pasha M A and Soper P J (1994) Integrating a knowledge base with an open hypermedia system and its application in an industrial environment. *Proceedings of the 3rd international workshop on information and knowledge management, CIKM94, Workshop on Intelligent Hypertext,* NIST Gaithersburg, November 1994.

Ishii, Hiroshi & Miyake, Naomi (1991) Toward an Open Shared Workspace: Computer & Video Fusion Approach to Team Work Station. *Communications of the ACM*, Dec 1991, Vol 34, No:12

Jackson P (1990) Introduction to Expert Systems. Addison-Wesley Publishing Company, ISBN 0-201-17578-9.

Jennings N R (1995) Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions. *Artificial Intelligence* 75, 195-240.

Jennings N R. Cooperation In Industrial Systems, *Proc ESPRIT Conf.*, Brussels, Belgium, 253-263 (Obtained via Queen Mary Westfield World Wide Web page, 1995)

Jo H H, Parsaei H R, and Sullivan W G (1993) Principles of Concurrent Engineering, in *Concurrent Engineering: Contemporary Issues and Modern Design Tools,* eds H.R. Parsaei and W.G. Sullivan, Chapman & Hall.

Knaus Rodger and Jay Chris (1990) Transporting Knowledge Bases: A Standard, *AI Expert,* November 1990.

Krause F L, Kimura F, Kjellberg T, Lu S C Y (1993) Product Modelling, *Annals of the CIRP* vol 42/2/1993

Lawley M (1992) System Workbench for Integrating and Facilitating Teams (SWIFT) *Knowledge-Based Engineering Systems Research Laboratory Annual Report.* Dept of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign.

Londono F, Cleetus K J and Reddy Y V, 'A Blackboard Scheme for Cooperative Problem Solving by Human Experts' Proceedings of MIT-JSME Workshop on Cooperative Product Development, November 1989.

Lu, Stephen C-Y (1992) Research, Development, and Implementation of Knowledge Processing Tools to Support Concurrent Engineering Tasks. *Knowledge-Based Engineering Systems Research Laboratory Annual Report.* Dept of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign.

Majaro S (1992) *Managing Ideas for Profit, the creative gap.* McGraw-Hill Book Company.

Manola Frank, (1990) Object-Oriented Knowledge Bases, *AI Expert*, March, 1990.

Mayer R J & Painter M K (1991) Roadmap for Enterprise Integration, Autofact '91 Conference Proceedings, November 10-14, Chicago, Illinois.

McBrien P, Seltveit A H, Wangler B (1992) An Entity-Relationship Model Extended to Describe Historical Information. *Proceedings of CISMOD92, Bangalore India.*

McKay A, Bloor M S, de Pennington A, (1995)(1) A Framework for Product Data. *IEEE Trans Knowledge and Data.* (accepted for publication)

McKay A, Juster N P, Harding J A, Popplewell K (1995)(2) Tendering for Improved Competitiveness: a case study. ISATA (28th International Symposium on Automotive Technology & Automation), Stuttgart, 18-22 September, 1995.

Meerkamm H (1994) Design for X - A Core Area of Design Methodology. *Journal of Engineering Design* Vol 5, No 2.

Meyer, Christopher (1993) *Fast Cycle Time, How to Align Strategy & Structure for Speed,* Free Press, New York, ISNBN 0-02-921181-6.

Molina A, Ellis T I A, Young R I M, Bell R (1994) Modelling Manufacturing Resources, Processes and Strategies to Support Concurrent Engineering, 1st International Conference on Concurrent Engineering Research & Applications, August 29-31, 1994, Pittsburgh, PA, USA.

Molina A, Al-Ashaab A H, Ellis T I A, Young R I M, Bell R (1995), A Review of Computer Aided Simultaneous Engineering Systems, *Research in Engineering Design,* 7, -- 38-63.

Moynihan, Gary P., (1993) Application of expert systems to Engineering Design, in *Concurrent Engineering: Contemporary Issues and Modern Design Tools,* eds H R Parsaei and W G Sullivan, Chapman & Hall.

Nevins J L & Whitney D E (1991) in *Computer Integrated Design and Manufacturing* (D D Bedworth, M R Henderson & P M Wolfe) Ch 4, McGraw-Hill, Inc. New York.

Oh V (1993) Intelligent Design Assistant Systems for Engineering Design. *Technical Report EDC-1993/02, Lancaster University Engineering Design Centre.*

Parsaei H R & Sullivan W G (1993) eds *Concurrent Engineering: Contemporary Issues and Modern Design Tools* Chapman & Hall

Pinto J K & Slevin D P (1987) Critical Factors in Successful Project Implementation, *IEEE Transactions on Engineering Management,* Vol, EM-34, No:1 February, pages 22-27.

Popplewell K, Harding J A (1995) Engineering Moderation: Supporting Concurrency in Engineering Using Hybrid Knowledge Representation. IFIP WG5.7 Working Conference on Managing Concurrent Manufacturing to Improve Industrial Performance, September 11-15, 1995, Seattle, Washington, USA.

Rational (1993) *The Booch Method: A Case Study for Rational Rose* Rational, 3320 Scott Boulevard, Santa Clara, California 95054

Rich E & Knight K (1991) *Artificial Intelligence,* McGraw Hill.

Roth K & Ricks D A (1994) Goal Configuration in a Global Industry Context. *Strategic Management Journal* v 15, n2, p103-120

Sadler James (1994) *The Development of an Expert System to Aid the Identification of Conflict in a Simultaneous Engineering CAE System.* MSc project submitted September, 1994, Dept of Manufacturing Engineering, Loughborough University of Technology.

Schein Edgar H. (1984) Coming to a new awareness of organizational culture. In *Sloan Management Review*, winter 1984, 3-16.

Schoemaker P J H (1993) Strategic Decisions in Organizations - Rational and Behavioural Views. *Journal of Management Studies* v30 n1 p107-129.

Scott, M G (1994) Concurrent engineering in a global manufacturing context. In proceedings of *Information Systems for Competitive Manufacture*, Dept. of Manufacturing Engineering, Loughborough University of Technology, September 1994.

Sheth A P & Larson J A (1990) Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases *ACM Computing Surveys,* Vol 22, No 3, September 1990.

Stork D & Sapienza A (1992) Task and Human Messages Over the Project Life Cycle: Matching Media to Messages. *Project Management Journal* vol XXII, No:4, pages 44-49.

Sycara K P (1990) Co-operative Negotiation in Concurrent Engineering Design, *Co-operative Engineering Design,* Springer Verlag Publications

Toye G, Cutkosky M R, Leifer L J, Tenenbaum J M, Glicksman J, (?) SHARE: A Methodology and Environment for Collaborative Product Development, *In Post-Proceedings of the IEEE Infrastructure for Collaborative Enterprises (CDR-TR #19930507)*

Vancouver J B & Schmitt N W (1991) An Exploratory Examination of Person-Organization Fit - Organizational Goal Congruence. *Personnel Psychology* V44 N2 p333-352.

Whitney Daniel E, (1990) Designing the Design Process, *Research in Engineering Design*, 1990 2:3-13.

Winner R I, Pennell J P, Bertrand H E, Slusarczuk M M G (1988) *The Role of Concurrent Engineering in Weapons System Acquisition,* IDA Report R-338, Institute for Defence Analyses, 18081 N. Beauregrad Street, Alexandria, Virginia 22311-1772

Wittig T, Jennings N R & Mamdani E H, (1995) ARCHON - A Framework for Intelligent Co-operation. *Queen Mary Westfield Publication*, world wide web page

World Wide Web (1994) URL http://leva.leeds.ac.uk/www-moses/moses.html

Wu J K, Liu T H and Fischer G W, 1992, PDES/STEP-Based Information Model for CAE and CAM Integration, The University of Iowa, *International Journal of Systems Automation: Research and Application*, October, 1992, pp 375-393.

GLOSSARY OF TERMS

The definitions of important words and phrases used in this thesis are listed here, with the section number in which the main or initial usage can be found.

Agent a combination of human and software expertise, interacting with the CAE system. (3.1)

Concurrent Engineering An holistic methodology for the co-ordination of distributed, heterogeneous expertise to achieve cost-effective, market-driven products in minimum time scales. (2)

Data relates simply to words or numbers the meaning of which may vary and is dependent upon the context in which the data is used. (1)

Design for X designing a product from a particular design perspective which has its own design criteria, rules and heuristics to which the product design should conform. For example, design for manufacture. (4.2.3)

Expert System any computer program which demonstrates expert performance in a given domain. (3.1)

Engineering Moderator a specialist manager of coordination program whose role is to drive concurrency within the MOSES system. (4.2.5)

Information is data which is structured or titled in some way so that it has a particular meaning. (1)

Knowledge is information with added detail relating to how it may be used or applied. (1)

MOSES Model Oriented Simultaneous Engineering Systems. The MOSES architecture for future CAE systems is based on the use of two information models which can be accessed by any number of information models via an integration environment. (4.2)

Manufacturing Model A manufacturing model is an information model which contains information of available manufacturing processes, resources and strategies for an organisation. (4.2.2)

Organisational Culture 'The pattern of basic assumptions that a given group has invented, discovered or developed in learning to cope with its problems of external

adaptation and internal integration and that have worked well enough to be considered valid and therefore to be taught to new members as the correct way to perceive, think and feel in relation to those problems.' (2)

Product Model A product model is an information model which contains all the information about a product from its conception to disposal. (4.2.1)

APPENDIX I

Data Definitions for Knowledge Representation Model as Implemented in DecObjectDB

```
// Written by J A Harding, Dept Manufacturing Engineering, LUT.
// MOSES PROJECT - October 1994
//
// MOSES RULES SCHEMA
// rules.ddl
//
// Last changed 1/2/95

#include <working_memory.h>

enum logical {True, False, Unknown};

enum logical_comparator {less_than, less_than_or_equal, greater_than,
greater_than_or_equal, equal, none};

enum logical_operator {AND, OR};

enum proceed_type {cont, stop};

enum value_type {float_val, integer_val};

enum memory_vars {sft, sft_ext, m_sft_in, m_sft_out, sft_ext_in, sft_ext_out,
bear1, bear2};

enum memory_slots {ms_shaft, ms_shaft_ext, ms_bearing_A, ms_bearing_B,
ms_reserve_factor, ms_uts, ms_uss, ms_full_load_power, ms_full_load_speed,
ms_percent_start_torque, ms_calculation};

enum field_names {fn_name, fn_tappered, fn_normal, fn_min_rad, fn_input_sec,
fn_output_sec, fn_input_loc, fn_input_hold, fn_output_loc, fn_output_hold,
fn_cyl_sec_A, fn_cyl_sec_B, fn_groove_A, fn_groove_B, fn_keyway_A, fn_keyway_B};

enum feature_type {cyl_shaft, tap_shaft, trans_null, trans_rad, trans_u_rad,
trans_cham, trans_step, term_rad, term_cham, bl_r_h, flat_b_r_h, slt, kway,
open_kway, grve};

enum creat_type {comp, feat};


// RULE_SET Class
class RULE_SET : public ooObj
  {
```

```
  // number of rule_set_elements currently associated with rule set
  private:
    char                            description[100];
    ooHandle(RULE_SET_ELEMENT)      the_rules[] <->  the_rule_set : prop(delete);
    char                            terminate_ins[100];
    int                             rule_set_number;
    int                             number_of_elements;
    int                             hasfired;
  public:
    RULE_SET();
    RULE_SET(char* thename);
    int fire_first(ooHandle(WORKING_MEMORY) my_memory);
    int fire_all(ooHandle(WORKING_MEMORY) my_memory);
    int fire_first_whilst(ooHandle(WORKING_MEMORY) my_memory,
                                                    ooHandle(CONDITION) conH);
    int fire_all_whilst(ooHandle(WORKING_MEMORY) my_memory,
                                                    ooHandle(CONDITION) conH);
    char* get_termination_instructions();
    int get_rule_set_number();
    void print_description();
    void add_predefined_rule_to_set();
    void add_new_rule_to_set();
  };


// RULE_SET_ELEMENT Class
class RULE_SET_ELEMENT : public ooObj
  {
  private:
    int                             element_number;
    ooHandle(RULE)                  the_rule <-> rule_element : prop(delete);
    ooHandle(RULE_SET)              the_rule_set  <-> the_rules[] : prop(delete);
  public:
    RULE_SET_ELEMENT(char* thename, int number, ooHandle(RULE) aruleH);
    void re_number_element(int number);
    int get_element_number();
    ooHandle(RULE) get_rule();
  };


// RULE Class
class RULE : public ooObj
  {
  private:
    char                            description[100];
    ooHandle(CONDITION)             the_condition <-> the_rule : prop(delete);
    ooHandle(RESULTING_ACTION)      the_result    <-> the_rule : prop(delete);
    ooHandle(RULE_SET_ELEMENT)      rule_element  <-> the_rule : prop(delete);
    char                            completion_ins[100];
    int                             hasfired;
```

```
  public:
    RULE();
    RULE(char* thename);
    int fire(ooHandle(WORKING_MEMORY) my_memory, char* instructions);
    // Returns 1 if action carried out and 0 if rule does not fire
    void print_description();
    void initialise_result_message();
  };


// CONDITION Class
class CONDITION : public ooObj
  {
  protected:
    char                          description[100];
    ooHandle(RULE)                the_rule <-> the_condition : prop(delete);
    logical                       negate;
  public:
    //  NOTE THERE IS NO CONSTRUCTOR HERE AS THIS IS
    //  INTENDED AS AN ABSTRACT SUPERCLASS FOR SIMPLE_CONDITION AND
    //  COMPOUND_CONDITION
    virtual void print();
    virtual int get_condition_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
    void readin_description();
    void print_description();
    void print_negate();
    logical get_negate();
  };


// SIMPLE_CONDITION Class
class SIMPLE_CONDITION : public CONDITION
  {
  private:
    ooHandle(EXPRESSION)          the_element <-> the_simp_con : prop(delete);
  public:
    SIMPLE_CONDITON();
    SIMPLE_CONDITION(char* thename);
    int get_condition_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// EXPRESSION Class
class EXPRESSION : public ooObj
  {
  protected:
    ooHandle(SIMPLE_CONDITION)    the_simp_con <-> the_element : prop(delete);
  public:
```

```
    //  NOTE THERE IS NO CONSTRUCTOR HERE AS THIS IS
    //  INTENDED AS A SUPERCLASS FOR ALL THE DIFFERENT EXPRESSIONS
    virtual int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// COMPOUND_CONDITION Class
class COMPOUND_CONDITION : public CONDITION
  {
  private:
    ooHandle(SIMPLE_CONDITION)          first_element : prop(delete);
    logical_operator                    conjunction;
    ooHandle(CONDITION)                 second_element : prop(delete);
  public:
    COMPOUND_CONDITON();
    COMPOUND_CONDITION(char* thename);
    int get_condition_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };

// RESULTING_ACTION Class
class RESULTING_ACTION : public ooObj
  {
  protected:
    ooHandle(RULE)                      the_rule <-> the_result : prop(delete);
    char                                result_message[100];
  public:
    //  NOTE THERE IS NO CONSTRUCTOR HERE AS THIS IS INTENDED AS
    //  AN ABSTRACT SUPERCLASS FOR SIMPLE_RESULTING_ACTION AND
    //  COMPOUND_RESULTING_ACTION
    virtual char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void initialise_result_message(char* the_message);
  };


// SIMPLE_RESULTING_ACTION Class
class SIMPLE_RESULTING_ACTION : public RESULTING_ACTION
  {
  public:
    //  NOTE THERE IS NO CONSTRUCTOR HERE AS THIS IS INTENDED AS
    //  AN ABSTRACT SUPERCLASS FOR ALL THE ACTUAL CLASSES OF
    //  RESULTING_ACTION
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
  };


// COMPOUND_RESULTING_ACTION Class
class COMPOUND_RESULTING_ACTION : public RESULTING_ACTION
  {
```

```
  private:
    ooHandle(SIMPLE_RESULTING_ACTION)          first_element : prop(delete);
    ooHandle(RESULTING_ACTION)                 second_element : prop(delete);
  public:
    COMPOUND_RESULTING_ACTION();
    COMPOUND_RESULTING_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
  };


// USER_INPUT_RESPONSE_EXPRESSION Class
// The primary function of objects of this class is to write a pre-defined
// question to the screen, and obtain the user's response. The message is
// requested when an object of this class is created, and is held in the
// question attribute. If the user types 'y' or 'Y' in response to the
// question, this expression returns 1 (TRUE), if the user types any other
// character, this expression returns 0 (FALSE).
class  USER_INPUT_RESPONSE_EXPRESSION : public EXPRESSION
  {
  private:
    char                                       question[200];
  public:
    USER_INPUT_RESPONSE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
     // Returns 1 for True, 0 for False, & -1 for Error
  };


// ALWAYS_TRUE_EXPRESSION Class
// The primary function of objects of this class is to ensure that
// its related resulting action is ALWAYS carried. ie This expression
// ALWAYS returns 1 (TRUE).
class  ALWAYS_TRUE_EXPRESSION : public EXPRESSION
  {
  public:
    ALWAYS_TRUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
     // Returns 1 for True, 0 for False, & -1 for Error
  };


// MENU_SELECTION_MADE_EXPRESSION Class
// The primary function of objects of this class is to ensure that a valid
// selection is made from a menu. There can be a maximum of 20 elements in
// the menu, currently. If one of the menu list elements is chosen,
// its number will be entered into the TEMP VALS integer value slot of the
// working memory object, my_memory, and this expression will return 1 (TRUE).
// If the value 0 is entered, ie the select nothing from the menu option,
// this expression will return 0 (FALSE). (If the size of the array, menu_
// menu_elements is increased above 20, the size of the rulenames_list array
```

```
// in class  FIRE_A_SELECTED_RULE_ACTION can also be increased).
class  MENU_SELECTION_MADE_EXPRESSION : public EXPRESSION
  {
  private:
    char                                        menu_header[100];
    char                                        menu_elements[20][100];
    char                                        menu_comment[100];
    int                                         number_of_menu_elements;
  public:
    MENU_SELECTION_MADE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// WI_MENU_SELECTION_MADE_EXPRESSION Class
// The primary function of objects of this class is to ensure that a valid
// selection is made from a pop-up menu. There can be a maximum of 20 elements in
// the menu, currently. If one of the menu list elements is chosen,
// its number will be entered into the TEMP VALS integer value slot of the
// working memory object, my_memory, and this expression will return 1 (TRUE).
// If the value 0 is entered, ie the select nothing from the menu option,
// this expression will return 0 (FALSE). (If the size of the array, menu_
// menu_elements is increased above 20, the size of the rulenames_list array
// in class  FIRE_A_SELECTED_RULE_ACTION can also be increased).
class  WI_MENU_SELECTION_MADE_EXPRESSION : public EXPRESSION
  {
  private:
    char                                        menu_header[100];
    char                                        menu_elements[20][100];
    char                                        menu_comment[100];
    int                                         number_of_menu_elements;
  public:
    WI_MENU_SELECTION_MADE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// COMPONENT_OF_SPECIFIED_TYPE_EXISTS_EXPRESSION Class
// The primary function of objects of this class is to check the product model
// to see if a component of a specified type already exists.
// Used by SDFFE to see if a shaft currently exists. The type of the
// component is specified by the attribute comp_type. If a component of
// the specified type is found in the product model this expression returns 1
// (TRUE). If no component of the specified type exists,
// this expression will return 0 (FALSE).
class  COMPONENT_OF_SPECIFIED_TYPE_EXISTS_EXPRESSION : public EXPRESSION
  {
  private:
```

```
  char                                comp_type[100];
 public:
  COMPONENT_OF_SPECIFIED_TYPE_EXISTS_EXPRESSION(char* thename);
  int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
  // Returns 1 for True, 0 for False, & -1 for Error
 };


// SDFFE_MEMORY_EQUALS_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the SDFFE Working Memory contains a string, integer,
// or float which equals the value specified at creation of this object,
// which is stored in one of the attributes the_string, the_int or the_float.
class SDFFE_MEMORY_EQUALS_SPECIFIED_VALUE_EXPRESSION : public EXPRESSION
 {
 private:
  char                                slot1[20];
  char                                slot2[20];
  char                                slot3[20];
  char                                slot4[20];
  char                                var_is_a[20];
  char                                the_string[100];
  int                                 the_int;
  float                               the_float;
 public:
  SDFFE_MEMORY_EQUALS_SPECIFIED_VALUE_EXPRESSION(char* thename);
  int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
  // Returns 1 for True, 0 for False, & -1 for Error
 };


// SDFFE_MEMORY_LESS_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the SDFFE Working Memory contains a string, integer
// or float which is less than the value specified at creation of this object,
// and which is stored in one of the attributes the_string, the_int or the_float.
class SDFFE_MEMORY_LESS_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
 {
 private:
  char                                slot1[20];
  char                                slot2[20];
  char                                slot3[20];
  char                                slot4[20];
  char                                var_is_a[20];
  char                                the_string[100];
  int                                 the_int;
  float                               the_float;
 public:
  SDFFE_MEMORY_LESS_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
```

```
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// SDFFE_MEMORY_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the SDFFE Working Memory contains a string, integer
// or float which is greater than the value specified at creation of this object,
// and which is stored in the attribute the_string, the_int or the_float.
class SDFFE_MEMORY_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
    char                                    the_string[100];
    int                                     the_int;
    float                                   the_float;
  public:
    SDFFE_MEMORY_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// STRAT_MEMORY_EQUALS_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the STRAT Working Memory contains a string, integer,
// or float which equals the value specified at creation of this object, and
// which is stored in one of the attributes the_string, the_int or the_float.
class STRAT_MEMORY_EQUALS_SPECIFIED_VALUE_EXPRESSION : public EXPRESSION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
    char                                    the_string[100];
    int                                     the_int;
    float                                   the_float;
  public:
    STRAT_MEMORY_EQUALS_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };
```

```
// STRAT_MEMORY_LESS_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the STRAT Working Memory contains a string, integer
// or float which is less than the value specified at creation of this object,
// and which is stored in one of the attributes the_string, the_int or the_float.
class STRAT_MEMORY_LESS_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
    char                                    the_string[100];
    int                                     the_int;
    float                                   the_float;
  public:
    STRAT_MEMORY_LESS_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// STRAT_MEMORY_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the STRAT Working Memory contains a string, integer
// or float which is greater than the value specified at creation of this object,
// and which is stored in the attribute the_string, the_int or the_float.
class STRAT_MEMORY_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
    char                                    the_string[100];
    int                                     the_int;
    float                                   the_float;
  public:
    STRAT_MEMORY_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };
```

```
// PRINT_MESSAGE_ACTION Class
// The primary function of objects of this class is to write a pre-defined
// message to the screen.  The message is requested when an object of this
// class is created, and is held in the message attribute.
class  PRINT_MESSAGE_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                message[200];
  public:
    PRINT_MESSAGE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
  };


// POSTSCRIPT_DISPLAY_ACTION Class
// The primary function of objects of this class is to display a postscript
// file, whose name is stored in the message attribute of objects of this
// class, and which must be stored in the Postscript_files directory.  The
// postscript file is displayed using Pageview, on the workstation specified
// by the value of the machine_name attribute in the working memory object
// (my_memory), which must be passed as a parameter to the execute_action method
// of objects of this class.
class  POSTSCRIPT_DISPLAY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                message[100];   // postscript file name
    char                                machine_name[20];
  public:
    POSTSCRIPT_DISPLAY_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
  };


// UPDATE_SDFFE_MEMORY_ACTION Class
// The primary function of objects of this class is to update a pre_defined
// slot in the SDFFE Working Memory with a value which already exist
// in the TEMP_VALUES object within the SDFFE working memory.
class  UPDATE_SDFFE_MEMORY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                slot1[20];
    char                                slot2[20];
    char                                slot3[20];
    char                                slot4[20];
    char                                var_is_a[20];
  public:
    UPDATE_SDFFE_MEMORY_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };
```

```
// READ_INTO_MEMORY_ACTION Class
// The primary function of objects of this class is to update a slot in the
// TEMP_VALUES object in the Working Memory of an expert with a value provided
// by the user at run time.
class READ_INTO_MEMORY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        prompt_message[200];
    char                                        var_is_a[20];
  public:
    READ_INTO_MEMORY_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// SPEC_VALUE_INTO_MEMORY_ACTION Class
// The primary function of objects of this class is to update a slot in the
// TEMP_VALUES object in the Working Memory of an expert with a value provided
// by the product model in the form of a quality or quantity within the spec.
// The required context string to be searched for among the product requirements
// has to be specified when initialising objects of this class.
class SPEC_VALUE_INTO_MEMORY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        context_string[200];
    char                                        value_is_a[20];
  public:
    SPEC_VALUE_INTO_MEMORY_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// FIRE_A_SELECTED_RULE_ACTION Class
// The primary function of objects of this class is to select a particular rule to fire,
// dependent upon the current value of the integer slot in the TEMP_VALUES
// object. The names of the rules which might possibly be fired are stored
// in the attribute rulename_list. SO currently, objects of this class can
// select from a maximum of 20 possible rules which might be fired. This
// number could be increased if the size of the menu_elements array attribute
// of MENU_SELECTION_MADE_EXPRESSION class is increased. The number of rules
// which can be selected from is stored in the attribute, num_of_rules, and
// this, and the rule names, must be specified when initialising objects of
// this class.
class FIRE_A_SELECTED_RULE_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
```

```
    char                                        rulename_list[20][100];
    int                                         num_of_rules;
  public:
    FIRE_A_SELECTED_RULE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// CREATE_BASIC_FEATURE_ACTION Class
// The primary function of objects of this class is to create a BASIC feature
// of a type specified by the attribute feature_type, in the product model
class CREATE_BASIC_FEATURE_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        feature_type[100];
    char                                        feature_name[100];
    char                                        functional_info[100];
    char                                        slot1[20];
    char                                        slot2[20];
    char                                        slot3[20];
    char                                        slot4[20];
  public:
    CREATE_BASIC_FEATURE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// CREATE_SPECIFIED_COMPONENT_ACTION Class
// The primary function of objects of this class is to create a component
// of a type specified by the attribute comp_type, in the product model
class CREATE_SPECIFIED_COMPONENT_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        comp_type[100];
  public:
    CREATE_SPECIFIED_COMPONENT_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// CREATE_SPECIFIED_DEFINITION_ACTION Class
// The primary function of objects of this class is to create a component
// definition of a type specified by the attribute defn_type in the product model
class CREATE_SPECIFIED_DEFINITION_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        defn_type[100];
```

```
  public:
    CREATE_SPECIFIED_DEFINITION_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// MATERIAL_VALUE_INTO_MEMORY_ACTION Class
// The primary function of objects of this class is to update a slot in the
// TEMP_VALUES object in the Working Memory of an expert with a value provided
// by the product model in the form of a property of a material.
class MATERIAL_VALUE_INTO_MEMORY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                    property_str[200];
    char                                    value_is_a[20];
  public:
    MATERIAL_VALUE_INTO_MEMORY_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// EXECUTE_NEURAL_NETWORK_ACTION Class
// The primary function of objects of this class is to update a slot in the
// TEMP_VALUES object in the Working Memory of an expert with a value obtained
// by running a neural network to select a BEARING.
class EXECUTE_NEURAL_NETWORK_ACTION : public SIMPLE_RESULTING_ACTION
  {
  public:
    EXECUTE_NEURAL_NETWORK_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// UPDATE_STRAT_MEMORY_ACTION Class
// The primary function of objects of this class is to update a pre_defined
// slot in the STRAT Working Memory with a value which already exist
// in the TEMP_VALUES object within the STRAT working memory.
class UPDATE_STRAT_MEMORY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
  public:
    UPDATE_STRAT_MEMORY_ACTION(char* thename);
```

```
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// EXECUTE_NAMED_FUNCTION_ACTION Class
// The primary function of objects of this class is to execute a function
// whose name is stored in the attribute 'name', and which requires a
// number of parametes, as stored in the attribute 'num_params'.
class EXECUTE_NAMED_FUNCTION_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                    name[30];
    int                                     num_params;
  public:
    EXECUTE_NAMED_FUNCTION_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// SPECIFIED_VALUE_INTO_MEMORY_ACTION Class
// The primary function of objects of this class is to update a slot in the
// TEMP_VALUES object in the Working Memory of an expert with a pre-defined value
class SPECIFIED_VALUE_INTO_MEMORY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                    value_is_a[20];
    int                                     the_int;
    float                                   the_float;
    char                                    the_string[100];
  public:
    SPECIFIED_VALUE_INTO_MEMORY_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// ADD_DIMENSION_ACTION Class
// The primary function of objects of this class is to add a dimension
// to a previously created feature. The feature is identified by the
// slot attributes. The dimension to be added is identified by the
// dimension attribute, and the type of dimension (ie nominal, +tol or
// -tol) is identified by the dim_type attribute. The value of the
// dimension is taken from the temp float slot in working memory

class ADD_DIMENSION_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                    slot1[20];
```

```
    char                                        slot2[20];
    char                                        slot3[20];
    char                                        slot4[20];
    char                                        feature_name[100];
    char                                        dimension[20];
    char                                        dim_tol[20];
  public:
    ADD_DIMENSION_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };



// CHANGE_FEATURE_TYPE_ACTION Class
// The primary function of objects of this class is to change a feature of
// one type into a feature of another type.  The feature to be changed is
// identified by the slot attributes.
class CHANGE_FEATURE_TYPE_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        feature_name[100];
    char                                        feature_type[100];
    char                                        slot1[20];
    char                                        slot2[20];
    char                                        slot3[20];
    char                                        slot4[20];
  public:
    CHANGE_FEATURE_TYPE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };



// JOIN_SHAFT_SECTION_ACTION Class
// The primary function of objects of this class is to change a feature of
// one type into a feature of another type.  The feature to be changed is
// identified by the slot attributes.
class JOIN_SHAFT_SECTION_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        feature_name1[100];
    char                                        feature_name2[100];
    char                                        join_type[20];
  public:
    JOIN_SHAFT_SECTION_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };
```

```
// UPDATE_SURFACE_FINISH_ACTION Class
// The primary function of objects of this class is to change the value of the
// surface finish attribute of a feature object which already exists in the product model
class  UPDATE_SURFACE_FINISH_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        feature_name[100];
  public:
    UPDATE_SURFACE_FINISH_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// UPDATE_FUNCTIONAL_INFO_ACTION Class
// The primary function of objects of this class is to change the value of the
// functional information string attribute of a feature object which already exists
// in the product model
class  UPDATE_FUNCTIONAL_INFO_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        feature_name[100];
    char                                        info_string[100];
  public:
    UPDATE_FUNCTIONAL_INFO_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };
```

```
// Written by J A Harding, Dept Manufacturing Engineering, LUT.
// MOSES PROJECT - February 1995
//
// MOSES MODERATOR SPECIFIC RULES SCHEMA
// mod_rules.ddl
// (Moderator specific rules)
//
// Last changed 13/4/95

#include <working_memory.h>
#include <rules.h>


// DESIGN_MOD_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the DESIGN_MOD Working Memory contains a string, integer,
// or float which equals the value specified at creation of this object,
// which is stored in one of the attributes the_string, the_int or the_float.
class DESIGN_MOD_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION : public EXPRESSION
  {
  private:
    char                                slot1[20];
    char                                slot2[20];
    char                                slot3[20];
    char                                slot4[20];
    char                                var_is_a[20];
    char                                the_string[100];
    int                                 the_int;
    float                               the_float;
  public:
    DESIGN_MOD_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// DESIGN_MOD_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the DESIGN MOD Working Memory contains a string, integer
// or float which is less than the value specified at creation of this object,
// and which is stored in one of the attributes the_string, the_int or the_float.
class DESIGN_MOD_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                slot2[20];
    char                                slot3[20];
    char                                slot4[20];
    char                                var_is_a[20];
    char                                the_string[100];
```

```
    int                                         the_int;
    float                                       the_float;
  public:
    DESIGN_MOD_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// DESIGN_MOD_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the DESIGN MOD Working Memory contains a string, integer
// or float which is greater than the value specified at creation of this object,
// and which is stored in the attribute the_string, the_int or the_float.
class DESIGN_MOD_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                        slot1[20];
    char                                        slot2[20];
    char                                        slot3[20];
    char                                        slot4[20];
    char                                        var_is_a[20];
    char                                        the_string[100];
    int                                         the_int;
    float                                       the_float;
  public:
    DESIGN_MOD_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION(char*
                                                                  thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// OBJECT_OF_SPECIFIED_CLASS_EXPRESSION Class
// The primary function of objects of this class is to determine if an object found in the product
// model is of a pre-defined class
class OBJECT_OF_SPECIFIED_CLASS_EXPRESSION : public EXPRESSION
  {
  private:
    char                                        the_kind[20];
  public:
    OBJECT_OF_SPECIFIED_CLASS_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// OBJECT_IS_FUNCTIONAL_EXPRESSION Class
// The primary function of objects of this class is to determine if an object found in the product
```

```
// model has a non-null functional info string attribute value
class OBJECT_IS_FUNCTIONAL_EXPRESSION : public EXPRESSION
  {
  private:
    char                                                slot1[20];
    char                                                slot2[20];
    char                                                slot3[20];
    char                                                slot4[20];
    char                                                var_is_a[20];
    char                                                the_string[100];
    int                                                 the_int;
    float                                               the_float;
  public:
    OBJECT_IS_FUNCTIONAL_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// UPDATE_DESIGN_MOD_W_M_ACTION Class
// The primary function of objects of this class is to update a pre_defined
// slot in the DESIGN MOD Working Memory with a value which already exist
// in the TEMP_VALUES object within the SDFFE working memory.
class  UPDATE_DESIGN_MOD_W_M_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                                slot1[20];
    char                                                slot2[20];
    char                                                slot3[20];
    char                                                slot4[20];
    char                                                var_is_a[20];
  public:
    UPDATE_DESIGN_MOD_W_M_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// GET_OBJ_TYPE_ACTION Class
// The primary function of objects of this class is to update a pre-defined
// slot in the Working Memory with the class of an object
class  GET_OBJ_TYPE_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                                slot1[20];
    char                                                slot2[20];
    char                                                slot3[20];
    char                                                slot4[20];
```

```
  public:
    GET_OBJ_TYPE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// GET_OBJECT_DETAILS_ACTION Class
// The primary function of objects of this class is to update slot in the
// Moderators  Working Memory with a values relating to the changed
// object.
class  GET_OBJECT_DETAILS_ACTION : public SIMPLE_RESULTING_ACTION
  {
  public:
    GET_OBJECT_DETAILS_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// ACTIVATE_OTHER_PROCESS_ACTION Class
// The primary function of objects of this class is to activate another pre-defined
// process, eg to seek advice on significance of a change to an object.
class  ACTIVATE_OTHER_PROCESS_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        path_list[100];
    char                                        message[100];
  public:
    ACTIVATE_OTHER_PROCESS_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// IDENTIFY_MODERATOR_ACTION Class
// The primary function of objects of this class is to determine what action the Engineering
// Moderator should undertake
class  IDENTIFY_MODERATOR_ACTION : public SIMPLE_RESULTING_ACTION
  {
  public:
    IDENTIFY_MODERATOR_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };
```

```
// DESIGN_EX_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the DESIGN EXPERT Working Memory contains a string, integer,
// or float which equals the value specified at creation of this object,
// which is stored in one of the attributes the_string, the_int or the_float.
class DESIGN_EX_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION : public EXPRESSION
  {
  private:
    char                                            slot1[20];
    char                                            slot2[20];
    char                                            slot3[20];
    char                                            slot4[20];
    char                                            var_is_a[20];
    char                                            the_string[100];
    int                                             the_int;
    float                                           the_float;
  public:
    DESIGN_EX_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// DESIGN_EX_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the DESIGN EXPERT Working Memory contains a string, integer
// or float which is less than the value specified at creation of this object,
// and which is stored in one of the attributes the_string, the_int or the_float.
class DESIGN_EX_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                            slot1[20];
    char                                            slot2[20];
    char                                            slot3[20];
    char                                            slot4[20];
    char                                            var_is_a[20];
    char                                            the_string[100];
    int                                             the_int;
    float                                           the_float;
  public:
    DESIGN_EX_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// DESIGN_EX_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the DESIGN EXPERT Working Memory contains a string, integer
```

```
// or float which is greater than the value specified at creation of this object,
// and which is stored in the attribute the_string, the_int or the_float.
class DESIGN_EX_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
    char                                    the_string[100];
    int                                     the_int;
    float                                   the_float;
  public:
    DESIGN_EX_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// DESIGN_EXPERT_EXISTS_EXPRESSION Class
// The primary function of objects of this class is to determine if the Engineering Moderator has
// knowledge of a particular design agent's knowledge
class DESIGN_EXPERT_EXISTS_EXPRESSION : public EXPRESSION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    the_expert_name[100];
  public:
    DESIGN_EXPERT_EXISTS_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };



// ADD_NEW_DESIGN_EXPERT_ACTION Class
// The primary function of objects of this class is to add details of a new
// design agent to the Design Expert Working Memory of the Engineering Moderator
class ADD_NEW_DESIGN_EXPERT_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                    the_expert_name[20];
  public:
    ADD_NEW_DESIGN_EXPERT_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
```

```
    void print(void);
  };



// ADD_DESIGN_EXPERT_KNOWLEDGE_ACTION Class
// The primary function of objects of this class is to add knowledge, in the
// form of Rulesets and Rules for the design agent specified in the Knowledge
// acquisition working memory of the Engineering Moderator.
class ADD_DESIGN_EXPERT_KNOWLEDGE_ACTION : public
SIMPLE_RESULTING_ACTION
  {
  private:
    char                                          the_expert_name[20];
  public:
    ADD_DESIGN_EXPERT_KNOWLEDGE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };



// ADD_KNOWLEDGE_ACTION Class
// The primary function of objects of this class is to add knowledge, in the
// form of Rulesets and Rules into a database specified by the user at runtime.
class ADD_KNOWLEDGE_ACTION : public SIMPLE_RESULTING_ACTION
  {
  public:
    ADD_KNOWLEDGE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };



// UPDATE_DESIGN_EX_W_M_ACTION Class
// The primary function of objects of this class is to update a pre_defined
// slot in the DESIGN EXPERT Working Memory with a value which already exist
// in the TEMP_VALUES object within the DESIGN EXPERT working memory.
class UPDATE_DESIGN_EX_W_M_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                          slot1[20];
    char                                          slot2[20];
    char                                          slot3[20];
    char                                          slot4[20];
    char                                          var_is_a[20];
  public:
    UPDATE_DESIGN_EX_W_M_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
```

```
    void print(void);
  };


// KNOW_ACQ_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the KNOWLEDGE ACQ  Working Memory contains a string, integer,
// or float which equals the value specified at creation of this object,
// which is stored in one of the attributes the_string, the_int or the_float.
class KNOW_ACQ_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION : public EXPRESSION
  {
  private:
    char                slot1[20];
    char                slot2[20];
    char                slot3[20];
    char                slot4[20];
    char                var_is_a[20];
    char                the_string[100];
    int                 the_int;
    float               the_float;
  public:
    KNOW_ACQ_W_M_EQUALS_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// KNOW_ACQ_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the KNOWLEDGE ACQ Working Memory contains a string, integer
// or float which is less than the value specified at creation of this object,
// and which is stored in one of the attributes the_string, the_int or the_float.
class KNOW_ACQ_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                slot1[20];
    char                slot2[20];
    char                slot3[20];
    char                slot4[20];
    char                var_is_a[20];
    char                the_string[100];
    int                 the_int;
    float               the_float;
  public:
    KNOW_ACQ_W_M_LESS_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };
```

```
// KNOW_ACQ_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// pre_defined slot in the KNOWLEDGE ACQ Working Memory contains a string, integer
// or float which is less than the value specified at creation of this object,
// and which is stored in the attribute the_string, the_int or the_float.
class KNOW_ACQ_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION : public
EXPRESSION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
    char                                    the_string[100];
    int                                     the_int;
    float                                   the_float;
  public:
    KNOW_ACQ_W_M_GREATER_THAN_SPECIFIED_VALUE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// UPDATE_KNOW_ACQ_W_M_ACTION Class
// The primary function of objects of this class is to update a pre_defined
// slot in the KNOWLEDGE ACQ Working Memory with a value which already exist
// in the TEMP_VALUES object within the KNOWLEDGE ACQ working memory.
class  UPDATE_KNOW_ACQ_W_M_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                    slot1[20];
    char                                    slot2[20];
    char                                    slot3[20];
    char                                    slot4[20];
    char                                    var_is_a[20];
  public:
    UPDATE_KNOW_ACQ_W_M_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// WI_USER_INPUT_RESPONSE_EXPRESSION Class
// The primary function of objects of this class is to write a pre-defined
// question to the screen (in a dialogure box), and obtain the user's response. The
// message is requested when an object of this class is created, and is held in the
// question attribute. If the user selects Yes in response to the question, this expression returns
```

```
// 1 (TRUE), if the user selects NO this expression returns 0 (FALSE).
class  WI_USER_INPUT_RESPONSE_EXPRESSION : public EXPRESSION
  {
  private:
    char                                        question[200];
  public:
    WI_USER_INPUT_RESPONSE_EXPRESSION(char* thename);
    int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
    // Returns 1 for True, 0 for False, & -1 for Error
  };


// WI_PRINT_MESSAGE_ACTION Class
// The primary function of objects of this class is to write a pre-defined
// message to the screen, in a message box. The message is requested when an object of this
// class is created, and is held in the message attribute.
class  WI_PRINT_MESSAGE_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        message[200];
  public:
    WI_PRINT_MESSAGE_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
  };


// WI_READ_INTO_MEMORY_ACTION Class
// The primary function of objects of this class is to update a slot in the
// TEMP_VALUES object in the Working Memory of an expert with a value provided
// by the user at run time, via a dialogue box
class  WI_READ_INTO_MEMORY_ACTION : public SIMPLE_RESULTING_ACTION
  {
  private:
    char                                        prompt_message[200];
    char                                        var_is_a[20];
  public:
    WI_READ_INTO_MEMORY_ACTION(char* thename);
    char* execute_action(ooHandle(WORKING_MEMORY) my_memory);
    void print(void);
  };


// CD_REQ_CHANGE_EXPRESSION Class
// The primary function of objects of this class is to determine if a
// change made to a cost_dely_product_model requirements object is
// significant. This is a single implementation done specifically for
// RRIPG Case Study demo.
class CD_REQ_CHANGE_EXPRESSION : public EXPRESSION
  {
  public:
```

```
  CD_REQ_CHANGE_EXPRESSION(char* thename);
  int get_expression_value(ooHandle(WORKING_MEMORY) my_memory);
  // Returns 1 for True, 0 for False, & -1 for Error
};
```

```
// Written by J A Harding, Dept Manufacturing Engineering, LUT.
// MOSES PROJECT - October 1994
//
// MOSES WORKING MEMORY SCHEMA
// working_memory.ddl
//
// Last changed 9/2/95

// SDFF_SPEC_VALUES Class
class SDFF_SPEC_VALUES : public ooObj
  {
  protected:
    char                                   DB_Name[100]; // Database containing Spec
    float                                  full_load_power;
    float                                  full_load_speed;
    float                                  percent_start_torque;
    char                                   duty[100]; // eg high, medium or low
    float                                  no_of_revolutions;
    float                                  life_expectancy;
    float                                  length_z;
    char                                   torque[100];
    char                                   axial[100];
    ooHandle(SDFF_WORKING_MEMORY)          wm5 <-> spec_requirements : prop(delete);
  public:
    SDFF_SPEC_VALUES();
    void print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot);
    float get_float(char* slot);
  };


// SDFF_MATERIAL_VALUES Class
class SDFF_MATERIAL_VALUES : public ooObj
  {
  protected:
    char                                   DB_Name[100]; // Database of materials
    char                                   mat_name[100];
    char                                   mat_grade[100];
    float                                  UTS;
    float                                  USS;
    float                                  youngs_mod;
    ooHandle(SDFF_WORKING_MEMORY)          wm6 <-> material_details : prop(delete);
  public:
    SDFF_MATERIAL_VALUES();
    void print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot);
```

```
    float get_float(char* slot);
  };


// SDFF_FEATURE Class
class SDFF_FEATURE : public ooObj
  {
  protected:
    char                                      feature_type[100];
    char                                      feature_name[100];
    ooHandle(SDFF_BEARING_JOURNAL)            b1_journal <-> axial_locate : prop(delete);
    ooHandle(SDFF_BEARING_JOURNAL)            b2_journal <-> radial_locate : prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)         shaft_req1 <-> axial_locate : prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)         shaft_req2 <-> radial_locate : prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)         shaft_req3 <-> torque_transmit :prop(delete);
  public:
    SDFF_FEATURE();
    void print();
    void update_string(char* slot, char* update_val);
    char* get_string(char* slot);
  };


// SDFF_BEARING_JOURNAL Class
class SDFF_BEARING_JOURNAL : public ooObj
  {
  protected:
    ooHandle(SDFF_FEATURE)                    axial_locate <-> b1_journal : prop(delete);
    ooHandle(SDFF_FEATURE)                    radial_locate <-> b2_journal : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)             wm1 <-> bearing_A : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)             wm2 <-> bearing_B : prop(delete);
    float                                     min_rad;
    char                                      bearing_chosen[100];
  public:
    SDFF_BEARING_JOURNAL();
    void print();
    void update_string(char* slot, char* slot_attrib, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot, char* slot_attrib);
    float get_float(char* slot);
  };


// SDFF_SHAFT_REQUIREMENTS Class
class SDFF_SHAFT_REQUIREMENTS : public ooObj
  {
  protected:
    ooHandle(SDFF_FEATURE)                    axial_locate <-> shaft_req1 : prop(delete);
    ooHandle(SDFF_FEATURE)                    radial_locate <-> shaft_req2 : prop(delete);
    ooHandle(SDFF_FEATURE)                    torque_transmit <-> shaft_req3 :prop(delete);
```

```
    ooHandle(SDFF_SHAFT_SECTION)            shaft1 <-> input : prop(delete);
    ooHandle(SDFF_SHAFT_SECTION)            shaft2 <-> output : prop(delete);
  public:
    SDFF_SHAFT_REQUIREMENTS();
    void print();
    void update_string(char* slot, char* slot_attrib, char* update_val);
    char* get_string(char* slot, char* slot_attrib);
  };


// SDFF_SHAFT_SECTION Class
class SDFF_SHAFT_SECTION : public ooObj
  {
  protected:
    ooHandle(SDFF_SHAFT_REQUIREMENTS)       input <-> shaft1 : prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)       output <-> shaft2 : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)           wm3 <-> shaft_extension : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)           wm4 <-> main_shaft : prop(delete);
    float                                   min_rad;
  public:
    SDFF_SHAFT_SECTION();
    void print();
    void update_string(char* slot, char* slot_attrib, char* slot_attrib_slot, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot, char* slot_attrib, char* slot_attrib_slot);
    float get_float(char* slot);
  };


// TEMP_VALUES Class
class TEMP_VALUES : public ooObj
  {
  protected:
    char                                    astring[100];
    float                                   afloat;
    int                                     anint;
    ooHandle(SDFF_WORKING_MEMORY)           wm7 <-> temp_vals : prop(delete);
    ooHandle(STRAT_WORKING_MEMORY)          wms7 <-> stemp_vals : prop(delete);
    ooHandle(DESIGN_MODERATION_W_M)         wmm1 <-> temp_vals : prop(delete);
    ooHandle(DESIGN_EXPERT_W_M)             wmm2 <-> temp_vals : prop(delete);
    ooHandle(KNOWLEDGE_ACQU_W_M)            wmm3 <-> temp_vals : prop(delete);
  public:
    TEMP_VALUES();
    void print();
    void win_print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    void update_int(char* slot, int update_val);
    char* get_string(char* slot);
    float get_float(char* slot);
```

```
    int get_int(char* slot);
  };


// WORKING_MEMORY Class
class  WORKING_MEMORY : public ooObj
  {
  public:
//   There is no CONSTRUCTOR for this class as it is an abstract super class and therefore should
//   never be created
    virtual void print();
    virtual void update_string(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                        char* slot_attrib_slot_attrib, char* update_val);
    virtual void update_float(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                        char* slot_attrib_slot_attrib,  float update_val);
    virtual void update_int(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                        char* slot_attrib_slot_attrib, int update_val);
    virtual char* get_string(char* slot, char* slot_attrib,  char* slot_attrib_slot,
                                                  char* slot_attrib_slot_attrib);
    virtual float get_float(char* slot, char* slot_attrib,  char* slot_attrib_slot,
                                                  char* slot_attrib_slot_attrib);
    virtual int get_int(char* slot, char* slot_attrib,  char* slot_attrib_slot,
                                                  char* slot_attrib_slot_attrib);

  };


// SDFF_WORKING_MEMORY Class
class  SDFF_WORKING_MEMORY : public WORKING_MEMORY
  {
  protected:
    char                                  DB_Name[100]; // Database for output
    char                                  Shaft_Name[100]; // Name of current shaft design
    char                                  Comp_Name[100]; // Name of shaft in PM
    ooHandle(SDFF_BEARING_JOURNAL)        bearing_A <-> wm1 : prop(delete);
    ooHandle(SDFF_BEARING_JOURNAL)        bearing_B <-> wm2 : prop(delete);
    ooHandle(SDFF_SHAFT_SECTION)          shaft_extension <-> wm3 : prop(delete);
    ooHandle(SDFF_SHAFT_SECTION)          main_shaft <-> wm4 : prop(delete);
    ooHandle(SDFF_SPEC_VALUES)            spec_requirements <-> wm5 :prop(delete);
    ooHandle(SDFF_MATERIAL_VALUES)        material_details <-> wm6 : prop(delete);
    ooHandle(TEMP_VALUES)                 temp_vals <-> wm7 : prop(delete);
    char                                  machine_Name[100];  // Machine Displays on
    float                                 dynamic_forces;
    float                                 min_rad;
  public:
    SDFF_WORKING_MEMORY();
    SDFF_WORKING_MEMORY(char* thename);
    void print();
    void update_string(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                        char* slot_attrib_slot_attrib,  char* update_val);
    void update_float(char* slot, char* slot_attrib,  char* slot_attrib_slot, char* slot_attrib_slot_attrib,
```

```
                                                                    float update_val);
    void update_int(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib,
                                                                    int update_val);
    char* get_string(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
    float get_float(char* slot, char* slot_attrib,  char* slot_attrib_slot, char* slot_attrib_slot_attrib);
    int get_int(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
  };


// STRAT_SPEC_VALUES Class
class STRAT_SPEC_VALUES : public ooObj
  {
  protected:
    char                                    DB_Name[100]; // Database containing Spec
    int                                     number_req;
    int                                     date_due;
    int                                     init_bat_size;
    int                                     subs_bat_size;
    int                                     init_lead_time;
    int                                     subs_lead_time;
    float                                   depth_x;
    float                                   height_y;
    float                                   length_z;
    int                                     pmat_code;
    int                                     pform_code;
    ooHandle(STRAT_WORKING_MEMORY)          wms5 <> sspec_requirements : prop(delete);
  public:
    STRAT_SPEC_VALUES();
    void print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    void update_int(char* slot, int update_val);
    char* get_string(char* slot);
    float get_float(char* slot);
    int get_int(char* slot);
  };


// STRAT_WORKING_MEMORY Class
class STRAT_WORKING_MEMORY : public WORKING_MEMORY
  {
  protected:
    char                                    DB_Name[100]; // Database for output
    ooHandle(STRAT_SPEC_VALUES)             sspec_requirements <-> wms5 :prop(delete);
    char                                    machine_Name[100];
    ooHandle(TEMP_VALUES)                   stemp_vals <-> wms7 : prop(delete);
  public:
```

```
  STRAT_WORKING_MEMORY();
  STRAT_WORKING_MEMORY(char* thename);
  void print();
  void update_string(char* slot, char* slot_attrib,  char* slot_attrib_slot,
                                        char* slot_attrib_slot_attrib,  char* update_val);
  void update_float(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib,
                                                                                    float update_val);
  void update_int(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib,
                                                                                      int update_val);
  char* get_string(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
  float get_float(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
  int get_int(char* slot, char* slot_attrib, char* slot_attrib_slot,  char* slot_attrib_slot_attrib);
};
```

```
// Written by J A Harding, Dept Manufacturing Engineering, LUT.
// MOSES PROJECT - October 1994
//
// MOSES WORKING MEMORY SCHEMA
// working_memory.ddl
//
// Last changed 9/2/95

// SDFF_SPEC_VALUES Class
class SDFF_SPEC_VALUES : public ooObj
  {
  protected:
    char                                    DB_Name[100]; // Database containing Spec
    float                                   full_load_power;
    float                                   full_load_speed;
    float                                   percent_start_torque;
    char                                    duty[100]; // eg high, medium or low
    float                                   no_of_revolutions;
    float                                   life_expectancy;
    float                                   length_z;
    char                                    torque[100];
    char                                    axial[100];
    ooHandle(SDFF_WORKING_MEMORY)           wm5 <-> spec_requirements : prop(delete);
  public:
    SDFF_SPEC_VALUES();
    void print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot);
    float get_float(char* slot);
  };


// SDFF_MATERIAL_VALUES Class
class SDFF_MATERIAL_VALUES : public ooObj
  {
  protected:
    char                                    DB_Name[100]; // Database of materials
    char                                    mat_name[100];
    char                                    mat_grade[100];
    float                                   UTS;
    float                                   USS;
    float                                   youngs_mod;
    ooHandle(SDFF_WORKING_MEMORY)           wm6 <-> material_details : prop(delete);
  public:
    SDFF_MATERIAL_VALUES();
    void print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot);
```

```
    float get_float(char* slot);
  };


// SDFF_FEATURE Class
class SDFF_FEATURE : public ooObj
  {
  protected:
    char                                        feature_type[100];
    char                                        feature_name[100];
    ooHandle(SDFF_BEARING_JOURNAL)              b1_journal <-> axial_locate : prop(delete);
    ooHandle(SDFF_BEARING_JOURNAL)              b2_journal <-> radial_locate :  prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)           shaft_req1 <-> axial_locate : prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)           shaft_req2 <-> radial_locate : prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)           shaft_req3 <-> torque_transmit: prop(delete);
  public:
    SDFF_FEATURE();
    void print();
    void update_string(char* slot, char* update_val);
    char* get_string(char* slot);
  };


// SDFF_BEARING_JOURNAL Class
class SDFF_BEARING_JOURNAL : public ooObj
  {
  protected:
    ooHandle(SDFF_FEATURE)                      axial_locate <-> b1_journal : prop(delete);
    ooHandle(SDFF_FEATURE)                      radial_locate <-> b2_journal : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)               wm1 <-> bearing_A : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)               wm2 <-> bearing_B : prop(delete);
    float                                       min_rad;
    char                                        bearing_chosen[100];
  public:
    SDFF_BEARING_JOURNAL();
    void print();
    void update_string(char* slot, char* slot_attrib, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot, char* slot_attrib);
    float get_float(char* slot);
  };


// SDFF_SHAFT_REQUIREMENTS Class
class SDFF_SHAFT_REQUIREMENTS : public ooObj
  {
  protected:
    ooHandle(SDFF_FEATURE)                      axial_locate <-> shaft_req1 : prop(delete);
    ooHandle(SDFF_FEATURE)                      radial_locate <-> shaft_req2 : prop(delete);
    ooHandle(SDFF_FEATURE)                      torque_transmit <-> shaft_req3 :prop(delete);
```

```
    ooHandle(SDFF_SHAFT_SECTION)              shaft1 <-> input : prop(delete);
    ooHandle(SDFF_SHAFT_SECTION)              shaft2 <-> output : prop(delete);
  public:
    SDFF_SHAFT_REQUIREMENTS();
    void print();
    void update_string(char* slot, char* slot_attrib, char* update_val);
    char* get_string(char* slot, char* slot_attrib);
  };


// SDFF_SHAFT_SECTION Class
class SDFF_SHAFT_SECTION : public ooObj
  {
  protected:
    ooHandle(SDFF_SHAFT_REQUIREMENTS)         input <-> shaft1 : prop(delete);
    ooHandle(SDFF_SHAFT_REQUIREMENTS)         output <-> shaft2 : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)             wm3 <-> shaft_extension : prop(delete);
    ooHandle(SDFF_WORKING_MEMORY)             wm4 <-> main_shaft : prop(delete);
    float                                     min_rad;
  public:
    SDFF_SHAFT_SECTION();
    void print();
    void update_string(char* slot, char* slot_attrib, char* slot_attrib_slot, char* update_val);
    void update_float(char* slot, float update_val);
    char* get_string(char* slot, char* slot_attrib, char* slot_attrib_slot);
    float get_float(char* slot);
  };


// TEMP_VALUES Class
class TEMP_VALUES : public ooObj
  {
  protected:
    char                                      astring[100];
    float                                     afloat;
    int                                       anint;
    ooHandle(SDFF_WORKING_MEMORY)             wm7 <-> temp_vals : prop(delete);
    ooHandle(STRAT_WORKING_MEMORY)            wms7 <-> stemp_vals : prop(delete);
    ooHandle(DESIGN_MODERATION_W_M)           wmm1 <-> temp_vals : prop(delete);
    ooHandle(DESIGN_EXPERT_W_M)               wmm2 <-> temp_vals : prop(delete);
    ooHandle(KNOWLEDGE_ACQU_W_M)              wmm3 <-> temp_vals : prop(delete);
  public:
    TEMP_VALUES();
    void print();
    void win_print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    void update_int(char* slot, int update_val);
    char* get_string(char* slot);
    float get_float(char* slot);
```

```
    int get_int(char* slot);
  };


// WORKING_MEMORY Class
class  WORKING_MEMORY : public ooObj
  {
  public:
//   There is no CONSTRUCTOR for this class as it is an abstract super class
//   and therefore should never be created
     virtual void print();
     virtual void update_string(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                              char* slot_attrib_slot_attrib, char* update_val);
     virtual void update_float(char* slot, char* slot_attrib,  char* slot_attrib_slot,
                                              char* slot_attrib_slot_attrib,  float update_val);
     virtual void update_int(char* slot, char* slot_attrib,  char* slot_attrib_slot,
                                                char* slot_attrib_slot_attrib, int  update_val);
     virtual char* get_string(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                                         char* slot_attrib_slot_attrib);
     virtual float get_float(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                                         char* slot_attrib_slot_attrib);
     virtual int get_int(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                                         char* slot_attrib_slot_attrib);
  };


// SDFF_WORKING_MEMORY Class
class  SDFF_WORKING_MEMORY : public WORKING_MEMORY
  {
  protected:
    char                                    DB_Name[100]; // Database for output
    char                                    Shaft_Name[100]; // Name of current shaft design
    char                                    Comp_Name[100]; // Name of shaft in PM
    ooHandle(SDFF_BEARING_JOURNAL)          bearing_A <-> wm1 : prop(delete);
    ooHandle(SDFF_BEARING_JOURNAL)          bearing_B <-> wm2 : prop(delete);
    ooHandle(SDFF_SHAFT_SECTION)            shaft_extension <-> wm3 : prop(delete);
    ooHandle(SDFF_SHAFT_SECTION)            main_shaft <-> wm4 : prop(delete);
    ooHandle(SDFF_SPEC_VALUES)              spec_requirements <-> wm5 :prop(delete);
    ooHandle(SDFF_MATERIAL_VALUES)          material_details <-> wm6 : prop(delete);
    ooHandle(TEMP_VALUES)                   temp_vals <-> wm7 : prop(delete);
    char                                    machine_Name[100];
    float                                   dynamic_forces;
    float                                   min_rad;

  public:
    SDFF_WORKING_MEMORY();
    SDFF_WORKING_MEMORY(char* thename);
    void print();
    void update_string(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                            char* slot_attrib_slot_attrib,  char* update_val);
```

```
    void update_float(char* slot, char* slot_attrib,  char* slot_attrib_slot, char* slot_attrib_slot_attrib,
                                                                                    float update_val);
    void update_int(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib,
                                                                                      int update_val);
    char* get_string(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
    float get_float(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
    int get_int(char* slot, char* slot_attrib, char* slot_attrib_slot,  char* slot_attrib_slot_attrib);
  };


// STRAT_SPEC_VALUES Class
class  STRAT_SPEC_VALUES : public ooObj
  {
  protected:
    char                                        DB_Name[100]; // Database containing Spec
    int                                         number_req;
    int                                         date_due;
    int                                         init_bat_size;
    int                                         subs_bat_size;
    int                                         init_lead_time;
    int                                         subs_lead_time;
    float                                       depth_x;
    float                                       height_y;
    float                                       length_z;
    int                                         pmat_code;
    int                                         pform_code;
    ooHandle(STRAT_WORKING_MEMORY)              wms5 <-> sspec_requirements : prop(delete);
  public:
    STRAT_SPEC_VALUES();
    void print();
    void update_string(char* slot, char* update_val);
    void update_float(char* slot, float update_val);
    void update_int(char* slot, int update_val);
    char* get_string(char* slot);
    float get_float(char* slot);
    int get_int(char* slot);
  };


// STRAT_WORKING_MEMORY Class
class  STRAT_WORKING_MEMORY : public WORKING_MEMORY
  {
  protected:
    char                                        DB_Name[100]; // Database for output
    ooHandle(STRAT_SPEC_VALUES)                 sspec_requirnts <-> wms5 :prop(delete);
    char                                        machine_Name[100];
    ooHandle(TEMP_VALUES)                       stemp_vals <-> wms7 : prop(delete);
```

```
public:
  STRAT_WORKING_MEMORY();
  STRAT_WORKING_MEMORY(char* thename);
  void print();
  void update_string(char* slot, char* slot_attrib, char* slot_attrib_slot,
                                              char* slot_attrib_slot_attrib, char* update_val);
  void update_float(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib,
                                                                                      float update_val);
  void update_int(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib,
                                                                                        int update_val);
  char* get_string(char* slot, char* slot_attrib,  char* slot_attrib_slot, char* slot_attrib_slot_attrib);
  float get_float(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
  int get_int(char* slot, char* slot_attrib, char* slot_attrib_slot, char* slot_attrib_slot_attrib);
};
```

APPENDIX II

Tables showing implemented Sub-Classes of Expression and Simple Resulting Action

TABLE 1

Implemented Sub-Classes of Expression	
USER INPUT RESPONSE EXPRESSION	Objects of this class display a pre-defined question on the screen and obtain the user's response. The message is defined when an object of this class is created. If the user answers y (yes) to the question this expression returns 1 (True) if the user gives any other response, this expression returns 0 (False)
WI USER INPUT RESPONSE EXPRESSION	Objects of this class display a pre-defined question on the screen, using a dialogue box and obtain the user's response. The message is defined when an object of this class is created. If the user answers y (yes) to the question this expression returns 1 (True) if the user gives any other response, this expression returns 0 (False)
ALWAYS TRUE EXPRESSION	Objects of this class always return 1(True). They are used to ensure that a particular resulting action is always carried out.
MENU SELECTION MADE EXPRESSION	Objects of this class display a pre-defined menu. The menu items are defined when an object of this class is created. If the user enters an integer representing a valid menu selection, the expression returns 1 (True) if the user enters any other value, this expression returns 0 (False).
WI MENU SELECTION MADE EXPRESSION	Objects of this class display a pre-defined pop-up menu. The menu items are defined when an object of this class is created. If the user makes a valid selection from the menu, this expression returns 1 (True) otherwise this expression returns 0 (False)
COMPONENT OF SPECIFIED TYPE EXISTS	Objects of this class check the current product model to see if a component of a pre-defined typed already exists. The component type is defined when an object of this class is created. If such a component exists, this expression returns 1 (True) otherwise this expression returns 0 (False)

SDFFE MEMORY EQUALS SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the SDFFE's working memory equals a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is equal to the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
SDFFE MEMORY LESS THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the SDFFE's working memory is less than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is less than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
SDFFE MEMORY GREATER THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the SDFFE's working memory is greater than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is greater than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
STRAT MEMORY EQUALS SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Manufacturing Strategist's working memory equals a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is equal to the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
STRAT MEMORY LESS THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Manufacturing Strategist's working memory is less than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is less than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
STRAT MEMORY GREATER THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Manufacturing Strategist's working memory is greater than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is greater than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)

DESIGN MODERATION MEMORY EQUALS SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Design Moderation working memory equals a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is equal to the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
DESIGN MODERATION MEMORY LESS THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Design Moderation working memory is less than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is less than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
DESIGN MODERATION MEMORY GREATER THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Design Moderation working memory is greater than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is greater than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
OBJECT OF SPECIFIED CLASS	Objects of this class determine if an object within the product model is of a specified class. If it is of the specified class, this expression returns 1 (True) otherwise this expression returns 0 (False)
OBJECT IS FUNCTIONAL	Objects of this class determine if an object within the product model has a non-null value in its functional info string attribute. If it has a non-null value, this expression returns 1 (True) otherwise this expression returns 0 (False)
DESIGN EXPERT MEMORY EQUALS SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Design Expert working memory equals a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is equal to the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
DESIGN EXPERT MEMORY LESS THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Design Expert working memory is less than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is less than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)

DESIGN EXPERT MEMORY GREATER THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Design Expert working memory is greater than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is greater than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
DESIGN EXPERT EXISTS	Objects of this class check if the Engineering Moderator has knowledge of the existence of a particular design agent. This expression returns 1 (True) otherwise this expression returns 0 (False)
KNOWLEDGE ACQUISITION MEMORY EQUALS SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Knowledge Acquisition working memory equals a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is equal to the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
KNOWLEDGE ACQUISITION MEMORY LESS THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Knowledge Acquisition working memory is less than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is less than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
KNOWLEDGE ACQUISITION MEMORY GREATER THAN SPECIFIED VALUE	Objects of this class check if a pre-defined attribute in the Knowledge Acquisition working memory is greater than a pre-defined value. The attribute and the value are both defined when an object of this class is created. If the attribute's value is greater than the pre-defined value, this expression returns 1 (True) otherwise this expression returns 0 (False)
CD REQ CHANGE EXPRESSION	Objects of this class determine if a change made to a cost and delivery product model requirements object is significant. Single implementation specifically done for the RRIPG Case Study Demonstration. If the change is significant this returns 1 (True), otherwise this expression returns 0 (False)

TABLE 2

Implemented Sub-Classes of Simple Resulting Action	
PRINT MESSAGE ACTION	Objects of this class display a pre-defined message on the screen. The message is defined when an object of this class is created.
WI PRINT MESSAGE ACTION	Objects of this class display a pre-defined message in a message box on the screen. The message is defined when an object of this class is created.
POSTSCRIPT DISPLAY ACTION	Objects of this class display a postscript file, whose name is stored in the message attribute of objects of this class.
UPDATE SDFFE MEMORY ACTION	Objects of this class update a pre-defined attribute in the SDFFE Working Memory with the value which exists in the Temp attribute of SDFFE Working Memory. The attribute to be updated is defined when this expression is created.
UPDATE STRAT MEMORY ACTION	Objects of this class update a pre-defined attribute in the STRAT Working Memory with the value which exists in the Temp attribute of STRAT Working Memory. The attribute to be updated is defined when this expression is created.
UPDATE DESIGN MODERATION MEMORY ACTION	Objects of this class update a pre-defined attribute in the DESIGN MODERATION Working Memory with the value which exists in the Temp attribute of DESIGN MODERATION Working Memory. The attribute to be updated is defined when this expression is created.
UPDATE DESIGN EXPERT MEMORY ACTION	Objects of this class update a pre-defined attribute in the DESIGN EXPERT Working Memory with the value which exists in the Temp attribute of DESIGN EXPERT Working Memory. The attribute to be updated is defined when this expression is created.
UPDATE KNOWLEDGE ACQUISITION MEMORY ACTION	Objects of this class update a pre-defined attribute in the KNOWLEDGE ACQUISITION Working Memory with the value which exists in the Temp attribute of KNOWLEDGE ACQUISITION Working Memory. The attribute to be updated is defined when this expression is created.
WI READ INTO MEMORY ACTION	Objects of this class update the Temp attribute of a working memory object (or any sub-class). It is updated with a value typed in by the user, using a dialogue box, at runtime.

READ INTO MEMORY ACTION	Objects of this class update the Temp attribute of a working memory object (or any sub-class). It is updated with a value typed in by the user at runtime.
SPEC VALUE INTO MEMORY ACTION	Objects of this class update the Temp attribute of a working memory object (or any sub-class), with a value extracted from the product model. The Temp attribute is updated by a value from the specification section of the product model. The value will be a quality or quantity related to a product requirement. Details of the product requirement to be used are determined when this object is created.
FIRE A SELECTED RULE ACTION	Objects of this class fire one of several pre-defined rules, the one actually fired at run-time is chosen dependent on the current contents of working memory.
CREATE BASIC FEATURE ACTION	Objects of this class can create a feature of a pre-defined type in the product model. The type of the feature created is determined when this action is created.
CREATE SPECIFIED COMPONENT ACTION	Objects of this class can create a component of a pre-defined type in the product model. The type of the component created is determined when this action is created.
CREATE SPECIFIED DEFINITION ACTION	Objects of this class can create a component definition in the product model
MATERIAL VALUE INTO MEMORY ACTION	Objects of this class update the Temp attribute of a working memory object (or any sub-class), with a value extracted from the product model. The Temp attribute is updated by a value from the materials section of the product model. The material attribute from which the value is to be extracted is determined when this action is created
EXECUTE NEURAL NETWORK ACTION	Objects of this class update the Temp attribute of a working memory object (or any sub-class), with a value calculated by running a neural network simulation to select a bearing
SPECIFIED VALUE INTO MEMORY ACTION	Objects of this class update the Temp attribute of a working memory object (or any sub-class), with a pre-defined value, which is set when this action is created
ADD DIMENSION ACTION	Objects of this class update a dimension of a feature object in the product model with the value currently in the Temp attribute of a working memory object (or any sub-class).
CHANGE FEATURE TYPE ACTION	Objects of this class can change a feature object in the product model to a feature of a different type

JOIN SHAFT SECTION ACTION	Objects of this class can join two pre-existing feature objects in the product model
UPDATE SURFACE FINISH ACTION	Objects of this class can change the value of the surface finish attribute of a pre-existing feature object within the product model
UPDATE FUNCTIONAL INFORMATION ACTION	Objects of this class can change the value of the functional information string attribute of a pre-existing feature object within the product model
GET OBJECT TYPE ACTION	Objects of this class find the class of an object which exists within the product model
GET OBJECT DETAILS ACTION	Objects of this class extract details of an object which exists within the product model
ACTIVATE OTHER PROCESS ACTION	Objects of this class activate another pre-defined process, for example this could be used by the EM to request advice from the Manufacturing Strategist.
IDENTIFY MODERATOR ACTION	Objects of this class may be used to determine the course of action to be followed by the EM
ADD NEW DESIGN EXPERT ACTION	Objects of this class enable the EM knowledge of existing design agents to be updated by adding a new agent to the known system
ADD NEW DESIGN EXPERT KNOWLEDGE ACTION	Objects of this class enable the EM knowledge of existing design agents to be updated by adding new knowledge of a design agent
ADD KNOWLEDGE ACTION	Objects of this class can be used to add knowledge in the form of ruleset and rule objects to a knowledge base which is specified by the user at run-time.