TOWARDS THE APPLICATION OF ARTIFICIAL INTELLIGENCE TECHNIQUES FOR DISCRETE EVENT SIMULATION

by

ANDREW FLITMAN

B.Sc. (York)

A thesis submitted in partial fulfilment

of the requirements for the degree of

"Doctor of Philosophy"

UNIVERSITY OF WARWICK

School of Industrial and Business Studies

September 1986

CONTENTS

CHAPTER THREE

CHAPTER FOUR

CHAPTER SEVEN

CHAPTER EIGHT

APPENDICES

TABLES

FIGURES

ACKNOWLEDGEMENTS

Several individuals have contributed to the success of this research, many of whom cannot be named in this acknowledgement. First thanks must go to my supervisor Dr. Robert Hurrion for all the help and advice he gave me throughout the period of this research.

Of the other people that have helped during the course of this research I specifically wish to thank the following. Margeret Keeton, formerly of Ace International, for providing me with relevant press release material; Chris Barton, for assisting in the early stages of the expert system development; and Dr. M. Larcombe, of the Warwick Computer Science Department, for providing useful information on Robotic Systems.

I also gratefully acknowledge the support of the SERC who funded this research.

To all my friends who have offered encouragement during this research I proffer my thanks. In particular to Ailison Marshall, Louise Richie, Richard Taylor, Donna Ridge, Judith Houghton, Chris Barton, Jacqui Spirling, and Andrew Stubley.

Final thanks go to my mother, without whose selfless support this research would not have been possible.

ABSTRACT

The possibility of incorporating Artificial Intelligence (A.I) techniques into Visual Interactive Discrete Event Simulation was examined. After a study into the current state of the art, work was undertaken to investigate the usefulness of PROLOG as a simulation language. This led to the development of a working Simulation Engine, allowing simulations to be developed quickly. The way PROLOG facilitated development of the engine indicated a possible usefulness as a medium for controlling external simulations.

Tests on the feasibility of this were made resulting in the development of an assembler link which allows PROLOG to remotely communicate with and control procedural language programs resident on a separate microcomputer. Experiments using this link were then made to test the application of A.I. techniques to current visual simulations. Studies were carried out on the controlling of the simulation, the monitoring and learning from a simulation, the use of simulation as a window to expert system performance, and on the manipulation of the simulation.

This study represents a practical attempt to understand and develop the possible uses of A.I. techniques within visual interactive simulation.

The thesis concludes with a discussion of the advantages attainable through such a merger of techniques, followed by areas in which the research may be expanded.

(INCORPORATING ARTIFICIAL INTELLIGENCE TECHNIQUES; VISUAL INTERACTIVE DISCRETE EVENT SIMULATION; PROLOG; SIMULATION LANGUAGE; ASSEMBLER; REMOTE COMMUNICATION; PROCEDUARAL LANGUAGE)

- CHAPTER 1 -

THE PROBLEM

Introduction

Currently computer visual interactive simulations are written to test proposed production schemes before they are implemented. An expert manager manipulates variables in order to get the best mix.

Problems with this set up include:

i) The flexibility of the simulation is limited, since the control rules are 'hard coded' and cannot be interactively manipulated.

ii) The system assumes the manager can manipulate the simulation without decision aids.

Such problems with simulations are noted by Moreira da Silva (1982), Radzikowski (1983) and Rubens (1979).

It is suggested that the logical element of a simulation can be separated from the mechanics, being placed on a different processor. If the logical element is written in a modular form (eg production rules) the possibility of interaction with the rule base is available. Such a rule base could increase the flexibility of the simulation and also, by monitoring the simulation mechanics, act as a decision support system for the Analyst.

In addition to these points other advantages could also accrue by the use of such Artificial Intelligence (A.I.) techniques within simulation:

a) An A.I. program (or Expert System) could be linked with a simulation in order to control resource levels within that simulation. Thus the simulation could provide a visual way of testing an expert system. This would ease expert system development, since it could be tested whilst it was being built on a visual model of the true situation it is designed to control. Thus errors and omissions could be spotted and corrected. Furthermore such a visual picture of the experts performance could be utilised to convince users as to its reliability.

b) By monitoring a user interacting with a simulation it may be possible for an A.I. program to 'learn' how to manipulate that simulation. That computer learning is possible can be seen in Samuel (1963) and Rich (1983). This could provide the basis for rapid development of simulation control and process control systems.

Research Objectives

This PhD aims to investigate the suitability of A.I. languages, and in particular PROLOG, as a medium for simulation rules. Whether PROLOG is well suited to this task could be tested by the development of a PROLOG simulation engine . It was then anticipated that a system could then be developed whereby PROLOG could control a standard simulation on a separate processor using a standard procedural language simulation. The choice of separate processors is deliberate to ensure that the simulation is separated from its controlling logic. The analyst at the simulation (as opposed

to logic) machine should have the opportunity of interrogating the PROLOG logic program to gain confidence in the expert system. Having investigated the technical incorporation of A.I. techniques into simulation, an investigation is carried out into possible applications and problems. This includes the development of a learning program whose performance in controlling a simulation is comparable to that of the human user and teacher.

A Review of the Chapters

In chapter 2 we discuss the current literature relating to Simulation, Decision Support Systems, and Artificial Intelligence. Deficiencies in current decision support technology and simulation stem mainly from the inherent assumption that the problem under consideration is structured. Artificial Intelligence techniques on the other hand are specifically designed for unstructured problems. Thus the use of current O.R. techniques with modern Artificial Intelligence Systems should be a medium for dealing with the semi-structured problems prevailing in industry. Having considered the literature an argument is put forward supporting the research undertaken in this thesis.

Chapter 3 deals with the investigation of PROLOG as a suitable medium for containing simulation logic. This has lead to the development of a working simulation engine, containing the modelling logic of simulations.

In chapter 4, the use of PROLOG as a language for development of expert systems is considered. In particular the development of an expert which can be viewed as containing simulation controlling logic is considered.

Chapter 5 considers the interfacing of PROLOG simulations and PROLOG expert systems, using examples from previous chapters. Ultimately this leads to the development of an interface that allows communication between separate microcomputers.

In chapter 6 we discuss the interfacing of a PROLOG system with standard procedural language simulations. This builds on the work of the previous chapter and is illustrated with a simple example.

Chapter 7 investigates possible applications of A.I. components to visual interactive simulations. Experiments are carried out, with the effects on the expert and simulation discussed.

We conclude in chapter 8 with a summary of the research and a discussion of resultant conclusions. Indications of future research are then made.

- CHAPTER 2 -

RESEARCH BACKGROUND

As stated in the outline of the problem, the research combined several diverse areas of study. In order to establish just what has been achieved in these areas and in the solution to the problem in particular, a detailed literature study was undertaken.

Reviewed in this chapter, therefore, is the background to the research presented in this thesis.

Because the research has involved incorporating new technologies into operational research, this chapter is necessarily diverse. Literature sections are included on the following :

- Introduction to decision making and support
- Current Problem solving methodologies
- Artificial Intelligence

After the above review, an argument is put forward supporting the research contained in this thesis.

Introduction - Decision making & Support

Much work has been undertaken to understand exactly what is meant by Decision Making. Definitions exist in many

forms (Mayer (1977) and George (1970)) but most stem from the pioneering work of Simon (1960) who described decision making as a three - phase process :

"finding occasions for making a decision; finding possible courses of action; and choosing among courses of action"

It is not intended to pursue a detailed study of such definitions in this thesis beyond giving this basic outline.

Decision support requires an understanding of decision making both in individuals and organisations. Specialist literature on decision making can be classified into several main schools of thought (Keen and Scott Morton (1978)). Each of these schools of thought stresses one particular aspect of decision making (for example the use of rules of thumb, and the problems of organisational politics). It is difficult for the Decision Support System (D.S.S.) designer to pick any one model. In reality, though, they will often be constrained by the decision situation. Of course, in order to make decisions, it is necessary to solve problems. Simple Decision Support Systems (D.S.S.) aid the (human) problem solver rather than tackle such difficulties themselves. However, with the new computing techniques, designed to enable computers to perform tasks currently thought of as needing human expertise, systems are now being developed which attempt to solve problems. Early work was centred around algorithms thought to mimic the human (Wilson,1970). These were largely unsuccessful and very

slow. A more modern approach also felt by some to be related to human problem solving (eg. Young, 1979) is the use of simple production rules or heuristics. This is proving much more successful and will be described in detail later in this chapter.

Having given a brief overview to decision making and decision support, we shall now discuss where D.S.S. might be usefully implemented.

Structured vs. Unstructured Tasks

Simon (1960) proposed to classify decisions as being of two types :

> programmed: "decisions are programmed to the extent that they are repetitive and routine"
>
> non-programmed: "decisions are non-programmed to the extent that they are novel ... there is no cut and dried method for handling the problem"

Keen and Scott Morton (1978) took a different view of Simons classification renaming ´programmed´ to ´structured´ and ´non-programmed´ to ´unstructured´, they then designated problems that lay between these two types as semi-structured. A semi-structured task, they defined, is one where at least one of the activities is unstructured.

It is these semi-structured tasks that can usefully benefit from D.S.S., since structured tasks can be automated, and the D.S.S. philosophy argues that for unstructured tasks computer tools are inapplicable.

Semi-structured tasks have been characterised by many people including Lee and Hurst (1983) and Radzikowski (1983).

Radzikowski (1983) characterises them as follows :

- solution objectives are ambiguous,numerous and not operational
- the process required to achieve an acceptable solution cannot be specified in advance
- it is difficult to determine, either in advance, or after the fact, which steps are directly relevant to the quality of the decision

In their assessment on the suitability of D.S.S. for such problems Lee and Hurst (1983) state

"Once modelling techniques for semi-structured problems are established, the effective utilisation of decision support technologies can be applied to implement modelling techniques, in particular to seek solutions to such problems"

Also Keen and Scott Morton (1978) state that

"A D.S.S. provides a coherent strategy for going beyond the traditional use of computers in structured situations, while avoiding ineffectual efforts to automate inherently unstructured ones"

Problem Solving - The O.R. Approach

For most people not in the O.R. world, O.R. is synonymous with the quantitative modelling of structured problems. Moreira da Silva (1982) gives reasons for this :

1. the low proportion of case studies in publications
2. the high proportion of publications where the human part of the system is forgotten altogether.

The emphasis within O.R. texts of the optimising quantitative model has been criticised by many within its ranks, for example Boothroyd (1978). He argues that optimised quantitative modelling has its uses, as long as the model is incorporated as an integral part of the whole decision making process.

Thus it can be seen that O.R. techniques are useful mainly only when the problem at hand is structured. On semi-structured problems, the techniques can be used with some effect on the structured subproblems. A difficulty here, however, is the inflexibility of these O.R. modelling techniques. By using such a model on a subproblem it is implicitly implied that no change will occur in that

sub-problem. Such cases are in reality quite rare. Thus, for semi-structured problems, O.R. can at best serve only as an approximation.

Further, such rigid models waste the pool of experience of the manager whose cognitive processes will not be structured and would have to adapt to the O.R. model (rather than the other way round).

Problem Solving - Discrete Event Simulation

Introduction

Discrete Event Simulation is a well established O.R. modelling technique. It allows the decision maker to explore consequences of decisions without actually having to apply those decisions to the harsh real world.

Brief History

As early as 1964 there were a large amount of simulation languages available (Tocher, 1964). Most of these were dedicated to a specific machine only and Tocher concluded that

> "it is a sad fact that the choice (of simulation language) will most likely be made by the type of machine available to (the user)"

After a detailed comparison of the languages available Tocher concluded that

"for occasional use, a simple language, which is easy to understand and learn may be more valuable than one of the sophisticated languages that have many facilities, but ... (are) ... much more complicated to use and understand".

Since 1964, the number of simulation languages "purporting to aid the unwary user of simulation" has greatly increased (Crookes, 1982). In 1981 there were some 137 such offerings.

Crookes was clearly concerned by this, but was at least happy to report "a level of model verification not previously attainable, by the use of pictorial outputs from a running simulation" (a major benefit of Visual Interactive Simulation - see later). Thus the early difficulties of writing good simulations (see Conway (1963) and Conway et al (1959)) had been largely overcome.

Another major advance has been the great number of simulation languages and packages implemented on microcomputers (O´Keefe,1984).

Packages designed to aid the simulation analyst have included program generators of which CAPS (Clementson, 1974) is the best known. This has been hailed by Crookes (1982) as a welcome development. Working with final year undergraduates of Management Science at Warwick it has been found that whilst using CAPS is undoubtedly easier than learning the language ECSL on which it is based, the problems in the CAPS system can confuse as much as

the CAPS system itself aids. Although a bold move towards the automated programming of simulations, providing a package which does all the easy work for you (though not necessarily in the same way you would) can only be the start of a research drive in that direction.

A more modern approach, put forward by O'Keefe (1984), is the Interactive Menu Driven Interpreter. This allows rapid development and immediate execution of visual interactive simulation models. The model can be tested whilst it is being written and hence the distinction between building and interaction disappears. The interpreter is programmed using a package of PASCAL simulation subroutines. Although this system is very much experimental, its scope for further development is encouraging.

Although simulation is a useful modelling technique, it is not used as widely as it could. This is primarily due (Balmer and Paul, 1986) to cost and difficulty of use. With this in mind work is currently being undertaken at the London School of Economics on a project called C.A.S.M. (Computer Aided Simulation Modelling). Described by Balmer and Paul (1986), the C.A.S.M. project has been set up to research into ways of automating as much as possible the use of simulation modelling. This work is being undertaken with cost and difficulty problems in mind.

Special area Simulation Languages

Many languages have been developed for simulating certain specific environments. One example is GRASP (Donney et al, 1984) - this is a computer aided design system for modelling and evaluating industrial robot workplaces. GRASP satisfies a range of simulation needs within the context of designing, implementing and operating industrial robotic systems. GRASP has been used to help solve a wide range of practical industrial robot problems.

The Approaches to Modelling

A few underlying methodologies to simulation modelling have been put forward. The earliest, developed by Tocher (1962) and still the most favoured in the U.K. (Crookes, 1982) is the three-phase approach. The three phases are :

1. A, time scan : find the next time at which one or more bound activities are scheduled to be executed
2. B phase : the actual execution of bound activities found necessary in the previous phase
3. C phase : the testing (and execution of the action part) of each conditional activity

The executive program cycles round these three phases until a preset duration or predetermined terminating condition is reached.

The three-phase method involves the recognition of two distinct types of occurrence (O'Keefe,1984)

1. Bound - which is predictable and can therefore be scheduled
2. Conditional - which is dependant upon certain conditions (normally the availability of certain objects within the system being modelled)

Crookes´ (1982) preference to this method of modelling is due to "economy" - by this he means the ease of modification due to the relationship between independent conditional rules and the three-phase method.This allows for modular programs to be constructed. The other principal methods of modelling are now outlined.

1. The next design historically is a subset of the three-phase method. It is a two-phase system known as the event method. Its two phases are the first two phases of the three-phase system - it does not use the third conditional phase. It was used in the first simulation language published in the U.S.A.
2. The process system. This is very different from the user viewpoint. The user is required to provide a program in the form of a number of ´processes´, each of which is a life description of one type of object in the simulation. A process needs to contain references to the progress of objects of other processes where these interact with its own objects progress. The advantage of this form of modelling is

that "the process is a natural form of expression and is easily understood".

Crookes (1982)

"it is apparent that scope for standardisation in...(simulation)... is limited until the conflict is resolved in favour of one of the (modelling) methods".

Recently Crookes et. al. (1986) have introduced a three-phase simulation system written in PASCAL for use on microcomputers, minicomputers and mainframes. This system is essentially a suite of PASCAL subroutines which perform various aspects of simulation programming (such as Entities, Sampling, and Histograms). Included in the latest version of this is an interactive simulation program generator, based in concept on the CAPS system (Clementson, 1974).

Cellular Simulation

The methodology of cellular simulation (C.S.) is described in Spinelli and Crookes (1976). The basis behind a C.S. is the splitting up of a simulation into non-overlapping activity groups (or cells). Each cell can thus be regarded as a simulation in its own right. An individual activity may not be in more than one cell. C.S. is said to reconcile the event based approach with the activity approach. In terms of the three-phase model, a great saving in the execution time is possible if we observe that there is no point in testing a C activity within a cell

unless that cell has had a B activity executed since the last time the clock was stopped. This can save a substantial volume of checking of those activity tests that must fail. Spinelli and Crookes also indicate the possibility of modelling large systems in separate cells, written at different times and for different purposes and later combined. This is an ingenious application of quite a simple concept. Its chief advantage is in its computational efficiency.

Problem Solving - Visual Interactive Simulation (V.I.S.)

Visual Simulation

A simulation model can be visually enhanced in one of four ways :

1. providing a trace - the values of selected variables are displayed over time
2. displaying time series or histograms etc.
3. tables composed of data that is updated by the simulation can be effectively used
4. displaying a picture corresponding to the real-life system being modelled.

Crookes (1982) described the advantages of visual simulation. "it aids the writer ... (for program) ... verification. It aids the ... client ... (to believe) ...

the computer program to be a fair representation of his real world".

Visual Interactive Simulation

Hurrion conceived the term V.I.S.. His PhD in 1976 lead to the development of VISION (Hurrion, 1980), which in turn lead to the well known FORTRAN based SEE-WHY system. The technique has since been extended by research students at Warwick - Secker (1977), Brown (1978), Rubens (1979) and Withers (1981).

Hurrion (1980) states that the basic design criteria of V.I.S. is to have

"

i) a simulation language in which it is possible to write complex industrial problems

ii) the ability of a 1-1 correspondence between elements in the model and elements as described visually on the vdu(s)

iii) flexibility at run time "

The user may choose the structure of a V.I.S. to be either event, activity or cellular based. In describing the advantages of V.I.S., Hurrion (1980) states that

"the visual interactive approach has ceased in making simulation a 'black box' technique, but now opens up the method for management to look inside. It now becomes a

transparent box which greatly assists in the problems of communication and model credibility"

Typical applications of V.I.S. are to be found in industry (such as manufacturing), simulating production processes. Such applications have been made on both internal Warwick projects (Hurrion and Secker, 1978) and on a series of joint research projects between Warwick University and various companies Secker (1977), Brown (1978), Bowen (1978), Bowen et al. (1978) and Rubens (1979). Hurrion and Secker (1978) observe that, by watching a simulation model progress through time, and having the ability to interact with it, a user can improve his analysis and understanding of the original problem situation. This is in many ways similar to the experience obtained by the physical process of developing a model (Pollard, 1986).

Applications to less structured problems (Rubens, 1979) encounter additional problems. In particular there is a need for development of good interfaces that "enable the decision maker to use his creative thinking and pattern recognition capacities to their maximum potential" (Moreira da Silva, 1982). Rubens points out that the V.I.S. approach "does offer a number of novel advantages over traditional batch mode modelling, such benefits can easily be lost if insufficient attention is given to the ergonomic and psychological issues surrounding the man-machine synergism."

The work on structuring V.I.S. software so that the display design is independent of the actual model is important since it allows freedom in the interaction of the manager and the analyst (Fisher, 1982).

Later Withers (1981) has suggested the interactive development of visual simulation models. This is in some ways similar to the work by O´Keefe mentioned earlier (O´Keefe,1984). Withers´ project consisted of providing facilities for the interactive design of displays used directly by the manager in the very early stages of the project. There is also a 1-1 correspondence between display elements and the simulation. This leads to the development ´behind the scenes´ of the simulation with minimal analyst involvement.

To date, no practical application of Withers´ work has been reported. Withers´ work (on vessels) was based on a pencil-and-paper problem at British Steel and showed that most of it could be tackled using the Withers approach.

Problems with current V.I.S. are its dependance on the problem being suitable for discrete event simulation (Withers, 1982) and its limitations of interaction. It is not possible to change the underlying logic of the simulation whilst it is running. For example, in coding logic concerning movement of entities about a simulation, the analyst is setting this logic within the procedural high level language. It is possible to develop ad hoc solution by regarding such logic as being part of a database, which may

then be edited. But even this solution is at best limited in scope, since the changes it allows are only very small.

Problem Solving - Artificial Intelligence (A.I.) Techniques

Introduction

Because this field plays a crucial part in the research carried out in this theses, and because it is a new area for O.R., a large section of this chapter deals with applicable Artificial Intelligence (A.I) techniques. For the continuity of this chapter, that section appears after the sections on problem solving.

Problem Solving - Decision Support Systems (D.S.S)

Introduction

A D.S.S. is a computer system designed to assist with semi-structured decisions (see the above section 'Structured vs. Unstructured Tasks'). Such systems might divide into personal support systems, group support systems and organisational support systems. A D.S.S. may take a number of forms, for example :

- providing a mechanism for ad hoc data analysis
- estimating consequences of proposed decisions
- proposing (and/or making) decisions

Generally, then, D.S.S. are designed to aid or improve the process by which people make decisions. This is in contrast to routine data processing, which aims at greater efficiency, accuracy and cost savings.

Existing Decision Support Systems

Alter (1980) attempts to taxonomise existing D.S.S. into set functions such as

i) file drawer systems which allow immediate access to data files

ii) data analysis systems which allow manipulation of data by specific and general operators

iii) accounting models calculating the consequences of specific actions

iv) optimisation models providing guidelines for action by generating the optimal solution consistent with a series of constraints.

Keen and Scott Morton (1978) summarise the situations in which a D.S.S. can be useful as also involving at least some of the following characteristics.

"

1. The existence of a large data base, so large that a manager has difficulty accessing and making conceptual use of it

2. The necessity of manipulation or computation in the process of arriving at a solution

3. The existence of some time pressure, either for the final answer or for the process by which the decision is reached

4. The necessity of judgement, either to recognise or decide what constitutes the problem, or to create alternatives, or to choose a solution. The judgement may define the nature of the variables that are considered or the values that are put on the known variables."

Keen and Scott Morton give examples of several widely used D.S.S. including :

i. PMS (Portfolio Management System) - designed for investment managers. It is a graphics system with a variety of simple models working on large and complex databases.
ii. Projector - this is for the support of corporate finance planning.
iii. Brandaid - this is a marketing model.

A further example is found in Moreira da Silva (1982) who attempted to develop a D.S.S. generator. The research undertaken by Moreira da Silva had two objectives (Moreira da Silva, 1982) :

1. to extend the application of visual modelling to decision areas where discrete simulation is not appropriate.
2. to investigate the "potential for a generalised framework that could form the basis for development of decision making aids for different problems."

The research strategy followed "action research methodology". This is the analysis of a real situation followed by generalisation, in this case aiming to form a generator of aids for different areas of decision making. The action project (ICI in Huddersfield) was to develop a specific D.S.S.. This was subsequently used to "learn more about the planning activity and to stimulate the discussion and experiments of different approaches to support it."

In his thesis, Moreira da Silva states some of the problems of research, including communication and lack of information. He concludes that :

1. Current O.R. is not suitable for semi-structured problems.

2. The analysts role in D.S.S. goes a lot further than just technical development. In building the model the analyst gains a greater insight into the problem as a whole and will therefore develop a high level of problem orientated expertise.

3. D.S.S. can be used on semi-structured problems.

4. D.S.S. are at present "reluctant" to accept complex models.

Artificial Intelligence (A.I.) and D.S.S.

A.I., and in particular Expert Systems has for a surprising time now been thought of as perhaps offering something to the D.S.S. field. Expert Systems and O.R. have been considered to have substantial overlap (O'Keefe, 1985),

although very little has been done to merge these technologies. This overlap is in the form of knowledge acquisition (which has been developed within O.R.and practiced for some time) and knowledge acquisition (which is used in the O.R. modelling process).

Keen and Scott Morton (1978) feel that within computer science A.I. has perhaps the greatest potential with respect to decision making. Besides, if one can understand computer intelligence, one is well on the way to understanding human intelligence, and therefore human decision making (Winston, 1977) and (Young, 1979).

Shortliffe (1976) outlines the four core topics of A.I. as being:

1. Modelling and representing knowledge
2. Reasoning, deduction and problem solving
3. Heuristic search
4. A.I. systems and languages.

As Keen and Scott Morton (1978) point out, "the first three of these four themes are directly relevant to decision support". However, as indicated by Bramer (1985), the claims given for A.I., or in particular the applicable subarea of Expert Systems, in the early days far exceeded the actual accomplishments.

Keen and Scott Morton (1978) do however regard expert systems as one area of A.I. especially relevant to decision

support. Talking about MYCIN (an early expert system), Keen and Scott Morton conclude that its underlying principles are of "value to Management Information Systems professionals trying to build tools that handle complex problems ... (and) ... respond intelligently".

Much more recently Radzikowski (1983) has reviewed prospects for Expert System incorporation into D.S.S.. In a well thought out paper, Radzikowski points out that Expert Systems complement O.R. methods. The latter are best suited for structured problems, but are now being used increasingly for semi-structured ones (see also Boothroyd, 1978). Expert Systems on the other hand "are designed for unstructured decision situations, and they can serve as a structuring tool. Therefore incorporation of Expert Systems into D.S.S. is potentially beneficial and should produce a decision aid of a superior quality".

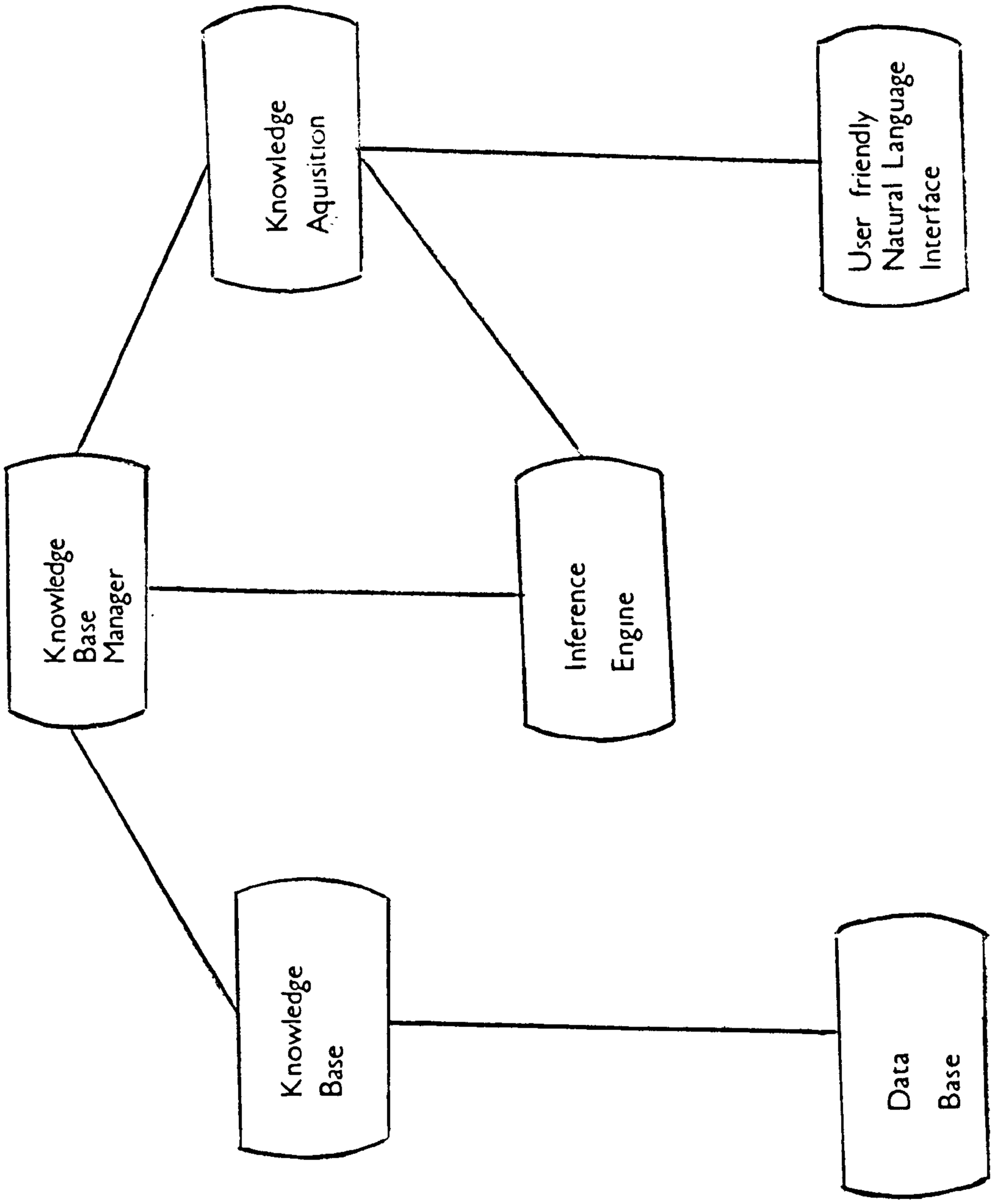


Figure 2·1 Structure of an Expert System

Radzikowski also points out the shortcomings of current D.S.S.. He points out that :

1. most existing information systems have attacked problems of a structured nature, whereas the most important problems are in fact unstructured.

2. "When we consider that every problem when faced for the first time appears to us as unstructured, it is evident that a substantial gap exists in the scope of decision situations covered by D.S.S.. Expert systems seem very well tailored for the role of a support system in structuring new decision situations, organising acquired experience, and isolating subproblems suitable for O.R. methods."

3. there is a discrepancy between managerial decision processes and O.R. solution procedures. Mintzberg (1973) points out that managers seldom make decisions as part of a deliberate, coherent, and continuous decision making process. The manager (on average) spends less than 5 minutes on any single activity.

4. it is standard practice to interface the decision maker with a D.S.S. through assistants, rather than directly.

Further, it has been suggested that the ideal D.S.S. would take an active role in leading the decision maker to a decision (Benet, 1983). For this the computer system must have

a. an understanding of what the decision maker is trying to do
b. an understanding of the decision makers environment
c. a knowledge based system.

Radzikowski (1983) states that the progressive sequence of Management Systems consists of :

- pure data processing systems
- Management Information Systems (with predefined data aggregation and reporting capacities)
- D.S.S. (an extensible system that can support ad hoc decision modelling in evolving decision space).

Onto this current list Radzikowski extrapolates to include D.S.S. with knowledgeable components and names them Decision Support Expert Systems (D.S.E.S.).

"The D.S.E.S. would ;
+ operate on explicit knowledge stored in a knowledge base
+ interface with the decision maker in natural language
+ choose proper QM techniques
+ create a qualitative model of the decision situation
+ present and explain the solution to the decision maker"

Radzikowski (1983) concludes finally that...

"The necessary preconditions for the merger of Expert Systems into D.S.S. are met".

Problem Solving - Future O.R.

Expert Systems and O.R.

Interest in the emerging technology of Expert Systems is beginning to result in research within the O.R. world. Work at the moment is naturally at a very infant stage and interest is not always matched by expertise. O.R. workers do, however, have some relevant points to make.

O´Keefe (1985) gives a brief outline of what expert Systems are available in the O.R. domain. These are notably in the traditional area of O.R. eg. scheduling and network planning. He also states that

"In using Expert Systems two distinct approaches are possible

1. using standard expert system methods
2. embedding techniques within Expert Systems "

There may also be, for reasons stated later, a third option...

3. embedding Expert System techniques within O.R.

A common area of interest for both Expert Systems and O.R. is knowledge acquisition. How do we elicit information from an expert ? One O.R. technique which could be of use for Expert Systems is the cognitive mapping process outlined by Eden, Jones and Sims (1983).

Most literature, however, has concentrated on the benefits to O.R. from Expert Systems. Kastner et al (1985) are of the opinion that Expert Systems that use large mathematical packages for a portion of the problem solving task ("Hybrid Expert Systems") are one of the most obvious first applications of Expert Systems to O.R.. Talking about the projected potential of Expert Systems for O.R. Kastner et al state that

"The area of O.R. is still largely untapped and it is only a matter of time before Expert Systems will be developed for traditional O.R. application areas".

However, as previously stated, little actual research into Expert systems has been undertaken by the O.R. community. One exception to this is the work undertaken by Doukidis and Paul (1985) at LSE. They have been involved in developing Expert Systems to help with the formulation (ie the logic) of a simulation model. Doukidis and Paul undertook research into a system that could help produce the Activity Cycle Diagram (A.C.D.) for the users (client and analyst). Doukidis and Paul regarded A.C.D.s as "an

excellent means of communication between the various people and computers that are involved in a discrete event simulation project".

Doukidis and Paul tried two techniques for tackling this problem :

i. production systems

ii. natural language understanding system approach (N.L.U.S.)

They found several problems with the production rule approach including ...

a. the IF...THEN structure was too simple : some knowledge had to be expressed explicitly in the interpreter which meant a lost advantage of modularity and uniformity.
b. the "is it true ... ?" type of question that is used in production systems is not convenient for this problem.

The N.L.U.S. approach tries to use normal English, or at least a structural limitation of English. Doukidis and Paul feel that this is a natural way for the client or analyst to describe the problem as he sees it. However, there are many problems with such "Natural" language system. In particular they tend not to be as natural as the user is lead to expect - see for example Hebditch (1984).

In experiments the researchers found the N.L.U.S. to work better than the production systems approach because of the latter´s disadvantages.

The prototype systems described in Doukidis and Paul (1985) indicate that computer-aided model formulation is feasible. Since then Paul and Doukidis (1986) have developed a working N.L.U.S. simulation formulation system written in PASCAL. They illustrate its use on the simulation of a small bar, similar to that used to illustrate CAPS (Clementson, 1974) and to the problem outlined in Appendix 3 of this thesis. Although the N.L.U.S. system is still limited in scope, its potential merits future research.

A further example of research into using Artificial Intelligence techniques within an O.R. problem is available in Grant (1986). This work concerned the development of an aid to schedule repair jobs on RAF squadrons. The level of resources and their capacities are completely predetermined from the scheduler´s viewpoint. Currently this scheduling is done manually without any aids. Since the task was knowledge intensive, it seemed that expert systems techniques could be of use. The usual O.R. techniques for scheduling applications are simulation, network methods, combinatorial procedures and heuristic approaches. Grant soon established that these techniques were inappropriate due to the complexity of the problem and the non-availability of certain key personnel. Thus Grant concluded that "an A.I. assisted heuristic scheduling program seemed to be the only

workable approach". He has so far reported a prototype version of this package written in an experimental language called NIAL. Although early days, Grant has concluded that use of such A.I. techniques are of great use for O.R. problems and in particular the area of simulations.

" (5) The use of A.I. planning techniques in simulations will be useful.

(6) The handling of decision rules in O.R. simulations is generally fixed. An embedded expert system would enable the simulation to incorporate an intelligent choice of rules to better model (human) decision making in the real life system being simulated."

Applicable A.I. Techniques

Introduction

"Artificial Intelligence (A.I.) is the study of how to make computers do things, at which, at the moment, humans are better" (Rich, 1983)

Some of the problems that fall within the scope of A.I. include Game Playing, Theorem Proving, General Problem Solving, Perception, Natural Language Understanding, and Expert Problem Solving. A.I. has developed its own techniques to deal with these problem areas. "An A.I. technique is a method that exploits knowledge that should be presented in such a way that

- it captures generalisations
- it can be understood by people who must provide it
- it can be easily modified
- it can be used to correct errors and to reflect changes in the world and our world view
- it can be used in many situations even if it is not totally accurate or complete
- it can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must normally be considered " (Rich, 1983)

Such Techniques have already been acknowledged to be of use in non-A.I. problems.

"... it is possible to apply A.I. techniques to the solution of non-A.I. problems. This is likely to be a good thing to do for problems that possess many of the same characteristics as do A.I. problems".

As previously described, one such field is discrete event simulation.

Four A.I. techniques or tools which have possibilities for application to non-A.I. problems are

a) expert systems

b) knowledge representation using heuristics and production rules

c) learning (or 'induction')

d) A.I. languages.

a) Expert Systems

Introduction

"Expert Systems are designed to put specialised knowledge in the hands of people who need it" (Hebditch, 1984).

"Expert Systems are computer programs that embody the expertise in a specific domain that would otherwise only be available from a high speed expert" (Johnson, 1983).

"an expert system is a computer system that embodies organised knowledge concerning some specific area of human expertise, such as medical diagnosis, chemical identification, number theory, etc., sufficient to be able to duty as a skilful and cost effective consultant" (Michie, 1979).

"expert systems are problem solving programs that solve substantial problems generally considered as being difficult and requiring expertise" (Stefic et al, 1983).

From the above definitions of expert systems a lot of programs would appear to fit the bill. It is not until the conditions under which an expert system would be used are considered, that the differences become apparent. No doctor, for example, would trust a 'black box' program which, given the symptoms, simply presented out a set of diagnoses. He would want to be able to ask the system how it got its answers, why it asked for certain data and why other

possible diagnoses were not given. Further, the doctor would surely not use a system whose performance was worse than his own or would only find diagnoses for rare diseases. In short the expert system is distinguished from a ´normal´ program by its transparency, performance and utility.

Having defined an expert system it is instructive to consider where they might be profitably applied (Johnson, 1983).

Martin Ernst, vice president of Advanced Information Technologies offers some decision making time criteria. Expert systems are limited to handling decisions that take humans anywhere from a few minutes to a few hours to make. Decisions less than 45 seconds are evidently not very critical, otherwise, Ernst states, the expert would presumably take more time with them. At the other end of the time spectrum "you could spend years watching your expert" waiting for that infrequent situation when a special rule applies.

Furthermore, problems requiring extensive analysis are also inapplicable - "you probably won´t be able to develop the rules before the rules or environment change".

Ernsts´ view of expert systems being necessarily complicated is not shared by everyone. In the ´Report to the Alvey Directorate on a Short Survey of Expert Systems in U.K. Business´ (d´Agapeyeff, 1984), the main finding was that

"simpler expert systems are practical and being implemented now by self taught teams with little risk at relatively low cost, to produce modest but unusual gains".

From this it was concluded that "it is necessary to correct the impression ... that expert systems are inherently complex, risky and demanding. This impression is a handicap to competitive developments in the supply and usage of Advanced Information Technology".

The report feels that A.I. researchers have tended to concentrate on difficult topics (eg uncertain knowledge, search and conflict resolution), even though there is a doubt as to how central these topics will be to expert systems in business over the next few years.

A case in point is DEC - one of the most successful companies in the world in expert system applications. Yet they depend upon software in which knowledge must be treated as certain. Another simplifying constraint is that there is no search - the next rule is directly determinable from the current situation.

Despite this there is a common view that plausible reasoning from uncertain knowledge (a difficult task to do correctly) is fundamental. Most products in this field in the UK are based on SRI's method of such reasoning as implemented in PROSPECTOR (see later). Yet during the survey carried out for the Alvey report, no team questioned about their treatment of knowledge, said that uncertainty was a fundamental existing feature although it was sometimes

anticipated to be of growing importance in future applications. Even then it was not certain that the uncertainty could not be handled by an ad hoc solution.

It seems likely that the range and utilisation of expert systems may depend on the provision of the following

a) good dialogue facilities (Hebditch, 1984). Expert systems are designed to put knowledge in the hands of people who need it. Therefore, in designing an Expert System, developers need to focus on the user dialog mechanism. MYCIN, for instance, uses a question and answer communication system so slow that it could jeopardise a patients health. With MYCIN being one of the most successful of expert systems, there is no reason to believe that expert system developers are particularly aware of the lessons learned by dp system designers. Imagination too is at a low ebb - error controls for instance concentrate on keeping errors out, rather than getting good data in.

b) the capacity to embed an expert system within a conventionally designed application.

c) the sharing of state tables and other control information between expert systems and other programs.

Applications of Expert Systems

Expert systems have found their use in several fields (Bond, 1981), (Buchanan, 1982), (Michie, 1979), for example medicine, chemistry, geology, engineering, signal

processing, configuring computer systems, intelligent computer interfaces, office scheduling and education.

There are two approaches to building expert systems each with its drawbacks

a) write a program from scratch

b) write a program from 'shells'

The former method needs a great deal of time and effort since the analyst must first decide how he is going to represent the knowledge to be contained in the expert system, and then express the knowledge in that form. A shell on the other hand, provides the basic building blocks, but does have the big restriction on enforcing knowledge to be represented in a specific way. Shells have been designed to make expert systems easier to write. In use they take the form of a special programming language with a built in inference mechanism. In the course of this research an educational version of ES/P Advisor has been tested. In addition another has been studied called SAVOIR.

i) ES/P Advisor : This is marketed by Expert Systems International, Oxford, and is a PROLOG based consultation system. The user writes in a language which is based on PROLOG to produce a program that essentially provides a computer description of some manual - eg. for house conveyancing, employee sickness benefits, etc. The system provides a consultation shell and a compiler.

ii) Savoir : Available on many machines, SAVOIR is written in PASCAL, and may be used to provide advisory expert

systems, intelligent front ends and complex non-interactive programs. It is claimed (ISI, 1984) that the inference engine can cater for about 1000 rules on an IBM PC with a 'very fast response time'. Significantly, both forward chaining and backward chaining are provided for. Forward chaining makes deductions from both supporting and refuting hypotheses and is useful were there are many final results but few pieces of evidence. Backward chaining works from hypotheses to the possible supporting evidence - questions are asked only if they will supply needed information. Backward chaining is of use where there are few hypotheses but many items of supporting or refuting evidence.

One example use of SAVOIR is the COUNSELLOR expert system. This gives advice and recommendations on the control of disease in the winter wheat crop.

PROSPECTOR - an Expert System Example

PROSPECTOR is a rule based judgemental reasoning system that evaluates the mineral potential of a site or region with respect to inference network models of specific classes of ore deposits. It is intended to aid geologists in evaluating the potential of an explanation site or region for occurrences of ore deposits of particular types.

The overall performance of PROSPECTOR depends on the number of models it contains. Each model (eg of a type of deposit) is coded as a separate data structure independent of the PROSPECTOR system. It is therefore important to distinguish models from the actual PROSPECTOR system which

should be viewed as a general inference mechanism for delivering relevant expert information about ore deposits, based on the models. A model consists of a network of connections or relations between field evidence and important geological hypotheses (these together are termed 'assertions'). The change in the probability of one assertion affects those of others depending on how they are linked. In particular such a relationship may not be certain. Here, each assertion counts as votes for or against a hypothesis. How assertions are related to a hypothesis is defined via 'plausible inference rules' each of which has a 'rule strength' (positive or negative) measuring the degree to which a change in probability of the evidence assertion(s) changes the probability of the hypothesis. This is achieved via Bayesian Decision Theory (see Appendix 2).

As with all complex computer systems, once a PROSPECTOR model has been written it must be tested for validity. For PROSPECTOR this is achieved by testing the expert system against the expert who designed the model. As well as testing PROSPECTORs solutions, its explanation facility can be used to confirm that PROSPECTOR is reaching decisions for similar reasons to the expert.

Expert Systems Methodology

In the field of expert systems whose range of applications is so wide, there is no formal methodology. Texts on building expert systems (for example Hayes-Roth et al, 1983) consist of little more than a series of case

studies. What is intended in this section is to outline the main overlapping areas of method (for example the use of production systems, heuristics, and basic design decisions and types). It is the wrong place here to outline the workings of any one particular expert system since architectures are very often implementation dependant. Where concepts used in the course of this research derive from specific research or expert systems elsewhere will be made explicit when they are introduced in this theses.

The work necessary to develop expert systems is called 'knowledge engineering'. Quinlan (1981) states that knowledge engineering needs three design decisions :

i) how is the knowledge to be represented

ii) what architecture should the system have

iii) how should we test for consistency and completeness

Many people see the problem as being one of 'expertise transfer'

One problem which must be solved early on is the decision on what language or system the expert system should be implemented in (this is of course related to the above decisions). Two possible decisions could be made :

i) use a specific expert system 'language' (or 'shell') with a built in inference mechanism (eg. SAVOIR, ROSIE, EMYCIN, META-DENDRAL).

ii) use a general A.I. language (such as LISP or PROLOG).

The former decision would greatly speed up implementation but at the cost of dictating the data-format needed and the operations possible.

Of expert systems currently in use the dominant old method of implementation is to use INTERLISP on a PDP-10. This is a modified version of LISP, containing facilities for explanation, rule acquisition and debugging. PROLOG is beginning to get wider acceptance as being suitable for expert system implementation, especially since its syntax closely resembles that of production rules, which are the accepted way of representing a large amount of knowledge in an expert system (see later). Most modern expert systems are now written in PROLOG. PROLOG does, however, have problems in terms of its speed (although Japanese fifth generation computers, designed around PROLOG, will certainly cure this problem in the late 1980's), debugging and explanation facilities.

Having made the decisions Quinlan outlined above, the knowledge engineer must construct the expert system. Waterman (1981) regards the construction of an expert system as a multistep process as follows :

i) analyse experts decisions to produce a detailed formal description of components of the decision process.
ii) implement this as a computer program.

iii) interact with experts to refine the program.

There are, of course, some domains in which nearly all humans are experts (eg speech recognition) and in such domains the knowledge engineer may double up as domain expert.

Basic Expert System Architectures

In studying successful expert systems it is apparent that two main underlying architectures prevail.

The top-down or MYCIN type - Bond (1981), Shortliffe (1976), Duda et al (1979), Hayes-Roth et al (1983), Swaan Arons (1983) - (examples include MYCIN, EMYCIN, PROSPECTOR, PUFF, SACON, CLOT).

Here each rule is a modular section of knowledge, which is immediately understood when verbalised in English. The database is a set of named objects each with a list of attributes ('parameters') and associated values. There are only a fixed number of object types known to the system. Rules are conveniently split into LHS and RHS.

LHS rules are a conjugation of tests upon objects. A test is the application of one of a fixed set of system predicates to 1 or 2 objects, or else a disjunction of such atomic tests.

RHS rules are a list of actions - usually that of assisting some assertion that is establishing the value of some attribute.

The value of an attribute may be definitely known, or only known to some degree of uncertainty. In the latter case a numerical 'Certainty Factor' (CF) should be stored with the value. It is possible that a given parameter may have several values simultaneous each with its own CF. Some workers (such as Gasching) use genuine probability theory, but the majority use CFs and combine them with their own theory of uncertainty.

As well as values and CFs other items need to be stored with each parameter.

1. its verbalisation form
2. a list of values it can take
3. a list of rules it occurs on the LHS of
4. a list of rules it occurs on the RHS of and can have its value changed by
5. a list of rules it occurs on the RHS of but which do not change its value.

The action of the system is 'top-down', ie it sets itself an ultimate goal and chains back through the rules to produce a goal tree. When a goal is created that requires input data, the system asks the user. The system continues until it attains its goal which ends, for example, with it presenting its recommendations.

<u>The Model Based or CASNET type</u> - Bond (1981), Weiss and Kulikowski (1981), Quinlan (1981), Kulikowski (1980), Weiss, Kulikowski and Safir (1978), Weiss et al (1978) - (examples include CASNET, INTERNIST, IRIS, EXPERT).

Rules are used for :

i) deriving hypotheses from experimental finding

ii) deriving a structure describing the world state

iii) describing time development of the model to allow investigation of the time course and causation.

A typical example is CASNET - whose database is divided into three 'planes'.

i) experimental observations (true/false/unknown)

ii) pathophysiological states

iii) disease states

The systems rules allow it to define information and to extend its understanding and knowledge of the patient.

The system is initially 'event-driven' by the input of clinical findings until enough confirmed tests have been established to constitute a causal model which is representative of the real world. The system then becomes more directed and asks for specific clinical data to be obtained to enable the model to be refined, clarified and further confirmed.

b) Knowledge Engineering using Heuristics and Production Rules

Heuristics

The word heuristics derives from the Greek 'heuriskein' meaning 'to discover'. A heuristic aims at studying the methods and rules of discovery (Polya, 1947) or assisting in

problem solving. To practitioners, heuristics are simple procedures, often guided by common sense, that are meant to provide good but but not necessarily optimal solutions to problems.

Builders of expert systems attribute the performance of their programs to the corpus of knowledge they embody, which includes a large number of judgemental rules (heuristics) which guide the system towards plausible paths. Yet what is the nature of heuristics, and how do they originate ? By examining two case studies, on the AM and EURISKO expert systems Lenat (1982) has drawn some tentative hypotheses :

i) heuristics can be thought of as compiled hindsight, and draw their power from the various sources of regularity and continuity in the world.
ii) heuristics rise through analogy, specialisation and generalisation.

A comparison of three decision strategies was carried out by Kleinmuntz and Kleinmuntz (1981). The decision strategies varied in complexity :

1. random trial and error
2. use of heuristics
3. statistical Expected Utility (EU) maximiser using Bayes´ theorem.

After trials of programs adopting these three strategies, EU was found to give the best decisions (specified by a human expert) closely followed by the heuristic strategy. The random strategy was inferior but to a lesser extent than one might have expected. Another important factor was computation time - EU was some 80 times slower than the other strategies.

Kleinmuntz and Kleinmuntz thought that the results indicated the need for a tradeoff between effort and decision quality, with the results indicating diminishing returns to performance from increased amounts of computation effort. In particular, in situations where a good decision rule is lacking, essentially random trial and error may be effective in discovering new solutions.

A further point to note is the amount of knowledge needed about the task. EU´s strategy required far more detailed information than the others, making it less adaptable to a changing environment.

Finally Kleinmuntz and Kleinmuntz point out that

"the heuristic strategy seems to be within human capabilities, indicating that humans should be able to attain comparable performance. Of course, training will be needed to build up the knowledge base ..."

This quote gives a clear definition of the uses of heuristics in expert systems. But when are heuristics an

appropriate tool and when would other methodologies be more appropriate ? Zanakis and Evans (1981) list several instances "where the use of heuristics is desirable and advantageous":

i) inexact or limited data

ii) a simplified model is used

(in both these cases a fast near optimal solution makes more sense than a slow exact solution to an inexact problem)

iii) an exact method is not available

...

vi) repeated need to solve the same problem frequently or on a real time basis

vii) a heuristic solution may be good enough for a manager if it produces results better than those currently realised

...

ix) as a learning device

x) other resource limitations eg budget, manpower.

Zanakis and Evans have also tried to answer the question 'what is a good heuristic ?' by listing various criteria such as simplicity, speed, accuracy, robustness, good stopping criteria that take advantage of search 'learning' and avoid stagnation, etc.

It is also pointed out that problem-specific heuristics will tend to be more efficient than general ones, but will also be less flexible.

This may suggest the advantage of a system which modifies the basic heuristic according to the specific problem. In the context of an expert system, this may form the basis of a program that learns by examples. A database of examples could be used to tune the expert systems set of heuristics towards a particular problem.

Thorngate (1980) attempts to compare ten general purpose heuristics by trying to determine how often each would select alternatives with highest-through-lowest expected value in a series of randomly generated decision situations. His results indicated that most of the heuristics, including some which 'ignored' probability information, regularly selected alternatives with highest expected value, and almost never selected alternatives with lowest expected value. Such results tend to imply (together with Kleinmuntz and Kleinmuntz) that overheads involved in developing and running elaborate 'accurate' heuristics, far outweigh the advantages gained.

Production (rule) Systems

In the late 1960's Newell and Simon started using production systems for cognitive modelling and then for A.I. in general.

Three basic components of a production system can be identified (Davis and King, 1977):

i) a set of rules

ii) a database

iii) an interpreter

Taking the most simplistic direction, this could lead to the following system design :

rules : simple IF ... THEN statements, ie an ordered pair of symbol strings. The set of rules has a predetermined ordered meaning.

database : simply a collection of symbols.

interpreter : this scans the left hand side (LHS) of each rule until a match against the database is found. Then the matched symbols are replaced by those found in the right hand side (RHS) of the rule (and so on).

Of course the set up in general will be more complicated :

a) rules : More generally, one side (it can be either, and rules may serve a dual purpose depending on the side chosen) is evaluated (a process of matching and detection) with respect to the database. If found to be true, the action specified by the other side is performed. How the rules are organised is important - 'Conflict Resolution' is the term used to describe the process of selecting a rule.

b) database : This is a collection of symbols whose interpretation depends on the application, which could be one of the following :

i) modelling human cognition (Young, 1979) - the database represents short term memory (STM) with a fixed length

(approximately 7 elements) and specific organisation (linear, hierarchical, etc).

ii) expert systems - the database represents facts and assertions, it is of arbitrary size and structure.

c) interpreter : This takes the form of a 'recognise-act' loop. This causes re-evaluation of the control state of the system at every cycle (as opposed to procedural approaches). A disadvantage to this is the high computing time.

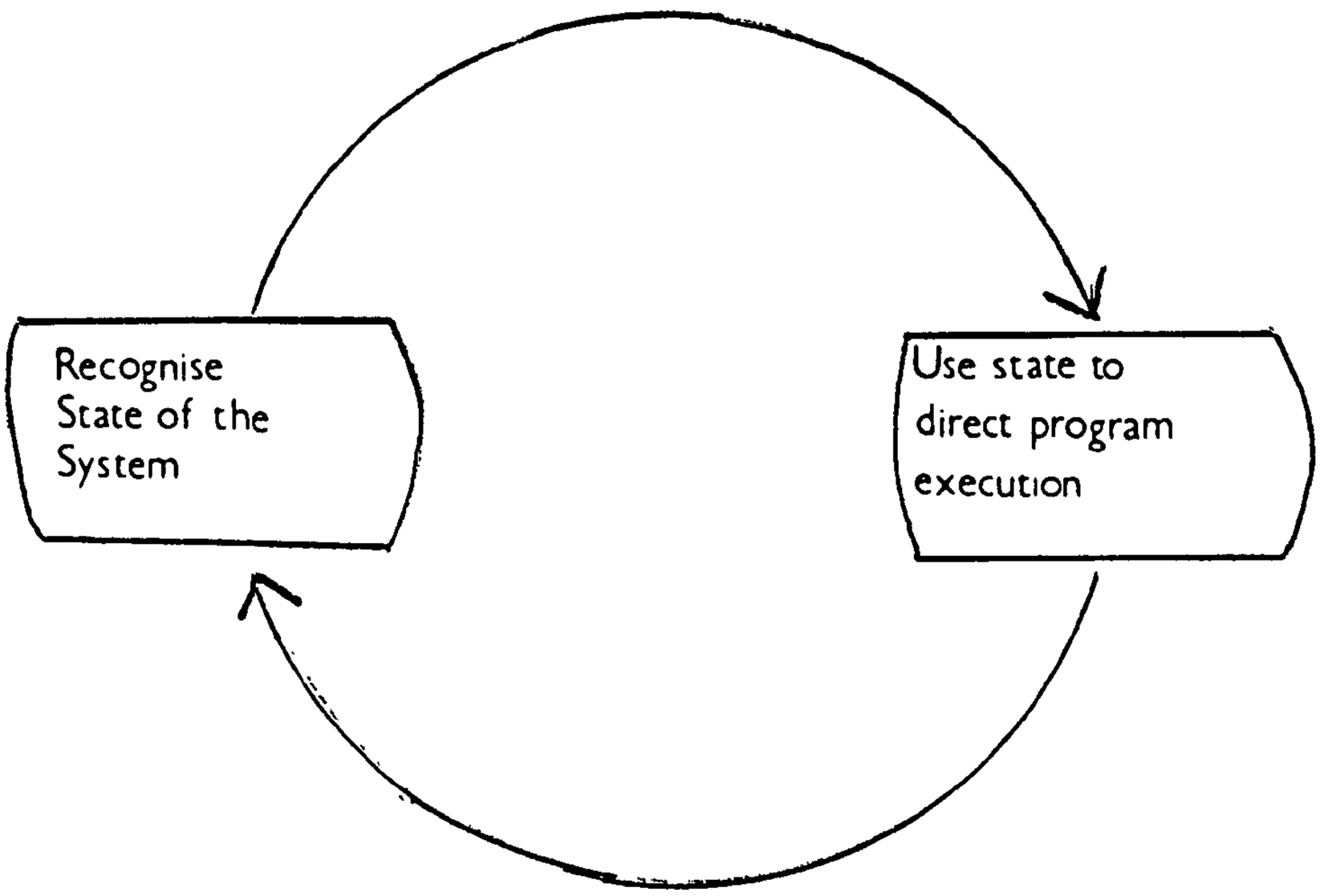


Figure 2-2 Recognise - Act Loop

Davis and King (1977) indicate the criteria used for conflict resolution :

i) rule order

ii) data order - rule fired which matches element highest up in database

iii) generality order - most specific rule chosen

iv) rule precedence

v) recency order

An Expert Systems View Of Production Systems

There are two current views of production systems

- psychological modelling (Young, 1979)

- expert systems

This section shall concentrate on the latter.

Such an approach starts with productions as representations of knowledge about a domain - and then attempts to build a program which displays competence in that domain. Efforts need not be human like (eg DENDRAL).

Rule based systems have a broad attention to changes in the database and are therefore well suited to situations found in 'industrial' applications with lots of variables and rich interconnections between actions and changes in the database. Rules allow the separation of data examination (LHS of rule) from data modification (RHS) - see Waterman and Hayes-Roth (1978). Rule systems are highly modular and allow easy development. New rules are simply added to the existing rule set - this allows for easy knowledge acquisition. In fact, in the case of one expert system (DENDRAL), the initial procedural approach proved sufficiently inflexible that the entire system was re-written in production rule terms.

Given the type of production system so far described two rule strategies are possible :

i) LHS matched, then RHS executed. This is called ´antecedent driven´ or ´bottom up´.

ii) RHS matched and then LHS set up for further matching. This is called ´consequent-driven´ or ´top-down´. A consequent driven system is usually given a premise to ´prove´ through deductive inference. The process of working backward through the rules from consequents to antecedents in search of a causal chain that will prove the given process is called ´backward chaining´ or ´consequent reasoning´.

c) Learning (or ´Induction´)

The Intelligent Database - Laurie (1984)

Current expert systems encapsulate the knowledge of human experts - they do not let the computer find out anything for itself. That would be done by an ´intelligent database´. These run round a database full of facts and from them extract a set of rules. These rules are expressed either as statements in a human like language, which conveys the substance of them to a person so that he can make the appropriate decisions on new data; or as programs which allow the computer to make the same decisions automated.

In many cases the database will be compiled for ordinary data processing purposes. The deduction of rules from it will be an incidental benefit as far as the user is concerned.

Examples include those based on the work of Quinlan (Expert-Ease) and on the work of Politakis and Weiss (see section below on 'AI research relevant to Learning Processes'). Expert Ease is a PASCAL based system for building a knowledge base by inducing rules from examples, using Quinlans induction algorithm ID3 (Quinlan, 1979). It essentially builds a decision tree from a set of examples. The database can be continually expanded with new examples. It is important that the use of any such induction process should be checked, and the resulting tree evaluated for its validity or at least plausibility before it is used in practice. In testing Expert Ease for this project it was found that in order for the system to provide a workable expert system, many additions to the original database are needed. Such additions, I found, often need the expert to have a detailed knowledge of the rule induced by Expert-Ease. This leads to the expert actually specifying the decision tree, rather than using the induction process. Expert-Ease is particularly vulnerable to error when used with example sets that are too small. The only heuristic available at this stage is 'the more examples the better' ! A further disadvantage is that no mechanism is available for the concept of uncertainty. Generally speaking, one might tentatively group intelligent databases into two main types (Rich, 1983) :

1. Those that learn by rote. The simplest kind of machine learning is the straightforward recording of data. Many

programs can record large quantities of data and yet are of little interest as learning programs. However, some programs, for example Samuel's checkers program (Samuel, 1963), that need to make static decisions can learn which decision to make after making it the first time. This saves a great deal of computational effort when decisions must frequently be made.

2. Those that learn by parameter adjustment. A large variety of A.I. programs rely on an evaluation procedure that combines information from several sources into a single summary statistic. In designing such programs it is often difficult to know in advance how much weight should be attached to each of the features being used. One way of finding the correct such weight is to begin with some estimate of the correct settings, and then to let the program modify the settings on the basis of its own experience.

There are of course many problems of information. The database may conceal more complicated rules than the rule finder can extract - possibly because they draw on information not in the database. Even supposing that databases of some sort of ancillary information were available on line, no conceivable processor could look through them fast enough to arrive at insights such as the brain can do (occasionally) in a flash.

A.I. Research Relevant to Learning Processes

As stated in d´Agepeyeff (1984) there appears to be a wide gulf between what A.I. researchers are working on and what is actually practicable for implementation of expert systems and other A.I. applications. However, some research is immediately relevant, and three such areas in the field of learning are :

i) discovering rules by induction

ii) using empirical analysis to refine expert system knowledge bases

iii) ´real-time´ plan creation and execution in dynamic environments.

i) discovering rules by induction Quinlan (1979) discusses the ´basic induction algorithm´ ID3 (as implemented in Expert-Ease) with particular regard to chess end-game problems. Presented here is a brief outline of the algorithm:

Let C = set of instances

A = set of attributes with permissible values

A1, A2, ... , AN.

Each member of C will have one of the values for A and we can therefore sort C into subsets C1, C2, ... , CN, where Ci contains those instances in C with value Ai.

Looking at this diagrammatically :

```
        /attribute A
        |A1 ----> C1
        |A2 ----> C2
        |...      ...
        \AN ----> CN
```

A similar process is then applied to each Ci. This is continued until each collection of instances is empty or all its members belong to the same class. A rule for attribute A can then be deduced. Of course this process assumes a large database of examples.

To make this clearer and for completeness, here is an example from Quinlan (1979).

problem: can the black king capture the white rook on its next move ?

attributes : black king is next to rook

white king is next to rook

Each of the attributes has possible values true (t) or false (f). The two classes will be 'can' and 'cannot'.

The training set will contain all possible instances

{tt:cannot, tf:can, ft:cannot, ff:cannot}

If the first attribute is selected we will have

```
/black king is next to rook
|t ----> {tt:cannot, tf:can}
\f ----> {ft:cannot, ff:cannot}
```

The same process is applied to the first (sub)collection, this time selecting the second attribute.

```
/black king is next to rook
|       /white king is next to rook
|t --> |t ---> tt:cannot
|       \f ---> tf:can
\f --> {ft:cannot, ff:cannot}
```

All subcollections now contain instances of only a single class; we can now replace the collections by the classes giving the rule :

```
/black king is next to rook
|       /white king is next to rook
|t --> |t ---> cannot
|       \f ---> can
\f --> cannot
```

This rule (found by induction) is identical to the program fragment

```
IF black king is next to rook
THEN IF white king is next to rook
     THEN cannot
     ELSE can
ELSE cannot
```

ii) using empirical analysis to refine expert system knowledge bases

The approach of Politikas and Weiss (1984) is to integrate performance information into the design of an expert model and to automatically provide advice about rule refinement. 'SEEK' generates advice in the form of

suggestions for possible experiments in generalising or specialising rules in an expert model.

The model consists of a set of tables for each possible expert conclusion. Each table provides a list of major and minor requirements for each conclusion to be valid. Also in each table are three rules stating the number of major and minor requirements for a conclusion to be definite, probable or possible.

The system is tested by comparing the models conclusions against known conclusions of the human expert. Where the model differs from the human, a set of experiments is devised by the system to modify its rule base. Rules may be strengthened (ie specialised) by adding to the number of major and minor requirements, or weakened (ie generalised) by reducing the number of requirements. The new database can then be tested against the known results as before.

iii) plan creation and execution The most advanced stages of research involve systems that have the ability to not only formulate courses of action but also to execute them. Such systems are called Complete Planners. McCalla and Reid (1982) describe such a system (ELMER) which in some sense learns from its past behaviour. ELMER has been written in LISP and is a taxi driver attempting to traverse a map from what it can see out of its window. Its world is thus dynamic and is therefore similar in nature to that of the blocks world HACKER (Sussman, 1975), although the two worlds are tackled in completely different ways; ELMER uses

hierarchical plans whereas HACKER uses linear sequences of actions.

d) A.I. Languages

Introduction

As with many applications, A.I. programming needs special tools, and in particular special languages. Ideally an A.I. language needs the following facilities : (Rich, 1983)

- particularly good facilities for manipulating lists
- pattern matching facilities for identifying data and determining control
- facilities for automatic deduction
- facilities for building complex knowledge structures
- control structures that facilitate goal directed behaviour in addition to more conventional data-directed behaviour
- the ability to intermix procedures and declarative data structures in whatever way best suits a particular task

Examples include those in the table on the next page (Rich, 1983).

Table 2·1 Artificial Intelligence Languages

IPL	A very early list-processing language
LISP	The most widely used A.I. language, in which the principal data structure is the list
INTERLISP	A fairly recent dialect of LISP, which is larger than pure LISP and provides a wider range of capabilities
SAIL	An ALGOL derivative with several additional features including support for an associative memory
PLANNER	An early language that facilitates goal-directed processing
KRL	A language that supports complex structures
PROLOG	A rule-based language built on top of a predicate logic theorem prover.

Of these languages the principal ones are LISP (INTERLISP) and PROLOG.

LISP : This language has influenced all other A.I. languages. Its principal data structure is the list and procedures have the same format as data, so that a program can construct a procedure and then execute it. The natural control structure is recursion, which is akin to many problem solving techniques.

PROLOG : This is a more modern production rules language in which programs are written as rules for proving relations among objects. An outline to PROLOG programming can be found

in Appendix 1. Its prominence in the U.K. and its relationship with fifth generation computer projects makes it an ideal language with which to investigate A.I. technique applications.

The Importance of PROLOG - the Fifth Generation

Currently in the UK the most successful AI language is PROLOG, even though most older expert system are written in INTERLISP. The main reasons are :

1. PROLOG is a newer language and it is therefore misleading to compare its use in numbers with INTERLISP. Even in the LISP dominated USA most recent expert systems are being developed in PROLOG.

2. PROLOG lends itself very well to production rule formalism.

3. The chief reason is the fact that research is currently being conducted in Japan for the next generation of computers - the fifth generation. Research is based on the language PROLOG, meaning that these machines will have architectures specifically geared toward PROLOG, resulting in very efficient performance. But why are these fifth generation computers being developed ? (Lemmons (1983), INGCT (1983)).

Current conventional computers have become numerical processing orientated, stored-program sequential processing systems. However, the situation has evolved in the following ways :

1. VLSIs (very large scale integrations) have substantially reduced hardware costs.
2. A new architecture for parallel processing is now required because device speed has approached the limit for sequential processing.
3. Parallel processing should be realised in order to utilise effective mass production of VLSIs.
4. The current computer technology lacks the basic functions for non-numeric processing of speech, text, graphics and patterns, and for AI fields such as inference, association, and learning.

It is for these reasons that the fifth generation computer systems (FGCS) project has been embarked upon. The functions required of such a system are as follows :

1. Problem Solving and Inference Function
2. Knowledge base Function.

This is aimed at providing storage and retrieval of not only data but also reasonable judgements and test results organised into a knowledge base.

3. Intelligent interface function

This is intended to allow computers to handle speech, graphics and images so that the computers can interact with humans flexibly and smoothly.

4. Intelligent programming function

The ultimate goal is to allow computers to take over the burden of programming from humans.

"the knowledge information processing systems realised by the FGCS are expected to expand extensively the fields where computers are applied, such as manufacturing, service, engineering, and office and business management".

To reiterate the importance of PROLOG to the FGCS project, a quote from Institute (1983) :

"Research in the initial stage of the FGCS project is based on the new programming language, the version 0 kernal language, which is extended on PROLOG"

The emphasis in Japan, it must be stressed, is different from that elsewhere. Work in Europe and the USA has tended to concentrate on the commercial applications of AI - namely expert systems. This should be contrasted with the non-commercial ethics of the FGCS program, in which logic programming is used which has a clear core in backward chaining through Horn clauses (eg PROLOG). Ostler (1985) states

"There are two ways for researchers to extend the capacities of expert system technology. One is to work on extending the materials from which expert systems can be constructed ... The other is simply to set about building expert systems, to build in, and so test, the new ideas one may have for potential about progress".

Limitations in Current Research (Reasons For This Thesis)

Having analysed the background to the general areas of simulation, O.R., Decision Support, and Expert Systems, it was felt that current research was lacking in several aspects.

Firstly although much has been said about the suitability of PROLOG to simulation logic (eg Adelsberger, 1984), little has actually been researched. One prototype simulation system that has been written in PROLOG called T-PROLOG (see Adelsberger, 1984) is a simple process based simulation language. It appears to be suitable for only very small problems, allows no interaction, and only works in a 'batch mode'. O'Keefe (1984) points out the 'possible' link between expert systems techniques and simulation logic .

"Rule based techniques, as evident in expert systems work, may provide a new method for writing and thinking about simulations ... A simulation is simply a set of rules ... such as

IF not (empty queue)

THEN remove entity

engage to sender

Thus available rule-based software and techniques such as popular expert system languages like PROLOG may be useful simulation tools ... The important matter is that being able to define a simulation as a series of rules in an

appropriate software environment may make it possible for simulation modelling to employ the facilities afforded by such environments ".

This same argument is also available in Crookes (1982). With PROLOGs natural representation of production rules this thought deserves investigation, particularly when the advantages of the production rule approach are considered. Crookes (1982) :

"The program units ... are more atom like than those of other systems ... Amending such a program is as easy as such a task could be".

Presented in this thesis is the research undertaken in this aspect. PROLOGs suitability has been considered to the three-phase simulation process, to visual simulation, and to interactive simulation. PROLOGs drawbacks are also investigated. The language developed has been kept as simple but flexible as possible. Tocher (1964) stated :

"for occasional use, a simple language, which is easy to understand and learn may be more valuable than one of the sophisticated languages that have many facilities, but ... (are) ... much more complicated to use and understand".

With this in mind, the research in this area has ultimately lead to the development of a PROLOG simulation engine illustrating the languages simulation capabilities.

As well as the current interest in a PROLOG based simulation language, there is much interest in the contribution A.I. techniques (and especially expert systems) themselves can make to O.R.. This interest is borne out by the number of expert system related papers being published in O.R. journals - eg Bell (1985), O´Keefe (1985), Kastner and Hong (1984), Nixon (1986). The possible use of A.I. techniques in non-A.I. fields has been indicated by Rich (1983):

"... it is possible to apply A.I. techniques to the solution of non-A.I. problems. This is likely to be a good thing to do for problems that possess many of the same characteristics as do A.I. problems".

Indeed in the months immediately preceding completion of this thesis the number of related articles published has greatly increased. These include a theoretical application of network flow models to image processing (understanding images such as speech waveforms is an important topic of A.I.) in Tso (1986), and the formulation of a pattern matching problem in terms of a dynamic programming model (Warwick and Phelps, 1986). Despite such interest, little seems to have been accomplished in incorporating Expert Systems into O.R.. Yet to many people, O.R. is simply the application of scientific techniques to problems in

industry, government and commerce. Further, O.R. techniques, emphasising the optimising quantitative model, have been questioned by many within the O.R. world (eg. Boothroyd, 1978). Phelps (1986), in talking about applying O.R. techniques and statistics in unclear medical problems states that "this is a situation were there is a strong case for a combination of the statistical, O.R. and A.I. approaches". Thus indicating the problems of using structured techniques to solve semi-structured problems. A similar conclusion is made in a comparison between O.R. and A.I. methods (Grant, 1986). Of all the areas of O.R. likely to benefit from Expert Systems, particular interest has been shown by the simulation community. A problem with discrete event simulation is its application to semi-structured problems (Rubens, 1979). In particular there is a need for development for good interfaces that "enable the decision maker to use his creative thinking and pattern recognition capacities to their maximum potential" (Moreira da Silva, 1982). This point is re-iterated by Radzikowski (1983) who states that expert systems "are designed for unstructured decision situations, and they can serve as a structuring tool. Therefore incorporations of expert systems into D.S.S. is potentially beneficial and should produce a decision aid of a superior quality". Further, it has been suggested that the ideal D.S.S. would take an active role in leading a decision maker to a decision (Benet, 1983). For this the computer system must have, amongst other things, a knowledge

based system. This view is echoed in a recent article by Grant (1986) whose conclusions include:

" (5) The use of A.I. planning techniques in simulations will be useful.

(6) The handling of decision rules in O.R. simulations is generally fixed. An embedded expert system would enable the simulation to incorporate an intelligent choice of rules to better model (human) decision making in the real life system being simulated."

de Swaan Arons (1983) considered the possibility of an expert model builder, feeling that the explosive market of microcomputers has "bought simulation within the reach of many, unfamiliar with the principles of simulation". Shannon et al (1985) cannot foresee the future of simulation without A.I.

"As we watch the latest developments in the areas of simulation ... and expert systems research, it is impossible to escape the conclusion that A.I.-based expert simulation systems will soon be available ... Indeed the future of simulation is bound up in the future of A.I. and the speed with which advances come".

"The necessary preconditions for the merger of expert systems into D.S.S. are met" (Radzikowski, 1983).

In talking about possible relevance of the A.I. approach to simulation Balmer and Paul (1986) state that

"Complex decision making on the part of human factors often plays a part in a simulation model. For instance, in

the simulation of the berth occupancy of a port, the complex allocation rules adopted by port managers must be adequately represented. These might be modelled in terms of an expert system component within the simulation model.

In a broader sense, a simulation model must itself encapsulate knowledge about some aspect of the (complex) system modelled, and it has been argued that the representation and access of this knowledge base itself constitutes a legitimate application of I.K.B.S. techniques."

Again, however, despite such interest little practical research has been performed. Following on from the research of PROLOG as a simulation language, the current trend towards PROLOG as an expert system language, and the large amount of interest within the O.R. community, it was felt necessary to research the benefits that could accrue by interfacing expert systems and simulations.

This research has been undertaken methodically, increasing the complexity of the tests after results from simple prototypes. Starting off by interfacing a PROLOG simulation and a PROLOG expert on one machine, the possibility of using two processors was then investigated. The test data used was a simple robotics problem because the area can be treated in isolation from outside influences. The research into the technical practicalities of interfacing expert with simulation, ended with considering the interfacing of a conventional procedural language based simulation (as currently used in industry) with a PROLOG

system. Because this is the first work of its kind, much attention has been made to consider the control aspects of the interfacing.

The advantages of linking expert systems and simulations has been given theoretical treatment by Shannon et al (1985). They feel that the advantages would stem from the differences in the two fields and that ...

"the primary ... (difference) ... is the desire to build into the modelling system most of the decisions that are now made by the simulation expert".

The benefits, according to Shannon et al, of incorporating expert systems within simulation are :

i) heuristics can be easily added to and modified.
ii) models cannot do anything that is not preplanned, as opposed to an A.I. based expert simulation system.
iii) expert systems also need simulations badly. Simulations can handle time dimension very well, but expert systems at present cannot.

This third point has also been put by O´Keefe (1985). Working on this problem has allowed the above advantages to be tested, as well as giving a greater insight into the facilities needed. Experiments on ´learning´ expert systems, control expert systems and general experts (in the field of

robot routing) have been conducted, all within the context of linking with simulations. The literature has been drawn upon to indicate what the expert must possess in order to gain user acceptance. The problems of remote provisions of such facilities as might be necessary for user acceptance, have also been investigated within this research.

As a result of this research at Warwick, it has been possible to propose benefits and draw conclusions that have not been previously available in the literature. It is felt that the possible applications of expert system methodology within discrete event simulation are much greater than previously envisaged. Indeed further work is now being undertaken by other research students at Warwick to continue the research.

In fact one aspect of this research has already been foreseen by Shannon et al (1985).

"Someone will devise a method of interfacing PROLOG like languages to existing simulation languages ... "

- CHAPTER 3 -

PROLOG AS A SIMULATION LANGUAGE

In the previous chapter current research was reviewed in areas critical to this thesis. In particular investigating PROLOG as a possible language for simulation was seen as both interesting in its own right, and as necessary groundwork for the research into the use of A.I. techniques within discrete event simulations.

Introduction

As stated in the first chapter, much interest has been expressed in the possibilities of using PROLOG as a simulation language. This link has been indicated by O´Keefe on talking about modelling the three-phase method:

"Rule based techniques, as evident in expert systems work, may provide a new method for writing and thinking about simulations ... A simulation is simply a set of rules ... such as

IF not (empty queue)

THEN remove entity

engage to sender

Thus available rule based software and techniques such as popular expert system languages like PROLOG may be useful simulation tools ... The important matter is that being able to define a simulation as a series of rules in an appropriate software environment may make it possible for

simulation modelling to employ the facilities afforded by such environments".

However, other than very small projects (see for example Adelsberger (1984), Broda and Gregory (1984)), little seems to have been done to test PROLOGs capabilities in this area. Theoretical relevant advantages to using PROLOG are :

i) Because PROLOG is basically a computer implementation of (first-order) logic, it facilitates a clear representation of the (logical) relationships between entities.

ii) Since PROLOG data and PROLOG programs have the same format, it is simple to develop a data-driven program that uses data from the last simulation time as program code for searching out the next event. The same attribute of PROLOG also means that the status of the model is easily changed. The model can be modified by adding data to the PROLOG system which is then interpreted as code. Crookes (1982) on talking about production rule structure states that:

"The program units ... are more atom like than those of other systems ... Amending such a program is as easy as such a task could be".

iii) Many managers have little experience of computer programming languages. It is important, therefore, to use a language that is easy to learn. PROLOGs relationship to logic means that its programs are simple logical statements of the problem to be solved, rather than algorithms describing how the problem is solved. In the experience of Clocksin and Mellish (1981) "novice programmers find that

PROLOG programs seem to be more comprehensive than equivalent programs in conventional languages".

iv) The PROLOG interpreters and compilers that are now available for most machines are based on the core found in Clocksin and Mellish (1981). This means that, if this core is adhered to, programs will be portable between different computers.

v) As previously described PROLOG is central behind the Japanese 5th generation computer systems project. Because of this PROLOG will soon be one of the most efficient high-level languages available. Although current PROLOG implementations are inefficient, it seems sensible to investigate its use now ready for use later.

Despite these advantages a reasonable question to ask is 'why bother ?'. There are already many purpose built simulation languages available (some 137 in 1982 - Crookes (1982)). In the context of this research several reasons are apparent:

i) Because most simulation languages are based on 2nd or 3rd generation languages, their efficiency on newer machines whose architectures (eg. vector processing) have demanded new languages, leaves a lot to be desired. In short, for efficient use of modern and future computing resources, the languages are out of date.

ii) By investigating whether PROLOG is suitable as a simulation language package base, we are trying to introduce new fields (such as A.I.) into the area of decision making.

Advancing O.R. into new fields is important if O.R. is to continue to be of any importance.

iii) The general area of this research has been investigating how well the discipline of discrete event simulation lends itself towards its enhancement using A.I. tools and techniques. With many A.I. applications now being written in a computer implementation of logic called PROLOG, it is natural to ask how well a simulation could be expressed in terms of simple logical statements. If computer simulation could be naturally represented by a PROLOG program, it follows that the logic behind the simulation is well suited to many A.I. techniques. The best way to test the suitability of PROLOG to simulation logic is to write a working simulation logic in that language. This has led to the development of a general simulation engine which contains logic for discrete event simulations. The ease with which this has been achieved gives a good indication of any such suitability.

It was with this last point in mind that my simulation engine in PROLOG was developed. The theoretical relationship between PROLOG and the three-phase method (as outlined in the previous chapter and the above quote by O´Keefe) dictated the modelling technique used. In testing PROLOGs suitability for simulation logic it was decided to make the package as problem independent as possible, by separating the dependant characteristics into a separate file and having them processed by general procedures. The reason for

this decision was to make sure that the logic behind the simulations (the three-phase model, timing and so forth) was explicitly modelled. As well as wanting to see whether such modelling was possible, it was also important to determine how natural a tool PROLOG was. For the sufficiently determined any job can be achieved with just about any tools.

Note on the System Development

Clearly the system that will now be described was not developed in one version from scratch. As with all good software, the acknowledged method of stepwise refinement was adopted. This involved the development of a simple version first, which was then tested and modified until all bugs were removed. After this, complicating parts of the system were added one by one, each one checked before the next modification was introduced. This method of software development helps ensure very robust programs, since they have been tested from the core outwards. The logic on which such programs are based is therefore tested as the program is built.

The Problem Dependant Section

When modelling using the three-phase system, large sections of simulation logic are independent of the problem under consideration. These sections form a core which co-ordinates the three phases, moves entities and their

attributes, and outputs results. This core forms the simulation engine which is described more fully in the next section. This section outlines how problem dependant characteristics are presented using PROLOG. An example of the use of the simulation engine, together with specific programming details is available in Appendix 3. For convenience the problem details are repeated here.

The Sample Problem

A small bar operates in a city centre and wishes to find out whether it employs enough bartenders and stores enough glasses.

There are presently two bartenders, each arrives for work promptly (within a few seconds of each other). Their duties involve serving customers and washing - serving having the highest priority. After a bartender has served 50 customers he may take a rest.

The customers arrive and then queue for service. After being served they drink up and then may play darts. After a rest a customer may either leave the bar or queue up for another drink. All glasses are washed immediately after drinking.

It has been found that 10% of customers play darts and that a customer may order anything from 1 to ten drinks at the bar. Other facts are:

number of glasses: 50

service time: random in range [0, 2]

washing time: random in range [0, 0.5]

drink time: random in range [5, 30]

customer inter-arrival time: random in range [0,1]

darts playing time: random in range [20, 60]

resting time (customer): random in range [10, 20]

resting time (bartender): 5 minutes

opening times of bar: 7pm - Midnight

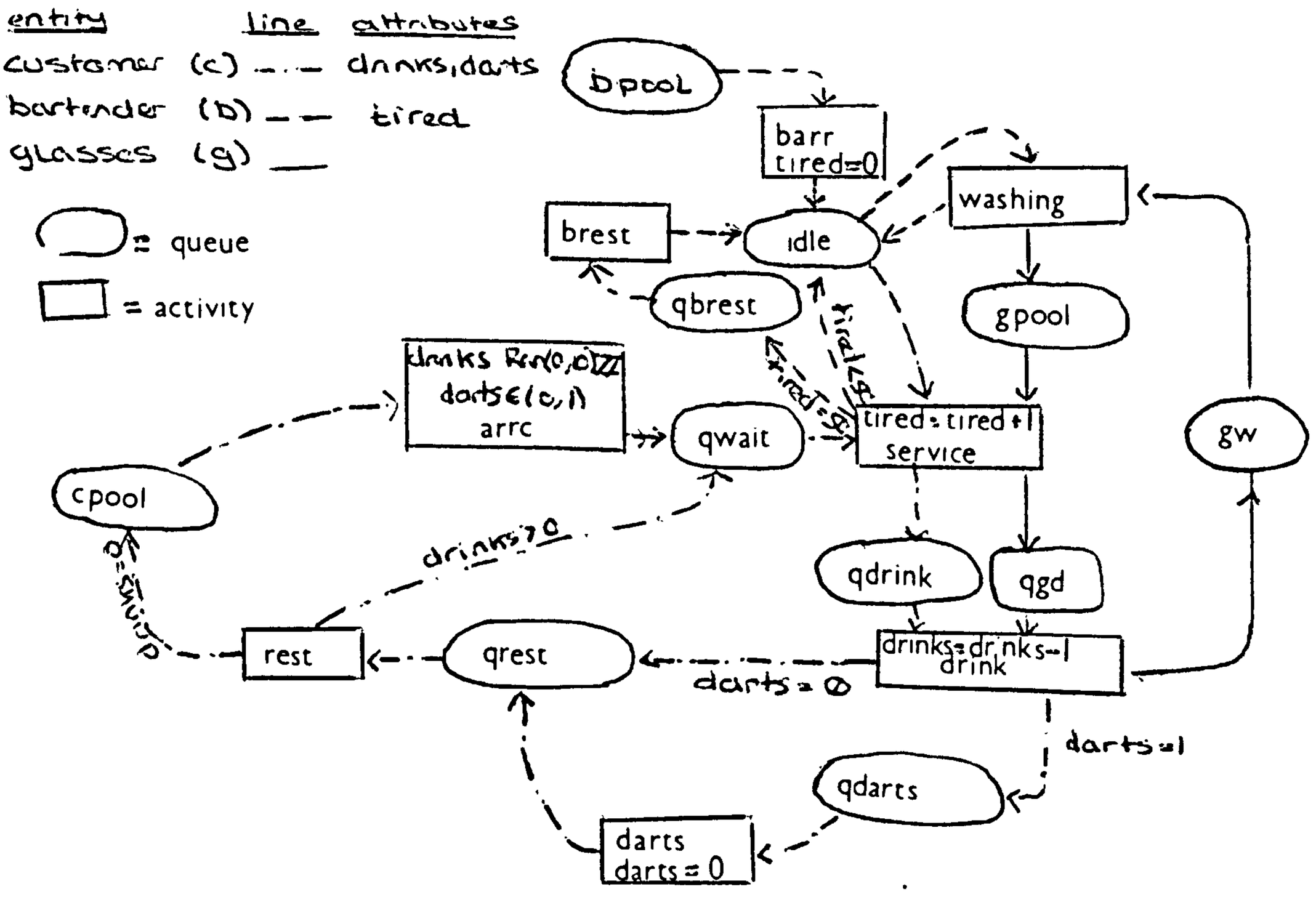


Figure 3-1 Entity Cycle Diagram For Bar Problem

When using the simulation engine the user need only concern himself with producing the problem dependant code. Such code falls naturally into two parts.

i) a part indicating the state of the system at the time

the simulation begins.

ii) a part indicating problem dependant logic, eg. the state entity diagram, attribute calculation and timing.

i) state information

As stated above, the predicates (facts) in this section indicate the state of the system at the time the simulation begins. In fact these predicates indicate the system state at each stage of the simulation. Thus the simulation engine could be viewed as a set of PROLOG procedures that update this database. The state information is simply recorded as a set of PROLOG facts (see Appendix 3) which define the following information about the problem.

- the entities
- the number of each entity in the system
- the activities and their priorities
- the queues
- the initial queue sizes.
- the queues whose sizes are to be output to the user
- the world pool queues (ie those queues for each entity which lie outside the simulated system and store unused entities)
- the names of attributes (if any) associated with each entity
- the initial scheduled events (including end of simulation)
- the number of realisations of each activity at the start of the simulation
- the number of realisations allowed for each activity at any one time

ii) Logic Information

Whilst, for simple problems, most discrete event simulation logic is independent of the actual problem, certain elements will vary from problem to problem.

1. The State Entity Diagram. This consists of a series of PROLOG facts which link the queues and the activities in their correct logical sequence. As such it was found that it can logically be defined in two parts:

a) those sections where a queue (or several queues) leads into an activity. This is defined by the 'quact' database.

b) those sections where an activity leads into one or more queues. This is defined by the 'actqu' database.

Complications occur when a conditional branch is encountered, so at first just simple cases were considered.

a) quact(Queue List, Activity).

This states that each of the queues in the Queue List are queues into Activity, eg.

quact([q1(a),q2(b),q3(c)],act(a,b,c)) represents :

(a b c are entity names)

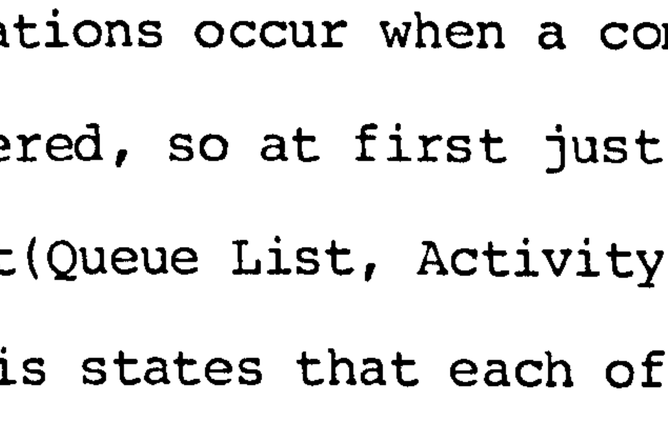


Figure 3·2: 'quact' section

The logic of the simulation engine dictates that for 'act' to be able to start 'q1', 'q2' and 'q3' must all be non empty (this is checked in the main simulation engine).

b) similarly 'actqu(act(a,b,c),[[q4(a),q5(b),q6(b)]]).' represents

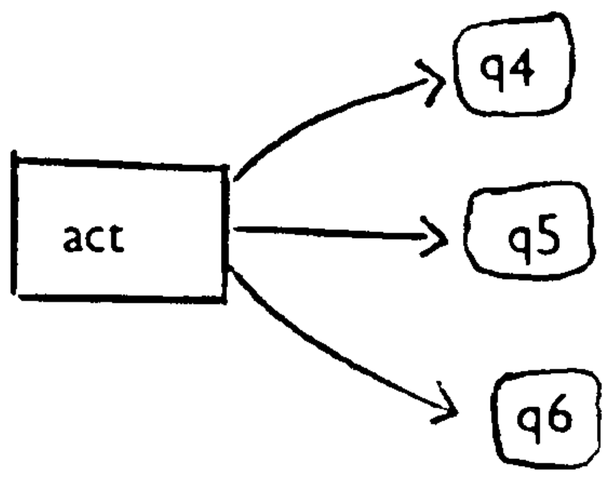


Figure 3-3 'actqu' section

For simple problems such a representation is sufficient to describe the whole state entity diagram. Most problems to be simulated, however, are more complicated, because many entities will have attributes which determine their path through a state entity diagram. For example, from the sample problem above we have

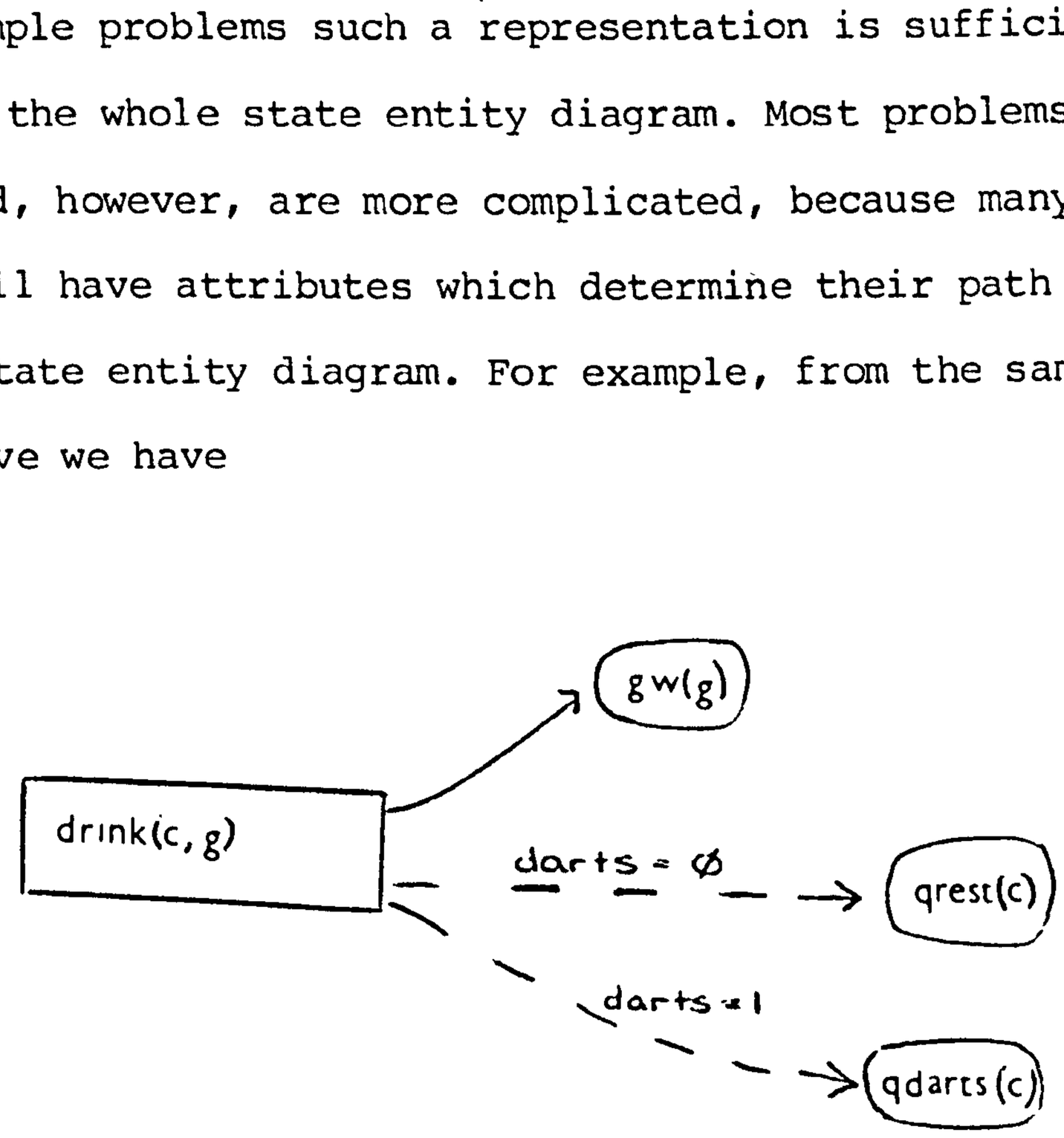


Figure 3 4 Conditional Branch

In this case a branch decision is made from an activity. One simplifying assumption made for this simulation engine is that all branch decisions are made from activities. Any decisions from queues can be simply converted into decisions from activities by inserting a dummy activity.

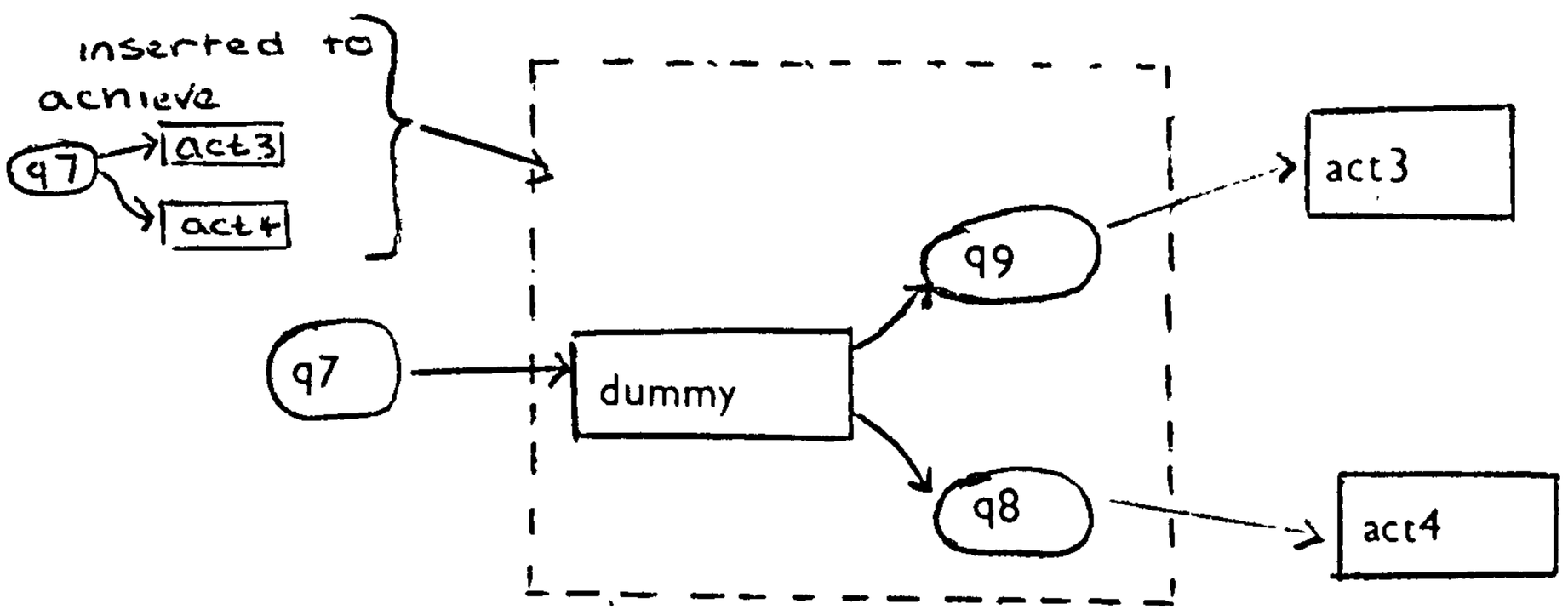


Figure 3·5 Inserting a dummy activity

Because of this assumption, the added complication of branching was to affect only the 'actqu' database. The solution adopted has been the inclusion of a condition in the right hand parameter of the 'actqu' database, eg.

```
                                    first queue list
actqu(drink(c,g),[[[darts,'=',0],qrest(c),qw(g)],[qdarts(c),
                  qw(g)]]).
                  condition
```

A condition (if present) is the first element of a queue list and is always of the form

[attribute,operator,constant],

where operator in {'=','<','>'}

If a condition is found to be true, the queues in the rest of the queue list are used. Otherwise the next queue list is used (and the conditions tested if necessary). Extensions to this have been developed, which allow for multiple decision points at any activity, thus making the system completely general. For example

```
actqu(act(a),[[[at,'=',0],q1(a)],[[at,'=',1],q2(a)],
                [[at,'=',3],q3(a)],[[at,'>',3],q4(a)]]
```

represents:

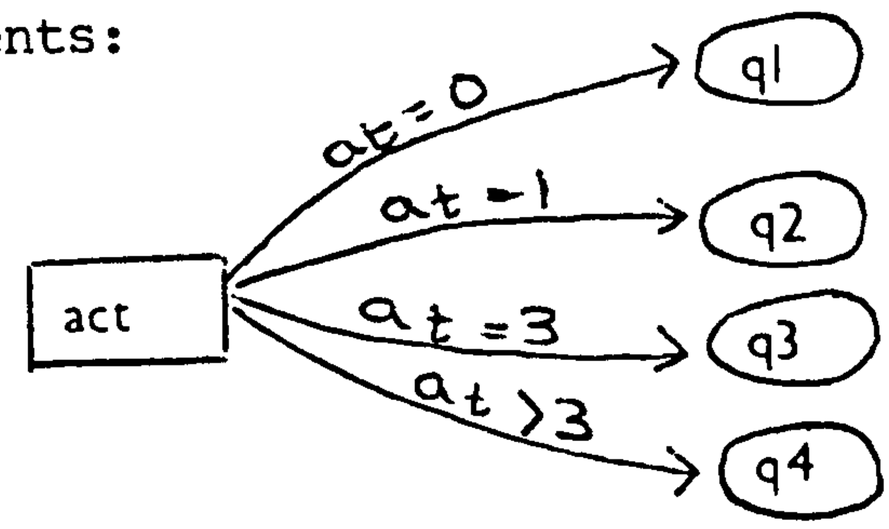


Figure 3.6 A General Branch From An Activity

2. Attribute Calculation. For non trivial examples the user needs the ability to specify where and how each attribute is calculated. This is controlled by the 'attribute_calculation' database which has the following form.

```
attribute_calculation(attribute,activity where calculated,
                                procedure where calculated).
```

For example take from the above problem the 'darts' attribute. This is calculated in two activities - 'arrc' and 'darts'. Thus we have

```
attribute_calculation(darts,arrc,ardac).

attribute_calculation(darts,darts,dadac).

ardac :-  random(N), NN is N/10, Nl is fix(NN),
          M is Nl+1, assert(attribute(M)).

ardac :-  assert(attribute(0)).

dadac :-  assert(attribute(0)).
```

The names of the procedures where attributes are calculated ('ardac' and 'dadac') are arbitrary, but must be unique and included in the problem dependant section.

3. Timing of Activities. As with the procedures outlining attribute calculation above, the user must also define the timing of events. In the main simulation engine there is included a random function (providing a random number between 0 and 100). The user defined timing procedures are of the form :

```
time(event,T) :- calculation of T
```

For example,

```
time(drink(c,g),T) :- clock(X),random(N),Ti is N/4,
                      T is X+5+Ti.
```

As can be seen from Appendix 3, the problem dependant section is very easy to write. As such it would be possible to write a program generator to help the user produce it. However, such a task falls outside the scope of this

research. This is because the research is concerned primarily with the introduction of A.I. techniques into simulation, rather than concentrating on automating just one aspect (such as simulation in PROLOG).

The Underlying Simulation Logic (The Engine)

In developing the simulation engine, several key factors of simulation logic were identified. These were :

i) the three phases

ii) attributes

iii) textual output of results

iv) graphical output of results

v) interaction

It was these points that needed to be translated into PROLOG code. Their implementation is dealt with in this section, together with any pitfalls apparent when using PROLOG. The details omitted here are chiefly simple data processing routines that handle PROLOG structures. A full listing of the PROLOG simulation engine is available in Appendix 3.

a) The A and B Phases

The A and B phases involve finding the next event to be completed, advancing the clock to that time and moving the entities that have just been processed. If the end of simulation is treated as another event, this PROLOG section can also detect end of simulation.

The first stage in considering PROLOGs suitability for this (and any other section) was to attempt to define the A/B phase in terms of a simple logical statement or production rule. Considering this lead to the following production rule :

IF <executing a/b phase>

THEN <find next event to be completed> AND

<finish that event and move on entities>

<find next event to be completed> is performed by a PROLOG predicate 'search_event(Y,Z)'. This first checks a database containing all the scheduled future events as a set of PROLOG facts. These are of the form

event(<event name>,time).

The <event name> with the smallest time is taken and its event fact deleted from the database. The current clock time (another PROLOG fact - see 'Problem Dependant Section') is changed and a check is made as to whether this event is the end of simulation (when <event name> = end). Thus after calling, Y is set to the name of the latest event and Z is set to zero, unless the end of simulation is reached.

All that now needs to happen for these phases is either the end of simulation or the moving of the relevant entities involved in the event to the successive queues. This is

controlled by another PROLOG routine 'bb_ph'. This causes exit from the simulation if necessary, otherwise, by looking at the entity cycle diagram for the problem it moves on the entities. Thus in outline, the A/B phases are represented in PROLOG as follows :

```
b_phase(Y,Z) :- search_event(Y,Z),bb_ph(Y,Z).

bb_ph(_,Z) :- Z==1,fin.

bb_ph(Y,_) :- moveon(Y),!.
```

This outline provides the model for the A/B phases as implemented in the PROLOG simulation engine. Slight complications occurred with the introduction of attributes, but this will be outlined later in this section.

b) The C Phase

Having modelled the A/B phase logic, the engine next needed to check whether any activities following the B phase could now be started. This is the C phase of the simulation. As with the A/B phase, implementation in PROLOG was relatively simple. Essentially this requires the simulation to pick up the list of all activities in the system (from the problem dependant section), stored as a PROLOG fact, and process each activity in turn. If an activity can be started (checked from the entity cycle diagram), an event is scheduled and its event added to the database. Events are then moved from queues to activities. Timing calculation for an event is determined via procedures written in the problem

dependant section. Use of PROLOGs recursive control was made to provide a priority system for activities. This is achieved because activities are processed in the C phase in the order in which they appear in the activity list (see Appendix 3). Once all the activities have been checked, the C phase is complete and the A/B phase may start again.

The three phases form the core of simulation using PROLOG. The simplicity with which this was achieved is a testament to statements made by Crookes (1982) and O'Keefe (1984) (in above introduction) amongst others, indicating the possible relationship between the three phase approach and production rules. On the debit side, as shall be explained later, the introduction of complicating features such as attributes, illustrated some of PROLOGs limitations as regards data manipulation.

c) Output of Results

The system for textual output (see below for graphics facilities) was purposefully kept as simple as possible. This was so as not to distract from the main aim of testing PROLOGs ability to manipulate simulation logic efficiently (as opposed to simply wishing to write a new simulation language).

Essentially, this facility prints each queue length neatly on the screen after each C phase together with the current clock time and the name of the newly computed event. Because there may often be many queues the user is not interested in, the queues to be printed are indicated as a

list in a PROLOG fact as part of the problem dependant section. As with many features of the simulation, this is basically a recursive process, the PROLOG system continually backtracking to obtain the queue sizes of all the queues in the list in turn.

Stated simply, the PROLOG implementation of this facility was programmed as :

```
record(Y) :- time_output,act_output(Y),
             queue_output,nl.
queue_output :- queue_list(L),member(Q,L),
                qu_output(Q),fail.
queue_output.
```

where,

qu_output(Q)	neatly prints the size of queue Q
time_output	neatly outputs the clock time
act_output(Y)	neatly outputs the name Y (the activity)

This provides output such as the following (the banner is generated by the 'initialise' predicate - see next section).

TIME	ACTIVITY	cpool	gpool	qw	idle
.055	barr	23	50	0	1
.1	barr	23	50	0	2
1.26	arrc	22	49	0	1
2.34	arrc	21	48	0	0
2.97	service	21	48	0	1
3.09	service	21	48	0	2
...	...	..	..	.	.

d) Three Phase Coordination

The basic model for three phase simulation, with results output could be controlled in one of two ways:

i) via the 'simulation' predicate as follows :

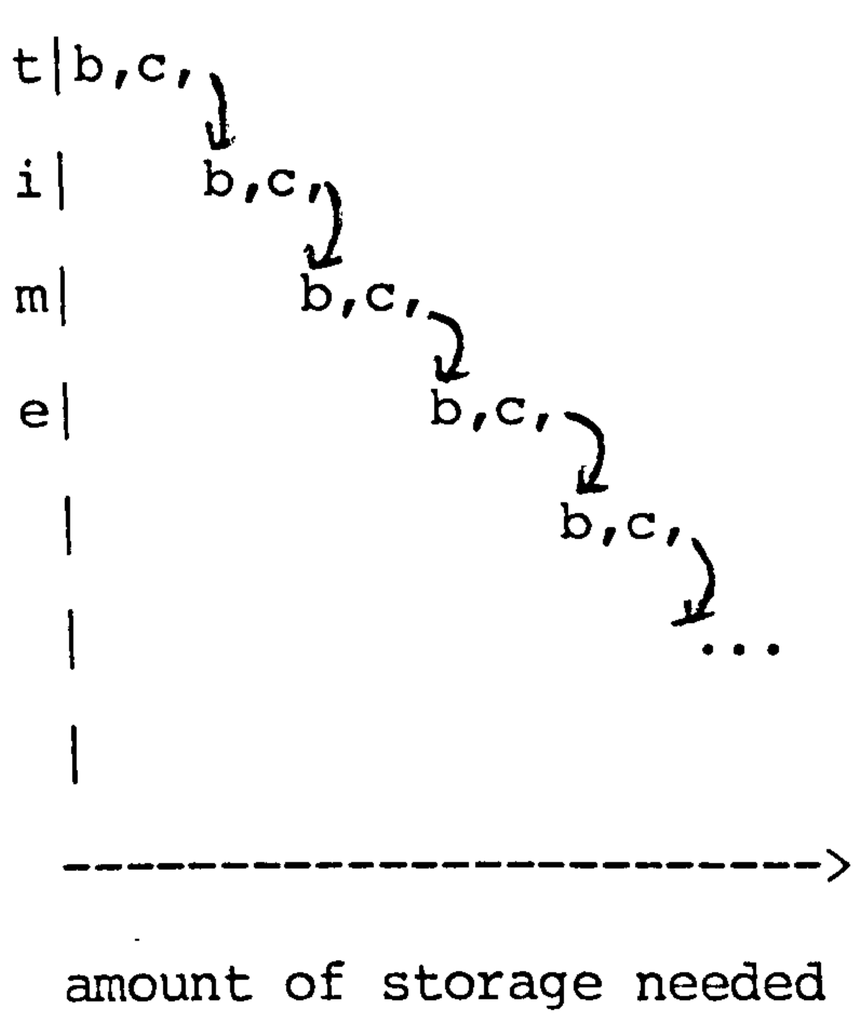
```
simulation :- initialise, update.

update :- b_phase(Y,Z),updtl(Y,Z).

updtl(_,Z) :- Z==1.

updtl(Y,_) :- c_phase,record(Y),update.
```

This produced the desired cycle between the A B and C phases, but had problems due to the iterative nature of the 'update' procedure. 'update' would continually call itself in a recursive manner until the end of simulation time was reached. This caused pointers to be retained in the PROLOG system pointing to code that would never be reused. As the simulation ran, space became more constrained and the search

time for that space increased. Eventually space would run out and the simulation would crash. For example, assuming the following (simplified) version of 'update',

```
u:-b,c,u.
```

then the procedure would execute as follows :

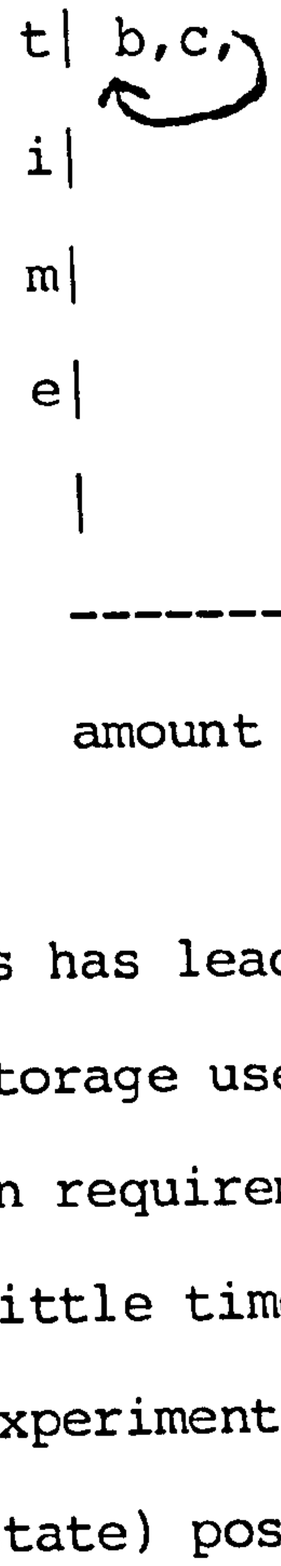


amount of storage needed

ii) to use PROLOGs backtracking together with the 'repeat' predicate, ie.

```
u:-repeat,b,c,fail.
```

With 'b' and 'c' incorporating cuts to control the backtracking, this has led to the following pattern of invocation :



This has lead to a huge saving in storage and speed.

- storage used in first iteration is reused, with no growth in requirement
- little time spent searching for space
- experiments indicate infinite simulation time (steady state) possible.

Thus we now have the following modified simulation control predicate :

```
simulation :- initialise, update.

update :- repeat,b_phase(Y,Z),updtl(Y,Z).

updtl(_,Z) :- Z==1.

updtl(Y,_) :- c_phase,record(Y),!,fail.
```

As the first phase performed is the b_phase, this predicate assumes that in the problem dependant section some starting activities have been scheduled. If, however, no such activities are present then the simulation must first

execute the c_phase to generate some. Thus the logic for the simulation becomes

```
sim :- initialise,c_phase,update.
```

The ease with which this extension to the simulation engine has been made illustrates how well the PROLOG modular production rule construction allows for efficient program modification.

e) Attributes

In any but the very simplest of problems, entities in a simulation have attributes associated with them. These attributes set out to distinguish differences between entities of the same type. In the simulation of a night in a public bar for instance (see above), one entity type is customers. However, not all customers are the same. Some drink more than others, some play darts and so forth. These attributes dictate the entity's path round the state entity cycle. Such attributes need to be :

i) calculated at specific points in the simulation

ii) permanently associated with specific entities

iii) used to determine the entity's path at decision points

iv) deleted as soon as the entity leaves the simulation (and re-enters the world pool).

It is these characteristics, therefore, that needed to be tested for ease of implementation in PROLOG. The definition of state entity cycles has already been discussed in the

problem dependant section as has attribute calculation definition. The incorporation of attributes into the underlying simulation logic at first seemed to necessitate great changes. In reality, however, this proved not to be the case. Just as entities are moved between elements of the state entity diagram, so attributes can be moved between PROLOG data structures. Two approaches were possible

i) each entity could be replaced with a list of attributes

ii) attributes could be contained in a separate list and moved around data structures parallel to the state entity diagram.

Since the engine as so far written moves numbers of entities and not individual elements, the former approach was not feasible on this engine. Hence the latter approach was adopted. The structure used to store attributes is the 'attribute_list' database. This is of the form

attribute_list(queue or activity,attribute,attribute list)

eg, attribute_list(qwait,darts,[1,0,0,1])

No assumption is made here about the maximum number of attributes per entity. Initially of course none of the attributes have any value, and so the above database at the start of the simulation is of the form

attribute_list(queue or activity,attribute,[]).

This initial database is created in the 'initialise' predicate.

The basic manipulating procedures needed for attributes are:

i) removing an attribute from the front of a list (after an event is completed) and adding it to the tail of the following queue. This caused modification to the b_phase and was found naturally to form part of the 'moveon' predicate.

ii) calculating an attribute, if necessary. From the problem dependant section the simulation engine can determine when (and how) an attribute value need be calculated or modified (for example incrementing or decrementing). Attribute calculation forms part of the c_phase. The attribute value gets inserted into the 'attribute_list' third parameter in the position dictated to by its entities service time. This required the provision of some PROLOG sorting routines and proved to be the most un-modular part of the whole simulation implementation. This is due to the fact that this method of attribute handling was chosen principally to fit in with the rest of the simulation engine. It did, however, ensure that whenever an activity was completed it was always the front attributes that were moved.

iii) to use the attributes to dictate the path through the entity cycle. The logic for this has already been discussed in the problem dependant section above.

f) Interaction

One inherent feature of the PROLOG language is the ability to interrupt the execution of a program at any time. New commands and database changes can then be tried before resuming the execution at the point at which it was stopped. The interaction facilities developed for the engine make use of this. It works by providing the user a simple to use menu. This menu is connected to a suite of pattern matching PROLOG predicates which change the relevant problem specific parts of the database, after requesting the relevant information. Facilities available are:

i) changing time at which recording starts

ii) changing an entities population

iii) changing a queue size

iv) changing the end of simulation time

v) changing the no. of allowed concurrent realisations for an event

vi)changing the entity cycle diagram

- delete part of cycle
- replace part of cycle
- add to cycle.

The interaction facility is designed to be easy to use (see Appendix 3 for an example session and listing of the facility).

One simplifying feature that may be hard to overcome is that no attempt is made to verify the consistency of the changes. Despite this, with care it is possible to radically

change the problems characteristics whilst the simulation runs. PROLOGs modular production rule structure means that it is a natural language to be used for such database changes.

Most of the interaction facilities available with this simulation engine are also obtainable on other simulation languages. One exception is the capacity to modify state entity cycles whilst the simulation is in progress. It is because of PROLOGs mixing of data with program code that such a change is possible. To change a piece of program the system needs only to regard it as a section of data that can be deleted. Similarly new program code can be written by regarding it initially as data added to the database.

g) Visual Display

A problem with the simulation engine as described above concerns its reporting of results. The information it provides is limited and not very user friendly. To overcome this, a suite of assembler programs have been written to allow moving graphics of the type available in visual interactive simulation systems. PROLOG systems have the facility of allowing communication between PROLOG and assembler code. Thus it is possible within PROLOG programs to control peripherals by simply calling the correct assembler. A suite of assembler programs has been written in the course of this research that allow lines and shapes of various colours to be drawn on the screen. With the PROLOG

program continually updating the picture, an impression of movement is obtained.

Flexibility of the Simulation Engine

From the research described above, it is seen that for basic discrete event simulations, PROLOG is a good medium to contain the logic. It was decided that it would be instructive to test the flexibility of the simulation to subtle changes in the logic. Two cases were considered.

i) future demand being computed using another program, written in some other high level language.

ii) dynamic programming of future events, eg. all future events can change depending on the current simulation state.

Both these logic changes amounted to the need for a revised 'c_phase' logic. Recall that the C phase of a simulation calculates any events that can take place and moves on the relevant entities. The new C phase had to do the same overall items but in a different way.

i) Future events were calculated by a PASCAL program which generates a text file containing a set of PROLOG facts of the form :

f_event(<queue>,<time>).

(see Appendix 7).

This file is then consulted by the PROLOG system, adding the future events to the database. A future event will have just been requested if and only if an f_event database entry

exists for a time between the current clock time and the previous clock time. If no such entry exists, the 'c_phase' simply moves on the entity to the next event to be completed. It was noted that the 'f_event' database needed also to be modified by the 'c_phase' - removing any 'f_event' entries which have been satisfied.

ii) If a new event has just been requested then a call needs to be made to a future event schedule routine (or expert system). After this program has generated its answer, the 'c_phase' converts it into a series of events for the 'event' database.

The problem area considered on which to do the experiments was one of an Automated Guided Vehicle (A.G.V.) needing to traverse a map. The choice of this problem area also tied up with early work on an expert-simulation link. This work is described in the chapter 'Controlling PROLOG Simulations using Expert Systems'. Nodes on the map could generate demand for a visit by the A.G.V.. This demand would lead to a set of events moving the A.G.V. along different arcs of the map. When new demand occurs it may be that a new route needs to be calculated for the A.G.V. and hence a new set of future events. This route calculation was performed by an expert system, but discussion of the simulation/expert interface will be left to a later chapter.

This is a simplified version of many flexible manufacturing system simulations.

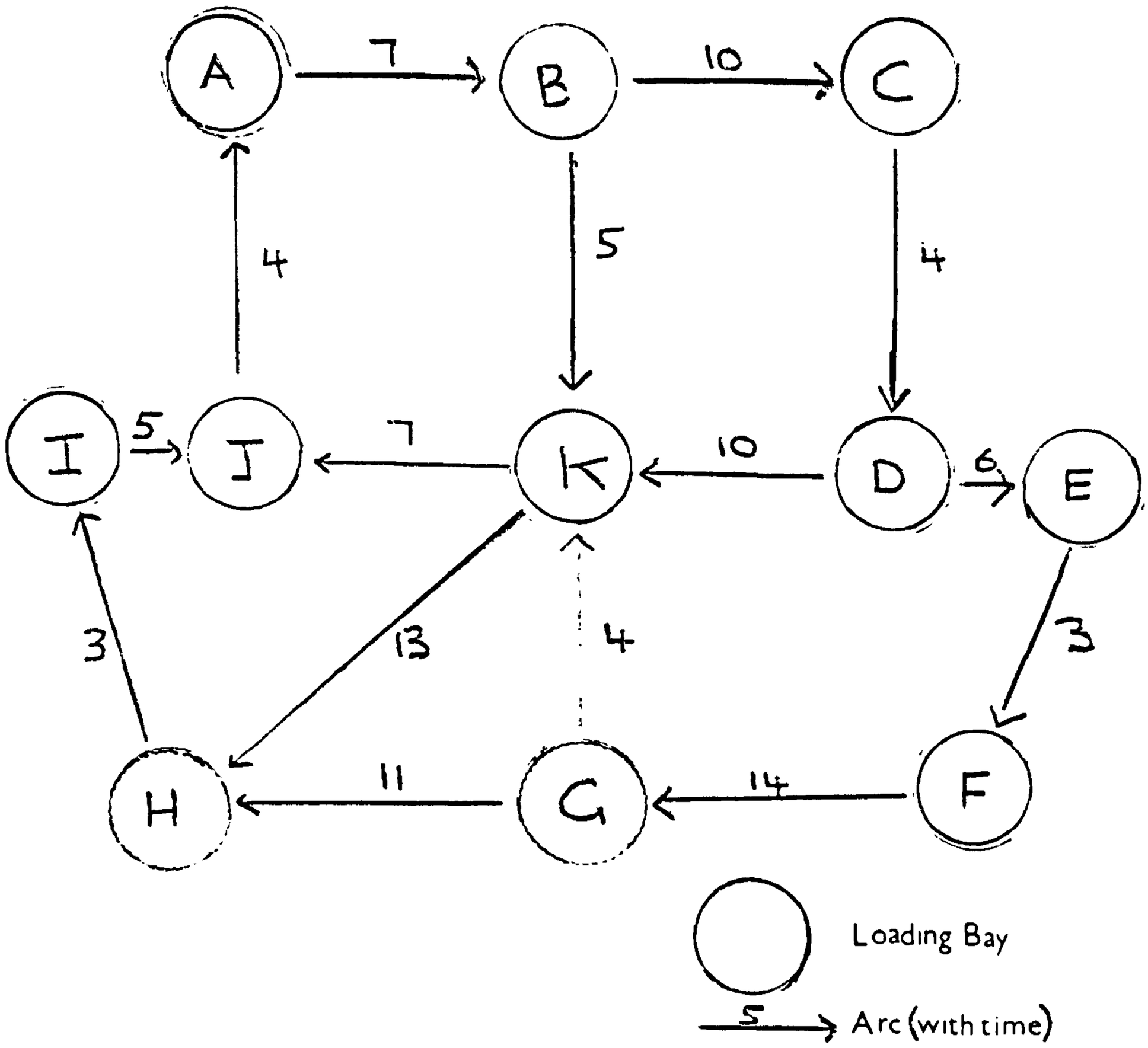


Figure 3·7 A Map Of the AGV System

As far as the simulation is concerned the points at which demand may occur are regarded as queues. The process of going to a specified queue (node) involves going along various defined routes or arcs which the simulation regards as activities. Whenever a new demand comes in from a node on the map, the A.G.V. needs to respond by changing its projected route to include the new destination.

The ease with which this was achieved indicates again the modularity of PROLOG and a close relationship between underlying simulation logic and production rule formalism.

Listings of the changes needed for this section are available in Appendix 7.

Conclusions

The results of this research have indicated that PROLOG is well suited both theoretically and practically as a simulation language. The natural partition by the PROLOG system of simulation logic and problem characteristics means that the simulation engine is very simple to use. Writing simulation programs is reduced to writing a few facts about the problem (and not the mode of solution). This is important, since as Tocher (1964) states:

> "for occasional use, a simple language, which is easy to understand and learn may be more valuable than one of the sophisticated languages that have many facilities, but ... (are) ... much more complicated to use and understand".

It is therefore possible to develop a program generator for this system (although it is not a matter for this research).

There are problems with the simulation engine: the arithmetic capability of PROLOG is very limited and simulations will at present run only very slowly. Even to these problems there appear to be simple solutions. Arithmetic could be performed via assembler code using PROLOGs interfacing ability. Also, PROLOGs slow speed is mainly due to its implementation at Warwick. Here we use the interpreted PROLOG-1, whereas it is claimed a twenty-fold increase in speed is possible using the mixed compiler and

interpreter of PROLOG-2. Of course, with PROLOG based machines being developed in Japan and the U.S.A., speed in the future should cease to be a problem.

Thus, although currently slow and limited in arithmetic capacity, the simulation engine compares favourably in potential with simulation languages and commercial packages such as SEE-WHY. Writing simulations is very easy and a high level of interaction is afforded. In particular the potential ability to change the state entity diagram whilst the simulation is running is a novel feature. This ability owes itself both to the use of PROLOG as a language, and the fact that an interpreter is used. This allows alternative paths to be tested on a single simulation run.

Perhaps the most important conclusion to be drawn from this work is that it implies PROLOG may well be suited to containing the controlling logic for simulations written in more conventional simulation languages.

The simulation engine defines the simulation modelling process in a very simple way, the engine working on a database largely consisting of a logical definition of the model. The impression is that A.I. techniques via PROLOG may indeed have some place in the field of discrete event simulation.

In emulating discrete event simulation methods using PROLOG the groundwork has been laid for investigating expert system technology within simulation.

- CHAPTER 4 -

PROLOG AS A SIMULATION CONTROL EXPERT SYSTEM LANGUAGE

Introduction

At the centre of the initial O.R. research proposal was the question of how and if Expert System technology could be usefully developed in such a way that it could be incorporated into the current use of discrete event simulations. This problem was tackled in several stages.

Although Artificial Intelligence system (especially expert systems) development methodology was covered during the literature search, little has been said formally about how to develop an expert system in PROLOG. With this in mind, the first stage was to develop an expert system in PROLOG of a simplified problem keeping as far as possible to accepted Expert System techniques. The problem area chosen was designed to be as simple as possible whilst having certain features in common with simulation manipulation.

Having gained experience on this problem a more realistic Expert System was developed. The initial (simplified) stages were conducted with an MSc student at Warwick (Barton, 1984a). This expert system manipulated the simple closed world of routing a single robot around a factory. This closed world enabled a simulation to be easily written, and entailed an expert which was as simple as possible.

The third stage in this part of the research involved linking the simulation with the expert such that the expert controlled the simulation.

In this chapter the work on the first two of the above stages with respect to the expert element is outlined. The development of the simulation of the A.G.V. (robot) problem using the PROLOG simulation engine is described in the chapter 'PROLOG as a Simulation Language'. The next section describes the first PROLOG system. Lessons learned from this development were invaluable for development of the second and consequent expert systems. It is for this reason that it is included in this thesis.

An Introductory Expert System

Introduction

To develop the principles on which future expert systems would be written, a simple problem area was chosen. For this first entry to Expert System development the problem area chosen was that of the game Mastermind. For this game the human user needs to think of a five digit number that the expert must guess. For each guess of the expert, two results are forthcoming:

1. the number of correct digits
2. the number of digits correctly placed.

With this information the Expert System should be capable of inferring better guesses next time. The Mastermind problem was chosen because :

i. it involved manipulating a set of numbers with the aim of optimising a score. In a similar way one may regard a managers interaction with a simulation as a manipulation of

a set of numbers (parameters) in order to achieve some aim (such as optimising output). The problems are clearly similar - the essential difference being that Mastermind is a more defined and clear cut manipulation of numbers.

ii. an unwritten rule of writing a computer program is to develop using 'stepwise refinement' - ie start on a simplified version of the problem and modify towards the final package. The rules of the Mastermind game can be easily modified to produce a simplified version of the problem.

iii. there was no need to find an expert, since the author of this thesis can play the game himself.

System Development

Initial work was greatly simplified, based on the following assumptions :

- the order of the digits was not important
- each digit could only be used once in the five digit number
- only information regarded as certain was used by the system.

With these assumptions, a first version was produced. This was written in the conventional algorithmic form for computer programs.

Second Version

Problems with this first version were soon apparent.

- very few production rules were in evidence. This lead to a system that was very hard to modify and it was not in keeping with accepted expert system methodology.

- the method of checking that new guesses were consistent with previous ones was hopelessly inefficient. It used an undirected search which used a lot of storage space and was very slow. In order to find the correct digits required in tests an average of 8 guesses (600 seconds) for a solution space of only 126. Further, its design was against the accepted structure for an expert system : that of using production rules.

With these drawbacks in mind the aim of the next version was to conform to the one unifying architecture of Expert Systems - it needed to be restructured into a series of production rules. In this example, as in the work done by DEC (see the chapter 'Research Background') there is never any conflict resolution needed. The next rule to be fired depends on the pattern in the database - the pattern matching the rules head parameter.

Each move is represented as a list of digits together with a score giving the number of correct digits. In processing moves and their scores, a pattern is generated which triggers the relevant rules. To decide which rule to trigger the system first analyses the move (and score) which are the newest, and produces a coded pattern consisting of a list of three numbers

[flag,number 1,number 2].

It is the flag which is used to pattern match to the production rules. This process of pattern matching is an important concept which in larger systems enables knowledge to direct the system towards a solution.

These patterns for the analysis-of-results production rules were stored as facts (p_match(X)) and a trace of previous rules executed (or rather their patterns) was stored as an ordered set of facts (p_m(pattern)). Rule execution for this stage was controlled by the 'anmod' predicate, which used backtracking to preserve storage space.

```
anmod :- repeat,retract(p_match(L)),rule(L),fail.
```

This resulted in 19 production rules for the analysis of results.

This produced a better and more flexible expert system, reducingthe number of guesses and time by about 30%. However, when it was decided to take away the first simplifying assumption, a new problem became apparent. By now requiring the order of digits to be taken into account the solution space was increased from 126 (9!/4!5!) to 15120 (9!/4!).

Although an improvement on the first expert system, requiring an average of about 7 moves (850 seconds) to obtain digits with their positions, this was still some 50% slower than the human. This latest version fails chiefly because of the time it takes (and hence the storage space) to find a solution. Its search strategy is basically a

random one amongst possible solutions which may or may not be consistent. Such a strategy whilst simple to implement is clearly highly inefficient. Yet whilst use is only made of certainty no other research strategy is feasible, without long cumbersome routines which detract from accepted expert system architecture.

It was thus decided to incorporate elements of uncertainty (´odds´ or ´probabilities´) into the system to allow efficient searching for solutions. Although Expert Systems which employ rigid probability theory do exist (noticeable Gashing´s uncertainty model based on Bayes´ theorem as employed in PROSPECTOR - Gashing (1982)), It was decided to use an ad-hoc form of probability theory, thereby following the more normal expert systems practice.

Probabilities were replaced with what we shall call ´inverted odds´. Their relationship is illustrated below:

```
probability(digit present) = 1/5 -> inverted-odds(digit) = 1
            "              = n/5 ->          "           = n
            "              = 1   ->          "           = 5
```

For ease of calculation inverted-odds were defined as follows:

```
inverted_odds(digit) = average(scores of moves in which
                               digit appears)
```

As a it was decided that 'probability' (ie odds) information should only be held about whether a digit is present, not where it may be present.

Conclusions of Study

The results of comparisons between the different versions of the expert system showed that in terms of speed the random search method is preferable to intelligent algorithms using directed search and probability information. Whilst both methods required an average of 7 moves, the latter approach required some 1125 seconds as opposed to 850 for the former. These results compare with an average of 7 moves (610 seconds) for the human. Of course the mastermind problem has a small solution space - a larger one could be expected to adversely affect the random search more than the directed ones. These results also illustrate the tradeoff involved between computation time and the probability of getting the right answer. It is relevant to note that the intelligent algorithms performed best when there was a lot of probability information. By this it is meant that there is a spread of digits along the odds scale of 0 to 5. When most digits had equal probabilities the 'intelligent' searches were very inefficient. This observation perhaps implies an integrating of the two search methods, using random search until sufficient information leads us to use intelligent search.

The other principal conclusion drawn, which was used in building the next Expert System concerned the use of production rules. These made system modification and improvement very simple. Such an ability is essential for

expert system development, since it is highly likely that modifications to the rule base would have to continually be made.

The A.G.V. Expert

Introduction

The problem area chosen around which to develop an expert system for investigation with simulation was that of routing an A.G.V. around a maze. The reasons for this choice were :

i) The problem was of a closed world which is easy to simulate. There are little outside influences which affect the problem (Barton, 1984b). Since this research is concerned with linking the technologies, it was important not to complicate the issue in the early stages with a difficult simulation.

ii) The possible use of PROLOG for certain aspects of robotics can be seen by looking at what robot languages cannot achieve (Soroka, 1979). These programming languages are deficient in a number of ways. They separate the control level from the user interface so that new algorithms cannot usually be tested. They communicate only stumblingly with external devices. They achieve completeness by requiring complexity such that simple things are often hard to do. They emphasis the procedural component of robotic programming making it difficult for example to program multiple arms working together.

The question of where the idea of using expert systems to control robots would be useful was put to Professor M Larcombe of the robotics laboratory at Warwick University. Larcombe thought the following :

i) optimising the route of robots through a warehouse would not significantly improve warehouse operations. The complexity of having several robots would be difficult to overcome. This view was given in the context of using procedural languages. I feel that with non procedural languages such as PROLOG, facilities for several intelligent robots could be developed. This has been shown later in this thesis (chapter 7).

ii) The concept of 'intelligent A.G.Vs' travelling through a maze may be more sensible in areas other than warehousing, eg restocking supermarket shelves, large scale catering services, cleaning airports etc.

The Problem

As stated in the chapter 'PROLOG as a simulation language' the problem concerns the routing of an A.G.V. around an airport. The A.G.V. must respond to demand from any of the nodes by choosing the best route that passes through all such nodes. The A.G.V. picks up a part at the node requesting service and transports it to the depot node (node A). The A.G.V. can carry more than one part, but it is subject to a maximum capacity.

A maze representing the factory was constructed with 11 nodes and 15 arcs. The nodes represent the various points where the robot might be instructed to visit.

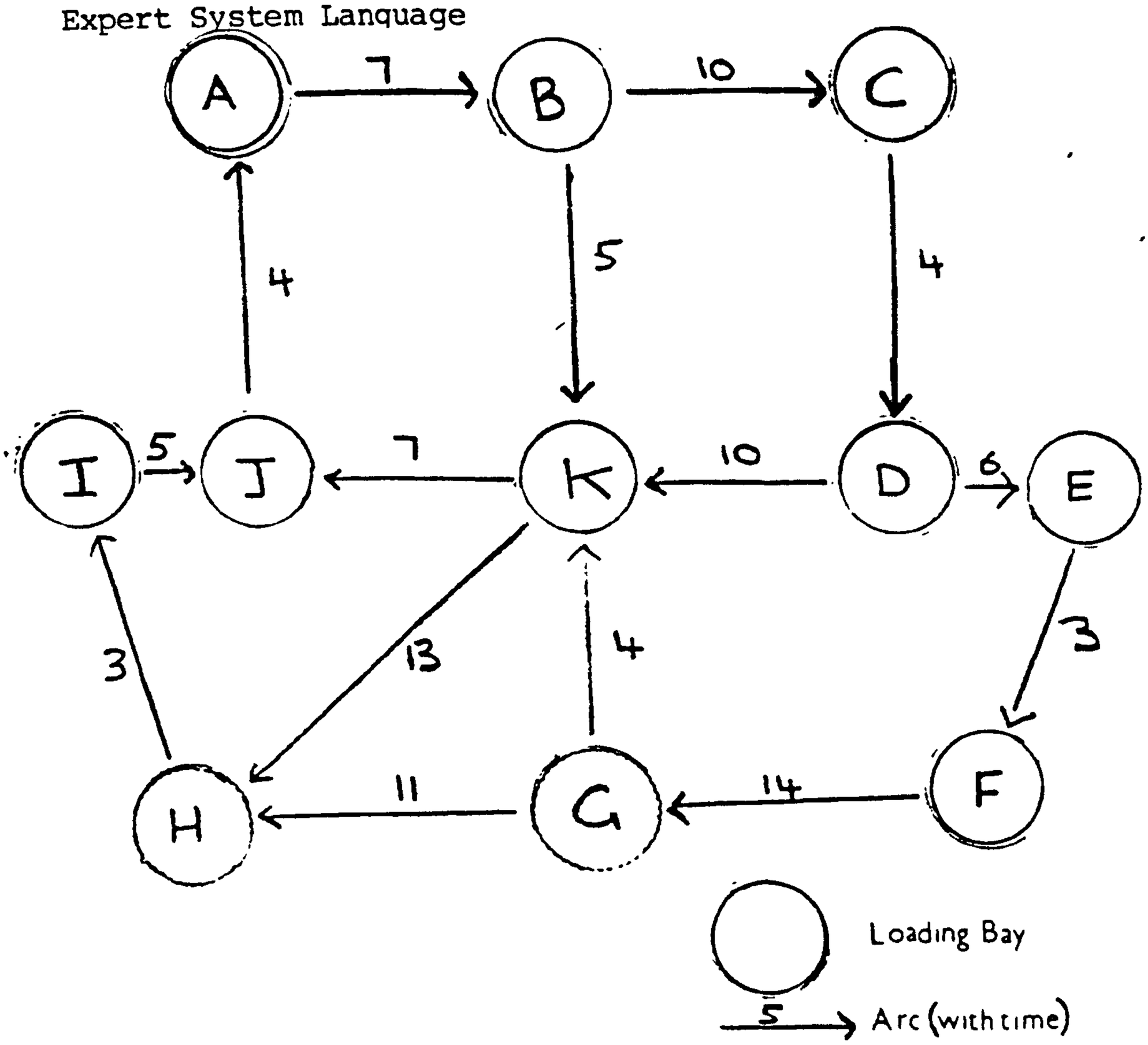


Figure 41 A Map Of the AGV System

The Basic System

A much simplified application was developed using the above problem (devised by an MSc student (Barton, 1984)). This initial project consisted of :

i) the PROLOG expert, used in conjunction with a hand simulation.

ii) a BASIC task generator, which produced random demand

iii) a hand simulation chart.

The hand simulation was used to keep track of what the expert was telling the A.G.V. to do. The BASIC task generator was used to generate random demand at nodes, which would then be input to the expert with other outstanding demand to generate a 'best route'.

The 'expert' system works using an exhaustive depth first search to find all routes between the current node and the destination node. A comparison is then made on the lengths of these routes to arrive at the shortest. Where more than one node has outstanding demand, the route chosen is checked to ensure that it passes through all the nodes that need the A.G.V. The map is stored as a set of PROLOG facts of the form :

arc(<from node>,<destination node>,<distance>).

(eg.

A -----5-----> B is stored as

arc(a,b,5).)

Several pitfalls were apparent with this expert system :

i) the search method used was exhaustive - ie little intelligence was inherent in the method.

ii) no use was made of production rules.

iii) no explanation facility was provided. For any true expert system, the user must be able to interrogate the system to find out how the system achieves its results.

iv) the algorithm does not allow routes to be calculated when the start and end nodes are the same.

v) the user was not allowed to direct the system towards a solution.

vi) in certain cases the algorithm simply does not work. The reasons for which are outlined below.

vii) the system does not respond to the A.G.V. getting full. It does not therefore redirect it to node A once a certain capacity is reached.

Removing the pitfalls

a) choice of search method.

It was decided to include four such search mechanisms, the last three of which may not supply the most optimal route, but could choose a route very quickly. The system can execute the required method using a system of pattern matching against the parameter of the fact

method(X).

which is input by the user. Pattern matching is against a series of production rules, which allows extensions to the number of methods to be easily made.

i) method(1). : exhaustive depth first search, which relies solely on PROLOGs automatic rule order conflict resolution. This is fine when the map contains only a small number of nodes, but in general it would be inefficient and slow.

ii) method(2). : maximum distance search. This is also depth first, but it automatically stops searching a branch once the distance exceeds a maximum set by the user as the parameter X in

max(X).

iii) method(3). : Set start search. This is useful when the user can see an obvious start to the route. The start is input by the user as the list parameter to

start(L) eg. start([a,b,c]).

iv) method(4). : Maximum distance set start search. This combines methods 2 and 3 and needs both sets of data from the user.

b) providing additional heuristics.

Some faults have been made apparent in the first version when attempting to use it. When asked to find a route from a start node to a finish node via various intermediate nodes the original algorithm will occasionally fail. It does so when no route is possible, but when a route would be possible if one of the intermediate nodes were swapped with the finish node.

Resolution of this problem was possible. Looking at the maze (above) it could be seen that the nodes were not labelled randomly. The writer of the maze tended to label the nodes in a rough order as illustrated in the diagram below. Such an ordering is common in these types of maps.

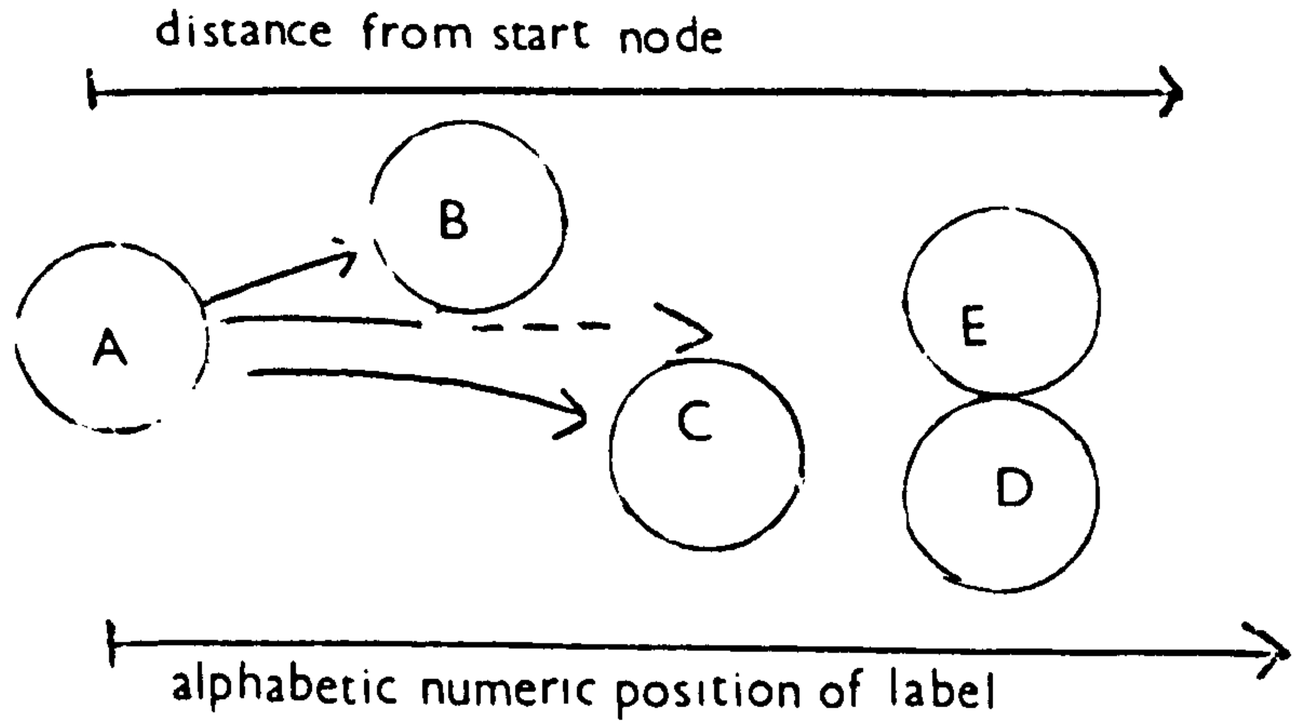


Figure 4 2 Ordering of Map-Labels

Whilst this order was not and cannot be precise it was still an underlying trend and a natural heuristic. If one assumes that labelling is always carried out in this way then it

becomes apparent that the original program fails when alphabetically some or all of the intermediate nodes do not lie between the start and finish nodes. This being the case I have devised a set of heuristics which finds the end (finish node) most likely to result in a found route, from the list of nodes to visit (there cannot of course be any flexibility in the choice of start node).

start node, S } find end node E E in N
other nodes, N }

IF there exists M in N such that M<S
THEN
E = max(L) (where L subset N, and R in N, and R<S<->R in L)
ELSE (ie if there does not exist M in N s.t. M<S)
E = max(N)

(in some cases these heuristics will not work, due to inaccuracies in its underlying assumption. In such cases an additional heuristic to find the end node E is required).

IF other heuristics fail
THEN try each member of N in turn until one succeeds

A further two heuristics have been developed to facilitate the sending of the A.G.V. to node A when it is full (or nearly full). These two heuristics require an extra

parameter to be passed to the expert - the current size of the A.G.V.´s load.

IF agv has ´near capacity or more´

THEN find the best route (R1) from the current node to node A, satisfy any demand along this route, find best route (R2) from node A going through any nodes where demand is outstanding, form overall best route as concatenate of R1 and R2.

IF agv has not ´near capacity or more´

THEN find best route from current node through outstanding demand nodes.

c) an explanation facility.

As outlined in the first chapter, one major problem expert systems must overcome is that of user acceptance. Programs that just act as a ´black box´ working on complicated problems do not inspire confidence. It is important therefore that the expert has the capacity to provide an outline of how it found the answer. Typically for expert systems, a trace is provided telling the user which production rules were used, in the correct order. With each production rule is associated some text, which can be printed out with the trace. This is the method adopted by me for this expert system.

Each time a heuristic (production rule) is used two facts are added to the end of the PROLOG database. These are of the form:

pred(<heuristic name>).

tprams(<parameters to heuristic>).

If the user asks how the expert reached a decision, the expert deletes these facts one by one, and matches the <heuristic name> (and parameters) of the deleted fact with a rule of the form:

text(<heuristic name>, parameters) :- <text>.

Thus there is text associated with each production rule. The user may engage the explanation facility by typing the goal 'how' after the expert has defined a route. Whenever the expert is asked to calculate a new route, it resets its tracing database so that only a trace of the most recent decision is ever presented to the user.

A full listing of this expert system, together with the small modifications outlined in the next chapter 'Controlling PROLOG Simulations Using Expert Systems' is available in Appendix 4.

This expert system has been tested and seen to work with reasonable response times. Its reliance on depth-first search does, however, mean that its performance on larger maps is degraded.

Conclusions

The expert system of the A.G.V. problem (together with the PROLOG simulation described in the previous chapter) has provided an ideal starting medium for the investigations in the rest of this thesis. The expert itself is simple, but has been constructed to contain all the essentials of an expert system (such as production rules, pattern matching, explanation facility). The work on the earlier Mastermind expert system indicated (as in common with much of the A.I. industry) that the incorporation of probability information into expert systems can slow rather than quicken expert response time. This is due to slow processing time and vastly increased expert development time. Perhaps PROLOG based machines will help change this situation.

Although the A.G.V. expert designed here is for the simple case of a single A.G.V. it has been shown possible to use it as the basis for a general type of A.G.V. expert that can cope with more than one machine. This is outlined in a later chapter 'Using Simulations with Expert Systems'.

Having developed expert systems and simulations in PROLOG, the next stage in the research was to integrate the two together. Initial work was carried out on a single processor. Later remote communication between expert and simulation was considered. This work is described in the next chapter.

- CHAPTER 5 -

CONTROLLING PROLOG SIMULATIONS USING EXPERT SYSTEMS

Introduction

So far in this research PROLOGs suitability to containing simulation logic has been investigated, as has also PROLOG as an expert system language. These two areas of work have laid the groundwork for attacking the central theme of whether expert systems could enhance current simulation techniques. Ultimately this would involve tests using computer simulations written in an industrial package. Yet the tools have already been developed in this research to enable a simpler starting point to be chosen. With the ability to write both simulations and expert systems in PROLOG it was easier to start with considering the merger of systems written in the same language. Indeed my initial experiments involved expert and simulation on the same processor. Lessons learned from that research were then applied when attempting to separate the expert from the simulation. From the results of this research the key area of linking expert with a procedural language based simulation could be tackled.

Thus the research presented in this chapter is concerned with the technical possibility of merging the technologies. Practical applications of such a merger could not be investigated until a realistic simulation environment was used. Yet all the work presented in this chapter was

undertaken with such future research in mind. Thus the expert-simulation interface had to be as easy to use and problem independant as possible.

The problem area chosen was that of the single A.G.V. routing problem. The PROLOG simulation and the problem are described in 'PROLOG as a simulation language' and the expert is outlined in 'PROLOG as a Simulation Control Expert System Language'.

Linking Simulation and Expert on The Same Machine

One of the main reasons for producing the simulation in the same language as the expert is the apparent ease of merging the systems together. This is because data types will be the same in both systems, and they will also share a common method for procedure calling. This advantage is compounded in PROLOG because it is an interpreted language. This means that both the simulation and the expert may be loaded into PROLOGs work area at the same time. Since an interpreter translates code only when it is being executed, the fact that it consists of more than one program is of no significance to the executing PROLOG system.

In order to gain the greatest insight into merging a simulation and expert it was necessary to make the codes as separate as possible. To facilitate this they were joined together via a single interfacing predicate in the simulation. This interfacing routine would be executed

whenever consultation with the expert is required. For the single A.G.V. problem, the expert was used to schedule new events and thus needed to be called during the C phase. The predicate forming the interface was called 'form new schedule' and it had in its simplest form the following structure :

```
form_new_schedule :-
<form parameters needed by the expert>,
<consult the expert>,
<initial translation of experts advice>,
<final translation into an executable recommendation>.
```

i). <form parameters needed by the expert>. For the A.G.V. problem the expert needs to be given a list of nodes to visit, as well as the start node.

ii). <consult the expert>. When expert and simulation are on the same machine (and written in the same language) as in the trial problem, this involves calling the expert predicate directly.

iii). <initial translation of experts advice>. This involves translation of the experts advice into a list of elements identifiable to the simulation. In the current example this produced an ordered list of future event names.

iv). <final translation into an executable recommendation>. This stage involves taking the partial translation and producing an executable form of the experts advice. In the

A.G.V. example this takes the form of producing a schedule of timed events.

This structure is fine for the most simple type of interaction between two systems. But in many cases the user may wish to communicate with the the expert in order to guide/speed up the decision process, or to question the method adopted, eg.

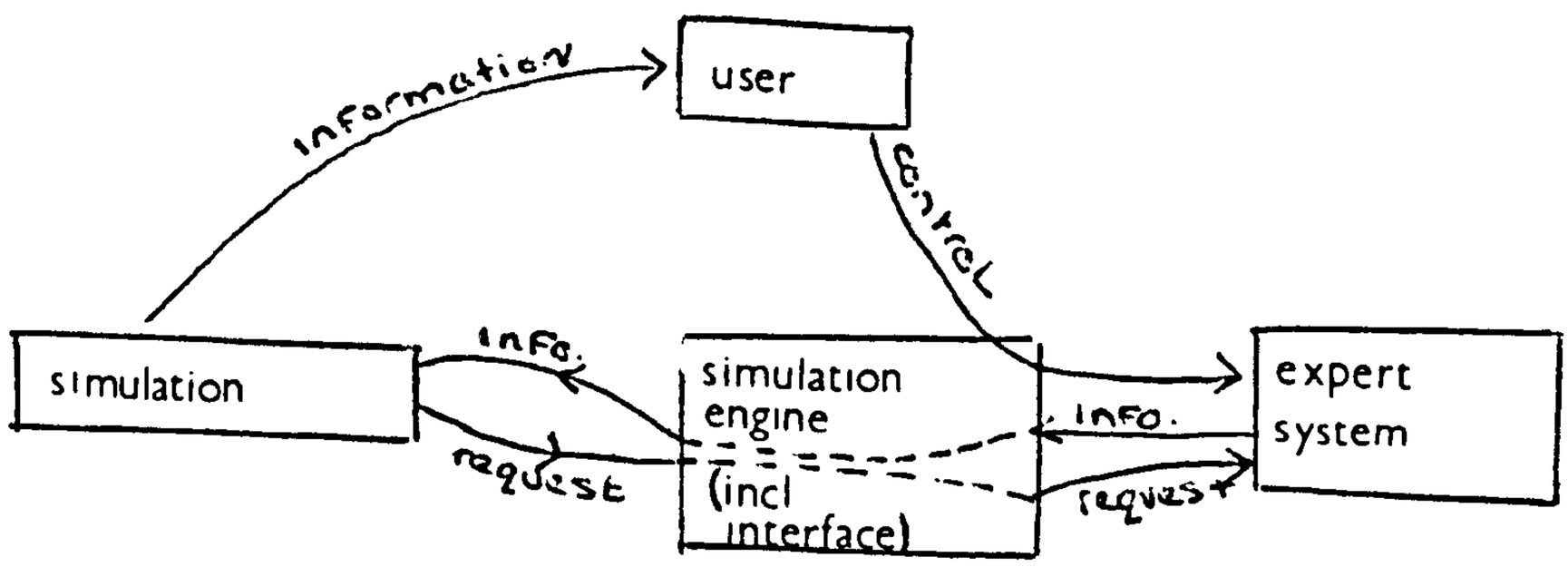


Figure 5·1 The Expert/Simulation/User Interface

Thus the expert needed greater refinement in the interface. From the A.G.V. problem the following was found to be the required form of such an interface :

```
form_new_schedule :-

<form parameters needed by the expert>,

<inform the user of the parameters and invite

interaction>,

<control experts facility for user interaction>,

<consult the expert (including the explanation

facility)>,

<initial translation of experts advice>,

<final translation into an executable recommendation>.
```

Such an interface allowed the user all the facilities available on the expert, including the choice of method (see previous chapter). With the interface as described above the expert was found to work well with the simulation. However, the expert in reality was just an integral part of the simulation program. All that was really achieved here was the separating simulation function from expert. In order to test the usefulness and plausibility of an expert-simulation link fully it was necessary to experiment linking these programs between different machines. The first result of this research was a working linking program which allowed communication between PROLOG programs on different IBM PC compatible machines. This link will now be described.

The Remote Communications Link

As the project currently described stands, it is possible to run a simulation in PROLOG and also write simple expert systems in PROLOG. Further we can run simulations which call expert systems within PROLOG on the same machine. Having proved this to be possible, several pitfalls exist :

- we are confined to having the simulation written in PROLOG. Although this is possible, it is at the present time a slow language.
- the expert needs to be on the same machine (or processor) as the simulation. Problems arise concerning storage space and expert availability.

To overcome these pitfalls it was felt necessary initially to write a link which would enable PROLOG programs on different computers to interact. It was anticipated that, with this link achieved, it would be possible to replace the PROLOG simulation with one written in another language. This simulation would still be able to interact with the expert system as long as it used the same protocol. It was considered that a similar situation could be achieved using a multi-tasking computer. This would alleviate the need for a special linking program to be written. However, with the current dominance of small microcomputers, it was felt that

the development of a link would provide increased relevance of this project to current simulation environments.

The link between computers was achieved using the asynchronous adapter connected to the serial ports of IBM PCs and compatible machines (such as the Olivetti M24s).

Communications via an Asynchronous Adapter

Communications from a PROLOG expert system via an RS232 data link to another program (a simulation) on a different computer is achieved using a modified version of the standard ^S/^Q protocol (also known as DC1/DC3 or XON/XOFF) for handshaking (Sargent, and Shoemaker, 1984). The code for the inter-processor link is written in the assembler language MACRO 86. Assembler is used because it is a language at a lower level than conventional programming languages. This means that the user has greater control on the use of computer resources and devices. On the debit side it is a much harder type of language to program with. Indeed conventional languages are converted into Assembler by the first part of a compilation program.

A schematic diagram of the interplay between expert system (slave) and simulation (master) is as follows :

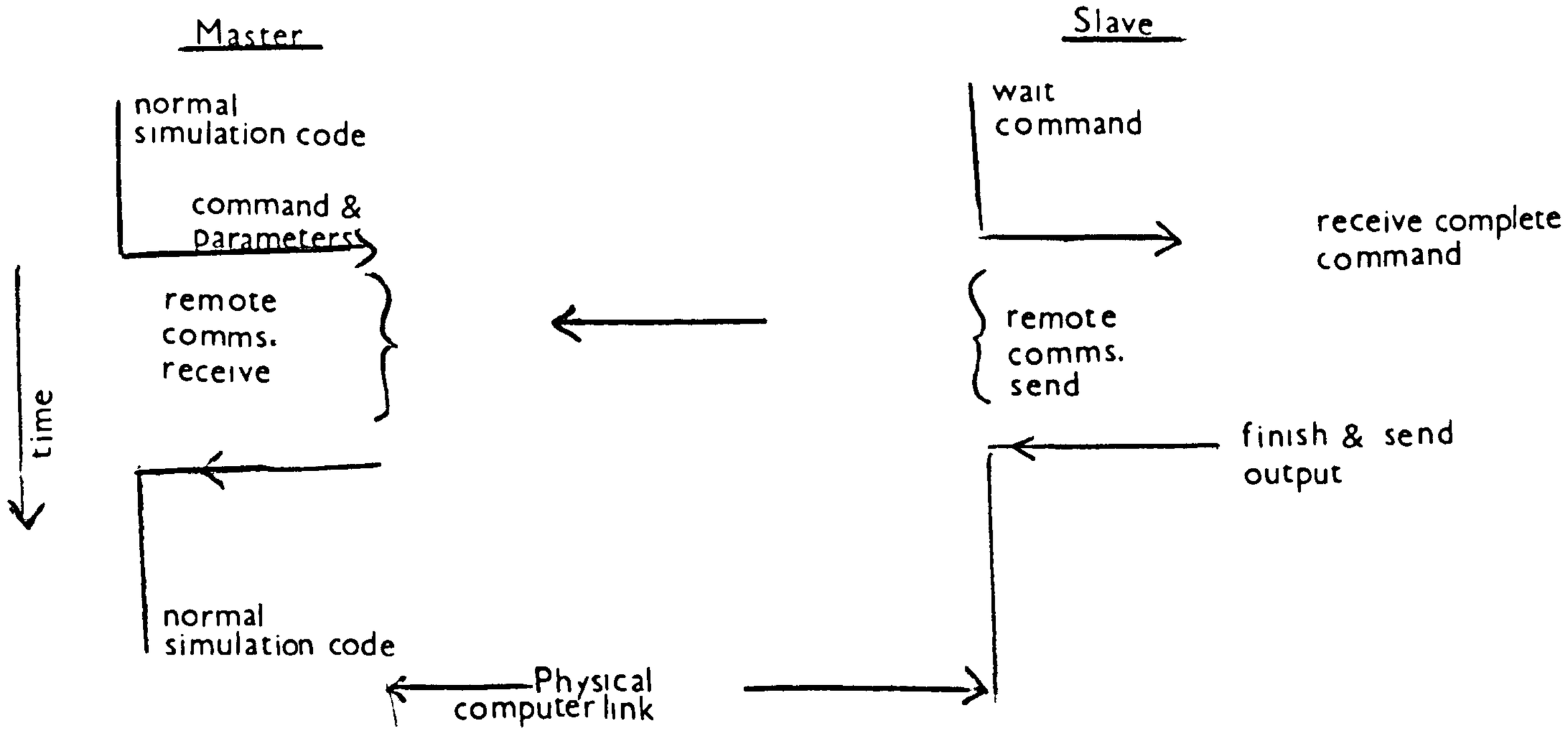


Figure 5·2 Slave/Master Communication

The slave waits to receive a command from the master. The command consists of a predicate (name) followed by input parameters. This command is then executed by the slave, whilst the master awaits its completion. During command execution various forms of interaction between master and slave may take place. When all interaction for the command is complete, the slave informs the master and sends him any results. Specific details of how synchronisation between the computers is obtained and how PROLOG communicates with the assembler are in Appendix 5.

The PROLOG communications protocol PROLOG programs operate on a variety of data structures. In order for a master program to interrogate a slave, it is necessary for that master to send commands and parameters to the slave. The command would be the name of a PROLOG predicate. The parameters may be integers, atom names or lists, and must be sent in the correct order.

For example, suppose the master wants to execute the slave predicate 'pred1' which has two input parameters and one output parameter thus:

```
first input parameter    : integer, 10 (<256)
second "    parameter    : list, [a,b,1,2] (mixed
                           integers and atoms)
output      parameter    : list, [h,i]
```

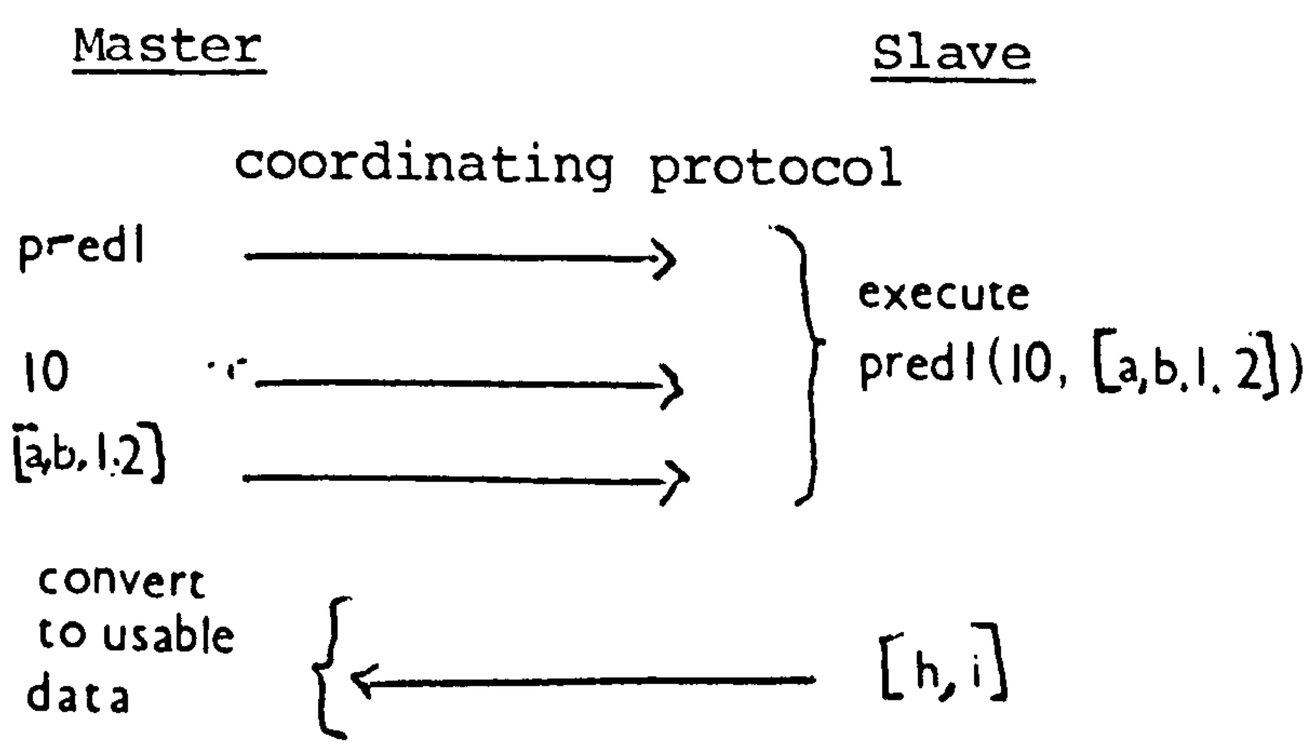


Figure 5·3 Sending Parameters

The slave also has the ability to request the user of the master for an input list, integer (<256) or atom, as well as printing on the masters terminal any message, question,

integer, list or atom. Each of these communications options (including the ones starting and ending a predicate execution) have their own distinguishing control signals. These are shown diagrammatically in Appendix 6. The listing of the assembler program that operates the inter PC link is available in Appendix 6. As can be seen from Appendix 6 the link program is large and complex. It has, however, been fully tested by myself and another PhD student at Warwick.

The assembler routines that control the link are manipulated by a suite of PROLOG predicates. In addition some simple problem dependant routines need to be written by the user, specifying the number and types of parameters for called expert predicates etc. These hid from the user the intricacies of using the link.

A suite of PROLOG predicates have been written to simplify the use of the link (a listing of them is available in Appendix 6). They cover the following functions:

TABLE 5.1- PROLOG COMMUNICATION PREDICATES

name	use
command	wait for a command and execute it
commsend(X)	send a command X and interact
commend	master ends communication
commmsg(X)	slave sends string X
readri(X)	remote read integer
readra(X)	remote read atom
readrl(X)	remote read list
writera(X)	remote write atom
writeri(X)	remote write integer
comlsend(X)	send a list X
comlsnd(X)	send a list X to be printed
comlrec(X)	receive a list X
readi(M)	receive an integer M
reada(M)	receive an atom M
nlr	remote write a new line

Most of the above are simple routines whose operation can be seen from the listing, but some are more complicated and require further explanation.

a) wait for a command and execute it.

This is the basic waiting command used by the slave to await an instruction.

```
1. command :- manaut(m),external_code(12,[],X),callo(X).

2. command :- repeat,trimcore,external_code(12,[],[X]),
                  comcall(X).

3. comcall(1000).

4. comcall(X) :- callo(X),!,fail.

5. callo(X) :- call(X).

6. ?- spy callo(1).
```

Before any commands can be passed from simulation to expert, the expert must be waiting for an instruction. This is achieved by the 'command' predicate. When entering the PROLOG system with the linking software at the expert end, the user is asked whether automatic or manual mode is needed (the response is asserted to the database). With automatic mode, immediately after execution of one command, the expert will wait for the next. With manual mode, after execution of a command the PROLOG interpreter returns to the top level. This allows changes to be made to the rule base if required.

The manual version mode of 'command' is line 1 above. The predicate 'external_code(12,_,[X])' waits for a command name, and instantiates X to it, which is then executed ('callo(X)'). The spy point on 'callo' (line 6) causes the command name X to be printed on the experts terminal, allowing the user a trace of what the expert is being asked to do.

Automatic mode is controlled by line 2. 'trimcore' is a system predicate which returns wasted memory to the main

pool. As automatic mode waits for the the next command after execution of the current one, there needs to be a way of indicating when the end of simulation was reached. This is because otherwise the expert would 'hang up' waiting for the next command when none was forthcoming. Use was made of PROLOGs loose typing of variables - when the simulation sends the 'end' signal, external_code(12,[],[X]) instantiates X to 1000. Such a value of X is pattern matched by the 'comcall' predicate (lines 3 and 4). If the 'end' signal is not forthcoming 'comcall(X)' causes backtracking to repeat the command predicate again.

b) send a command and execute.

This is the basic interaction predicate used by the simulation to pass commands to the expert.

```
1.0  commsend(X) :- external_code(7,[X],[]),
1.1                 remparams(X),
1.2                 repeat,
1.3                 external_code(11,[],P),
1.4                 remote_list_request(P),!.
2.0  remote_list_request([500]) :-
2.1                 read(X),
2.2                 comlsnd(X),!,fail.
3.0  remote_read_request([]).
```

'external_code(7,[X],[])' sends the predicate name X to the remote expert. Line 1.1 calls a problem dependant

routine which sends the parameters needed by command X - 'remparams(X)' must be included in the main simulation file (see below). Line 1.3 is a call to the external code operation that allows all the interaction to take place during command execution (such as receiving typed messages, requests for data, etc, from the slave), except when a request for a list from the simulation user to the expert is received. In that case the output parameter P is instantiated to the list [500]. This is pattern matched by 'remote_read_request' (lines 2.0, 2.1. 2.2) which reads the list from the screen and sends it to the slave, element by element (line 2.2 - see next section), before backtracking to line 1.3 again.

c) sending and receiving lists.

Lists are passed across machines by the donor initially sending a 'list coming' protocol (external_code(14,[],[])) followed by each element in turn. After the last element has been sent a 'list finished' protocol is sent (external_code(10,[],[])). Receiving lists is basically the converse of the above procedure, ie from the set of list elements received a list is constructed, stopping when the 'end of list' signal is received.

The link assembler and the calling PROLOG routines are used at both the simulation and the expert. Thus diagrammatically the set up is as follows :

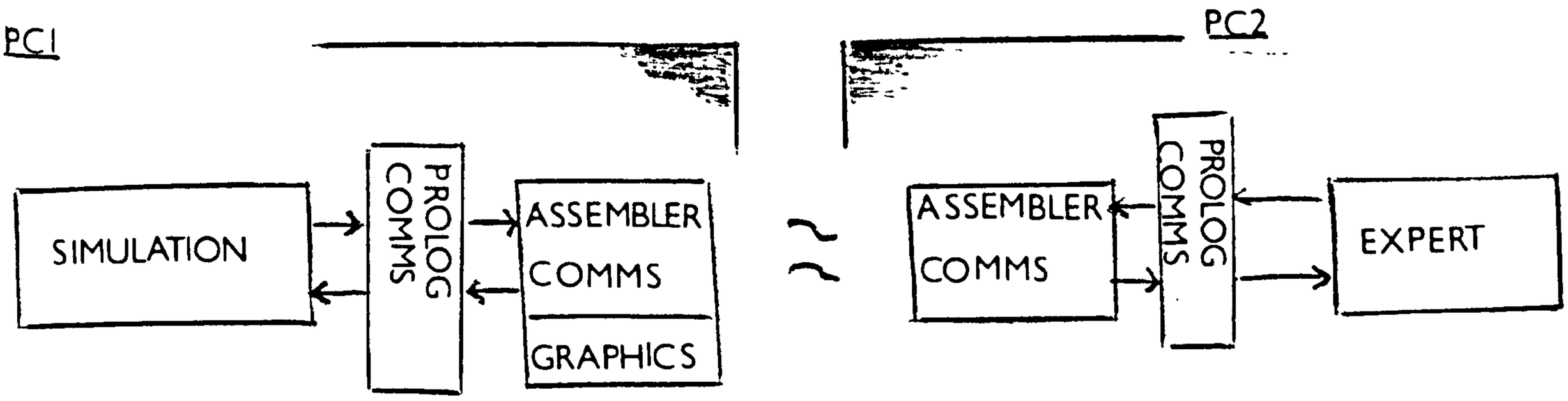


Figure 5·4 Interprogram Connections

Linking Simulation and Expert - Different Machines

The above set up has been tested on a couple of problems, including the A.G.V. one above. It proved to perform in the same way as the earlier one machine approach. Now, however, the expert was physically removed from the simulation.

In investigating the practicalities of such a link, it was necessary to consider how much modification to expert and simulation were needed in order to separate them. A great deal of modification might be viewed as detracting sufficiently from the simulation (or expert system) structure, so as to fundamentally change the method in which they are written. For the A.G.V. example that has been considered both the simulation and expert were written without interlinking in mind. They were then modified to facilitate the link (new versions listings in Appendix 7).

Consequently this formed the model by which the extent of system modification was assessed. This is now discussed.

The Expert

The changes needed at the expert end fell into two groups :

i) rerouting textual output to the remote simulation screen

ii). modifying expert predicates to conform with the simulations needs.

In the event both of these groups needed only minor modification.

i) rerouting textual output. In the PROLOG system text is output to the screen via the 'output' predicate.

```
ou.1      output([]).
ou.2      output([X|L]) :- put(X),output(L).
```

Conversion to remote screen output simply involved adding a predicate above these in the database.

```
ou.0      output(X) :- commmsg(X),!.
```

('commmsg' is a PROLOG link predicate - see above).

With this in place all text is passed to the remote simulation. This means that the expert is unusable 'stand alone' with this line in it. However, with another modification it is possible to allow the facility of stand

alone or linked. This change involves the prompting of the user as the expert is being consulted. After lines ou.1 and ou.2 (without ou.0) have been added to the database the following code could be included:

```
n.1  ?- output("stand alone or linked (s./l.) ? "),
n.2  seen,read(X),sal(X),see('dummy2.pro').
n.3  sal(s).
n.4  sal(l) :- asserta(output(X):-commmsg(X),!).
```

Line n.1 requests the expert end user to say whether the expert is to be used stand alone or linked. If the user replies in the affirmative towards a link, line n.4 is called which adds rule ou.0 to the top of the database (before ou.1 and ou.2). Although this last modification requires a little extra work, it is felt that it provides a useful contribution. This is because it increases flexibility of the expert, whilst not changing any of its structure.

ii) modification of system predicates. Expert predicates (ie those routines in the expert system that are explicitly called by the simulation) need to be changed in order to

a) receive parameters (if any)

b) end the interaction with the simulations user (which commences automatically with the command being called)

c) send the results of the consultation (if any).

The process by which this is achieved is the same for all expert predicates. In this section 'intel' will be taken as an example.

To find the shortest route between a start node S and other nodes N (a list) with the current A.G.V. load Q the user needs to type at the expert (unlinked mode)

```
?-intel(S,N,P,Q).
```

(The expert would respond with a path list P)

Thus, for modification to a linked expert system the three stages above correspond to

a) read S,N,Q from the simulation

b) end interaction (after calling intel(S,N,P,Q))

c) send simulation result P.

All this is achieved by writing an additional rule:

```
intel :-
        reada(S),
        comlrec(N),
        readi(Q),
        intel(S,N,P,Q),
        commend,
        comlsend(P).
```

('reada','comlrec','readi','commend' and 'comlsend' are all PROLOG link predicates as indicated above).

In cases where no input (and/or output) parameters are needed those parts of the above format are simply left out, eg.

```
r:-
rr,commend.
```

The beauty of this is that in addition to being very simple it allows the expert to act as both 'stand alone' and linked.

The Simulation

Changes to the simulation are of a slightly different nature. This is because in this particular example the simulation needs an expert system. Thus if that expert is not in the same machine a link has to be provided. Otherwise the simulation simply would not work. The concept of such a simulation being 'stand alone' does not exist. In other applications of the expert-simulation link (described in a later chapter) such a concept is not so unreasonable, but these are not discussed here.

For this example simulation, and others where the simulation is dependant on an expert to provide control information, what we are interested in is whether the underlying simulation structure is appreciably altered (necessitating a new simulation methodology).

Differences between the linked and non-linked versions lie in the way the expert is called. In the non-linked

version the expert is called simply as any other predicate

is, for example:

```
form_new_schedule :-

        ...

        intel(S,N,P,Q),

        ...
```

For the linked version however, we needed a method for sending the command name and then any parameters that are needed. This is achieved by the PROLOG linking predicate ´commsend´ which is described above. It requires a suite of PROLOG predicates to be included in the simulation file of the form:

```
remparams(expert predicate name) :-

        <specify and send parameters>.
```

Before ´commsend´ is called the parameters must be added to the database so that they can be picked up by ´remparams´. For the example just given :

```
form_new_schedule :-

     ...

     assert(paramr(1,S)),

     assert(paramr(2,N)),

     assert(paramr(3,Q)),

     commsend(intel),

     comlrec(P),

     ...

remparams(intel) :-

     retract(paramr(1,X)),

     writera(X),

     retract(paramr(2,Y)),

     comlsend(Y),

     retract(paramr(3,Z)),

     writeri(Z).
```

For cases where no input parameters are needed the 'remparams' database contains a simple PROLOG fact, eg.

```
remparams(r).
```

Since 'remparams' forms a database completely independent of the simulation, this procedure in no way alters the underlying simulation methodology. It is also very easy to program.

Finally another advantage of the link is it allows for parallel processing of PROLOG programs. This means that once

the simulation has sent a command to the expert, it may continue execution. The expert can then compute any results as requested. When ready the expert will then wait until the simulation requests the results. Thus as well as being a useful tool for indicating the plausibility of expert-simulation interaction, the link also enables PROLOG programs to run faster.

Conclusions

The research presented in this chapter has outlined the technical development of a tool that enables expert systems and simulations to usefully communicate. This tool takes the form of an Assembler program that facilitates transfer of data to a similar program on another machine. Although with a multi-tasking machine such a link would not be necessary (whilst some programmed interface most certainly would) it is recognised that most modern computer environments centre around microcomputers such as the IBM PC. Thus developing the link was a deliberate choice designed to increase the current practicality of this research.

The large linking Assembler program has been fully tested at Warwick University and allows full separation of PROLOG programs. All but the most complex of PROLOG data structures can be passed between computers. The link can also be used to allow parallel processing of PROLOG programs, thereby helping to alleviate some of the current

problems concerning PROLOGs speed. The linking software is operated via a suite of easy to use PROLOG predicates which hide the major complexities from the user. In using the link to provide connection between an expert system and a simulation, it has been found that only minimal superficial changes need to be made to both expert and simulation. Thus using the link does not cause fundamental change to the way either expert or simulation are written. In addition it has been shown to be simple to provide full expert facilities to the simulation and its user. Thus the user can direct and interrogate the expert, from the simulation monitor. By looking at the form of the program interface at the simulation to the expert, it has been possible to generalise it. Thus we now have the blueprint for all interfacing of this kind.

Of course, as the research now stands, there are limitations.

i) The link only connects two PROLOG programs.

Although there is great interest in the possible future use of PROLOG, it nonetheless remains true that most simulations are written in some other high level language. A link only becomes of truly practical value when connecting PROLOG with some other high level language such as FORTRAN or PASCAL. All the PROLOG to PROLOG link provides is a demonstration that such a link should be possible, as well as providing a basis for such a development. Indeed, because of the intentionally modular form of the Assembler link it

was hoped that it could be used as the strong basis for a link between PROLOG and a high level language.

ii) Usefulness of an expert-simulation integration has not been considered.

Even if an expert-simulation link were possible, this research has not yet started to investigate its possible uses. In the A.G.V. routing example above, only one use was tested. This was of using an expert system to contain rules controlling elements of the simulation. In this example the expert system consisted of a modular, transparent and changeable form of rules normally hardcoded into a simulation. Although, with simulation and expert both being written in PROLOG, one could argue the expert to simply be part of the simulation anyway, the distinction would become clearer if the simulation were written in a compiled high level language. It may be true that other applications of expert systems to simulation (and vise versa) can be usefully adopted.

It is these two research limitations that have been tackled in the rest of this PhD. They form the next two chapters of this thesis.

- CHAPTER 6 -

LINKING PROCEDURAL LANGUAGE SIMULATION AND PROLOG EXPERT

The work presented in this chapter provides the tools whereby visual interactive simulations can be investigated for the possible enhancement using A.I. programs. The link connecting the two systems allows remote communication to a PROLOG program whilst allowing the full interactive facilities of the simulation.

Introduction

As currently described the expert-simulation interface works between two PROLOG programs. Although the feasibility of simulation development in PROLOG has been shown, it no doubt remains true that current simulations tend to be written in high level conventional languages. The question therefore arises as to whether it is possible to link a PROLOG expert with such a simulation and thus investigate the possibility of intelligent simulation environments. Technical difficulties would arise because of such factors such as

1) different data structures between two languages

2) the languages may differ in execution (ie they may be interpreted or compiled)

3) different variable types between the two languages

4) same variable types, but different internal representation between the two languages.

The approach used to investigate the possibility of an interprocessor link and therefore the possibility of linking remote expert systems with simulations was to develop such a link. Further if such a link could be developed, it would be possible to investigate the practicality and usefulness of merging the two technologies.

Current day simulation packages are based on procedural languages. In addition, most other computer programs are written in such languages. It was decided therefore to develop a link between a procedural language and PROLOG (a non-procedural language).

An Interprocessor Link

The handshaking protocol used in the PROLOG-PROLOG link described in the previous chapter worked satisfactorily. It was therefore decided to adopt the same protocol. The difference between the PROLOG end and the procedural language end lay in the Assembler interface to the main programs. The PROLOG Assembler link and the associated PROLOG routines needed no modification. The procedural end Assembler link was envisaged to be essentially the same as the PROLOG one (although, since it would be used only by the simulation, not all functions would be needed) with modifications at the front end (input) and rear end (output) so that it could interface with procedural type languages. New associated procedural language routines needed to be written. These were mainly problem independent ones

(specifying parameter types etc.) similar to those for the PROLOG simulation.

Modifications to the Link Assembler

No modification was needed for the PROLOG end of the link. Indeed for the procedural language end the core of the program also remained the same. Modification proved necessary because of the differing methods by which procedural languages communicate with the Assembler. Although the precise technique would vary from language to language (and implementation to implementation) the principles will be the same. The working link is for Fortran and the link will now be described, using that medium.

Communication to Assembler is achieved via a common block. The same area of code is defined by, and can be accessed by, both Assembler and procedural languages.

```
Assembler (Macro86)

     ATA  SEGMENT BYTE COMMON 'DATA'
     CHAR DB   120 DUP (?)
     ATA  ENDS
```

```
Procedural Language (Fortran)

     integer*1 char(120)
     common/ata/char
```

PROLOGs access to Assembler code (for the implementation adopted at Warwick) is achieved by having 51

bytes common to both sets of code (see Appendix 5). In this 51 bytes, provision is given for a subroutine number, input variables, and output variables. In order to make this link usable by a procedural language, a front end has been written to the Assembler which allows the procedural language to specify the subroutine number and input variables. A back end has also been written to receive output variables via the common data area ATA. The back end simply transfers output parameters (atoms or integers) to the common array after function execution. A listing of the front and back end interfaces for the link is available in Appendix 9. Input variables are slightly more complicated to handle and are dealt with below.

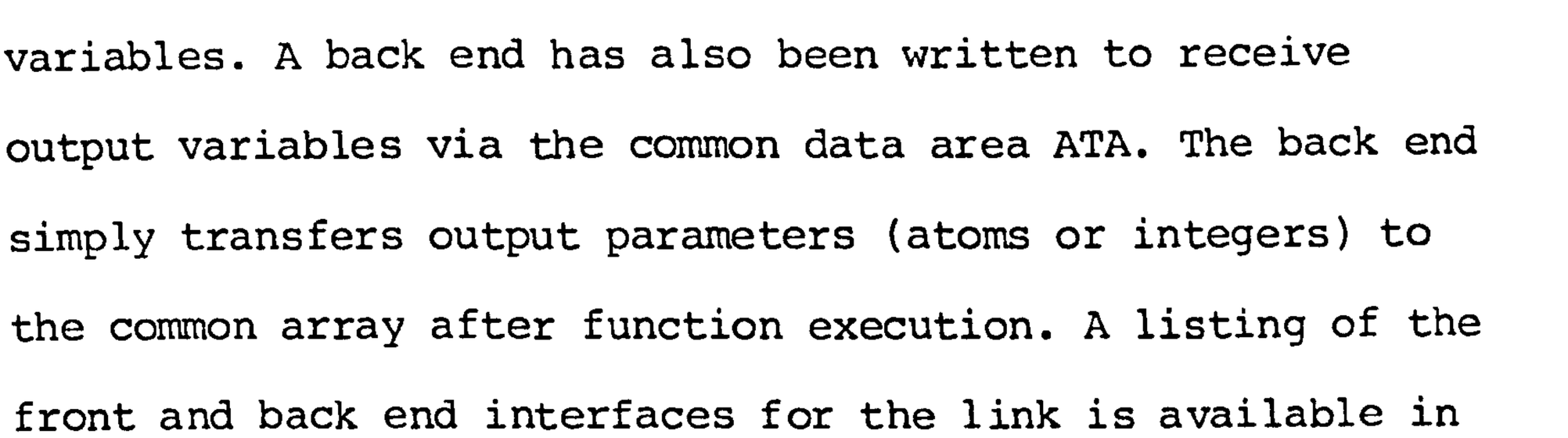


Figure 6·1 Procedural Language/Assembler Common Area

Input Variables

i) integers (8 bits)

These are required by the remote write ('writeri') and send a string ('commmsg') subroutines (codes 3 and 8 - see

Appendix 6). The front end checks the subroutine number (placed in char(1)) against these values and then if found then the integer is deemed to be the value in char(2). All other elements of the char array are zero. The front end places the subroutine number and the integer value into the areas allocated for them by the link program designed for PROLOG. The link is then used as in the PROLOG-PROLOG link (previous chapter).

ii) atoms

These are required by the remote write ('writera') subroutine (number 7). If this number is found in char(1) the front end deems the atom to be the characters in char(2), char(3) and so on, with the atom being delimited by a semicolon. For example the atom 'hello' is represented by:

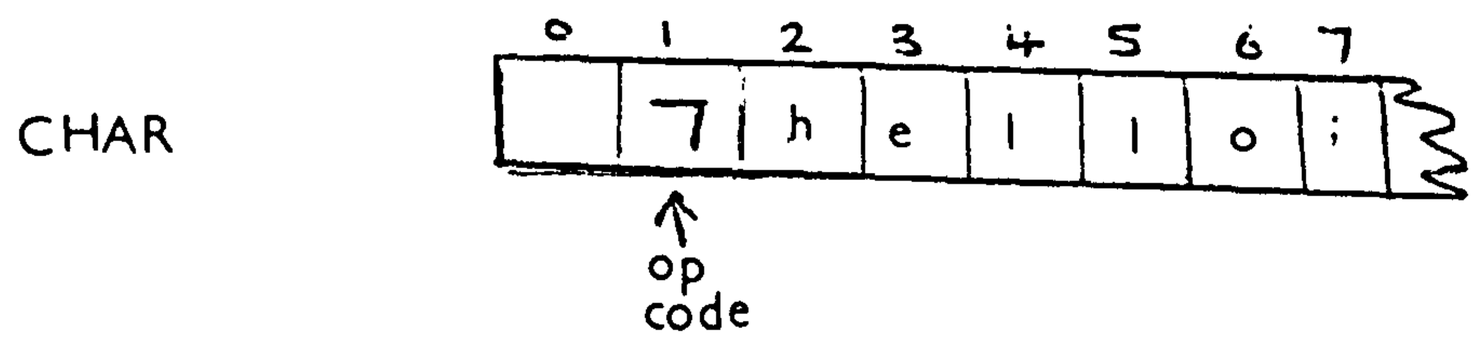


Figure 6·2 Common Area Example

This atom is then processed by the front end to have the same internal form as a PROLOG atom, ie.

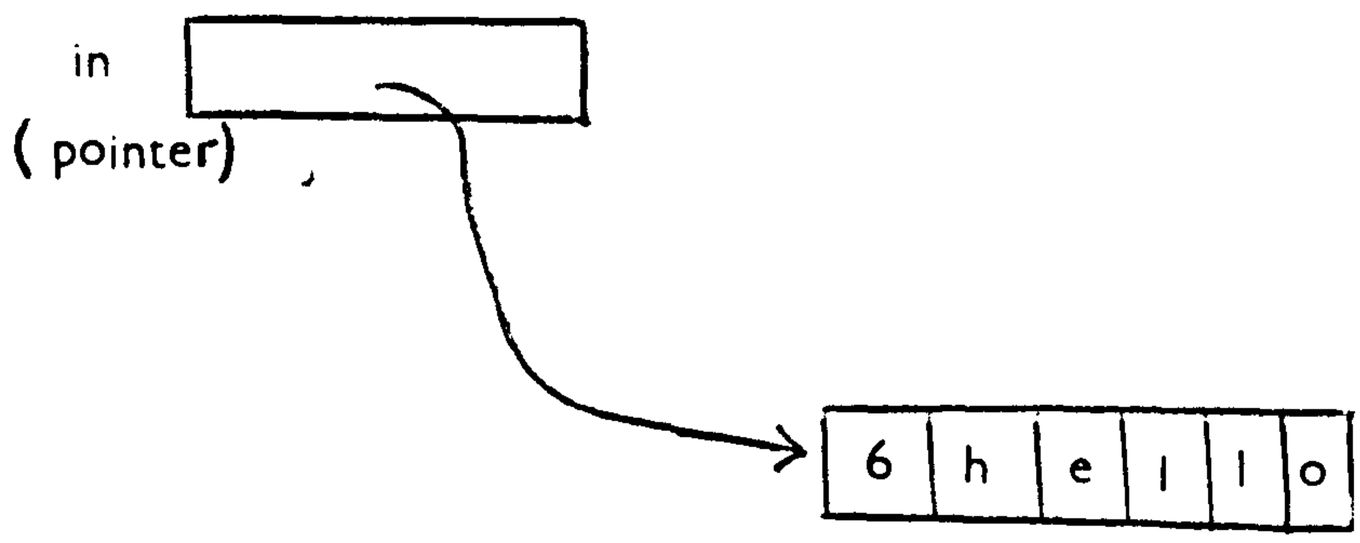


Figure 6-3 Example Internal Form of PROLOG Atom

The link is then performed in the same way as for the PROLOG-PROLOG Link.

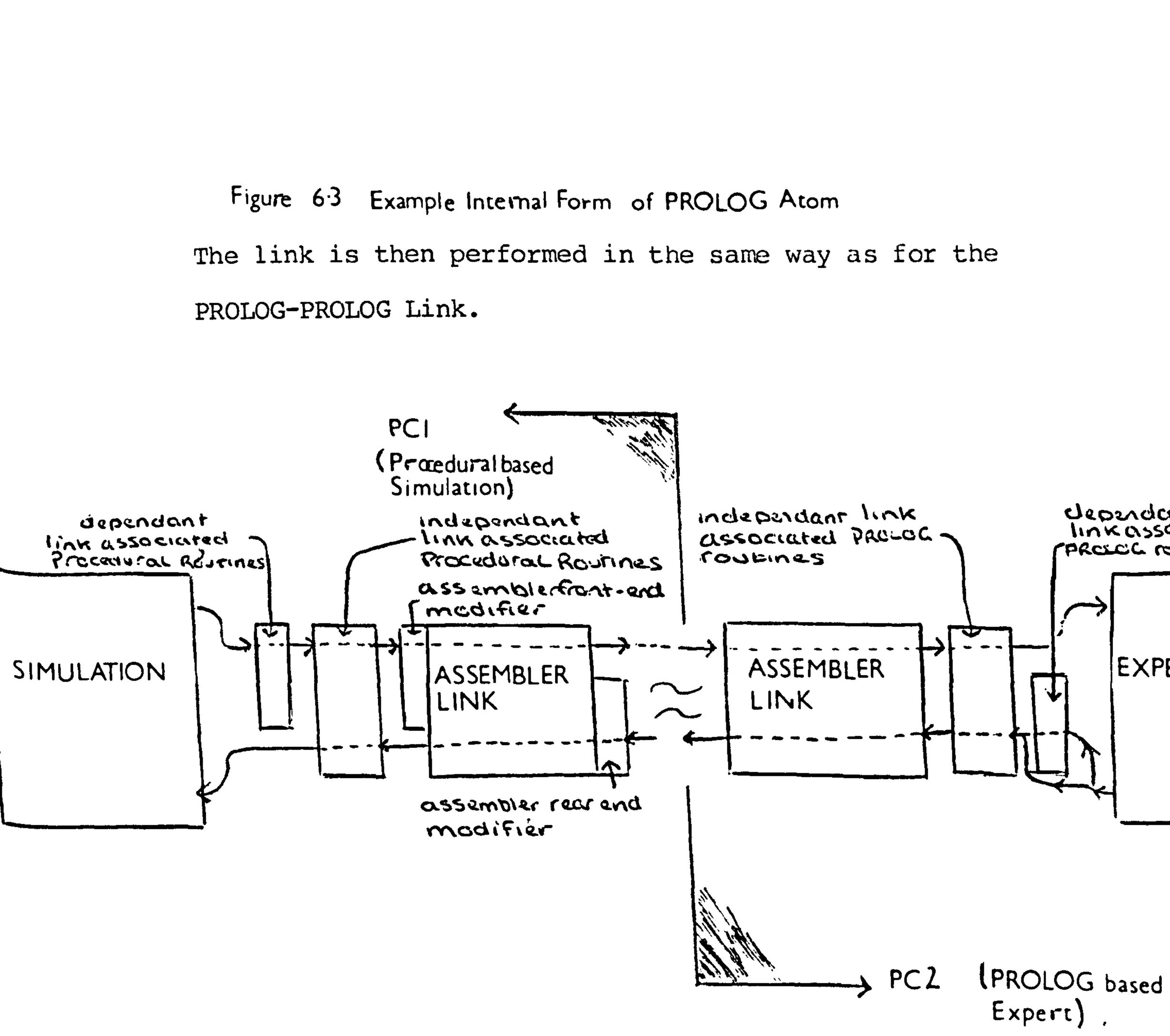


Figure 6-4 PROLOG/Procedural Language Interprogram Connections

Procedural Language Interaction with PROLOG

In order for a conventional procedural language to work effectively with a PROLOG expert system it has proved necessary for procedural languages to be able to handle PROLOG data structures. This is because it is necessary to converse with the expert in a manner natural to it. With procedural languages it is possible to combine local data structures to form approximations of other data types. On the other hand, non procedural languages such as LISP and PROLOG, often regard their data structures form as being a vital element for program execution. This is because data can often be executed as code. Thus trying to approximate data structures of other languages tends to fundamentally change program execution.

Three PROLOG data structures needed to be handled by the simulation.

i) integers : no problems. The link can only send single byte positive integers anyway and these can be used as integer*1 (single byte) variables by FORTRAN.

ii) atoms : again no real problems. These can be handled by integer*1 arrays, one ASCII character per element, with a semi-colon indicating end of atom.

iii) lists : Problems arise because lists can be of mixed type, elements can be either integer or atom or both. Procedural Language arrays on the other hand can handle only one type. Lists, then, have to be represented in these conventional (3rd generation) languages using two arrays,

one for integers and one for atoms. The main (integer) array allocates 1 byte for each list element. An integer element (remembering that the link can only handle 1 byte positive integers) are simply stored in the main array (plist). Atom elements are given a negative pointer value in the main array which points to a 2-D array row which would hold the atom terminated with a semicolon. 'End of list' is denoted by the value -100 in the main array. For example the PROLOG list

[1,2,low,34,hi]

would be represented in a typical high level procedural language as

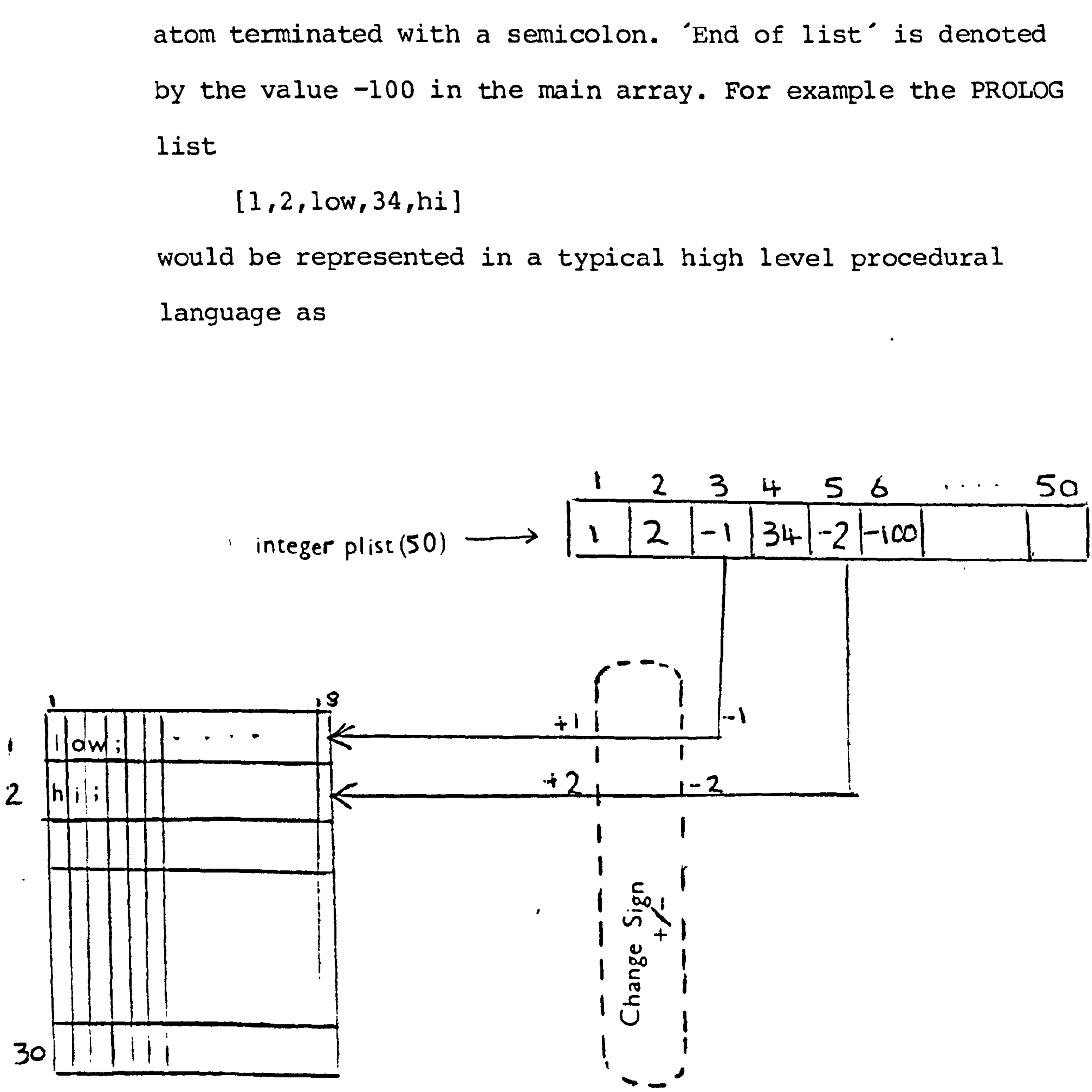


Figure 6.5 Using a Procedural Language to Represent a PROLOG List

Representing PROLOG lists in procedural languages, then, has highlighted the difficulties in having two such languages communicating with each other. However, by the use of procedural language subroutines, it has been possible to remove any difficulties in using the package. The user, wishing to use the link, performs the same operations as s/he would with a PROLOG simulation.

These procedural language subroutines fall into two groups:

i) those that are direct equivalents of PROLOG link manipulation routines (see previous chapter)

ii) those that are designed to make the link easier to use.

The test language used to produce these subroutines was Fortran. A listing of these subroutines is available in Appendix 8.

i) those that have PROLOG equivalents

Table 5.2 Procedural Language Subroutines with PROLOG Equivalents

PROLOG	Procedural Language
writera(A)	wrra
writeri(I)	wrri
reada(A)	rda
readi(I)	rdi
commsend(X)	csend
external_code(16,[],[])	ends
comlsnd(L)	lsnd
comlsend(L)	lsend
comlrec(L)	rdl

ii) those designed to make the link easier to use.

When testing the link using a procedural language simulation it was realised that some simple operations (for example filling the common data area with a command name) were rather lengthy to program. Consequently a suite of easy to write problem dependant routines were devised. These are simple to write because they use data manipulation routines included in the link manipulation files.

a) sending commands

Because the range of commands to be sent in any one particular application is finite, the atoms for each command can be given an index number. To send a command all one need do is supply the index to the subroutine 'cfill' and then call the send command subroutine 'csend'.

'cfill' is a problem dependant routine, whose use is best illustrated by an example. Suppose two commands ('hello' and 'goodbye') are available for use of the expert by the simulation, then 'cfill' is written as follows (the problem independent parts are in capitals):

```
      SUBROUTINE CFILL(IMESS)
      INTEGER    IMESS
      CALL STORE(IMESS,1,'hello;')  ;1 is the index of hello
      CALL STORE(IMESS,2,'goodbye;')
      RETURN
      END
```

Then to call the command 'hello', the simulation need only

incorporate

```
call cfill(1)

call csend
```

b)simplification of list construction

In a similar manner to 'cfill' above, a problem dependant 'pfill' helps with the construction of lists. For example, assume the following 'pfill' (problem independent parts in capitals):

```
SUBROUTINE PFILL(IMESS)
INTEGER   IMESS
CALL STORE(IMESS,1,'albert;') ;1 is the index of albert
CALL STORE(IMESS,2,'will;')
CALL STORE(IMESS,3,'win;')
RETURN
END
```

Then to send the PROLOG expert the list [albert,1,2,win], the simulation must include the following code:

```
plist(1) = -1   ;first element of list is an atom

plist(2) = 1

plist(3) = 2

plist(4) = -2

plist(5) = -100 ;end of list marker
```

```
call pfill(1,1) ;fill atom array row 1 with atom no. 1

call pfill(3,2)
```

c) sending parameters

As for the earlier PROLOG simulation, provision has been made for the calculation of parameters before they are needed. In addition a problem dependant subroutine needs to be written indicating what the sent parameters should be and then sending them (this is equivalent to the 'remote_params' predicate for PROLOG simulations).

1. storing of parameters prior to being sent: the version produced for this thesis allows the storage of 1 list, 3 atoms and 20 integers. Integers may be stored in an integer array. An atom is stored in one of three integer arrays. To store an atom 'hi' in the first of these arrays for example the user can use the predefined 'fatom' subroutine:

```
call fatom(1,'low;')
```

The provision for lists has already been described above.

2. sending the parameters: This is achieved via a problem dependant subroutine that checks the name of the command

just sent and sends off the respective parameters, having picked them up from storage. This is best illustrated by an example. Suppose two commands to PROLOG are possible: 'hello' and 'goodbye'; 'hello' has two parameters - an integer (stored in intp(1)) and an atom (stored in atml). The simulation analyst would need to write the following 'rprams' (the problem independent sections are in lowercase)

```
      SUBROUTINE RPRAMS
      INCLUDE 'LSIM'
      CALL TATOM('hello;',FLG)        ;if command was hello
                                      set FLG = 1
      IF (FLG.EQ.1) goto 10
      CALL TATOM('goodbye;',FLG)      ;if command was goodbye
                                      set FLG = 1
      IF (FLG.EQ.1) goto 20
      GOTO 990                        ;cover all possibilities
10    char(2) = intp(1)               ;set char(2) to integer
                                      parameter
      call wrri                       ;send integer
      call fchar(1)                   ;fill char with atom in
                                      atml
      call wrra                       ;send atom
      GOTO 990
20    CONTINUE
990   RETURN
      END
```

In order to test this link a simple example was chosen. The simulation package used in the procedural language (Fortran) simulation example was MICRO-VISION (Hurrion, 1981). This, essentially, is the reduced initial version of the SEE-WHY system, suitable for use on microcomputers.

Simple use of the Link

The sample problem consisted of the operation of a (simple) bank at peak operating times. Customers arrive randomly at the bank (the average time between successive arrivals is 20 seconds). The service time is modelled using an exponential distribution with a mean of one minute 15 seconds. The bank has 5 service booths. As well as producing a simulation it was required to produce an expert controlling the variable factors.

This problem is of a fundamentally different type to that described for the PROLOG-PROLOG link in the previous chapter. That problem required the expert to contain some of the simulation logic. In this example, however, the simulation is complete and independent. The 'expert system' (or A.I. program) takes the place of the simulation user in controlling resource levels, in order to keep the simulated system running smoothly. Thus the expert acts as a process control system. Indeed if this link could be shown to work on such systems then a valuable use could have been found - that of testing process control expert systems. By using the link it would be possible to:

i) Test prototype versions on a simulation of the process. From this the human expert/knowledge engineer could spot errors in the expert system by watching an animated version of the process. This could greatly speed up expert system development. In addition the simulation could be used to thoroughly test the expert in a safe environment, reducing the chance of unseen bugs in the expert.

ii) By producing an animated picture of the consequences of the experts decisions, the user could test the expert to his own satisfaction. Thus the important aspect of user acceptance could be easily handled at this point.

In this simple problem only one factor of production is variable, that of the number of servers. The number of servers needed at any one time is determined by the size of the customer queue. This is one unit of information that needs to be sent to the expert. Other information could include the current clock time. With this information the expert could recommend a set number of servers. Clock time could be used to ensure that the expert only recommends staffing level changes at set times (eg. shifts). However, for this simple application, only the queue size will be sent. But when ?

The structure of MICRO-VISION models have at their core the following logic

```
10    <advance to next event>

      <move entities>

      ...

      <start new events where possible>

      ...

      GOTO 10
```

It was decided that, in order to keep the expert as informed as possible, the interaction with the expert should occur in this loop. This would mean that after each event, the simulation would send the queue length, whether or not it had changed. Thus the expert was informed often enough to allow it to make changes as soon as they were required. The expert, however, was not forced to make changes just because it was called. It could decide to leave the number of servers as they were. This interaction has essentially the same structure as for the PROLOG simulation in the previous chapter, so the loop structure then became

```
10    <advance to next event>

      <move entities>

      <prepare parameters>                    |
      <send command and interact>             | Interface
      <receive parameters>                    | to Expert
      <translate parameters (1 and 2)>        |

      <start new events where possible>

      GOTO 10
```

Translating the parameters also involved incorporating them back into the model.

As in the previous chapter it was important to see how all of this affected the simulation programming. The old and new code matched up like this:

```
old code:
      10   call advanc(ievent,itime,iele)
           ...
           call starts
           goto 10


 new code:
      10   call advanc(ievent,itime,iele)
           ...
           call starts
      *    call incomm              ;re-initialise comms port
           intp(1) = isize(qu)      ;store size of customer queue
                                    as first integer parameter
           call cfill(1)            ;load correct command into
                                    common char array
           call csend               ;send the command and
                                    parameter
           call clrc                ;clear the common 'char' array
           call rdi                 ;read the integer number of
                                    servers from the expert
      *    call setatt(config,3,ointg)
           goto 10
```

In addition the variable declarations for the link were available in another file, which was read into the start of the main program.

Finally, for the link to operate the two main problem dependant routines 'rprams' and 'cfill' needed to be written (see above).

```
      subroutine rprams
      include 'lsim'
      integer*1 flg
      call tatom('control;',flg)
      if (flg.eq.1) goto 10
      goto 990
10    intg = intp(1)
      call wrri
990   return
      end

      subroutine cfill(imess)
      integer*2 imess
      call store(imess,1,'control;')
      return
      end
```

In order to use the communications package, the simulation object code needed to be linked with the procedural language link manipulation routines object code and the actual Assembler object code.

With this set up the simulation was found to work well in conjunction with a remote expert, but improvements could be made:

i) The program could be made easier to read by placing all the communicating interface (the code between the two * above) in a separate routine, which is then called within the main loop.

```
10   call advanc(ievent,itime,iele)
     ...
     call starts
     call expint
     goto 10
```

ii) At present the simulation cannot work in a stand alone mode. However, by introducing an extra facility to the problem dependant interaction mode, this can be overcome. By giving the user the ability to set a flag ('istal', say) we could have

```
10    call advanc(...)

      ...

      call starts

      if (istal.eq.0) goto 20

      call expint

20    continue

      goto 10
```

(istal = 1 means 'interact with expert'
istal = 0 means 'stand alone').

Thus, as far as the simulation is concerned, introduction of the link is possible with minimal changes to both simulation structure and simulation use.

The expert system here was designed to be as simple as possible, since the object of the exercise was primarily to produce a working example. However, any level of complexity of the expert would not alter the simulation. Thus expert refinement can occur using the simulation model as an indicator of the experts success. Here, though, the expert simply returns a number of servers which is dependant upon the size of the queue parameter it receives. In other words the expert consists of a set of production rules of the form:

IF <queue size> > N THEN <number of servers = M>

As the PROLOG end of the link is the same as before, the interface with the simulation is unchanged from the

previous chapter. Thus with the above type of production rules we have:

```
control :-
        readi(I),
        control(I,J),
        commend,
        writeri(J).
control(I,J) :- I>4,J=5. ;IF queue>4 THEN no. servers=5
...
?-consult(prlink4). ;read in PROLOG link manipulation
```

With two IBM PC based machines this expert-simulation system was seen to work satisfactorily. The simulation worked very slightly slower than before, since it had to wait (about 2 seconds) for the expert to process the data. It was also worth noting that the inclusion of the expert link in no way altered the interactive nature of the simulation.

CONCLUSIONS

The work presented in this chapter has provided the tools by which modern visual interactive simulations can be investigated for the possible integration of A.I. programs. A working link has been developed which is adaptable to many standard programming languages. The link here has been

adapted to work with Fortran programs, but would also be compatible with Algol type languages and Pascal. The link allows remote connection to a PROLOG program, whilst allowing the full interaction facilities of the simulation.

The main problem in producing such a link has proved to be matching data types between a procedural language and PROLOG. The list data structure, so vital to PROLOG program execution proved the most difficult to reproduce in conventional languages. Despite this, it did prove possible to use standard data structures in a way that facilitated list representation.

The example used to test this link is of a fundamentally different type to that for the previous chapter. It is of the form of a process control system. Thus it illustrates another anticipated use of the link. Experiments on some uses of the link are presented in the next chapter, together with an outline of other possible applications. Where possible, generalisations are made indicating possible drawbacks.

- CHAPTER 7 -

USING SIMULATIONS WITH EXPERT SYSTEMS

Introduction

The work presented in the previous two chapters concerned the technical considerations of interfacing simulations and expert systems. It has shown that it is possible to link such systems, and indeed a working interface has been produced. Where this work is currently lacking, however, is in the consideration of how useful such a link would be. Before embarking on development of the interface, several theoretical applications seemed to indicate that the link would be of great use. In this chapter these uses are outlined, together with (except one case) practical experiments undertaken to test their feasibility. Anticipated use of the link fell into five groups. The final one below was not tested due to the fact that similar results could be obtained by other methods.

i) Manipulation of Parameters

An expert system could be linked with a simulation in order to control resource levels within the simulation. A simple example of this is the bank queue example from the previous chapter. There the 'expert' controlled the number of servers. The expert in such cases could be viewed as a process control system working on a simulation of some true life industrial process. Although this may not seem related to the bank queue problem, there are similarities. Thus,

instead of the expert controlling the number of servers in order to keep queue length reasonable, it could be controlling valve size in order to keep water flow (say) to some tolerance level. Thus an alternative way to look at this anticipated use of the link, is as a method for verifying process control systems. It allows the process control system to be tested to its limits in a totally safe environment.

ii) Containing Simulation Logic

Here, an expert system could be used to contain some of the logic of the simulation. Many of the problems to which simulation is called upon are semi-structured, and as Moreira da Silva (1982) points out, a problem of using simulations for such problems is that they are inflexible. Moreira da Silva (1982) states that there is a need for development of good interfaces that "enable the decision maker to use his creative thinking and pattern matching capacities to their maximum potential". Once decision rules are coded into a simulation they are fixed. If such rules could be coded separately from the simulation, in an easily modifiable language, the result would be a greatly more flexible simulation. Such a situation was confirmed during an interview conducted with M. Hunt, a member of the British Steel O.R. group, who are major users of the SEE-WHY system. He stated that

"Some of the decisions (of the simulation) are two complicated and changeable for FORTRAN code, so we make the simulation request the user for a decision at these points".

In such situations PROLOG appears to fit the bill of a remote easily modified language that could contain critical parts of the simulation logic. The link has been developed to allow changes in the expert system to be made at any time. Furthermore PROLOGs production rule syntax means it is well matched for current theory on representing human decision making heuristics.

iii) Expert System Acceptability and Development

This has already been touched upon above. If we can test the application of an expert system on a visual simulation of the problem then we can be much happier that it is satisfactory. This could help overcome a major problem of expert system implementation - that of user acceptance. If the simulation is written in a way that those eventually using the expert system can understand, then its critical users can be assured. The expert system could be tested against a known human expert on the same simulated problem, thus providing users with a control with which to compare the system.

Related to this use of the simulation as a 'window' into the expert system, is the possibility of using the link to help develop expert systems. By testing partially built expert systems on visual simulations of the area to which they will be put, it should be easier for the expert user

and knowledge engineer to spot problems in the expert. Thus the simulation will give an animated indication of where the expert needs improvements and where it performs well. In addition, if care is made to ensure that the expert is of a modular production rule form, it may be possible for the users to interactively change the expert rule base whilst the simulation is running and then to see the affects of their changes immediately. This could greatly speed up expert system development.

iv) Learning by Parameter Adjustment

As indicated in the first chapter, a main way of developing a learning expert system is by parameter adjustment. This means that the expert has a set of rules which are triggered according to the values of certain parameters. The levels of these parameters are adjusted according to the effects of monitoring a human performing the task later to be done by the expert. It is envisaged that the link could be used to help develop a process control expert system in this way. If one considers a process to be a set of values that dictate resource levels needed, then by monitoring an expert such values (parameters) could be monitored in the light of the simulation users' decisions. After a while of monitoring the user, the expert could then be asked to act as a process control system, based on the values it has calculated from the expert user. Of course, assuming this possible, main questions still remain such as:

a) how long should we monitor the user ?

b) how should we calculate parameter values ?

c) should we monitor more than one user ?

d) is one experienced user better than several novice users combined ?

These questions are dealt with in an experiment described later in this chapter.

v) Models for Expert Refinement

The idea for this particular application resulted following a visit to the Expert Systems´85 conference at Warwick University. At this conference there was a $100 000 expert system which worked as a chemical plant monitoring system. It operated as a standard expert system, but when difficult problems arose, it would run a model to simulate the problem and the process systems reaction to different decisions, so as to select the best decision. It is suggested that the simulation-expert link could give an economic way of providing similar facilities. This application has not been tested in this thesis, it is not envisaged that it could not produce any new advantages (except reduced costs and perhaps ease of implementation) over the expert system described above.

Manipulation of Parameters Experiment

This experiment used the same ´lorry´ problem as used by Hurrion and Secker (1978) to illustrate visual

interactive simulation. The problem relates to the operations of a simple coal yard.

"The problem consists of two types of vehicle known as NCB and MERCH, which arrive, randomly at a weighbridge. After weighing in the NCB vehicles remain in the yard for a fixed period of time doing specific tasks before filling up with coal, and weighing out. The MERCH vehicles, which also arrive randomly, need first to be weighed in. After this operation they proceed to a loader, where they are loaded with coal and from here leave the depot having weighed out. Coal trains, TRAINs, bring coal into the yard, are unloaded using the same loader required for MERCH vehicles, before they too leave the yard". Due to fuel and labour costs money is lost whenever trains or lorries need to queue. As well as this labour, depreciation, and fuel are all costs of using additional loaders and weighbridges (since labour on these tasks demands a higher rate of pay). It has been seen in the past that the cost of a train or lorry queuing is approximately half the cost of an extra resource on top of the standard two loaders and two weighbridges.

The expert system was be written to control the number of weighbridges and loaders. This problem is in many ways similar to the 'bank queue' problem in the last chapter and hence will not need much explanation here. The chief difference between the two problems is the amount of information that needs to be sent between expert and simulation. The size of the resource levels needed at any

one time is determined by the queues for that resource. There are four such queues - two for the weighbridge (weigh-in and weigh-out) and two for the loaders (merchants and trains). Thus four input parameters need to be sent to the expert. The expert in turn needs to pass back two resource levels.

These input and output parameters are passed across as two lists. This is a general way that any number of input and output parameters could be sent between simulation and expert.

The simulation has the same structure as for the 'bank queue' example in the previous chapter, and so it will not be discussed here. A listing is presented in Appendix 11. However, since the method of passing parameters between simulation and expert is one that can be generalised, presented below are commented versions of the 'expint' and 'rprams' subroutines (see previous chapter).

As regards performance, the link between expert and simulation took about 1.5 seconds on each call and enabled the system to be controlled simply and efficiently.

```
subroutine expint

integer db

include 'simdef'                ;define all variables, common

include 'lsim'                  blocks and equivalence

call incomm                     ;initialise ports' parameters

call plist(1) = isize(qwin)     ;fill 'plist' with parameters

call plist(2) = isize(qwout)    to be sent to the expert, end

call plist(3) = isize(qtral)    list with 'endl'

call plist(4) = isize(qmerl)

call plist(5) = endl

call clrc                       ;clear procedural/assembler

                                common area

call cfill(1)                   ;fill it with correct command

call csend                      ;send command (and parameters)

call clrc

call rdl                        ;read list from PROLOG into

                                'plist'

db = plist(1)                   ;use values received from

call setatt(wpool,1,db)         expert in the simulation model

db = plist(2)                   as required.

call setatt(lpool,1,db)

return

end
```

```
      subroutine rprams
      integer flg                    ;define variable information
      include 'lsim'
      call tatom('control;',flg)     ;is the command in the
                                     procedural/assembler common
                                     area 'control' ?
      if (flg.eq.1) goto 10          ;yes
      goto 990                       ;no
   10 call lsend                     ;send list in 'plist'
      continue
      return
      end
```

The expert system side of this system was also very similar to the 'bank queue' example.

```
control :-
        comlrec(L),                  ;receive list from simulation
        control(L,M),                ;calculate resource levels
        commend,                     ;end command interaction
        comlsend(M).                 ;send list M to simulation

control([A,B,C,D],[E,F]) :-          ;to calculate resource levels
        conw([A,B],E),               calculate no. of weighbridges
        coml([C,D],F).               and the the no. of loaders

conw([A,B],4) :- X is A+B,X>4.
...
```

A Window to Simulation and Expert System Logic

Introduction

There were several purposes to this experiment. Firstly to check the feasibility of removing simulation logic to a remote expert system in a non-trivial industrial problem. Secondly to see what aid the use of visual interactive simulation gives to the development of such expert systems. Thirdly it provided a good medium with which to test the generality of the logic contained in the AGV expert system described in chapter 4.

The Test Problem

The problem concerned the simulation of a flexible manufacturing system. The original simulation was undertaken at Warwick some time before this PhD. The factory consisted of 10 workstations joined by a simple route system. One of the workstations (node 10) was regarded as the input/output port of parts to the factory. These parts are conveyed to a waiting machine by one of two AGVs (robots), each bidirectional. After each part has been machined it is then carried back to the I/O port by one of the AGVs.

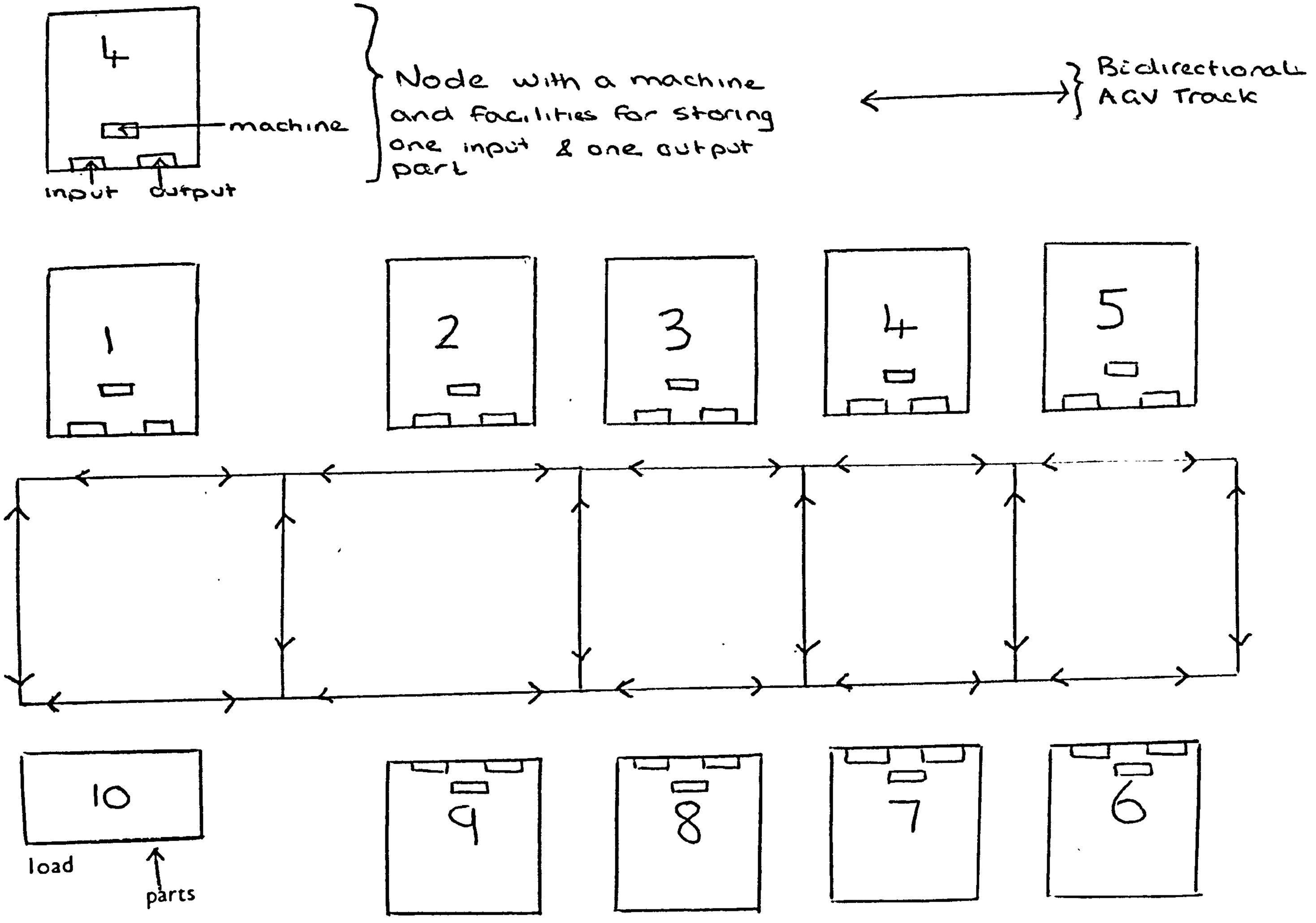


Figure 7-1. Diagram of FMS Layout

Understanding the Problem

In this old SEE-WHY simulation the AGVs are sent round the factory floor according to rules designed to maximise efficiency. These rules are 'hard coded' into the simulation which means that they are very hard to modify. It was hoped that the link system can be used to isolate these rules, making them more amenable to modification and also more immediately identifiable. Indeed a major problem with understanding this simulation was the fact that no documentation was available and the program code was largely uncommented. Therefore, in order to extract the rules governing movement of the AGVs it was necessary to simply watch the simulation for a couple of hours and extract the rules that could be seen working. Although not accurate, this does at least correspond to the knowledge extraction from a human expert.

As an added aid to expert development another simulation of the problem was developed. This was the same as the old version above except that it contained no controlling logic. Hence, whenever a decision point was reached (each time an AGV was at a node) the simulation stopped and requested the user to input the next node that AGV should visit. Producing this enabled the analyst to test out the rules deduced from watching the original simulation as well as giving the analyst a better feel for the problem. In addition it was anticipated that this simulation could provide the model for an expert system to control. All that

was needed was to replace the request to the user with a request to a remote computer using the link as previously described.

The Expert System

Using the above two simulations for rule extraction the following set of production rules (in order of priority) were formulated. Each of which tests that the AGV has reached a node.

1. IF reached node 10, part available THEN send AGV to nearest machine.
2. IF reached node 10, part not available, but part ready at one of the machines THEN send AGV to nearest machine with a part.
3. IF reached node 10, part not available in entire system THEN keep the AGV at node 10.
4. IF reached a node (1 to 9), AGV empty, part available at node THEN fill AGV and send to node 10.
5. IF reached a node (1 to 9), AGV empty, part available at another node THEN send AGV to nearest node with a part.
6. IF reached a node (1 to 9), AGV empty, no part available at any node (1 to 9) THEN send AGV to node 10.
7. IF reached a node (1 to 9), AGV full with machined part THEN send AGV to node 10.

8. IF reached a node (1 to 9), AGV full with unmachined part, no part ready in rest of system THEN send AGV to node 10.

9. IF reached a node (1 to 9), AGV full with unmachined part, part available at another node, current nodes' machine not available THEN send AGV to current destination mode.

10. IF reached a node (1 to 9), AGV full with unmachined part, part available at another node, current node not busy THEN send AGV to nearest of nodes where part is available.

11. IF reached a node (1 to 9), AGV full with unmachined part, part available at current node, current nodes' machine not busy THEN send AGV to node 10.

Notice how these production rules do not calculate the actual routes to be taken, but only the destination node. Route calculation has been achieved by using the AGV expert described in Chapter 4 with some minor modifications.

The expert system described in Chapter 3 works out the most optimal route between a source and a destination node. It gave the user four options. This modification removes these options, allowing only the depth-first search with maximum distance cut off point method. With the short arcs having a distance of 6 units and the long arcs a distance of

11 units, the maximum distance any route could possibly take is 50 units.

The expert in Chapter 4 needed only to deal with one AGV, and so for this application modification was needed to avoid collision. The original algorithm works using depth first, looking at a database of arcs. Collision could be avoided by deleting the offending arcs before the algorithm is used, the arc database later being reinstated to its original form. Assuming AGVs operate at the same speed, and that new routes to a destination are calculated each time an AGV reaches a node,collisions can occur when:

1. The other AGV is on the nearest arc moving in the opposite direction.

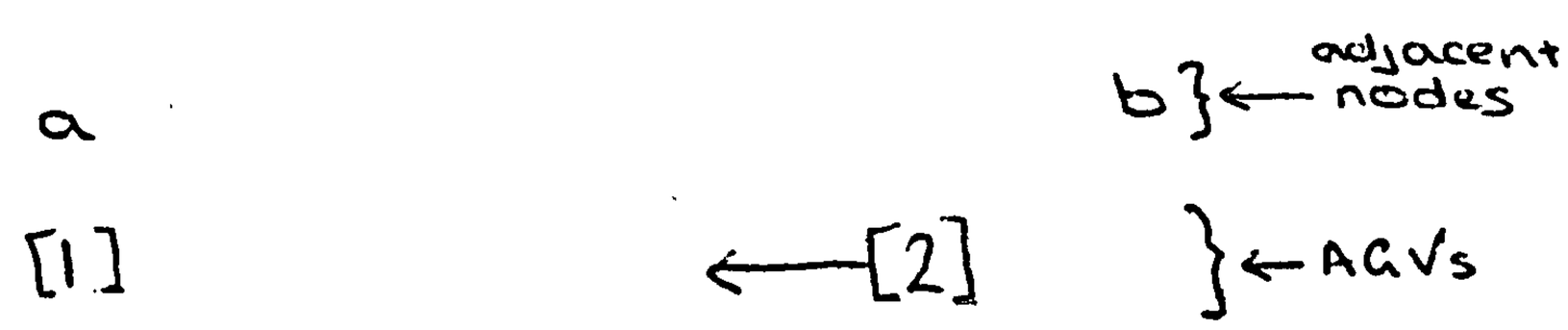


Figure 7-2 AGV Collision Type I

2. The other AGV is perhaps due to reach the next node at the same time as the current AGV.

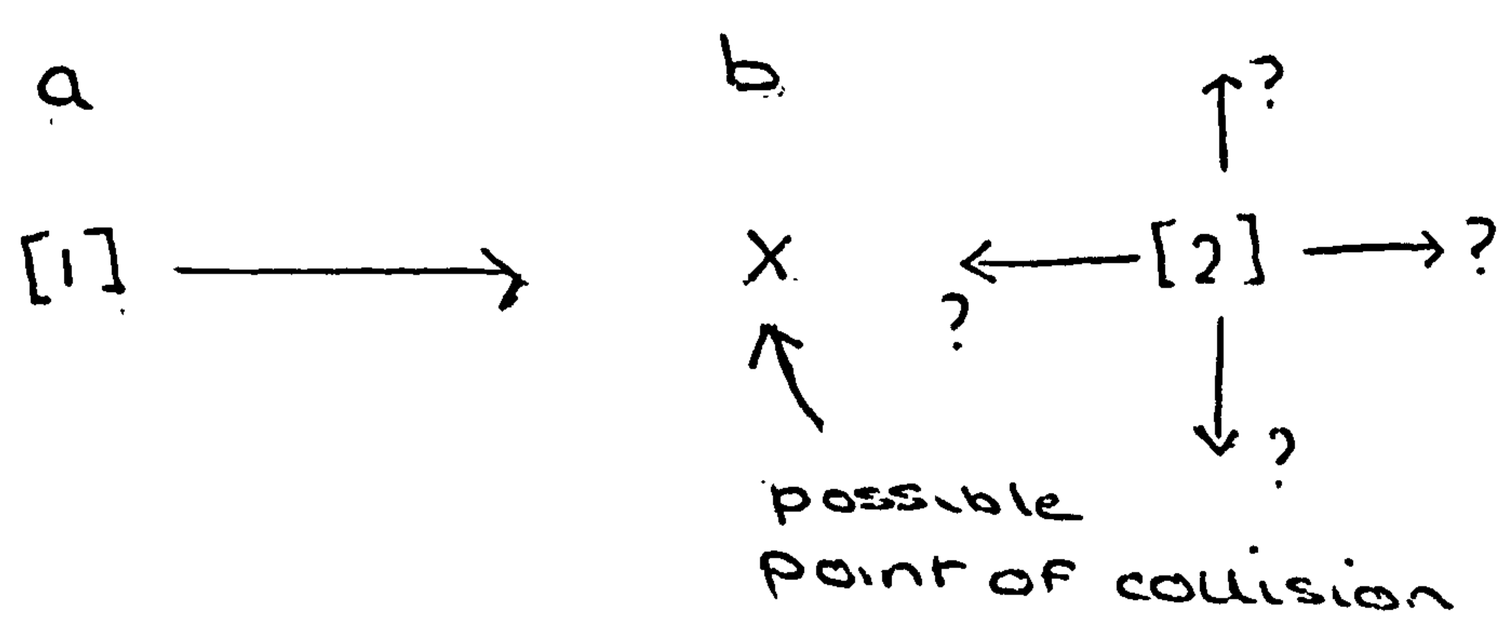


Figure 7-3 AGV Collision Type2

In both these cases the solution adopted was to temporarily delete the arc

arc(a,b,T)

and then calculate the route and assert the arc fact again.

As well as these modifications to the original AGV expert, two databases needed to be kept by the expert system. One was of the form

agvdest(AGV,destination node)

indicating which node the expert has decided for the AGVs destination (as opposed to just the next node on route to that destination). Checking this database ensured that both AGVs are not sent to the same node at the same time. A further database is kept, informing the system of the type of part (either. machined or unmachined) on each AGV.

partype(AGV,type(0=unmachined, 1=machined))

This database was used by the conditional part of some of the production rules outlined above. These two databases were continually updated by the production rules execution stages.

Thus it can be seen that the core of the AGV expert system can be generalised to many flexible manufacturing systems. With a suite of such general expert systems it would be possible to greatly decrease development time of intelligent visual interactive simulations.

The expert system needs a lot more information about the simulation state than previous examples. The actual information required was as follows.

- size of input, processing and output queues at machines 1 to 9.
- size of input queue to system at node 10.
- the number of the AGV (1 or 2) currently at a node, plus the node number, plus whether it is empty or full (0 or 1).
- for the AGV not at a node, its track number and direction is sent to the expert so he can compute routes avoiding collision.

As indicated in the previous experiment input to the expert can be sent in a list from the simulation. This list was compiled in the following order.

```
[AGV number,
 node,
 0,
 in(i),  (those nodes with non empty input queues)
 pr(i),  (those nodes with non empty processing queues)
 ou(i),  (those nodes with non empty output queues)
 in10,   (size of queue of parts into the system)
 track number of other AGV,
 direction of other AGV (0=backward,1=stationary,2=forward),
 empty/full status of current AGV (0=empty,1=full)]
```

Testing the System

The simulation controlled with the above expert system was tested on two computers. It was anticipated that changes would need to be made to the expert system, since it was felt unlikely that all the rules for successfully controlling the AGV would have been input. An advantage of the production rule format for containing simulation logic is its modularity and consequent ease of modification.

In looking at the experts performance via the simulation a visual picture of the experts decisions were presented. This made it much easier to spot errors in the logic controlling the AGV, and to formulate rules to overcome them. At the time this system was built it was recognised that expert development was greatly enhanced by provision of this animated performance indicator. Two errors in the expert system were immediately apparent.

1. Rule 8 has an incomplete conditional part, which sometimes caused an AGV with an unmachined part to be sent to node 10. The rule should read:
   IF reached a node (1 to 9),AGV full with unmachined part, machine at current node available, no part ready in rest of system THEN send AGV to node 10.

2. A form of cycling exists. An AGV waiting at node 10 effectively blocks another one getting there, or to some adjacent node. This is a fault of the over simplification of the collision avoidance algorithm. A

more accurate calculation of where collisions may occur was needed.

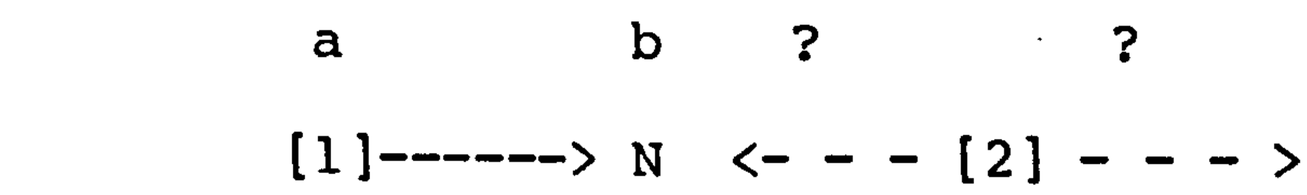
Accurate Collision Avoidance

i) collision type 1 (see above). This can only occur if the AGVs move in opposite directions, or one is stationary. This observation is included into the basic AGV expert logic by the addition of a new database. Given the node the 'decision' AGV is at and the track number of the other AGV, this database supplies the direction of the other AGV which would cause collision. This database is of the form

coll(node,track number,direction).

Testing the problem parameters with this database indicates whether a type 1 collision will occur. If it will the track is temporarily removed from the database, whilst calculation is made.

ii) collision type 2 (see above).



This type of collision will only occur if both

a) [2] is moving towards N

b) [2] is the same distance from N as [1]

To detect this requires the additional information of the position of the other AGV to be sent to the expert.

Given the node of the 'decision' AGV, the track number, track position and direction of the other AGV, a database of the form

col2(track number, node, position, direction).

indicates when this type of collision will occur.

Testing the Second Expert System

The expert as just described was tested in the same way as the first version. It ran better than that earlier version, however, another problem was spotted necessitating two new rules.

The 'stay at node 10' rule (rule 3) does not allow for an imminent collision from the other AGV. The solution adopted for the third version of the expert was to add the following two rules, each with a higher priority than rule 3.

a) IF AGV at node 10, no part available in entire system, other AGV is approaching from node 9 and is very close to node 10 THEN send 'decision' AGV to node 1.

b) IF AGV at node 10, no part available in entire system, other AGV is approaching from node 1 and is very close to node 10 THEN send 'decision' AGV to node 9.

This third version of the expert ran very well, with the AGVs coping easily with the problem. It controlled the AGVs without crashing them and in an efficient manner. In comparison with a human the computer gave equally good decisions, and in a much shorter time. It is worth noting that not all the information sent from the simulation is

consequently there is scope for a vastly increased set of rules. In addition comparison between completely different rule sets is possible. Listings of the AGV simulation and the expert system are available in Appendix 12.

Comparing this expert to the 'bank queue' and 'lorry' experts all can seen to be a simple set of production rules:

IF {simulation state} matches X THEN {response} = N

Given this similarity it would perhaps be worthwhile in the future to undergo research to try and develop a higher level language reducible to PROLOG. Such a knowledge representation language should be particularly well suited to experts designed to be linked with simulations. Such a language would be easier for the O.R. analyst to use and learn than PROLOG. Although for the two examples discussed initially the experts are small and hardly require a special language, for more complicated problems with a greater decision space such as for the AGV problem, such a language would probably speed up simulation/expert development.

A Monitoring Expert

In cases where an expert system is due to be used for process control problems, it was envisaged that the link could be used to help develop the expert, using the method of parameter adjustment outlined in chapter two. The

development process was envisaged to be achieved in several stages.

i) a discrete event simulation of the process is developed.

ii) the user manipulates the model in order to play the part of the process control system.

iii) simultaneously a PROLOG system monitors and records the actions of the user via the link. This information can be saved on disc.

iv) the information obtained by the PROLOG system is then used to form the basis of an expert system, which can be tested on the simulation via the link.

v) modifications to the expert may be made by amalgamating different experts together and/or refining the current expert by more monitoring of a user.

This experiment was carried out using the 'lorry' problem outlined above as a simple process. The apparatus for the experiment shall now be described with separate reference to the simulation and to the PROLOG system.

a) The Simulation

This was largely the same as for the previous example involving the 'lorry' problem. Differences occurred because in fact the PROLOG system contained three commands needed by the simulation:

i)control simulation; use the expert system to control the

process.

ii)monitor simulation; look at the user to adjust the computer experts parameters.

iii)save the recorded expert knowledge; save the calculated parameters.

Each of these commands required different parameters to be sent between expert and simulation.

(Table 7.1) Parameters Needed By Learning Expert

	Parameters	
	to expert	to simulation
control	weigh-in queue size	no. of weighbridges
	weigh-out queue size	
	lorry queue for loaders size	no. of loaders
	train queue for loaders size	
monitor	weigh-in queue size	none
	weigh-out queue size	
	lorry queue for loaders size	
	train queue for loaders size	
	no. of weighbridges	
	no. of loaders	
save	none	none

The choice of parameters was clearly dictated by the specific PROLOG commands' function. Thus 'control' needed to

know current demand for resources in order that it could calculate the required level of support. Having calculated this it then needed to send these results back to the simulation. The 'monitor' command needed to know what response (ie resource level) the user made to what demand level. Thus the expert needed to be sent all information about resource levels and demand. As in this mode the expert was not dictating to the user in any way, no values needed to be passed to the simulation.

The user could select 'control', 'monitor', or 'save' by setting a common variable 'icot' within the interactive facility of the simulation system.

```
icot = 1   -----> send monitor command
     = 2   -----> send control command
     = 3   -----> send save command
```

'icot' was then tested within an expanded 'expint' subroutine (the routine which interfaced with the expert) in order to branch to the correct section of code. This, together with changes to the 'rprams' and 'cfill' subroutines consistent with their definition, was the modification needed by the simulation.

```
      subroutine     expint
      integer   db
      include   'simdef'
      include   'lsim'
      common/ctype/icot
      call incomm
      goto (100,200,300), icot
100   plist(1) = isize(qwin)
      plist(2) = isize(qwout)
      plist(3) = isize(qtral)
      plist(4) = isize(qmerl)
      plist(5) = iatt(wpool,1)
      plist(6) = iatt(lpool,1)
      plist(7) = endl
      call clrc
      call cfill(1)
      call csend
      call clrc
      goto 990
200   plist(1) = isize(qwin)
      plist(2) = isize(qwout)
      plist(3) = isize(qtral)
      plist(4) = isize(qmerl)
      plist(5) = endl
      call clrc
      call cfill(2)
      call csend
```

```
      call clrc
      goto 990
300   call clrc
      call cfill(3)
      call csend
      call clrc
990   return
      end
```

A full listing of the Simulation is available in Appendix 13

b) The PROLOG system

This is best considered in its four separate sections, 'monitor', 'control', 'save', and 'test_expert'. Full listings are available in Appendix 13.

i) monitor. This records the users actions and stores them in a database. It does this by calculating exponentially weighted moving averages on the upper and lower bounds of queue lengths for each resource level.

For example, suppose the simulation has just started running with two loaders. The human user will keep the resource level at two loaders until the demand for loaders is either too great or two small. At this point the number of loaders is changed to match the demand. By monitoring the queue lengths throughout the time when two loaders were deemed a suitable number, and upper and a lower bound for this resource level can be calculated. This is then stored in the knowledge base as a fact:

bound(A,B,C,D). where,

A = resource name

B = resource level

C = upper bound for demand

D = lower bound for demand

eg. bound(1,2,3.123,0).

"resource ´1´ at quantity 2 is correct when demand lies between 0 and 3.123"

A similar process is carried out on weighbridges and loaders for all acceptable levels of resource. At some point a previously maintained resource level may well be retried. Thus a new upper and lower bound pair is calculated. This is then combined with the pair currently in the database to form a new stored ´bound´ fact. The method of combination used is the exponentially weighted average,ie.

<new bound> = A*<new observation> + (1 - A)*<old bound>

A = small constant (0.2)

Exponentially weighted average gives most bias to recent results. This is used rather than normal averages since it is expected that the user would improve his performance over time. One problem with exponentially weighted averages is the assumption that many observations are taken, since in the early stages of this average the first observation will have a very high weighting. Since some resource levels will

probably only rarely be used and thus observations are few, this is inadequate. Consequently for the first five observations of each resource level, a standard average is simulated by using a variable value for the constant 'A'.

observation 1 : A = 1

observation 2 : A = 0.5

observation 3 : A = 0.333

observation 4 : A = 0.25

observation 5 : A = 0.2

ii) control. This uses the information produced by the 'monitor' command in order to control the resources of the simulation. Two ways of looking at this parameter database were possible:

1. keep the resource levels constant as long as the demand (queue sizes) falls within the accepted limits. This allows maximum consistency and minimum disruption, but not minimum resource cost. This would be achieved by...
2. keep the resource level at a minimum for the current level of demand.

Both these logics were implemented by two closely related versions of the expert system (see Appendix 13).

At some points there may be demand levels for which the expert does not know a suitable resource level. These can be

called knowledge base gaps. The initial solution for 'control' when such a gap became apparent, was to set the resource level to a default of seven. This was easy to implement, but not logical since such gaps could occur anywhere, even for quite low demand levels. Thus a better solution was adopted. When such a gap becomes apparent, the expert system now sends an atom ('ping' - PROLOG inferred numerical gap) instead of a resource level. This indicates to the simulation that the problem is currently out of the expert systems learnt domain. The expert then re-enters monitor mode and the human is invited to interact with the simulation as before. Thus the link can be used to test a partially developed expert, allowing the user to append the system when knowledge gaps are apparent.

Another point to note as regards gaps in the knowledge base is the fact that upper and lower bounds are stored as real numbers, whereas queue sizes are always integer. Thus gaps could occur in the knowledge base because the system, if checking on actual values, would effectively round bounds down to the nearest integer. Because of this the 'control' system includes a mechanism whereby bounds are always rounded to the nearest integer before they are used. For accuracy of future bound calculation they are, however, still stored as reals.

iii) save. This is a simple PROLOG routine which enables a listing of the bound database to be written to a disc file.

This file can then be consulted at the beginning of a session and used by the 'monitor' or 'control' commands.

iv) test expert. As stated at the start of this chapter, an aim of this experiment was to consider how different human experts' databases might be combined. To enable this to be done it was necessary to have an objective way of measuring each databases' performance. Given that the aim is to minimise cost, then for each resource we are trying to minimise

V = M*Q + N*R

Q = queue size for resource

R = resource level

M and N = relative measures of cost between queuing and adding resources.

In the 'lorry' example an extra resource above two was twice the cost of an extra element queueing, and so M=1 and N=2.

If V were averaged out so that it gave a cost for resource R as an average per event, it could then be directly compared with other values from other databases. This is the method adopted here. Whilst the expert is in 'control' mode the compound values of V for loaders and weighbridges are stored in the database. These values are added to whenever the expert is called (after each event). The number of times this occurs is also added to the database. At the end of the simulation execution the user can execute the 'test_expert' predicate which evaluates the average scores for V.

The System Output

A typical 'bound' database after a short while (50 time units) in the monitor mode is as follows:

```
bound(1,1,3,0).
bound(w,1,0.34E+001,0).
bound(1,2,0.12E+001,0.4E+000).
bound(1,3,7,5).
bound(1,4,8,6).
bound(w,2,0.3632E+001,0.2512E+001).
bound(1,5,9,6).
bound(w,6,24,7).
bound(w,5,0.15E+002,0.12E+002).
bound(w,4,0.114E+002,0.94E+001).
bound(w,3,0.748E+001,0.492E+001).
bound(1,6,17,7).
```

Looking at this diagrammatically we get:

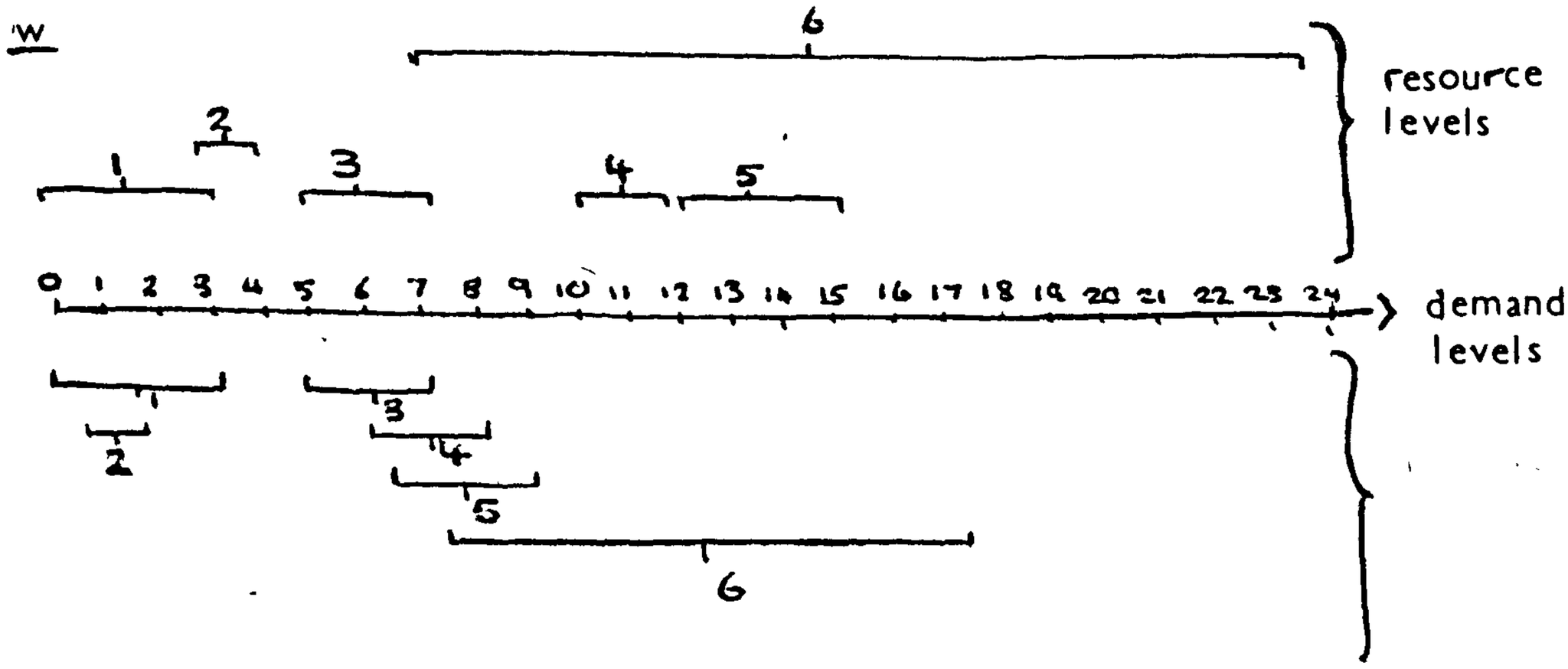


Figure 7.4 Gaps in a Learnt Database

Here we can see examples of gaps in the knowledge base:

weighbridges at queue level 4

loaders at queue level 4

It can also be seen that there is inconsistency in the database for w due to the wide range for resource level 6. For example a dropping of queues from 10 to 9 would result in an increase of weighbridges from 4 to 6. This is due to the fact that the 'monitor' period was very short (about 50 time units). A longer time in 'monitor' mode would have resulted in the expert system detecting more common reactions to queue sizes 8 and 9, thus plugging the gap between resource levels 3 and 4. That this is true can be seen by the fact that all the databases produced after 200 time units did not produce this inconsistency (see next section).

A Set of Experiments on the Utility of the Learning Expert

Given that the monitoring expert has been seen to work, we still need to ask how it might best be used. Some questions need to be answered.

1. Could we combine databases from several different human experts to gain an improved system ?
2. How should such a combination be performed ?
3. Would the performance of a human expert over several uses of the simulation be better than that of the combination of several less experienced experts ?

4. Can we regard the problem as composed of independent units whose database can be joined together with no detrimental effect ?

In order to test whether the learning expert could be used to control a simulation and to answer the above questions, a simple set of experiments were devised and these are detailed in this section.

Five students from different backgrounds with no prior knowledge of the problem were asked to control the simulation. For each of them the same random number stream was used. They used a special version of the simulation which automatically changed certain conditions at specific times. This meant that each user had to deal with a variety of conditions from low demand to high demand. In addition it ensured that all volunteers had to deal with exactly the same conditions for exactly the same number of simulated time units.

At the start of each session with a student, s/he was given a sheet outlining the 'lorry' problem (Appendix 14). The sheets also contained details on the experiments objectives. Each person was tested in isolation so that no inter-learning could take place. Having read the sheets the students were given a demonstration on how to use the simulation to change resource levels.

Each person who undertook the experiment had their databases (see Appendix 14) later used by the 'control' expert command. Using the 'test_expert' command these databases were given average costs for mixed demand,

performance where resource demand was low, and where resource demand was high. These tests were repeated for both types of expert logic (see 'control' above). These logic types were:

1. keep the resource levels constant as long as the demand (queue sizes) falls within the accepted limits. This allows maximum consistency and minimum disruption, but not minimum resource cost. This would be achieved by...
2. keep the resource level at a minimum for the current level of demand.

In the tables below, and throughout this discussion, the volunteers shall be identified by their subject of study.

a) Database Costs with Expert System under Logic 1

(Table 7.2)

	DEMAND					
	high		low		mixed	
	w	l	w	l	w	l
STUDENT						
law	29.121	13.411	11.541	9.859	21.971	11.967
maths	30.06	14.54	12.85	7.85	22.995	11.742
physics	26.911	14.21	12.566	9.06	21.159	12.145
business	30.192	14.725	13.68	8.372	20.338	12.971
french	26.93	15.64	10.52	9.0	22.547	11.783
average	28.643	14.505	12.231	8.828	21.802	12.122

where w = 'weighbridges', l = 'loaders'

b) Database Costs with Expert System under Logic 2

(Table 7.3)

	DEMAND					
	high		low		mixed	
	w	l	w	l	w	l
STUDENT						
law	27.05	15.32	10.9	8.729	20.54	12.66
maths	29.55	14.31	12.85	7.468	22.99	11.62
physics	29.25	14.36	12.01	8.607	22.477	12.09
business	30.008	14.438	11.65	7.728	20.559	13.089
french	26.693	16.016	10.807	8.435	22.648	11.747
average	28.525	14.889	11.643	8.193	21.843	12.241

where w = 'weighbridges', l = 'loaders'

Overall impressions from these results in some ways indicate the simplicity of the experiment undertaken. In particular there seems to be an inverse relationship between the performance of the databases as regards weighbridges and loaders. Where the weighbridge result for a database is better than for other database, the result for loaders is usually worse than average (in 5 out of 6 times above).

Similarly for the best loader results the corresponding weighbridge cost is above average (in 6 out of 6 times above). This implies a dependance between the two sections of the problem and so it is suggested that the best overall indication of a databases' performance is the combined average of both weighbridges and loaders. This is shown in the table below.

Overall Costs with Experts 1 and 2 (Table 7.4)

	DEMAND					
	high		low		mixed	
LOGIC->	1	2	1	2	1	2
STUDENT						
law	21.266	21.185	10.7	9.815	16.969	16.66
maths	22.3	21.93	10.35	10.159	17.369	17.303
physics	20.561	21.805	10.813	10.309	16.652	17.284
business	22.459	22.223	11.026	9.689	16.655	16.824
french	21.3	21.355	9.76	9.621	17.165	17.198
average	21.574	21.707	10.530	9.918	16.962	17.042

The first conclusion to be made from this study is that there is no significant difference between the performance of either expert system logic. With this sample, the average overall increased cost at any one time in using expert system 2 over expert system 1 is only

((17.042-16.962)/2)*unit_cost_of_a_resource

= 0.175*unit_cost_of_a_resource .

Thus the databases produced appear to control the expert system equally well, irrespective of which underlying logic was used. Since logic 2 uses minimum resource levels, it would probably suit a situation were resource prices were slowly rising, since it ensures that such resource levels are kept as low as possible. That this is true can be seen by using the same databases and simulation as before yet with the resource_cost:queue_cost ratio at 4:1 instead of 2:1 - see next table. If such a situation did arise the best thing to do would be to produce a new database, but as a short term solution expert system 2 would suffice.

Table 7.5 Expert Performance Under New Cost Ratio for Mixed Demand

	expert 1		expert 2	
	w	l	w	l
STUDENT				
law	31.971	18.569	29.686	17.612
maths	32.906	17.040	31.820	16.066
physics	31.267	17.791	30.248	16.814
business	30.571	18.453	29.112	17.013
french	31.877	16.956	30.034	15.721

A second look points to similar findings in that as well as there being little overall difference between the two expert systems' performances, it can be noted that a

database which performed well in one area on one expert logic tended to do well in that same area on the other expert. Take as an example the maths students database. This performed consistently well for loaders over both expert system logics.

Most important, though, is the fact that databases produced by the learning expert could be used to effectively control the simulation. Indeed performance was very similar to that of the humans for which the databases derived.

Having the results from the above tables, and noting that the strengths for different parts of the problem lay in different databases, it was relevant to ask whether any improvement could be made by combining parts of different databases together. It had already been noted that the two parts of the problem were not independent, so an improvement was not guaranteed by such a combination. In addition it was pertinent to ask how a straight average of all the databases would fair. Perhaps this pooling of expertise would improve the overall performance, or perhaps it would produce a mediocre database with high performance databases being cancelled out by low performance ones.

To test such possibilities several combinations of the databases were made and tested on their respective expert systems. These combinations were:

i. a straight average of all the databases. This was achieved by finding the numerical average of the upper and lower bounds for each [resource,level] pair.

ii. the best overall performers on expert 1. This meant the

weighbridge database of the business student, plus the loader database of the maths student.

iii. the best overall performers on expert 2. The weighbridge database of the law student, plus the loader database of the maths student.

iv. the best individual section performers on expert 1. This meant the low and high demand weighbridge databases from the french and law students, the low demand loader database from the maths student, and the high demand loader database from the law student.

v. the best individual section performers on expert 2. This meant the low and high demand weighbridge databases from the french student, and the low and high demand loader databases from the maths student.

Combinations ii and iii were achieved by simply using the 'bound' facts for weighbridges from one database with the 'bound' facts for loaders from another database. Combinations iv and v were somewhat more difficult in that for each resource the database would come from two different sources. The 'low' and 'high' demand parts of the database were defined as being the 'bound' facts for resource levels of 3 and under and 4 or more respectively. When combining the databases in this way (using a simple text editor) it was clear that knowledge base gaps between resource levels 3 and 4 could occur. These gaps were plugged by simply modifying the upper bound for resource level 3 and the lower bound for resource level 4 to meet halfway. The results of

these databases when tested are shown in the tables below.

The databases are in Appendix 14.

a) Costs with Expert System under Logic 1 (Table 7.6)

	DEMAND					
	high		low		mixed	
	w	l	w	l	w	l
COMBINATION						
i.	27.826	15.264	15.355	9.614	22.256	12.606
ii.	27.832	14.386	11.811	8.447	21.156	11.912
iv.	28.532	14.274	11.678	8.405	21.726	11.904

where w = 'weighbridges', l = 'loaders'

b) Costs with Expert System under Logic 2 (Table 7.7)

	DEMAND					
	high		low		mixed	
	w	l	w	l	w	l
COMBINATION						
i.	28.177	15.602	14.077	8.466	21.926	12.438
iii.	27.319	14.543	11.988	8.224	20.835	11.871
v.	27.910	14.885	11.720	8.427	21.402	12.289

where w = 'weighbridges', l = 'loaders'

c) Overall Average Costs of Databases (Table 7.8)

	DEMAND					
	high		low		mixed	
LOGIC->	1	2	1	2	1	2
COMBINATION						
i.	21.545	21.890	13.583	11.272	17.431	17.182
ii.	21.109	-	10.129	-	16.534	-
iii.	-	20.931	-	10.106	-	16.353
iv.	21.403	-	10.042	-	16.815	-
v.	-	21.398	-	10.074	-	16.846

Comparing these results with those for the individual databases, the following tentative conclusions have been drawn.

1. The combined databases do not perform as well in specific areas as those databases from which they were formed. For instance, combination iii (overall best for expert 1) did not perform as well for weighbridges as the business student database, or for loaders as well as the maths students database. This is despite being formed from the two databases. This is due, again, to the interdependence of the two parts of the problem. By changing one half of the database you are changing the conditions under which the other has to operate. This leads to different cost achievements. Thus it is concluded that combining databases

is most unlikely to result in any improvement in the respective individual areas of concern.

2. Looking at overall average costs, which give the best objective view of the experts´ entire performance, gives a rather different picture. In the six groups for which costs were calculated, the combinations ii, iii, iv, and v performed very well in comparison with the individual databases. In three of the groups they recorded the lowest cost, with the second lowest cost recorded in two other groups. This is significant since in cases for experts where no method of ´scoring´ is available (other than the judgement of the human expert) these results indicate that a high quality expert is likely to be attained through combination, and so this is a strategy should at least be tried.

3. In looking at the combinations results more closely it can be seen that the databases involving the best overall performers on the individual expert systems, performed better than other combinations in both high and general level demands. This was at first surprising since it was at first expected that combinations iv and v being constructed from both low and high demand databases would produce a better expert. However, when the inter-independence of the problem is taken into account, together with the problems of merging high and low demand databases together (with the resultant changes needed to the databases) such results become understandable.

4. The combination involving a straight average of all the individual databases produced a mediocre expert system. As stated above this is felt due to a 'cancelling out' effect on high and low performance databases.

5. Overall, it was shown that databases which learnt from different sources could be combined to produce new decision criteria which could then be applied to the original problem. Although the sample taken was very small, it was sufficient to test this basic premise.

Conclusions

The work carried out in this chapter has shown that the development of an interface between visual simulations and expert systems has several practical uses. These uses have been tested in this chapter, and as a result some possibilities for further extending the work of this thesis have been bought forward.

In the area of learning by parameter adjustment the work presented here indicates that the best way to develop expert systems for simulation control with this method is to use an intelligent combination of individuals databases.

Work on the expert system components of the systems presented in this chapter suggests that plenty of overlap exists in the methodology of development. The AGV expert has shown that for problem areas akin to industry, it would be

useful if a special knowledge representation language were developed for expert systems that form the intelligent component of a visual interactive simulation. This would relieve the O.R. analyst of having to learn PROLOG in any more detail than actually necessary. Also it would speed up development by replacing often repeated sections of code with a single statement.

The AGV problem also illustrated how an expert system could be used for more than one simulated problem. By carefully producing a suite of expert systems they may form the basis for intelligent components for several expert systems, further speeding up development time.

The most striking point is the ease with which intelligent components can be included in visual interactive simulations with good results.

- CHAPTER 8 -

CONCLUSIONS AND FUTURE RESEARCH

This chapter presents a summary of the research carried out for this thesis, together with conclusions and recommendations for future research.

Research Summary

The research concerned the introduction of modern A.I. techniques into the area of visual interactive simulation (V.I.S.). The principal problem with V.I.S. lies in its inflexibility. In simulating a problem using conventional languages the analyst is assuming the problem to be well structured. In reality few such problems exist. In addition many rules which govern elements of the simulation are very complicated and uncertain. Coding them with conventional languages can be very cumbersome and of course renders them difficult to modify. On the other hand, many techniques of A.I. are particularly well suited to unstructured tasks. As well as this, the modular approaches to system building means that programs can be easily modified, even whilst they are running. Consequently it appeared that a merger of the two technologies could be beneficial.

The first stage of the research was to undertake a detailed literature study on the topics of direct interest to this work. This covered the areas of decision making and

support, problem solving methodologies, and Artificial Intelligence. The models for simulation were examined, with particular detail to the much favoured three-phase method. With the research undertaken on current A.I. techniques a relationship between this method and the constructs of A.I. languages was established. Such a relationship was tested by the development of a simulation engine in PROLOG. The manner with which this engine was constructed indicated this languages high suitability both as a simulation language, and as language for containing simulation logic. With this in mind, experiments on developing intelligent systems with this language were undertaken. The intelligent AGV routing program that resulted was designed to control a robot around a general maze. This problem area was chosen since it meant that the intelligent program could form a module to contain the logic part of the simulation of a robot.

With the intelligent program (expert system) written experiments were performed to link this expert with simulations. A simulation written in PROLOG was first used as the simplest start to such work. Having merged the two systems successfully work was then undertaken to develop a package which allowed a PROLOG intelligent module to interact remotely with a simulation on another machine. This computing environment was chosen in preference to multi-tasking machines as it more closely related to the type of environment typically found in small and large industry. This remote provision was designed to allow

integration with a conventional simulation, to allow parallel processing to occur, and to enable standard personal computers (as widely used in industry) to be used. Once developed the link was experimented with to interface expert systems to simulations for several different functions. These allowed the PROLOG program to:

i) act as a process control system

ii) contain controlling simulation logic

iii) be visually tested and modified on a simulation

iv) learn by parameter adjustment from actions by the user on the simulation.

It was concluded from these tests that the introduction of A.I. techniques to V.I.S. adds a new dimension to that tool.

Conclusions

Several conclusions can be drawn from the research undertaken here. These are best sub-divided into several areas.

PROLOG as a Simulation Language

PROLOG is suited both theoretically and practically as a simulation language. The logic of simulations has been separated out with the simulation engine making simulations very easy to write. But as well as just being an alternative simulation language, the PROLOG simulation engine has the benefit of extending levels of interaction, so that the

state entity diagram can be changed whilst the simulation is running. This allows alternative paths to be tested in a single simulation run. Even the current problem concerning PROLOGs speed is only temporary. With new language compiler and computer developments execution speed of PROLOG is set to increase sharply. Already the newest implementations of PROLOG run some 20 times the speed of the version used in this PhD. PROLOGs other principal problem, that of poor arithmetic capacity, can be overcome by using assembler code for complicated calculations.

The work described in Chapter 2 in conjunction with the theoretical observations of Crookes (1982), O'Keefe (1984) and Adelsberger (1984) indicates that PROLOG is well suited to containing the controlling logic for simulations written in more conventional simulation languages. A principal purpose behind the investigation of PROLOG as a simulation language has been that it has provided a basis for investigating expert system technology. By emulating discrete event simulation methods using PROLOG it has been possible to investigate the advantages of having common code/data facilities.

Merging Simulation with A.I.

Having developed the inter-processor link it has been possible to test practically the merger of A.I. techniques with simulation. Also by producing the link such a merger has been tested on computer systems typically found in small as well as large industrial units. With this link it has

been shown that merging a simulation with an A.I. program is simple to perform. The link has been developed in such a way as to ensure ease of use. This was felt important, since a tool which is hard to use is less likely to be adopted by O.R. scientists than one which is relatively simple to use.

This link has been designed so that no change to the underlying structure of the simulation or expert system is required. Despite this full expert facilities are available to the simulation and its user. Thus the user can direct or interrogate the expert from the simulation monitor. By looking at the form of the program interface at the simulation to the expert it has been possible to generalise it and therefore have a model for all interfacing of this kind. This removes all ambiguity in how to adopt the link and ensures that overall system development time is not affected by considerations of the interface.

The working link developed is designed to work with conventional Algol type languages, FORTRAN and PASCAL. The link allows remote connection to a PROLOG program, whilst still allowing the full interaction facilities of the simulation.

It was found that PROLOG communication with conventional languages involved matching the data types between PROLOG and the language concerned. The list data structure, so vital to PROLOG program execution proved the most difficult to reproduce in conventional languages.

Despite this, it did prove possible to use standard data structures in a way that facilitated list representation.

Using A.I. with Simulation

Having concluded that A.I. programs could technically be incorporated into simulations, it was necessary to experiment on possible applications. This work is detailed in the previous chapter. That work showed that the development of an interface between visual simulations and expert systems has several practical uses.

i) Manipulation of Parameters. An expert system could be linked with a simulation in order to control resource levels within that simulation. Thus the link could be seen as a way for testing process control systems (PROLOG) on visual simulations. It allows the process control system to be tested to its limits in a totally safe environment.

ii) Containing Simulation Logic. Here, an expert system can be used to contain some of the logic of the simulation. Many of the problems to which simulation is called upon are semi-structured, and as Moreira da Silva (1982) points out, a problem with using simulations for such problems is that they are inflexible. Once decision rules are coded into a simulation they are fixed. Such rules can be coded separately from the simulation in PROLOG with the result of a greatly more flexible simulation.

iii) Expert System Acceptability and Development. Using the link we can test the application of an expert system on a visual simulation of the problem, thereby gaining user

acceptance of the system. In addition the expert system could be tested against a well known human expert on the same simulated problem thus providing users with a control with which to compare the system.

iv) Learning by Parameter Adjustment. As indicated in the literature chapter, a main way of developing a learning expert system is by parameter adjustment. This means that the expert has a set of rules which are triggered according to the values of certain parameters. The levels of these parameters are adjusted according to the effects of monitoring a human performing the task later to be done by the expert system. The link can be used to help develop a process control system in this way. Some tests have been performed to try and find the best way of using such a method for expert system development/refinement. Although by no means conclusive, this work indicates that a good way to develop expert systems for simulation control with this method is to use an intelligent combination of individuals databases. Such a combination would involve using those parts of individuals parameter databases that performed better than those of other individuals. Whilst individual sections of a parameter database are not independent of each other (and therefore performance of a section will change when other sections are modified) it has still been seen that a combined parameter database performs consistently well over all areas of a problem.

Future Work

In this section those areas thought to be worthy of future research are outlined. It would be probable, however, that in the course of such work new areas of research would be uncovered.

i) PROLOG as a Simulation Language. As stated in the chapter dealing with this topic, the version of PROLOG used in developing the simulation language is rather slow. Advances are continually being made, however, in this regard with the development of mixed interpreters and compilers. The structure for compiled PROLOG is somewhat more rigid than for interpreted PROLOG so that conversion of the Engine to run on faster versions of PROLOG would require some re-structuring of the Engine.

Because of the way the simulation engine was incrementally developed inclusion of attributes to the system was rather messy, since it was developed around the rest of the engine. An improvement (which would involve some detail changes in the entire package) would occur if entities in the Engine were replaced with attribute lists. This would eliminate the need for two data structures for each entity and would reduce the size of database in the system.

ii) Expert Systems for Simulations. Future work has already been alluded to earlier in this thesis. Firstly, on applications tested in this thesis, a similarity is apparent

between the different expert systems. This similarity is due to

a) the common method in producing expert systems of production rules

b) the common manipulation of simulation parameters for this application of expert systems in particular.

With this in mind investigations should be undertaken on how to reduce the burden of programming such expert systems. On idea may be to use a currently available shell with a modified front end that would deal with parameter communication between expert and simulation. A problem with shells is that they restrict representation of knowledge, so that the user must fit their problem into such a structure. Perhaps a better idea might therefore be to produce a set of PROLOG predicates that replace often used sections of code with single commands. Such predicates could hide the intricacies of PROLOG program development such as the use of backtracking and pattern matching.

Related to this is the idea that expert systems, such as the AGV expert developed and used in this thesis, could be developed as general systems that could deal with many different simulations. Such experts would be controlled by a set of production rules specific to the particular problem being simulated. This could be viewed as a further extension in the development of a shell for expert systems used in conjunction with simulations.

iii) Parameter Learning Expert Systems for Simulations.

Tests carried out in the previous chapter indicated that intelligent combination of individual parameter databases was the best way to utilise the method of learning by parameter adjustment. Two problems exist here that merit further work.

Firstly, the results obtained in the previous chapter were obtained from a only a very small sample. If more conclusive theories are to be developed from this work far more extensive tests need to be made. In addition statistical comparison of different databases should be carried out.

The second problem involved the mode of combination of databases, which was performed by hand. This is seen as a break in the automatic development of parameter based expert systems and involves some ad hoc decisions to be made by the analyst. Research should be carried out into the automation of this process. Such a system would take into account the results of the detailed study outlined immediately above.

iv) Using Simulations with Expert Systems. The areas to which the link has been applied have all been examined in the safe environment of a University. The next stage in testing the usefulness of such a set up should be in using it for actual current industrial problems. Even the flexible manufacturing example used in the previous chapter was taken from an old industrial problem. Thus it was possible to implement and test it wholly within the confines of the

University. Whilst this was sufficient to see whether such a simulation/expert link up was of practical use, it did not offer a realistic model of system development. Unseen problems could arise when attempting to produce a similar system under industrial conditions.

Also, as noted in the previous chapter, one currently perceived application for the interface of simulation and expert systems is in the area of models for expert refinement. The idea for this particular application resulted following a visit to the Expert Systems'85 conference at Warwick University. At this conference there was a $100 000 expert system which worked as a chemical plant monitoring system. It operated as a standard expert system, but when difficult problems arose, it would run a model to simulate the problem and the process systems reaction to different decisions, so as to select the best decision. It is suggested that the simulation-expert link could give an economic way of providing similar facilities.

v) Simulation Parameter Verification. The work of this PhD has led to the research topic of Simulation Parameter Verification being currently undertaken at Warwick University.

Visual simulation has greatly enhanced the understanding of a simulated problem by people not conversant with computer techniques. It is unfortunate that this advantage has directly led to a new problem. Since the model is so easy to understand and interact with, it can

sometimes occur that decisions are made on a single experiment with the model. For instance resource levels such as manning might be fixed on the basis of a single run, and then implemented on the factory floor. Since many timing factors within a simulation are governed by random statistical distributions it could well occur that in a single experiment extraordinary pressures occur within part of the simulated system. Thus decisions based on rare occurrences could be implemented on the factory floor.

For people with knowledge of simulation, the best way to proceed is via a careful set of experiments. Again unfortunately many people likely to use visual simulations are either unaware of this or unable to devise a set of experiments. It is this problem area that the research at Warwick is trying to tackle. A PROLOG program, attached to a simulation program via the link devises and executes a set of experiments on the simulation. The PROLOG program would initially enquire of the user what parts of the simulation are critical and need to be statistically examined. With these referenced a set of experiments would be devised and executed. After execution of the experiments, statistical results and recommendations should be output to the user.

BIBLIOGRAPHY

ADAMS, J.B.
"A Probability Model of Medical Reasoning and the MYCIN model", Mathematical Biosciences, 32, YEAR?, 177-186.

ADELSBERGER, H.H.
"PROLOG as a Simulation Language", in S. Sheppard, U. Pooch, D. Pegden (Eds) Proceedings of 1984 Winter Simulation Conference.

d'AGAPEYEFF, A.
Report to Alvey Directorate on a short Survey of Expert Systems in U.K. Business, Alvey, Feb 1984.

AGIN, G.J,
"Computer Vision Systems For Industrial Inspection and Assembly", Computer, 13, 1980, 11-20.

AIKINS, J.S.
"Prototypical Knowledge for Expert Systems", Artificial Intelligence, 20, 1983(2), 163-210.

ALTER, S.L.
Decision Support Systems - Current Practice and Continuing Challenges, Addison Wesley, 1980.

ALVEY, DIRECTORATE
"Alvey Programme - Annual Report 1984", IEEE, Nov 1984.

ALVEY, DIRECTORATE
The Esprit Programme 1985 - Fact Sheet, Alvey, 1985a.

ALVEY, DIRECTORATE
"Alvey News no. 9 February 1985", IEEE, 1985b.

BALL, C.R., NEWPORT, P.J.
Simulation of Conveyor 129, BL Systems Ltd, 1981.

BALLARD, B.W.
"The *-Minimax Search Procedure for Trees Containing Chance Nodes", Artificial Intelligence, 21, 1983, 327-350.

BALMER, D.W., PAUL, R.J.
"CASM - the Right Environment for Simulation", J.O.R.S., 37, pp 443-452, 1986.

BARR, A., FEIGENBAUM, E.A.
The Handbook of Artificial Intelligence Vol. 1, Pitman London, 1981.

BARSTOW, D.R.
"The Roles of Knowledge and Deduction in Algorithm Design", in Hayes, Michie, Pao (Eds) Machine Inteligence 10, Ellis Horwood, 1982, 361-381.

BARTON, C.
The Relevance of PROLOG to Robotics, MSc Disseration, SIBS, Warwick University, 1984a.

BARTON, C.
Memo for Record: Conversation with M Larcombe held 16 July 1984, Internal Memo, SIBS, Warwick University 1984b.

BARTON, C.
Memo for Record: Conversation with M Larcombe held 26 July 1984, Internal Memo, SIBS, Warwick University 1984c.

BELL, M.Z.
"Why Expert Systems Fail", J.O.R.S., 36, 1985, 613-620.

BENET, J.L. (ED)
Building Decision Support Systems, Addison-Wesley, 1983.

BIRMINGHAM, UNIVERSITY.
ECSL (Extended Control and Simulation Language); CAPS (Computer Aided Programming System) : Detailed Reference Manual, Lucas Institute for Engineering Production, 1980.

BIT, LTD.
PARYS: A new Generation of Management Software, Bit, 1985.

BOND, A. (ED)
MACHINE INTELLIGENCE - State of the Art Report, Pergamon Infotech, 1981.

BOOTHROYD, H.
Articulate Intervention, Taylor and Francis, 1978.

BOWEN, H.C., FENTON, R.J., ROGERS, M.A., HURRION, R.D., SECKER, R.J.
"Interactive Computing as an aid to Decision Makers", in Haley, K.B. (ed),O.R. 1978, North Holland, 1979, 829-842.

BRAMER, M.A.
"Expert Systems: The Vision and the Reality", in Bramer, M.A. (ed), Research and Development in Expert Systems, CUP, 1985a.

BRAMER, M.A. (ED)
Research and Development in Expert Systems (Proceedings of 4th Technical Conference of the BCS Specialist Group on Expert Systems, University of Warwick 1984), CUP, 1985b.

BRODA, K., GREGORY, S.
PARLOG for Discrete Event Simulation, Research Report DOC84/4, Dept. of Computing, Imperial College, March 1984.

BROWN, J.C.
Visual Interactive Simulation: Further Developments Towards a Generalised System and its use in 3 Problem Areas Associated with a High Technology Company, MSc Thesis, SIBS, Warwick University, 1978.

BUCHANAN, B.G.
"New Research on Expert Systems", in Hayes, Michie, Pao (eds), Machine Intelligence 10, 1982, 269-300.

BUNDY, A.
Artificial Intelligence: an Introductory Guide, Edinburgh, 1980.

BURGHES, D.N., WOOD, A.D.
Mathematical Models in the Social, Management and Life Sciences, Ellis Horwood, 1980.

CARHART, R.E.
"CONGEN: An Expert System Aiding the Strucural Chemist" in Michie, D. (ed), Expert Systems in the Micro-elecronic Age, EUP, 1979.

CHRISTY, D.P., WATSON, H.J.
"The Application of Simulation : A Survey of Industry Practice", Interfaces, 13, pp 47-55, 1983.

CLARK, K.L., McCABE, F.G., HAMMOND, P.
"PROLOG : a Language for Interpreting Expert Systems", in Hayes, Michie, Pao (eds), Machine Intelligence 10, pp 455-476, ELLIS HORWOOD, 1982.

CLEMENTSON, A.T.
Computer Aided Programming for Simulation, The Lucas Institute for Engineering Production, Birmingham University, 1974.

CLOCKSIN, W.F., MELLISH, C.S.
Programming in PROLOG, Springer-Verlag, 1981.

CONWAY, R.
"Some Tactical Problems in Digital Simulation", Management Science, 10, pp 47-61, 1963.

CONWAY, R.W., JOHNSON, B.M., MAXWELL, W.L.
"Some Problems of Digital Systems Simulation", Management Science, 9, pp 92-110, 1959.

CROOKES, J.G.
"Simulation in 1981", E.J.O.R., 9, pp 1-7, 1982.

CROOKES, J.G., BALMER, D.W., CHEW, S.T., PAUL, R.J.
"A Three-Phase Simulation System Written in PASCAL", J.O.R.S., 37, pp 603-618, 1986.

DAVEY, P.G.
Robots R and D in the U.K., SERC, 1982.

DAVIS, R., BUCHANAN, B., SHORTLIFFE, E.
"Production Rules as a Representation for a Knowledge Based Consultation Program", Artificial Intelligence, 8, pp 15-45, 1977.

DAVIS, R., KING, J.
"An Overview of Production Systems", in Elcock, E.W., Michie, D. (eds), Machine Inteligence 8, Chichester & Ellis Horwood, 1977.

DIGITAL
News - 21st September 1984, press issue from DEC, 1984.

DONNEY, M.C., EDWARDS, P.J., GREEN, J.A., MARSHALL, J.L., YONG, J.F.
"The Simulation of Industrial Robot Systems", OMEGA, 12, pp 273-281, 1984.

DOUKADIS, G.I., PAUL, R.J.
"Research into Expert Systems to aid Simulation Model Formulation", J.O.R.S., 36, pp 319-325, 1985.

DUDA, R., GASCHING, J., HART, P.
"Model Design in the Prospector Consultant System for Mineral Exploration", in Michie, D. (ed), Expert Systems in the Micro-Electronic Age, EUP, 1979.

EDEN, C., JONES, S., SIMS, D.
Messing About in Problems, Pergamon, 1983.

ESI
PROLOG-1 Language Reference Manual, ESI, 1983.

ESI
Introducing Expert Ease, ESI, 1985a.

ESI
Expert Ease; A Technical Overview, ESI, 1985b.

ESI
PROLOG-2 ; A Description for the Programmer, ESI, 1985c.

EVANS, C., ROBERTSON, A.
Cybernetics, Butterworths, 1968.

FEIGENBAUM, E.A.
"Themes and Case Studies of Knowledge Engineering", in Michie, D. (ed), Expert Systems in the Micro-Electronic Age, EUP, 1979.

FINDLER, N.V., MELTZER, B. (EDS)
Artificial Intelligence and Heuristic Programming, EUP, 1971.

FISHMAN, G.S.
Concepts and Methods in Discrete Event Simulation, Wiley, 1973.

FISHMAN, G.S.
Principles of Discrete Event Simulation, Wiley, 1978.

FLITMAN, A.M.
A Guide to PROLOG, York University, 1982.

FORRESTER, J.W.
Industrial Dynamics, M.I.T., 1969.

FOSTER, J.M.
List Processing, Macdonald and co., 1967.

FRANTA, W.R.
The Process View of Simulation, North-Holland, 1977.

FREIDBERG, R.M.
"A Learning Machine pt. 2", IBM Journal of Research and Development, 3, pp 282-287, 1959.

FUTO, I., SZEREDI, J.
"A Very High Level Discrete Simulation System T-PROLOG", Computational Languistics and Computer Languages, 15, pp 111-131, 1982.

GASCHING, J.
"Application of the PROSPECTOR system to Geological Exploration Problems", in Hayes, Michie, Pao (eds), Macine Intelligence 10, Ellis Horwood, pp 301-323, 1982.

GENERAL RESEARCH CORPORATION
TIMM the Intelligent Machine Model : Introduction, GRC Santa Barbara, California, 1983.

GENTIL, F., PRODO, G.
"Guided Vehicle Systems at Renault", in Proceedings of 1st International Conference in Automated Guided Vehicle Systems, IFS, pp 59-66, 1981.

GEORGE, F.
Models of Thinking, Allen and Unwin, 1970.

GEORGEOFF, M.G.
"Strategies in Heuristic Search", Artificial Intelligence, 20, pp 393-425, 1983.

GHOSAL, A., HEATHCOTE, E.
"On Dynamic Optimization Problems", Cybernetica, 15, pp 290-305, 1972.

GRANT, T.J.
"Lessons for O.R. from A.I.: a Scheduling Case Study", J.O.R.S., 37, pp 41-59, 1986.

GORRY, G.A., SCOTT MORTON, M.S.
"A Framework for Management Information Systems", Sloan Management Review, 13, pp 55-70, 1971.

GOTTINGER, H.W.
"Towards a Fuzzy Reasoning in the Behavioural Science", Cybernetica, 16, pp 113-135, 1973.

GREEN, B.F.
"Computer Models of Cognitive Processes", Psychometrika, 26, pp 85-91, 1961.

HAESSLER, R.W.
"Developing an Industrial Grade Heuristic Problem Solving Procedure", Interfaces, 13, pp 62-71, 1983.

HARMON, P. (Ed)
Expert Systems Strataqies Vol 1, Cahners, 1985.

HARRISON, N.
"Knowledge Base Builders", Systems International, Aug 1984.

HARTLEY, J.
FMS at Work, (chapter 1), IFS, 1984.

HAWKINS, D.
"An Analysis of Expert Thinking", International Journal of Man-Machine Studies, 18, pp 1-48, 1983.

HAYES-ROTH, F., WATERMAN, D., LENAT, D. (eds)
Building Expert Systems, Addison-Wesley, 1983.

HEBDITCH, D.
"How Friendly are Expert Systems ?", Datamation, pp 15-19, Feb 1984.

HOLLINGSWORTH, R.
Applications of Visual Interactive Modelling, Istel.

HOVLAND, C.I., HUNT, E.B.
"Computer Simulation of Concept Attainment", Behavioral Science, 5, pp 265-267, 1960.

HURRION, R.D.
The Design, Use and Required Facilities of an Interactive Visual Computer Simulation Language to Explore Production Planning Problems, PhD Thesis, University of London, 1976.

HURRION, R.D., SECKER, R.J.R.
"Visual Interactive Simulation : an Aid to Decision Making", Omega, 6, pp 419-426, 1978.

HURRION, R.D.
"An Interactive Simulation System for Industrial Management", E.J.O.R., 5, pp 86-93, 1980.

HURRION, R.D.
"Visual Simulation Using a Microcomputer", Computing and Operations Research, (U.K.), 8 pp 267-273, 1981.

INSTITUTE
Proceedings of the 1st International Conference on Automated Guided Vehicle Systems 2-4 June 1981, Stratford Upon Avon, IFS, 1981.

INSTITUTE OF NEW GENERATION COMPUTER TECHNOLOGY
Outline of Research and Development for 5th Generation Computer Systems, INGCT, April 1983.

ISI
SAVOIR - Technical Description, ISI, 1984a.

ISI
COUNCELLOR - A SAVOIR Case Study, ISI, 1984b.

ISI
SAVOIR - Powerful Flexible Expert Systems for General Application, ISI, 1984c.

ISI
EXPERT SYSTEMS - What They are and What They Need, ISI, 1984d.

JACKSON, P.C.
Introduction to Artificial Intelligence, Petrocelli Books, 1974.

JIPDEC
Preliminary Report on Study and Research on 5th Generation Computers 1979-1980, Japan Information Processing Development Center, 1981.

JOHNSON, J.
"Expert Systems for You", Datamation, Dec 1983.

KASTNER, J.K., HONG, S.J.
"A Review of Expert Systems", E.J.O.R., 18, pp 285-292, 1984.

KEEN, P.G.W., SCOTT MORTON, M.S.
Decision Support Systems - An Organisational Perspective, Addison-Wesley, 1978.

KING, R.A.
The IBM PC DOS Handbook, Sybex, 1983.

KLEINMUNTZ D.N., KLEINMUNTZ, B.
"Decision Strategies in Simulated Environments", Behavioral Science, 26, pp 294-305, 1981.

KNUTH, D.E.
The Art of Computer Programming Vol. 2, (ch3), Addison Wesley, 1969.

KULIKOWSKI, C.A.
"Artificial Intelligence Methods and Systems for Medical Consultation", in IEEE Transactions on Pattern Analysis and Machine Intelligence Vol. PAMl-2, pp 464-476, 1980.

LASKI, J.G.
"On Time Structure in (Monte Carlo) Simulations", O.R.Q., 16, pp 329-339, 1965

LAURIE, P.
"The Intelligent Database", Systems International, Aug. 1984.

LEE, J.K., HURST, E.G.
Solving Semi-Structured Problems and the Design of Decision Support Systems: Post-Model Analysis, ORSA/TIMS, Orlanda, November 1983.

LEMBERSKY, M.R., CHI, U.H.
"Decision Simulators Speed Implementation and Improve Operations", Interfaces, 14, pp 1-15, Aug. 1984.

LEMMONS, P.
"Japan and the 5th Generation", Byte, pp 394-401, Nov. 1983.

LENAT, D.B.
"The Nature of Heuristics", Artificial Intelligence, 19, pp 189-249, 1982.

LOVELAND, D.W.
Automated Theorem Proving : a Logical Basis, North Holland, 1978.

MARCE, L., JULIERE, M., PLACE, H.
"An Autonomous Computer Controlled Vehicle", in Proceedings of the First Internationa Conference on Automated Guided Vehicle Systems, IFS, pp 113-122, 1981.

MARTINS, G.R.
Better Simulation Models for Decision Support, The Rand Corporation, Santa Monica, California.

MAYER, R.E.
Thinking and Problem Solving : An Introduction to Human Cognition and Learning, Scott Foresman and Co, 1977.

McCALLA, REID.
"Plan Creation, Plan Execution and Knowledge Aquisition in a Dynamic Microworld", Int. J. Man-Machine Studies, 16, 1982.

van MELLE, W.J.
System Aids in Constructing Consultation Programs, UMI Research Press, 1981.

MICHALSKI, R.S., CHILAVSKY, R.L.
Knowledge Aquisition by Encoding Expert Rules vs. Computer Induction from Examples : a Case Study Involving Soybean Pathology, Int. J. of Man-Machine Studies, 12, pp 63-87.

MICHIE, D. (ED)
Expert Systems in the Micro-Electonic Age, EUP, 1979.

MICHIE, D.
"Expert Knowledge Re-Engineered", Computer Bulletin, pp 6-8, June 1981.

MINTZBERG, H.
The Nature of Managerial Work, Harper and Row, 1973.

MOREIRA da SILVA, C., HURRION, R.D., SWANN, W.H., TOSNEY, P.J.
A Decision Support System for the Planning and Control of Complex and Interrelated Manufacturing Units, S.I.B.S. Warwick University, 1980.

MOREIRA da SILVA, C.A.R.
The Development of a Decision Support System Generator via Action Research, PhD Thesis, S.I.B.S., Warwick University, 1982.

MORRIS, E.W.
"Developments in Guided Vehicle Systems - Possibilities and Limitations and Economics of Their Operation", in Proceedings of the 1st International Conference on Automated Guided Vehicle Systems, IFS, pp 67-68, 1981.

MORSE, A.C., KOHLER, R., SUTHERLAND, M.
The Development of an Intelligent Trainable Graphic Display Assistant for the Decisionmaker, Intelligent Software Systems Inc, 1982.

MYCROFT, A., O'KEEFE, R.A.
"A Polymorphic Type System for PROLOG", Artificial Intelligence, 23, pp 295-308, YEAR ??.

NIXON, E.
"Information Technology: the Way Ahead", J.O.R.S., 37, pp 1-12, 1986.

O'GREEN, J.
"(Getting) Computers Smarter", Popular Computing, pp 97-104, Jan. 1984.

O'KEEFE, R.M.
Developing Simulation Models an Interpreter for V.I.S., PhD Thesis, Faculty of Mathematical Studies, Southampton University, 1984.

O'KEEFE, R.M.
"Expert Systems and Operational Research - Mutual Benefits", J.O.R.S., 36, pp 125-129, 1985.

OSTLER, N.
"Japan : A Difference of Emphasis", Systems International, pp 69-70, April 1985.

OXFORD UNIVERSITY
Oxford English Dictionary, Clarendon Press, 1933.

PAUL, R.J., DOUKADIS, G.I.
"Further Developments in the use of Artificial Intelligence Techniques which Formulate Simulation Problems", J.O.R.S., 37, pp787-810, 1986.

PECK, S.N.
"Intermediate Generality Simulation Software for Production", J.O.R.S., 36, pp 591-595, 1985.

PHELPS, R.I.
"Artificial Intelligence - an Overview of Similarities with O.R.", J.O.R.S., 37, pp 13-21, 1986.

POLITAKIS, P., WEISS, S.M.
"Using Empirical Analysis to Refine Expert System Knowledge Bases", Artificial Intelligence, 22, pp 23-48, 1984.

POLLARD, H.
Corporate Planning Model for Laura Ashley, Delloitte Haskins and Sells (Decision Systems), 1986.

POLYA, G.
How to Solve It, Princeton UP, 1947.

POTTER, T., GUILD, I.
Robotics, Usborne, 1983.

QUINLAN, J.R.
"Discovering Rules by Induction From Large Collections of Examples", in Michie, D. (ed) Expert Systems in the Micro-Electronic Age, EUP, 1979.

QUINLAN, J.R.
"Fundementals of the Knowledge Engineering Problem", in Bond, A. (ed) Machine Intelligence - State of the Art Report, Pergamon Infotech, 1981.

RADZIKOWSKI, P.
Perspectives of the Business Decision Support Expert Systems, TIMS/ORSA, 1983.

RICH, E.
Artificial Intelligence, Mcgraw Hill, 1983.

RIETZ, F.
Cheaper, Better and Quicker - Warehouse Design using V.I.S., Istel.

ROBINSON, J.A.
"The Logical Basis of Programming by Assertion And Query", in Michie, D. (ed) Expert Systems in the Micro-Electronic Age, EUP, 1979.

RUBENS, G.T.
A Study of the Use of V.I.S. for Decision Making in a Complex Production System, MSc Thesis, S.I.B.S. Warwick University, 1979.

SAMUEL, A.L.
"Some Studies in Machine Learning Using the Game of Checkers", in Feigenbaum, E.A., Feldman, J. (eds) Computers and Thought, McGraw Hill, 1963.

SARGENT, M, SHOEMAKER, R.L.
The IBM PC From the Inside Out, Addison Wesley, 1984.

SCHANK, R.C., RIESBECK, C.K.
Inside Computer Understanding, Lawrence Erlbaum Associates, 1981.

SECKER, R.J.R.
That V.I.S Offers a Viable Technique for Examining Production Planning and Scheduling Problems, MSc Thesis, S.I.B.S., Warwick University, 1977.

SHANNON, R.E., MAYER, R., ADELSBERGER, H.H.
"Expert Systems and Simulation", Simulation, 44, pp 275-284, 1985.

SHORTLIFFE, E.H.
Computer Based Medical Consultations : MYCIN, Elsevier Publishing Co., 1976.

SIMON, H.A.
The New Science of Management Decision, Harper and Row, 1960.

SIMON, H.A.
"A Behavioral Model of Rational Choice", 1975.

SOROKA, B.I.
What Can't Robot Languages Do ?, University of Southern California, 1979.

SPINELLI CARVALHO, R., CROOKES, J.G.
"Cellular Simulation", O.R.Q., 27, pp31-40, 1976.

STEFIK, M., AIKINS, J., BALZER, R., BENOIT, J., BIRNBAUM, L., HAYES-ROTH, F., SACERDOTI, E.
"The Organisation of Expert Systems, a Tutorial", Artificial Intelligence, 18, pp 135-173, 1982.

SUSSMAN, G.J.
A Computer Model of Skill Aquisition, Elsevier Computer Science Library, 1975.

de SWAAN ARONS, H.
"Expert Systems in the Simulation Domain", Maths and Computers in Simulation, 25, pp 10-16, 1983.

SWARTOUT, W.R.
"XPLAIN: A System for Creating and Explaining Expert Systems", Artificial Intelligence, 21, pp 285-325, 1983.

THORNGATE, W.
"Efficient Decision Heuristics", Behavioral Science, 25, pp 219-225, 1980.

TOCHER, K.D.
"Review of Simulation Languages", O.R.Q., 16, pp 189-217, YEAR ??

TRAVIS POPE, S.
"Unix and A.I.", Systems International, pp 64-66, April 1985.

TSO, M.
"Network Flow Models in Image Processing", J.O.R.S., 37, pp 31-35, 1986.

WATERMAN, D.A.
"User Orientated Systems for Capturing Expertise : A Rule Based Approach", in Michie, D. (ed) Expert systems in the Micro-Electronic Age, EUP, 1979.

WATERMAN, D.A.
"Rule Based Expert Systems", in Bond, A. (ed) Machine Intelligence - State of the Art Report, Pergamon Infotech, 1981.

WATERMAN, D.A., HAYES-ROTH, F.
"An Overview of Pattern Directed Inference Systems", in Pattern-Directed Inference Systems, pp 3-22, Academic Press, 1978.

WEISS, S., KULIKOWSKI, C., AMAREL, S., SAFIR, A.
"A Model Based Method for Computer Aided Medical Decision Making", Artificial Intelligence, 11, pp 145-172, 1978.

WEISS, S., KULIKOWSKI, C., SAFIR, A.
"Claucome Consultation by Computers", Computers in Biology and Medicine, 8 pp 25-40, 1978.

WHALEN, T., SCHOTT, B.
"Issues in Fuzzy Production Systems", Int. J. of Man-Machine Studies, 19, pp 57-72, 1983.

WILSON, V.
Strategies for Problem Solving, Brandon/Systems Press, 1970.

WINOGRAD, T.
"Extended Inference Modes in Reasoning by Computer Systems", Artificial Intelligence, 13, pp 5-26, 1980.

WINSTON, P.H.
Artificial Intelligence, Addison Wesley, 1977.

WINSTON, P.H.
"Learning New Principles From Precedents and Exercises", Artificial Intelligence, 19, pp 321-350, 1982.

WITHERS, S.J.
Towards the On-line Development of Visual Interactive Simulation Models, PhD Thesis, S.I.B.S. Warwick University, 1981.

YAGER, R.R.
"An Approach to Inference in Aproximate Reasoning", International Journal of Man-Machine Studies, 13, pp 323-338, 1980.

YOUNG, R.M.
"Production Systems for Modeling Human Cognition", in Michie, D. (ed) Expert Systems in the Micro-Electronic Age, EUP, 1979.

YOVITIS, C.M.
Advances in Computers, vol. 22, Academic Press, 1983.

ZANAKIS, S.H., EVANS, J.R.
"Heuristic Optimisation; Why, When, and How to use it", Interfaces, 11, pp 84-91, Oct. 1981.

APPENDIX 1 - PROGRAMMING IN PROLOG

Introduction

This Appendix gives a brief outline to the concepts behind programming in PROLOG. This outline draws largely from that given in Adelsberger (1984), and assumes the PROLOG dialect found in Clocksin and Mellish (1981).

Computer programming in PROLOG consists of:

1. declaring some facts about objects and their relationships
2. defining rules about objects and their relationships
3. asking questions about objects and their relationships

Facts

For example to say "Mozart composed Don Giovanni" states that a relationship (composed) links two objects. This could be written in PROLOG in standard form:

```
composed(mozart, don-giovanni).
```

The name of the relationship is given first, and the objects are separated by commas and are enclosed by parenthesis.

A collection of facts (and later rules) is called a database. Eg.

```
composed(mozart, don-giovanni).
```

```
composed(verdi, rigoletto).

composed(verdi, macbeth).

composed(rossini, guglielmo-tell).
```

Questions and Variables

It is possible to ask questions in PROLOG. Two different types of question can be asked: is-questions (eg "did Mozart compose don-giovanni ?") and which-questions (eg "who composed don-giovanni?"). In PROLOG one would write:

```
?-composed(mozart, don-giovanni).

?-composed(X, don-giovanni).
```

For is-questions the answer is 'yes' or 'no': in the above example the answer would be 'yes'. For which-questions one has to specify one or more variables. 'X' is the variable in the above example, and the result would be

```
X = mozart
```

If there are more solutions to a question as in "which operas where composed by verdi?"

```
?-composed(verdi,X).
```

all solutions are listed:

```
X = rigoletto

X = macbeth
```

When PROLOG is asked a question containing a variable, it searches through all its facts to find an object that the variable could stand for.

Syntax

PROLOG programs consist of terms. A term is a constant, a variable or a compound term (structure). Constants are numbers or atoms. Names of atoms begin with a lower case letter. Variables are always capitalised. A structure is written by specifying its functor ('composed' in the above example), followed by its components (also called arguments) enclosed in parenthesis, separated by commas. Lists are a special form of compound term.

Conjunctions

Given the following database:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

In PROLOG one could ask "Is there anything that John and Mary both like?" in the following form:

```
?-likes(mary, X), likes(john, X).
```

The comma is pronounced 'and', and expresses the fact that one is interested in the conjunction of these two goals.

Rules

A rule is a general statement about objects and their relationships. Rules are used to say that a fact depends on a group of other facts. For example, to say that a person is someone's sister one would say:

```
'X is a sister of Y if
        X is female and
        X and Y have the same parents'
```

In PROLOG SYNTAX one would write:

```
sister-of(X,Y) :-
        female(X),
        parents(X, Z1, Z2),
        parents(Y, Z1, Z2).
```

The symbol ':-' is pronounced 'if'.

Lists

A list is an ordered sequence of elements that can have any length. Lists are written in PROLOG using square brackets, elements are separated by commas as in:

```
languages([gpss, simscipt, simula, slam]).
```

Some other lists:

```
[]
[[the,[boy]],[kicked,[the,[ball]]]]
```

The first list is the empty list, the second one represents the grammatical structure of a simple sentence.

The vertical bar is used to split a list into its head and tail:

```
?-languages([X|Y]).
  X = gpss
  Y = [simscript, simula, slam]
```

Recursion

Recursion is a powerful technique to express complex algorithms and structures in an easy way. In many cases algorithms can be expressed in two different forms: one using recursion and one using loops. A simple example is the

computation of a factorial function. In PROLOG, recursion is the normal and natural way.

The membership test for an element of a list is a simple demonstration of recursion in PROLOG:

```
member(X, [X|Y]).
member(X, [H|Y]) :- member(X, Y).
```

This can be read as:

'The element given as the first argument is a member of the list given as the second argument, if the list starts with the element (the fact in the first line) or if the element is a member of the tail (the rule in the second line).'

Possible question:

```
?-member(d,[a,b,c,d]).
  yes
?-member(e,[a,b,c,d]).
  no
```

It is possible to get all members of a list by asking:

```
?-member(X, [a,b,c,d]).
X = a
X = b
```

X = c

X = d

APPENDIX 2 - OUTLINE TO BAYESIAN DECISION THEORY

Introduction

As indicated in the literature review chapter, a major problem in expert system development is the treatment of uncertainty. One expert system which has tried to tackle this problem is PROSPECTOR. This tries to assess the degree to which a change in probability of an evidence assertion changes the probability of the hypothesis. This is achieved via Bayesian Decision Theory. This is not directly relevant to this thesis, and a brief outline is given here for completeness only.

Definitions

$prob(A|B) = P(A|B) = P(A \cap B)/P(B)$

odds of A = odds(A) = O(A); P(A) = p -> O(A) = p/(1-p)

Bayes' rule:

$$P(B|A) = (P(B)P(A|B))/(P(B)P(A|B))$$

E = 'evidence assertion'

H = 'hypothesis assertion'

We also need measures of the sufficiency for certain evidence to imply a given hypothesis, and of the necessity of an evidence for a hypothesis. We call these measures LS and LN respectively.

LS

This is just the likelihood ratio

$$LS = P(E|H)/P(E|\overline{H})$$ as LS ->∞ H is more likely

as LS -> 0 H is less likely

in other words a small LS implies a small sufficiency

LN

$$LN = P(\overline{E}|H)/P(\overline{E}|\overline{H})$$ as LN ->∞ lack of E ($\overline{E}$) implies H

as LN -> 0 lack of E ($\overline{E}$) implies $\overline{H}$

in other words a small LN implies a high necessity

Odds Likelihood

LS and LN are then used in the "odds-likelihood" forms of Bayes' formulae:

$$O(H|E) = LS * O(H)$$

$$O(H|\overline{E}) = LN * O(H)$$

hence for the production rule:

IF E

THEN (to degree LS,LN) H

the model designer must articulate E and H and supply the numerical values for LS, LN and O(H).

Generally E is not certainly present or absent. Here the expert system user provides a value in the range (-5,+5) which is used by PROSPECTOR by linearly interpolating between the extreme values of E (value=+5) and $\overline{E}$ (value=-5).

APPENDIX 3 - OUTLINE GUIDE TO THE SIMULATION ENGINE

Introduction

This contains information on

i) problem definition

ii) example problem

iii) running the package

iv) using the interactive facility

v) reserved words

vi) listings

Problem Definition

The following should be placed in a PROLOG text file.

1. schedule end of simulation and other starting events:

   event(<event name(attributes)>, time).

eg. event(end,300).

   event(service(c,b,g), 1.23).

2. initialise clock time:

   clock(time).

eg. clock(0).

3. set recording constant (time after which results should be recorded):

   recording(<time>).

eg. recording(10).

4. record how many occurrences of each event are happening at the start of the simulation:

   curr_realis(<event name and attributes>, <number>).

eg. curr_realis(service(c,b,g),0).

5. state initial queue sizes:

qusize(<queue name and attributes>, <number>).

eg. qusize(cpool(c),24).

6. list activities and attributes in decreasing order of priority:

activity_list(<list of activities(attributes)>).

7. list queues (in any order):

queue_list(<list of queue names>).

8. list queues in order of required output (not all queues need to be listed):

queue_lisp(<list of queue names>).

9. state the number of simultaneous occurrences of each activity allowed:

simoult_realis(<activity and attributes>, <number>).

eg. simoult_realis(service(c,b,g),2).

10. list the entities in the system (in any order):

entity_list(<list of entity names>).

eg. entity_list([c,b,g]).

11. where an entity has any attributes associated with it, the attributes names must be associated with the entities:

attribute_name(<entity_name>,<attribute name list>).

eg. attribute_name(c,[drinks, darts]).

12. the initial numbers for each of the entities must be stated:

number(<entity name>, <number>).

eg. number(c,25).

13. definitions of attribute calculations:

attribute_calculation(<attribute name>,<activity name>,
<function name>).

The named attribute is calculated during the named activity according to the procedure <function name>.

```
eg, attribute_calculation(drinks,drink,drac).
    drac:- current(Value),    ;returns Value as the
                              value of the attribute
    V is Value-1,
    assert(attribute(V)).     ;all attribute calculating
                              functions must finish by
                              asserting the calculated
                              value.
```

current(V) is a function built into the simulation engine which is used to facilitate the deletion of an attribute value, which may then be optionally replaced by another one. Another built in function is random(N) which returns a random real number in the range (0,100):

```
eg. ardac:-random(N),N>90,assert(attribute(1)).
    ardac:-assert((attribute(0)).
```

"calculate the attribute such that it has a 10% chance of being 1 and a 90% chance of being 0"

14. stipulate the time taken by each activity:

time(<activity name and attributes>, V):-
clock(X), <calculate Time for activity>, V is Time+X,!.

```
eg.  time(service(c,b,g),T):-clock(X),
     random(N),
     Ti is N/50,
     T is Ti+X,!.
```

15. define the state entity cycles:

    a) look at each activity in turn and find the queues that lead into it:

eg.

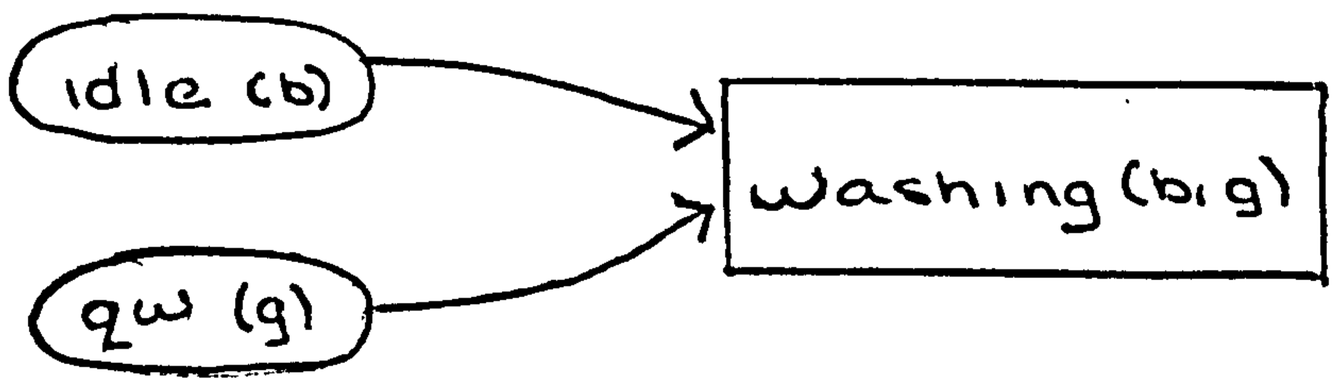


Figure A3.1 An Example 'quact' section of a state entity diagram

write this as

```
quact([idle(b),qw(g)], washing(b,g)).
```

b) look at each activity in turn and find the queues it leads into:

eg.

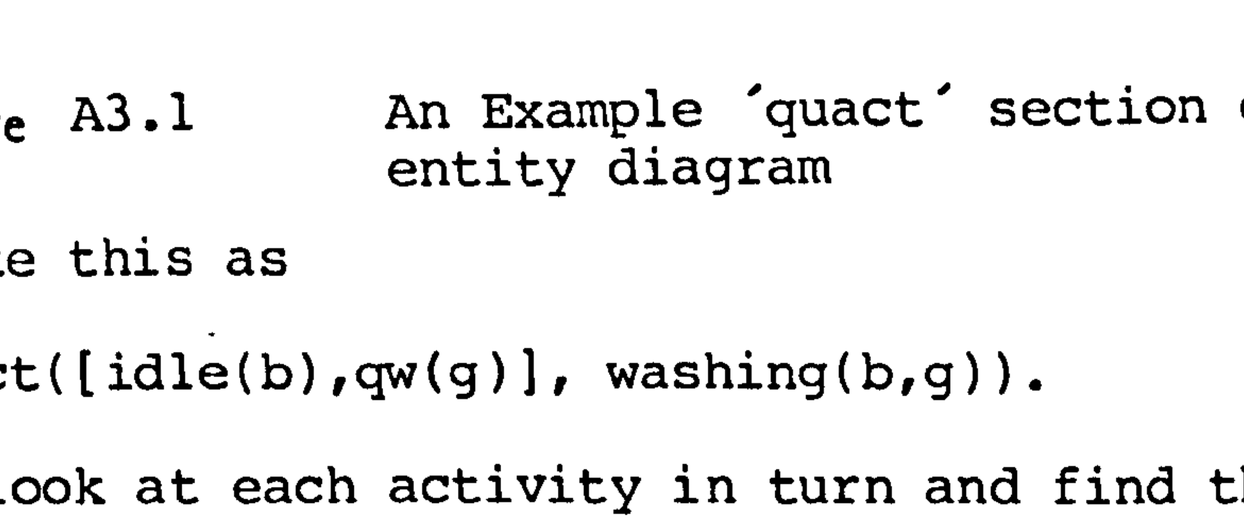


Figure A3.2 An Example 'actqu' section of a state entity diagram

actqu(washing(b,g),[idle(b), gpool(g)]).

eg.

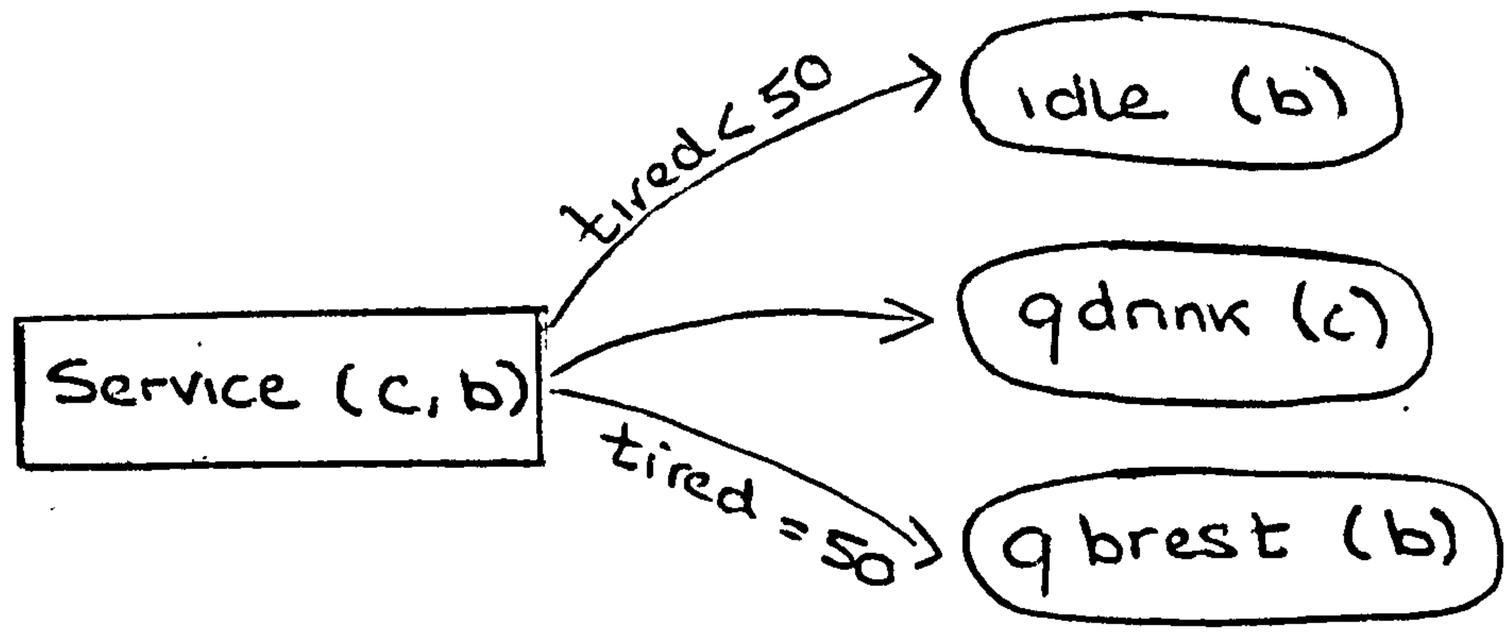


Figure A3.3 An Example Conditional 'actqu' section of a state entity diagram

```
actqu(service(c,b),[[[tired,'<',50],

qdrink(c),qidle(b)], [qdrink(c),qbrest(b)]]).
```

In the latter example the second argument to actqu consists of a list of lists. Each sublist contains a number of queues and possibly (as its first element) a condition, Conditions are of the form:

[<attribute>,<op>,<const>]. (op = '<', '=', or '>')

If attribute-op-const is true then the queues in the rest of the sublist are moved into by the entities. If it is false then the next sublist is considered. If there is no condition the queues in the sublist are moved into. Finally, if, after accepting a sublist, not all the entities have been moved, the last sublist is considered and any entities not yet moved are transferred into their queues in this last sublist. For

example the following statement is identical to the second example:

```
actqu(service(c,b),[[[tired,'<',50], idle(b)],
[brest(b), qdrink(c)]]).
```

The creation of a datafile is now illustrated with a sample problem.

A Sample Problem

A small bar operates in a city centre and wishes to find out whether it employs enough bartenders and stores enough glasses.

There are presently two bartenders, each arrives for work promptly (within a few seconds of each other). Their duties involve serving customers and washing - serving having the highest priority. After a bartender has served 50 customers he may take a rest.

The customers arrive and then queue for service. After being served they drink up and then may play darts. After a rest a customer may either leave the bar or queue up for another drink. All glasses are washed immediately after drinking.

It has been found that 10% of customers play darts and that a customer may order anything from 1 to 10 drinks at the bar. Other facts are:

number of glasses: 50

service time: random in range [0, 2]

washing time:	random in range [0, 0.5]
drink time:	random in range [5, 30]
customer inter-arrival time:	random in range [0, 1]
darts playing time:	random in range [20, 60]
resting time (customer):	random in range [10, 20]
resting time (bartender):	5 minutes
opening times of bar:	7pm - Midnight

This problem is designed to illustrate the scope of the simulation engine:

+ multiple attributes
+ integer and real attributes
+ incrementing and decrementing attribute values
+ conditional branching on event completion
+ varying event duration times
+ continuous time model
+ no events initially scheduled (other problems may have events scheduled to start the simulation off).

entity	line	attributes
customer (r)	— · —	drinks, darts
bartender (b)	— —	tired
glasses (g)	—	

= queue

= activity

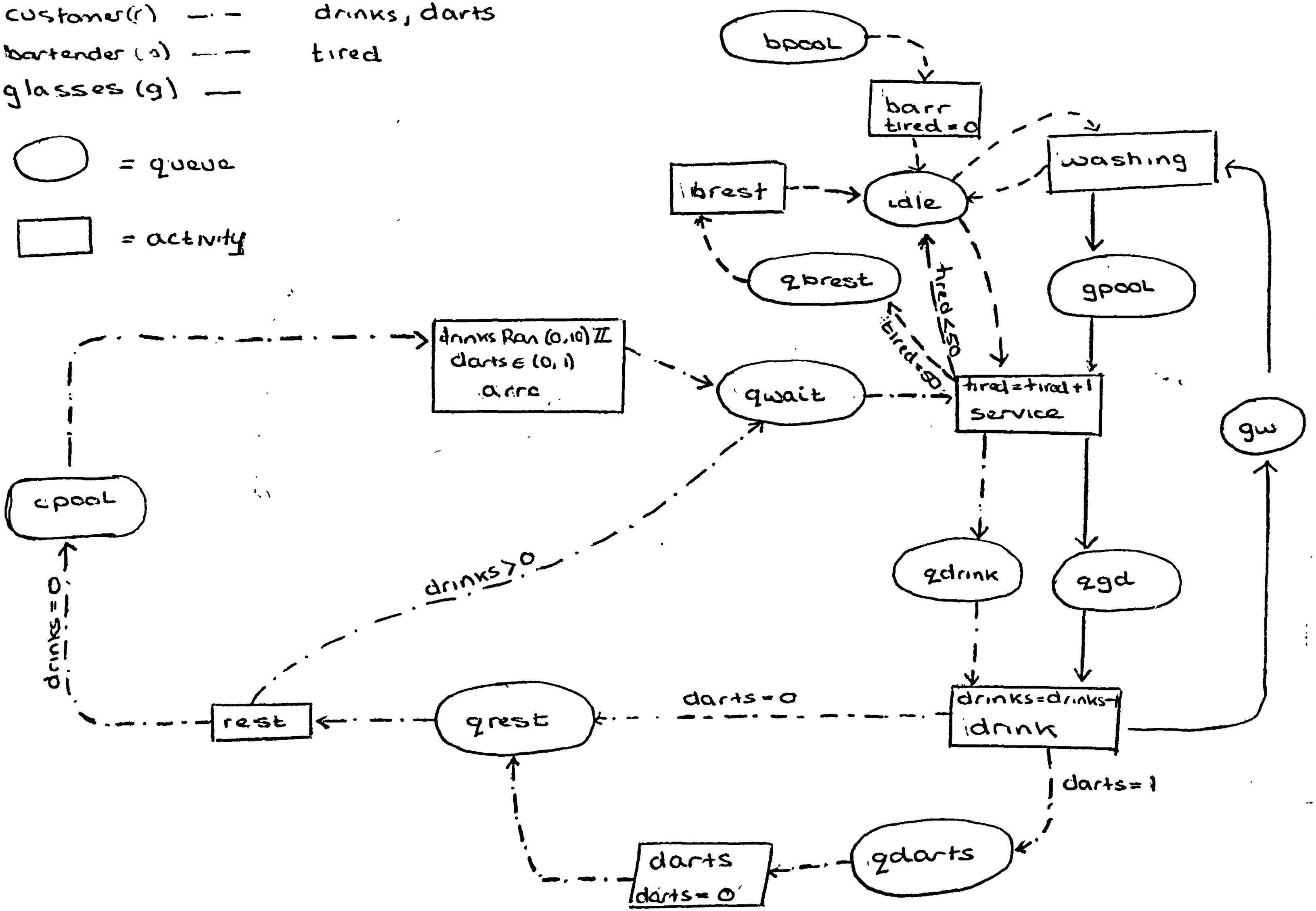


Figure A3.4 Entity Cycle Diagram for the Bar Problem

Problem Specific Data For the Bar Example

```
event(end,300).
clock(0).
recording(0).
curr_realis(arrc(c),0).
curr_realis(service(c,b,g),0).
curr_realis(washing(b,g),0).
curr_realis(drink(c,g),0).
curr_realis(darts(c),0).
curr_realis(rest(c),0).
curr_realis(barr(b),0).
curr_realis(brest(b),0).

qusize(cpool(c),24).
qusize(qwait(c),0).
qusize(qdrink(c),0).
qusize(qdarts(c),0).
qusize(qrest(c),0).
qusize(bpool(b),2).
qusize(qbrest(b),0).
qusize(idle(b),0).
qusize(gpool(g),50).
qusize(qgd(g),0).
qusize(qw(g),0).

activity_list([service(c,b,g),washing(b,g),drink(c,g),arrc(c
),
darts(c),rest(c),barr(b),brest(b)]).

queue_list([cpool,qwait,qdrink,qdarts,qrest,bpool,qbrest,idl
e,qgd,
qw,gpool]).

queue_lisp([cpool,gpool,qw,idle,qwait,qdarts]).

simoult_realis(service(c,b,g),2).
simoult_realis(washing(b,g),2).
simoult_realis(drink(c,g),50).
simoult_realis(darts(c),4).
simoult_realis(rest(c),50).
simoult_realis(brest(b),2).
simoult_realis(arrc(c),1).
simoult_realis(barr(b),1).

entity_list([c,b,g]).

attribute_name(c,[drinks,darts]).
attribute_name(b,[tired]).

number(c,25).
number(b,2).
number(g,50).

attribute_calculation(drinks,arrc,ardrc).
```

```
attribute_calculation(drinks,drink,drdrc).
attribute_calculation(darts,arrc,ardac).
attribute_calculation(darts,darts,dadac).
attribute_calculation(tired,service,setic).
attribute_calculation(tired,barr,batic).
attribute_calculation(tired,brest,brtic).

pools([bpool,gpool,cpool]).

ardrc:-random(N),NN is N/10,N1 is fix(NN),M is N1+1,
       assert(attribute(M)).
drdrc:-current(Value),V is Value-1,assert(attribute(V)).
ardac:-random(N),N>90,assert(attribute(1)).
ardac:-assert(attribute(0)).
dadac:-assert(attribute(0)).
batic:-assert(attribute(0)).
setic:-      current(V),Va is V+1,assert(attribute(Va)).
brtic:-current(V),assert(attribute(0)).

time(service(c,b,g),T):-clock(X),random(N),Ti is N/50,T is
Ti+X,!.
time(washing(b,g),Tim):-clock(X),random(N),Ti is N/200,Tim
is Ti+X,!.
time(drink(c,g),T):-clock(X),random(N),Ti is N/4,T is
X+5+Ti,!.
time(arrc(c),Time):-clock(X),random(N),Ti is N/99,Time is
Ti+1+X,!.
time(darts(c),Tim):-clock(X),random(N),Ti is N/2.5,Tim is
Ti+20+X,!.
time(rest(c),Time):-clock(X),random(N),Ti is N/10,Time is
Ti+10+X,!.
time(barr(b),Time):-clock(X),Time is X+0.05,!.
time(brest(b),Tim):-clock(X),Tim  is X+5,!.

quact([cpool(c)],arrc(c)).
quact([qwait(c),idle(b),gpool(g)],service(c,b,g)).
quact([idle(b),qw(g)],washing(b,g)).
quact([qdrink(c),qgd(g)],drink(c,g)).
quact([qdarts(c)],darts(c)).
quact([qrest(c)],rest(c)).
quact([bpool(b)],barr(b)).
quact([qbrest(b)],brest(b)).

actqu(service(c,b,g),[[[tired,'<',50],idle(b)],[qdrink(c),qb
rest(b),
qgd(g)]]).
actqu(washing(b,g),[[idle(b),gpool(g)]]).
actqu(drink(c,g),[[[darts,'=',0],qrest(c)],[qdarts(c),qw(g)]
]).
actqu(arrc(c),[[qwait(c)]]).
actqu(darts(c),[[qrest(c)]]).
actqu(rest(c),[[[drinks,'>',0],qwait(c)],[cpool(c)]]).
actqu(barr(b),[[idle(b)]]).
actqu(brest(b),[[idle(b)]]).
```

Running the Simulation Engine

To run the simulation engine, the following system files are needed:

1. problem file (see above).
2. simulat6.pro (main simulation engine)
3. interal.pro (interaction facility)
4. dummy.pro (empty file with EOF marker)
5. wreal.pro (real number formatter)
6. resout.pro (output of results)

The simulation engine can be run with or without the interaction facility:

a) Running a Simulation Without Interaction.

1. enter the PROLOG system and consult the main simulation engine (simulat6.pro).
2. In response to the prompt:

name of problem file ?

type the name of the file containing the problem specific information (without the .pro extension) followed by a full stop and a carriage return.

3. The simulation can then be run by typing:

a) simulation. (return) if the problem file contains scheduled initial events, or

b) sim. (return) if no events are initially scheduled.

4. After each completed C-phase, the lengths of the required queues will be printed on the screen (see example run below). After the simulation has run its full course the message:

End of Simulation

is displayed. The system may then be exited by typing:

halt. (return)

b) Using the Interaction Facility.

It is possible to change the characteristics of the problem at any time after the simulation engine has been consulted:

i) immediately after the engine has been consulted, typing 'change.' will enter the interaction facility.

ii) whilst the simulation is running, the interaction facility can be entered as follows:

- type control Y
- in action to the prompt type b (return)
- type 'change.' (return).

After quitting the interaction facility, typing control-Z (return) will cause the simulation to continue, with the modifications.

Below is a list of interaction facilities available, together with any restrictions.

A. Entity Cycle Modification. The following are available:

1. help (h.): lists possible options.
2. print actqu (aq.): This gives a list of all the 'actqu' facts, associating a number with each one. The number is used to identify an individual 'actqu' fact for the delete and change options (see below). When an 'actqu' fact is deleted, the numbers will be changed.

3. print quact (qa.): This gives a numbered list of 'quact' facts (as for 'aq.' above).
4. replace an actqu fact (ca.): This allows the user to replace a single (numbered) 'actqu' fact for another. The new 'actqu' fact will not have the same number as the one it replaced.
5. replace a quact fact (cq.): As for 'ca.' above.
6. delete a quact fact (rq.): This allows a numbered 'quact' fact to be deleted without replacement.
7. delete an actqu fact (ra.): As for 'rq.' above.
8. add to entity cycle diagram (ad.): This allows the user to add any number of actqu facts to the database. The set of new facts should be terminated with control-Z.
9. return to 'normal' interaction mode (re.): As well as signalling the end of state entity cycle changes this command causes the system to ask the user some questions about the database change (for internal housekeeping).

B. Other Interactions. The following are available.

1. help (h.): lists options available.
2. change recording constant (r.): This allows the user to change the time after which result output will commence.
3. change an entities population (e.): This modifies a specified entities population. The required change may not occur immediately. If a large reduction in population is requested it may be necessary for the

system to wait for some of the entities to return to the world pool. All reductions and additions are made via the world pool and this should be noted. If entities never return to this pool, a requested reduction in population may never be fully implemented.

4. change a queue size (q.): This allows a specified queue size to be changed. This command assumes that all entities in this queue have no current attribute values.
5. enter the modify entity cycle mode (ec.).
6. change end of simulation time (f.).
7. change the number of allowed simultaneous realisations for an activity.
8. return to simulation (re.): This allows return to the PROLOG interpreter. If the interaction facility was called during a simulation run, the run is continued by typing control-Z (return).

Below is a sample run of the simulation using the 'bar' problem above, this is followed by a complete listing of the simulation engine.

Example Run of The Simulation Engine

```
A>b:prolog86

+---------------------------------------- --+
| MS-DOS Prolog-1                           |
| Copyright 1983       Serial number: 000735|
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+


?- [simulat6].

SIMULATION PACKAGE - VERSION PROANDY 6
---------- ------- - ------- ------- -

name of problem file : barl.
simulat6 consulted.
?- sim.
TIME      ACTIVITY  cpool  gpool  qw  idle  qwait  qdart
----      --------  -----  -----  --  ----  -----  -----
0.055     barr      23     50     0   1     0      0
.1        barr      23     50     0   2     0      0
1.26      arrc      22     49     0   1     0      0
2.34      arrc      21     48     0   0     0      0
2.97      service   21     48     0   1     0      0
3.09      service   21     48     0   2     0      0
3.74      arrc      20     47     0   1     0      0
3.82      service   20     47     0   2     0      0
4.85      arrc      19     46     0   1     0      0
6.37      service   19     46     0   2     0      0
6.78      arrc      18     45     0   1     0      0
7.86      arrc      17     44     0   0     0      0
8.49      service   17     44     0   1     0      0
8.61      service   17     44     0   2     0      0
9.26      arrc      16     43     0   1     0      0
9.33      service   16     43     0   2     0      0
10.37     arrc      15     42     0   1     0      0
11.19     drink     15     42     0   0     0      0
11.36     washing   15     43     0   1     0      0
11.89     service   15     43     0   2     0      0
TIME      ACTIVITY  cpool  gpool  qw  idle  wait   qdart
12.29     arrc      14     42     0   1     0      0
13.45     service   14     42     0   2     0      0
14.04     arrc      13     41     0   1     0      0
15.31     service   13     41     0   2     0      0
15.41     arrc      12     40     0   1     0      0
Interrupt option (h for help): b
Entering Break
```

```
1?- change.
type your change command (h. for help):h.

possible commands are :
-------- -------- --- -

h. :prints this message
r. :change recording constant
e. :change an entities population
ec.:enter change entity cycle mode
q. :change a queue size
f. :change end of simulation time
s. :change no. of concurrent realisations
re.:return to simulation

type your change command (h. for help):q.

queue ? :gpool.
entity ? : g.
size ? (natural number) :0.
type your change command (h. for help):re.

return to simulation by ending break
TIME      ACTIVITY  cpool  gpool  qw    idle   qwait   qdart
----      --------  -----  -----  --    ----   -----   -----

yes
1?- ^Z

Leaving Break
16.71     drink     12     0      0     0      0       0
16.93     arrc      11     0      0     0      1       0
17.13     service   11     0      0     1      1       0
17.16     washing   11     0      0     1      0       0

Interrupt Option (h for help):b
Entering Break


1?-change.
type your change command (h. for help):f.

time ? :18.
type your change command (h. for help):re.

return to simulation by ending break
TIME      ACTIVITY  cpool  gpool  qw    idle   qwait   qdart
----      --------  -----  -----  --    ----   -----   -----

yes
1?- ^Z

Leaving Break
17.31     drink     11     0      0     0      0       0
17.68     washing   11     1      0     1      0       0

yes
?-halt.
```

Listing of the Main Simulation Engine

```
/* b_phase satisfies the B-phase of the simulation, and
   consists of finding what and when the next event to occur
will be
   (search_event), and moving on the relevant entity
(moveon)        */

search_event(Y,Z):-clock(X),event(Y,X ),s_event(X,Y,Z).
search_event(Y,Z):-clock(X),XX is
X,search2(X,Y,Z),s2_event(XX,Z).

s_event(X,Y,Z) :-Y\==end,
                 curr_realis(Y,CNo),CCNo is CNo-1,
                 retract(curr_realis(Y,CNo)),
                 asserta(curr_realis(Y,CCNo)),
                 retract(event(Y,X)),!.
s_event(X,Y,Z) :-Z is 1.

s2_event(XX,Z):- Z==1.
s2_event(XX,_):- retract(clock(XX)),!.

search2(I,Y,Z) :-
                 assert(min(6000)),
                 event(X,Y),
                 mino(P),
                 Y>I,Y<P,
                 remin(P,Y),fail.
search2(X,Y,Z) :-

retract(min(X1)),event(Y,X1),search3(X1,Y,Z),!.

mino(P) :- min(P),!.

remin(P,Y) :- retract(min(P)),assert(min(Y)),!.

/*
search2(X,Y,Z):- XX is X+1,search2(XX,Y,Z),!.
*/


search3(X,Y,Z):- Y\==end,
                 curr_realis(Y,CNo),CCNo is CNo-1,
                 retract(curr_realis(Y,CNo)),
                 asserta(curr_realis(Y,CCNo)),
                 retract(event(Y,X)),assert(clock(X)),!.
search3(_,_,Z) :-Z is 1.

/* moveon(Y) : move on the entity that has just completed
event Y.
                This involves moving & testing attributes as
well */

moveon(Y):-actqu(Y,X1),
           Y=..[_|Y1],assert(entities(Y1)),
```

```
            queue_select(Y,X1,X2),mem(X,X2),
            attrib_mod(Y,X),qu_siz_inc(X),fail.
moveon(Y):-actqu(Y,X),select_last(X,X1),
            mem(MX,X1),
            check_and_move(MX,Y),fail.
moveon(Y):-retractall(entities(_)).

/* note that in entities(X), X is a list of those entities
   that are due to still be moved on after the last event */

check_and_move(MX,_):-
            entities(E),
            MX=..[_,M],
            not(mem(M,E)),!.
check_and_move(MX,Y):-
            attrib_mod(Y,MX),
            qu_siz_inc(MX),!.

/* qu_siz_inc(X)
   1st sentence : checks whether an earlier interaction
requires
                  entity being moved to be removed from the
system
   2nd sentence : normal incrementation of qusize fact */

qu_siz_inc(X):-
            retract(remove(X,P)),modrem(X,P),
            X=..[_,X1],retract(entities(X2)),
            difference([X1],X2,X3),
            assert(entities(X3)),
            !.
qu_siz_inc(X):-
            qusize(X,XX),retract(qusize(X,XX)),XY is XX+1,
            assert(qusize(X,XY)),
            X=..[_,X1],retract(entities(X2)),
            difference([X1],X2,X3),
            assert(entities(X3)),
            !.

modrem(_,1).
modrem(X,R) :-
            R1 is R-1,
            assert(remove(X,R1)).

/* select_last(QL,X) ; X is the last queue_list from the
list
                        of queue lists QL */

select_last([X|[]],X) :- !.
select_last([_|L],X) :- select_last(L,X).

/* difference(X,Y,Z) : Z is Y-X (X,Y,Z are lists) */

difference(_,[],[]).
```

```
difference(X,[Y|L],Z) :- mem(Y,X),difference(X,L,Z).
difference(X,[Y|L],[Y|Z]) :-
          not(mem(Y,X)),difference(X,L,Z).

/* attrib_mod(Y,X) ; sets up the logic to move an attribute
from
                  act/qu Y to q/act X */

attrib_mod(Y2,X1) :-
          Y2=..[Y|_],
          X1=..[X|_],
          attribute_list(X,XN,XL),
          not(at_mod(XN)),
          modatt(Y,XN,Y1),
          assert(at_mod(XN)),
          add_to_attribute_list(Y1,X,XN,_),fail.
attrib_mod(_,_) :- retractall(at_mod(_)),!.

b_phase(Y,Z):-search_event(Y,Z),bb_ph(Y,Z).
bb_ph(_,Z):-Z==1,fin.
bb_ph(Y,_):- moveon(Y),!.

/* c_phase satisfies the C-phase of the simulation, and
consists of
   checking all activities to see whether they can be
started
   The predicate time(activity,Time) is true if 'activity'
is
   completed at time Time      */

activity_start(XX):-
        activity_list(L),mem(XX,L),act_start(XX).
act_start(XX):-
        poss_start(XX),

quact(QL,XX),all_q_pos(QL),q_siz_dec(QL),time(XX,Time),
        assert(t(Time)),attrib_calc(XX),

assert(event(XX,Time)),a_start(XX),retract(t(Time)),!.

a_start(XX):-
        curr_realis(XX,CNo),CCno is CNo+1,
        retract(curr_realis(XX,CNo)),
        asserta(curr_realis(XX,CCno)).

poss_start(XX):-
        simoult_realis(XX,SNo),curr_realis(XX,CNo),
        Poss is SNo-CNo,Poss>0.

all_q_pos([]):-!.
all_q_pos([QX|L]):-qusize(QX,Size),Size>0,all_q_pos(L).

q_siz_dec([]):-!.
q_siz_dec([QX|L]):-
```

```
        qusize(QX,Size),Size2 is Size-1,
        retract(qusize(QX,Size)),
        assert(qusize(QX,Size2)),
        q_siz_dec(L).

/* attrib_calc(X1) : this tests whether activity X1
neccessitates
                    the calculation of an attribute - which
is
                    then calculated if necessary */

attrib_calc(X1) :-
    X1=..[X|_],
    attribute_calculation(Nm,X,Z),
    assert(att(Nm)),assert(act(X1)),
    call(Z),retract(attribute(A)),
    add_to_attribute_list(A,X,Nm,X1),assert(mved(Nm)),
    retractall(att(_)),retractall(act(_)),fail.
attrib_calc(X1) :-
    X1=..[X|_],
    quact(QL,X1),mem(Q,QL),Q=..[Q1|Q2],
    mem(Q2M,Q2),attribute_name(Q2M,Q2L),
    mem(EN,Q2L),
     ck(Q1,EN),
    modatt(Q1,EN,L1),
    add_to_attribute_list(L1,X,EN,X1),fail.
attrib_calc(_) :- retractall(mved(_)),!.

ck(Q1,E) :-
          attribute_list(Q1,E,_),
          mved(E),!,fail.
ck(_,_).

/* current(Value) returns Value as the attribute being
   currently processed during the current attribute
calculation */

current(Value) :-
              retract(att(ATT)),
              retract(act(ACT)),
              curr_value(ACT,ATT,Value),!.

curr_value(ACT,ATT,Attrib) :-
              quact(QL,ACT),
              mem(Q1,QL),
              Q1=..[Q,_],
              modatt(Q,ATT,Attrib).

/* find_entity_name(Y,Nm) finds the name (Nm) of the entity
   which has an attribute called Y */

find_entity_name(Y,X) :-
     attribute_name(X,XX),
     mem(Y,XX),!.
```

```
/* add_to_attribute_list(A,X,Nm) adds the attribute A to the
   tail of the list L defined in the database as
   attribute_list(X,Nm,L) where X is a non world pool queue.
   if X is an activity name, we must insert into correct
place */

add_to_attribute_list(_,X,_,_):-
    pools(P),mem(X,P),!.
add_to_attribute_list(A,X,NM,_):-
    queue_list(Q),mem(X,Q),retract(attribute_list(X,NM,L)),
    a_t_at_list(A,L,L1),
    assert(attribute_list(X,NM,L1)),!.
add_to_attribute_list(A,X,NM,X1):-
    assert(pos(1)),
    place(X1,N),
    insert(A,X,NM,N),!.

a_t_at_list(A,[],[A]).
a_t_at_list(A,[B|BL],[B|L]) :-
    a_t_at_list(A,BL,L).

/* conc(X,Y,Z) ; X concatenate Y is Z (X,Y,Z are lists) */

conc([],L,L):-!.
conc([X|L1],L2,[X|L3]):-
    conc(L1,L2,L3).

place(X1,_) :-
    t(T),event(X1,T1),T>T1,retrP(P),fail.
place(X1,N) :-
    retract(pos(N)).

retrP(P) :- retract(pos(P)),P1 is P+1,assert(pos(P1)),!.

no_elements([],0).
no_elements([_|L],N) :- no_elements(L,N1),N is N1+1,!.

/* insert(A,X,NM,N) : insert attribute NM (value = A)
correctly
                      placed (position N) for activity X.
                      The correct place being dependant on
the
                      time to event completion */

insert(A,X,NM,N) :-
    retract(attribute_list(X,NM,L)),
    split(L,N,LF,LT),
    a_t_at_list(A,LF,L1),conc(L1,LT,L2),
    assert(attribute_list(X,NM,L2)).

split(L,N,LF,LT):-
    no_elements(L,LS),FS is LS-N+1,
    first_n_elems(L,FS,LF,LT).
```

```
first_n_elems(L,0,[],L).
first_n_elems([X|L],1,[X],L).
first_n_elems([Y|L],N,[Y|L1],L2) :-
    N1 is N-1,
    first_n_elems(L,N1,L1,L2),!.

modatt(Q1,EN,L) :-
    retract(attribute_list(Q1,EN,[L|L1])),
    assert(attribute_list(Q1,EN,L1)),!.

c_phase:-
        activity_start(XX),fail.
c_phase:-!.

/* as a simplification, time(XX,Y) a random real
no.([0,100]) + clock
   time as Y     */

random(Number):-
               ran_select,
               ranseed(N),MMM is 3125*N,
               M is MMM mod 8192,
               F is M mod 100,
               retract(ranseed(N)),
               MMM1 is 3125*F,
               M1 is MMM1 mod 8192,
               F1 is M1 mod 100,
               F2 is F1/100,
               Number is F+F2.
ranseeds([1,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,
67,71]).

ran_select :-
              retract(ranseeds([X|L])),
              assert(ranseed(X)),
              conc(L,[X],LX),
              assert(ranseeds(LX)),!.

/*
time(XX,Time):-clock(X),random(Number),Time is Number+X,!.
*/

/* update performs the B- and C-phases of the simulation */

update :- repeat,b_phase(Y,Z),updt1(Y,Z).
updt1(_,Z):-Z==1.
updt1(Y,_):-c_phase,trimcore,record(Y),!,fail.

/* initialise is true if output banners and database are set
up */

initialise:-
     d_base_initialise,
```

```
     output("TIME      ACTIVITY "),qu_ban_out,
     output("----      --------- "),under_ban_out.

/* simulation is initialise + B- and C-phase  */

simulation :- initialise,update.
sim :- initialise,c_phase,update.

/* mem(X,L) is true if X is a member of list L */

mem(X,[X|_]).
mem(X,[_|Y]):-mem(X,Y).

attrib_operators(['<','>','=']).

/* queue_select(X1,X2) ; from the list of queue lists X1,
   the queue list X2 is chosen which satisfies the attribute
conditions
   in X1.
*/

queue_select(_,[L1|L],L1) :-
     not(mem([X1,X2,X3],L1)),!.
queue_select(Y,[[[X1,X2,X3]|L]|L1],L) :-
     Y=..[Y1|_],
     attribute_list(Y1,X1,[X5|_]),
     evaltrue(X5,X2,X3),!.
queue_select(Y,[_|L],X) :-
     queue_select(Y,L,X),!.

evaltrue(X1,'=',X3) :- X1==X3.
evaltrue(X1,'<',X3) :- X1<X3.
evaltrue(X1,'>',X3) :- X1>X3.


/* d_base_initialise sets up tne initial database regarding
   movement of attributes around the system */

d_base_initialise :-
   queue_list(L),entity_list(E),mem(X,L),
   mem(Y,E),ch_valid(X,Y),
   attribute_name(Y,YM),mem(YYM,YM),
   assert(attribute_list(X,YYM,[])),fail.
d_base_initialise :-
   activity_list(A),mem(X,A),
   X=..[L|L1],mem(M,L1),
   attribute_name(M,M1),mem(M2,M1),
   assert(attribute_list(L,M2,[])),fail.
d_base_initialise :- !.

/* ch_valid(X,E) is true if queue X contains entity E */

ch_valid(X,E) :-
   Y=..[X,E],
```

```
   qusize(Y,_),!.

/* record(Y) is satisfied if all the relevant information is
output
   neatly after activity Y has been completed */

record(Y):-
            recording(X),clock(XX),!,
            (XX<X;
            (newban,time_output,Y=..[P|N],
             act_output(P),queue_output,nl,!)).

/* fin outputs end of simulation message          */

fin :- output("End of simulation"),nl,!.

/* consult other files :
   wreal    :real number formatter
   resout   :neat output of results
   interal  :interaction package
  (dummy    :blank file to facilitate consulting completion)
   Also ask user for problem file name (& consult it) */

?-consult(wreal).
?-consult(resout).
?-consult(interal).
?-nl.
?-output("SIMULATION PACKAGE - VERSION PROANDY 6"),nl.
?-output("---------- ------- - ------- ------- -"),nl,nl.
?-output("name of problem file : "),seen,read(X),consult(X),
 see('dummy.pro').


→
```

Listing of The Interaction Facility

```
change :- repeat,
          output("type your change command (h. for help):"),
          read(X),process(X).

process(h) :- nl,
              output("possible commands are :"),nl,
              output("-------- -------- --- -"),nl,
              nl,output("h. :prints this message"),nl,
              output("r. :change recording constant"),nl,
              output("e. :change an entities
population"),nl,
              output("ec.:enter change entity cycle
mode"),nl,
              output("q. :change a queue size"),nl,
              output("f. :change end of simulation
time"),nl,
              output("s. :change no. of concurrent
realisations"),nl,
              output("re.:return to
simulation"),nl,nl,!,fail.
process(re):- nl,
              output("return to simulation by ending
break"),nl,
             output("TIME     ACTIVITY "),qu_ban_out,
             output("----     --------- "),under_ban_out.
process(ec) :- repeat,
               nl,
               output("entity cycle mode (h. for help) :"),
               read(X),
               ecprocess(X),!,fail.
process(r) :- nl,
              output("time ? :"),read(T),
              retract(recording(_)),
              assert(recording(T)),!,fail.
process(s) :- nl,
              output("activity ? :"),read(A),
              output("entity list ? :"),read(E),
              output("no. of allowed concurrent realisations
? :"),
              read(R),
              simoult_realis(X,Y),
              X=..[A|_],
              retract(simoult_realis(X,Y)),
              conc([A],E,AE),
              P=..AE,
              assert(simoult_realis(P,R)),
              !,fail.
process(q) :- nl,
              output("queue ? :"),read(Q),
```

```
                output("entity ? :"),read(E),
                output("size ? (natural number) :"),read(S),
                X=..[Q,E],
                retract(qusize(X,O)),
                assert(qusize(X,S)),mnum(O,S,E),!,fail.
process(e) :- nl,
                output("entity ? :"),read(E),
                output("new number ? (integer) :"),read(N),
                retract(number(E,ON)),assert(number(E,N)),
                pf(N,ON,E),!,fail.
process(f) :- nl,
                output("time ? :"),read(T),
                retract(event(end,_)),
                assert(event(end,T)),!,fail.

ecprocess(h) :- nl,
                   output("ec mode commands are :"),nl,
                   output("-- ---- -------- --- -"),nl,nl,
                   output("h. :prints this message"),nl,
                   output("aq.:prints current 'actqu'
facts"),nl,
                   output("qa.:prints current 'quact'
facts"),nl,
                   output("cq.:change a 'quact' fact"),nl,
                   output("ca.:change an 'actqu' fact"),nl,
                   output("rq.:delete a 'quact' fact"),nl,
                   output("ad.:add to entity cycle
database"),nl,
                   output("ra.:delete an 'actqu' fact"),nl,
                   output("re.:return to normal change
mode"),nl,nl,
                   !,fail.
ecprocess(ad):- nl,
                   output("type new facts, terminate with
^Z"),nl,
                   consult(user),!,fail.
ecprocess(cq):- nl,
                   output("number ? :"),read(N),
                   assert(naq(N)),
                   quact(Q,A),
                   retnaq,
                   retract(naq(0)),nl,
                   retract(quact(Q,A)),
                   write(quact(Q,A)),
                   output(" has been deleted"),nl,
                   output("type in replacement"),nl,
                   read(X),assert(X),
                   !,fail.
ecprocess(ca):- nl,
                   output("number ? :"),read(N),
                   assert(naq(N)),
                   actqu(A,Q),
                   retnaq,
                   retract(naq(0)),nl,
```

```
                retract(actqu(A,Q)),
                write(actqu(A,Q)),
                output(" has been deleted"),nl,
                output("type in replacement"),nl,
                read(X),assert(X),
                !,fail.
ecprocess(ra):- nl,
                output("number ? :"),read(N),
                assert(naq(N)),
                actqu(A,Q),
                retnaq,
                retract(naq(0)),nl,
                retract(actqu(A,Q)),
                write(actqu(A,Q)),
                output(" has been deleted"),!,fail.
ecprocess(rq):- nl,
                output("number ? :"),read(N),
                assert(naq(N)),
                quact(Q,A),
                retnaq,
                retract(naq(0)),nl,
                retract(quact(Q,A)),
                write(quact(Q,A)),
                output(" has been deleted"),!,fail.
ecprocess(aq):- nl,
                assert(nno(0)),
                output("NO.    ACTIVITY ENTITIES/DESTINATION
QUEUES"),nl,
                output("---    -------- ------------------
------"),
                nl,nl,
                actqu(A,Q),
                nxtno(N),
                write(N),
                qul_output(N),
                A=..[A1|A2],
                act_output(A1),
                write(A2),
                output("/"),
                write(Q),nl,fail.
ecprocess(aq):- retractall(nno(_)),
                !,fail.
ecprocess(qa):- nl,assert(nno(0)),
                output("QUEUES TO DESTINATION
ACTIVITIES"),nl,
                output("------ -- ----------- ----------
"),nl,
                quact(Q,A),
                nxtno(N),write(N),
                qul_output(N),
                write(Q),tab(2),
                write(A),nl,fail.
ecprocess(qa):- retractall(nno(0)),
                !,fail.
```

```
ecprocess(re):- nl,
                interegate,
                output("returning to normal change
mode"),nl.

/* interegate : asks the user questions when quitting entity
cycle
                mode so that various database adjustments
can be
                made */

interegate :-
     output("have you deleted any queues from the system
(y/n) ? :"),
     get(121),
     interl,
     fail.
interegate :-
     output("have you deleted any activities from the system
(y/n) ? :"),
     get(121),inter2,fail.
interegate :-
     output("have you added any queues to the system (y/n) ?
:"),
     get(121),
     inter3,fail.
interegate :-
     output("have you added any activities to the system
(y/n) ? :"),
     get(121),
     inter4,fail.
interegate :- !.

/* interl : logic of database adjustments for queue deletion
;
          remove qusize fact,
          alter number fact for the entity (retn)
          remove queue from queue_list (remvequeue) */

interl :-
        repeat,
          output("name ? :"),read(Q),
          output("entity ? :"),read(E),
          Y=..[Q,E],
          retract(qusize(Y,N)),
          retn(N),
          remvequeue(Q),
          output("more (y/n) ? :"),
          get(110),!.

retn(N):-retract(number(E,N1)),
          N2 is N1 - N,
          assert(number(E,N2)),!.
```

```
remvequeue(Q) :-
          retract(queue_list(QL)),
          remq(QL,Q,QQL),
          assert(queue_list(QQL)),
          retract(queue_lisp(QP)),
          remq(QP,Q,QQP),
          assert(queue_lisp(QQP)),!.

remq([],_,[]).
remq([Q|L],Q,L1) :- remq(L,Q,L1).
remq([X|L],Q,[X|L1]) :- remq(L,Q,L1).

/* inter2 : logic of database adjustments for activity
deletion
             remove from activity_list (remveact)
             alter number of activities entities (numa)
             remove scheduled future events for the activity
(cntev)*/

inter2 :-
           repeat,
           assert(nbe(0)),
           output("name ? :"),read(A),
           output("entity list ? :"),read(EL),
           Y=..[A|EL],
           remveact(Y),
           cntev(Y),
           retract(nbe(N)),
           numa(EL,N),
           output("more (y/n) ? :"),
           get(110),!.

remveact(A) :-
              retract(activity_list(AL)),
              remq(AL,A,AAL),
              assert(activity_list(AAL)),!.

numa(EL,N) :-
               mem(E,EL),
               retnbr(E,N),
               fail.
numa(_,_) :- !.

retnbr(E,N) :-
               retract(number(E,Num)),N1 is Num-N,
               assert(number(E,N1)),!.

cntev(Y) :-
             retract(event(Y,_)),
             addnbe,
             fail.
cntev(_) :- !.

addnbe :-
```

```
        retract(nbe(N)),
        N1 is N+1,
        assert(nbe(N1)),!.

/* inter3 : logic of database adjustments for queue addition
;
            add queue to queue_list (retrqu)
            add qusize fact (size=0) to database
            form attribute database (attribute_list)
(attrib_db) */

inter3 :-
        repeat,
        output("name ? :"),read(Q),
        output("entity ? :"),read(E),
        retrqul(Q),
        Y=..[Q,E],
        assert(qusize(Y,0)),
        attrib_db(Q,[E]),
        output("more (y/n) ? :"),get(110),!.

retrqul(Q):-retract(queue_list(QL)),
            assert(queue_list([Q|QL])),!.

attrib_db(Q,E) :-
        mem(E1,E),
        attribute_name(E1,EM),
        mem(EEM,EM),
        assert(attribute_list(Q,EEM,[])),fail.
attrib_db(_,_) :- !.

/* inter4 : logic of database adjustments for activity
additions
            add to activity list - in priorities place
(add_to_act_list)
            add simoult_realis fact
            add curr_realis fact (0)
            form attribute database (attribute_list)
(attrib_db) */

inter4 :-
        repeat,
        output("name ? :"),read(A),
        output("entity list ? :"),read(EL),
        Y=..[A|EL],
        add_to_act_list(Y),
        output("simoultaneous realisations ? :"),
        read(SN),
        assert(simoult_realis(Y,SN)),
        assert(curr_realis(Y,0)),
        attrib_db(A,EL),
        output("more (y/n) ? :"),get(110),!.

add_to_act_list(Y) :-
```

```
        assert(nno(0)),
        activity_list(AL),
        output("current activities (in order of priority
:"),nl,nl,
        mem(A,AL),
        nxtno(N),
        write(N),
        tab(2),
        write(A),nl,fail.
add_to_act_list(Y) :-
        retractall(nno(_)),
        nl,nl,
        output("what number priority (>= 0) ? :"),read(P),
        insert2(Y,P),nl,!.

insert2(A,N) :-
        retract(activity_list(AL)),
        split2(AL,N,ALF,ALT),
        a_t_at_list(A,ALF,L1),
        conc(L1,ALT,L2),
        assert(activity_list(L2)).
split2(A,0,[],A).
split2(L,N,LF,LT) :-
            first_n_elems(L,N,LF,LT),!.

/* retnaq and nxtno are counter modifiers */

retnaq :-
          retract(naq(N)),
          N1 is N-1,
          assert(naq(N1)),!.

nxtno(N) :-
            retract(nno(N1)),
            N is N1+1,
            assert(nno(N)),!.

/* mnum(O,N,E) : work out new number of entity E, given
                 interaction changes */

mnum(O,N,E):-N>=O,
             Diff is N-O,
             retract(number(E,EN)),END is EN+Diff,
             assert(number(E,END)).
mnum(O,N,E):-Diff is O-N,
             retract(number(E,EN)),END is EN - Diff,
             assert(number(E,END)).

/* pf(N,ON,E) : modify size of world pool for entity E given
                the interaction changes - if the pool is not
                large enough to absorb all the required
change
                then add a 'remove' fact to the database
```

```
                    giving the size of the change still to be
made */

pf(N,ON,E):-N>=ON,
                Diff is N-ON,
                pools(P),
                mem(M,P),
                X=..[M,E],
                retract(qusize(X,S)),S1 is S+Diff,
                assert(qusize(X,S1)).
pf(N,ON,E):-Diff is ON-N,
                pools(P),
                mem(M,P),
                X=..[M,E],
                retract(qusize(X,S)),
                finddiff(X,S,Diff),
                retractall(remove(X,0)).

finddiff(X,S,Diff) :-
                S=<Diff,
                D is Diff-S,
                assert(qusize(X,0)),
                assert(remove(X,D)).
finddiff(X,S,Diff) :-
                SD is S-Diff,
                assert(qusize(X,SD)).
```

Listing of The Results Output Facility

```
output([X|L]):-put(X),output(L).
output([]):-!.

/* output new banner every page(N) lines */

page(20).

cnum(0).

newban :- page(N),
          retract(cnum(N)),
          output("TIME      ACTIVITY "),qu_ban_out,
          assert(cnum(0)),!.
newban :- retract(cnum(N)),N1 is N+1,
          assert(cnum(N1)),!.

/* qu_ban_out is satisfied if a list of the queue names
(each truncated
   to five characters if neccessary), separated by two
spaces is output */

qu_ban_out:-
      queue_lisp(QL),mem(X,QL),name(X,LL),
      length(LL,Len),qnam_trunc(LL,L,Len,Len2),
      output(L),Tab is 7-Len2,tab(Tab),fail.
qu_ban_out:-nl,!.

qnam_trunc(L,L,Len,Len):- Len<6,!.
qnam_trunc([A,B,C,D,E|_],[A,B,C,D,E],Len,Len2):-Len2 is 5,!.

/* under_ban_out is satisfied if the underlines
corresponding
   to the queue list output in qu_ban-out are output  */

under_ban_out:-
      queue_lisp(QL),mem(X,QL),name(X,L),length(L,Len),
      und_out(Len,Len2),Tab is 7-Len2,tab(Tab),fail.
under_ban_out:-nl,!.

und_out(0,0):-!.
und_out(Len,5):-Len>5,Le is 5,und_out(Le,Le),!.
und_out(Len,Len):-output("-"),L2 is Len-1,und_out(L2,L2),!.

/* the following predicates are used to supply the
information
   neccessary for neat output of results :
   digits_num(Int,No) is satisfied when the integer Int has
No digits;
   length(List,Leng) is satisfied whrn list List has Leng
no. of elements */

digits_num(Num,1):- Num<10,!.
digits_num(Num,2):- Num<100,!.
digits_num(Num,3):- Num<1000,!.
digits_num(Num,4):- Num<10000,!.
digits_num(Num,5):-!.

length([],0):-!.
length([_|L],Length):-length(L,LL),Length is LL+1.

/* time_output is satisfied if the time is output neatly */
```

```
time_output:-

clock(X),write_real(X),retract(digits_numr(Num)),
              Tab is 9-Num,tab(Tab).

/* act_output(Y) is satisfied if activity name Y is output
neatly */

act_output(Y):-

name(Y,List),output(List),length(List,Length),
                 Tab is 9-Length,tab(Tab).

/* queue_output is satisfied if the current queue lengths
are all output
   in the correct order neatly */

 queue_output:-qu_output(Q),fail.

queue_output:-!.
qu_output(Q):-
q_output(Q),qsiztot(Q,Size),write(Size),qul_output(Size).
qsiztot(Q,_):- assert(size(0)),
               entity_list(EL),mem(E,EL),
               QS=..[Q,E],qst(QS),
               fail.
qsiztot(_,S):- retract(size(S)),!.
qst(QS) :- qusize(QS,Sizel),
           retract(size(S)),Size is S+Sizel,
           assert(size(Size)),!.
qul_output(Size):-
               digits_num(Size,Length),
               Tab is 7-Length,tab(Tab),!.
q_output(Queue):-queue_lisp(L),mem(Queue,L).
```

Listing of The Real Number Formatting Routines

```
write_real(R) :-
                assert(digits_numr(0)),
                assert(wrblubct(R)),
                w_real,fail.
write_real(_) :- retractall(wrblubct(_)),retractall(e(_)),
                  retractall(fst(_)),!.

w_real :-
          retract(wrblubct(R)),
          exponent(R,Exp),
          digital(R,Exp).

exponent(R,2) :- R>=100,not(e(_)),assert(e(2)),!.
exponent(R,1) :- R>=10,not(e(_)),assert(e(1)),!.
exponent(R,0) :- R>=1,not(e(_)),assert(e(0)),!.
exponent(R,-1.0) :- R>=0.1,not(e(_)),assert(e(-1.0)),!.
exponent(R,-2.0) :- R>=0.01,not(e(_)),assert(e(-2.0)),
                    assert(fst(_)),!.
exponent(R,E) :-
                R>0.0,
                retract(e(E0)),
                fe(E0,E),
                assert(e(E)),!.

fe(2,1).
fe(1,0).
fe(0,-1.0).
fe(-1.0,-2.0).

dtal(N) :-
          retract(digits_numr(NN)),N1 is NN+N,
          assert(digits_numr(N1)),!.

digital(R,2) :-
                dtal(1),
                fdig2(R,Digit),



                write(Digit),
                R1 is R-(Digit*100),
                assert(wrblubct(R1)),fail.
digital(R,1) :-
                dtal(1),
                fdig1(R,Digit),
                write(Digit),
                R1 is R-(Digit*10),
                assert(wrblubct(R1)),fail.
digital(R,0) :-
                dtal(1),
                fdig0(R,Digit),
                write(Digit),
                R1 is R-Digit,
                assert(wrblubct(R1)),fail.
digital(R,-1.0):-
                dtal(2),
                fdim1(R,Digit),
                output("."),
                write(Digit),
                R1 is R-(Digit/10),
                assert(wrblubct(R1)),fail.
```

```
digital(R,-2.0):-
                fst(-2.0),
                dtal(3),
                fdim2(R,Digit),
                output(".0"),
                write(Digit),
                R1 is R-(Digit/100),
                assert(wrblubct(R1)),fail.
```

```
digital(R,-2.0):-
                dtal(1),
                fdim2(R,Digit),
                write(Digit),
                R1 is R-(Digit/100),
                assert(wrblubct(R1)),fail.

fdig2(R,9)  :- R>=900,!.
fdig2(R,8)  :- R>=800,!.
fdig2(R,7)  :- R>=700,!.
fdig2(R,6)  :- R>=600,!.
fdig2(R,5)  :- R>=500,!.
fdig2(R,4)  :- R>=400,!.
fdig2(R,3)  :- R>=300,!.
fdig2(R,2)  :- R>=200,!.
fdig2(R,1)  :- R>=100,!.
fdig2(_,0) :-!.
fdigl(R,9)  :- R>=90 ,!.
fdigl(R,8)  :- R>=80 ,!.
fdigl(R,7)  :- R>=70 ,!.
fdigl(R,6)  :- R>=60 ,!.
fdigl(R,5)  :- R>=50 ,!.
fdigl(R,4)  :- R>=40 ,!.
fdigl(R,3)  :- R>=30 ,!.
fdigl(R,2)  :- R>=20 ,!.
fdigl(R,1)  :- R>=10 ,!.
fdigl(_,0) :- !.
fdig0(R,9)  :- R>=9  ,!.
fdig0(R,8)  :- R>=8  ,!.
fdig0(R,7)  :- R>=7  ,!.
fdig0(R,6)  :- R>=6  ,!.
fdig0(R,5)  :- R>=5  ,!.
fdig0(R,4)  :- R>=4,  !.
fdig0(R,3)  :- R>=3  ,!.
fdig0(R,2)  :- R>=2  ,!.
fdig0(R,1)  :- R>=1  ,!.
fdig0(_,0) :- !.
fdiml(R,9)  :- R>=0.9,!.
fdiml(R,8)  :- R>=0.8,!.
fdiml(R,7)  :- R>=0.7,!.
fdiml(R,6)  :- R>=0.6,!.
fdiml(R,5)  :- R>=0.5,!.
fdiml(R,4)  :- R>=0.4,!.
fdiml(R,3)  :- R>=0.3,!.
fdiml(R,2)  :- R>=0.2,!.
fdiml(R,1)  :- R>=0.1,!.
fdiml(_,0) :- !.
fdim2(R,9)  :- R>=0.09,!.
fdim2(R,8)  :- R>=0.08,!.
fdim2(R,7)  :- R>=0.07,!.
fdim2(R,6)  :- R>=0.06,!.
fdim2(R,5)  :- R>=0.05,!.
fdim2(R,4)  :- R>=0.04,!.
fdim2(R,3)  :- R>=0.03,!.
```

APPENDIX 4 - LISTING OF THE GENERAL AGV EXPERT

```
/* FIRST SECTION : simple data processing predicates
   used by the rest of the expert system          */

member(E,[E|_]).
member(E,[_|T]):-member( E , T ).

output([]):-!.
output([X|L]):-put(X),output(L).

/* min(L,M) M is smallest element of L
   max(L,M) M is largest  element of L */

min(L,M) :-
     member(M,L),sthan(M,L).
max(L,M) :-
     member(M,L),lthan(M,L).

sthan(M,L) :-
     member(MM,L),M>MM,!,fail.
sthan(_,_).
lthan(M,L) :-
     member(MM,L),M<MM,!,fail.
lthan(_,_).

/* sthan1(S,L1,L2) : L2 is a list of the elements
                     of L1 smaller than S     */

sthan1(St,[],[]).
sthan1(St,[H|T],[H|R]) :- H<St,sthan1(St,T,R).
sthan1(St,[_|T],R) :- sthan1(St,T,R).

difference(_,[],[]) :- !.
difference(X,[Y|L],Z):-member(Y,X),difference(X,L,Z).
difference(X,[Y|L],[Y|Z]):-
not(member(Y,X)),difference(X,L,Z).

/* ascii(L1,L2) : find L2, list of ascii values of list L1
   char(L1,L2)  : find L2, list of chars of ascii nos. in L1

   convass : converts first two parameters to ascii
   assconv : converts first two parameters to character form
*/

ascii([],[]).
ascii([N1|N2],[MN|N3]) :-
          name(N1,[MN]),ascii(N2,N3).
ascii(N,NA) :-
          atomic(N),name(N,NA).

char([],[]).
char([N1|N2],[MN|N3]) :-
          name(MN,[N1]),char(N2,N3).
char(N,NA) :-
          atomic(N),name(NA,[N]).

convass(NL,N,ANL,AN) :-
          ascii(NL,ANL),ascii(N,[AN]),!.
assconv(NL, N ,ANL,AN) :-
          char(NL,ANL),char(N,AN),!.

conc([],X,X) :- !.
conc([A|B],C,[A|D]) :- conc(B,C,D).
```

```
/* SECOND SECTION deals with user interaction facilities */

/* database(N) : requests user for the extra information
          required for method no. N.          */

database(1).
database(2) :-
     output(" maximum distance : "),
     read(M),
     assert(max(M)).
database(3) :-
     output(" start of route : "),
     read(S),
     assert(start(S)).
database(4) :-
     database(2),
     database(3).

/* choose : allows user to specify required method no. */

choose :-
   output(" type required method no. : "),
   read(X),
   retractall(method(_)),
   assert(method(X)).

curr_dist1(X) :- curr_dist(X),!.

curr_dist(0).

/* THIRD SECTION deals with the expert system
   explanation facility */

/* retpred (& retpar) access the trace of
   heuristic calls */

 retpar(A,B,C,D) :-
          retract(tparams( A,B,C,D)),!.
retpred(X,A,B,C,D) :-
          retract(pred(X)),retpar(A,B,C,D).

/* how predicate executes explanation facility
   using the trace of heuristic calls */

how :-
     retpred(X,A,B,C,D),
        text(X,A,B,C,D),
     output(" and then..."),nl,
     get(32),
     fail.
how :- output(" I did a depth first AND-OR tree search "),
     nl,r.

/* explanation texts associated with each of the
   heuristics */

text(intel2,A,B,C,D) :-output("I tried heuristic 1"),nl.
text(intel3,A,B,C,D) :-output("I tried heuristic 2"),nl.
text(intel1,A,B,C,D) :-output("I tried heuristic 3"),nl.

/* enquiry_save(P,A,B,C,D) adds parameters of newly
   envoked heuristic P to the trace */
```

```
enquiry_save(P,A,B,C,D) :-
          assertz(pred(P)),
          assertz(tparams(A,B,C,D)),!.

/* FOURTH SECTION deals with search mechanism and
   heuristics of the expert system        */

/* intel(S,N,P) : controls expert process, ie.

   1. process the problem parameters
   2. envoke the 'sh' predicate.          */

intel(S,N,P) :-
          intel2(S,N,P).
intel(S,N,P) :-
          member(M,N),difference([M],N,NM),
          sh(S,NM,M,P),
          enquiry_save(intel1,S,NM,M,P).

intel2(Z,N1,P ) :-
     convass(N1,Z,N,Y),
     member(M,N),M<Y,sthan1(Y,N,N2),max(N2,E),difference([E]
,N,NE),
     assconv(NE,E,ANE,AE),!,
     enquiry_save(intel2,Z,ANE,AE,P),
     sh(Z,ANE,AE,P).
intel2(S,N1,P ) :-
     convass(N1,S,N,Y),
     max(N,E),difference([E],N,NE),
     assconv(NE,E,ANE,AE),!,
     enquiry_save(intel3,S,ANE,AE,P),
     sh(S,ANE,AE,P).

/* sh (s_route,retr,distance,path,pat_t,check_distance,end)

   envoke the required search method to find the best route.
    Method envoked is coded 1 to 4 as the first parameter of
   pat_t :

   1 = normal exhaustive depth first search
   2 = exhaustive depth first search with max. distance
       cut off point
   3 = exhaustive depth first search with specified start
   4 = exhaustive depth first search with specified start
       and max. distance.                          */

sh(Source,_,Dest,_):-
   path(Source,Dest,Path),
   distance(Path,Dist),
   assert(route(Path,Dist)),
   fail.

sh(_,List,_,_):-
   route(Path,_),
   member(M,List),
   not member(M,Path),
   retract(route(Path,_)),
   fail.

sh(_,List,_,_):-
   s_route(X,D),
   D<X,
   retr(D),
   fail.
```

```
sh(_,List,_,Path):-
    smallest_dist(X),
    routel(Path,X).


s_route(X,D) :-
    retract(route(N,D)),
    assert(routel(N,D)),
    smallest_dist(X).

retr(D):- retract(smallest_dist(_)),
    asserta(smallest_dist(D)),!.

distance([_|[]],0).

distance([X,Y|Z],Dist):-
    arc(X,Y,D1),
    distance([Y|Z],D2),
    Dist is D1+D2.


path(Source,Dest,Path):-
    method(X),
    pat_t(X,Source,Dest,Path,[]).

pat_t(1,Point,Point,[Point],_).
pat_t(1,New_source,Dest,[New_source|Rest],Points_so_far) :-
    arc(New_source,Next,_),
    not member(Next,Points_so_far),
    pat_t(1,Next,Dest,Rest,[Next|Points_so_far]).

pat_t(2,Point,Point,[Point],_).
pat_t(2,New_source,Dest,[New_source|Rest],Points_so_far) :-
    arc(New_source,Next,D),
    not(member(Next,Points_so_far)),
    curr_dist1(CD),CDD is CD+D,
    asserta(curr_dist(CDD)),
    check_distance(CDD),
      pat_t(2,Next,Dest,Rest,[Next|Points_so_far]).

pat_t(3,Source,Dest,Path,LL) :-
    start(L),
    end(L,End),
    pat_t(1,End,Dest,[_|Path1],LL),
    conc(L,Path1,Path).

pat_t(4,Source,Dest,Path,LL) :-
    start(L),
    end(L,End),
    pat_t(2,End,Dest,[_|Path1],LL),
    conc(L,Path1,Path).

end([X|[]],X) :-!.
end([_|L],X) :- end(L,X).

check_distance(CDD) :-
    max(M),
    CDD=<M.
check_distance(_) :-
    retract(curr_dist(_)),!,fail.
```

```
/* FIFTH SECTION deals with initial conditions & reseting */

/* reset system after a consultation */

r:-  retractall(routel(_,_)),
     retractall(route(_,_)),
     retractall(max(_)),
     retractall(method(_)),assert(method(1)),
     retractall(start(_)),
     retractall(curr_dist(_)),assert(curr_dist(0)),
retractall(smallest_dist(_)),asserta(smallest_dist(16383)),
     retractall(tparams(_,_,_,_)),retractall(pred(_)).

/* default method is no. 1. */

smallest_dist(16383).

method(1).

/* PROBLEM SPECIFIC map information */

arc(a,b,7).
arc(b,c,10).
arc(b,k,5).
arc(c,d,4).
arc(d,e,6).
arc(d,k,10).
arc(e,f,3).
arc(f,g,14).
arc(g,k,4).
arc(g,h,11).
arc(h,i,3).
arc(i,j,5).
arc(j,a,4).
arc(k,j,7).
arc(k,h,13).
```

APPENDIX 5 - LINKING PROLOG AND ASSEMBLER

Introduction

This Appendix details how synchronisation between two IBM PC compatible machines is obtained using MACRO 86, and also how PROLOG communicates with the Assembler.

Communications via an Asynchronous Adapter

(ref : Sargent, and Shoemaker, 1984). Communications from a PROLOG expert system via an RS232 data link to another program (a simulation) on a different computer is achieved using a modified version of the standard ^S/^Q protocol (also known as DC1/DC3 or XON/XOFF) for handshaking. The code for the interprocessor link is written in the assembler language MACRO 86. Assembler is used because it is a language at a lower level than conventional programming languages. This means that the user has greater control on the use of computer resources and devices. On the debit side it is a much harder type of language to program with. Indeed conventional languages are converted into Assembler by the first part of a compilation program.

A schematic diagram of the interplay between expert system (slave) and simulation (master) is as follows :

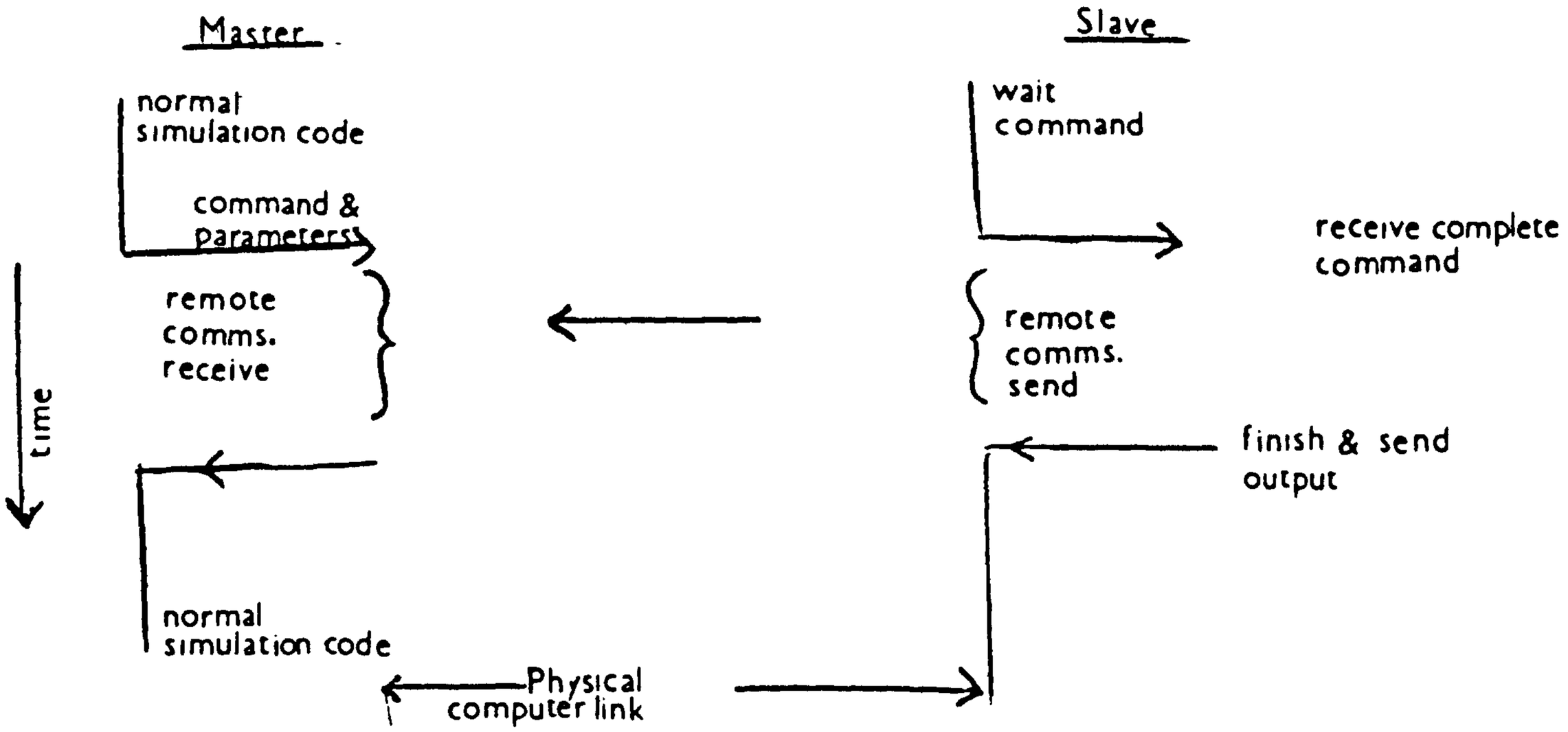


Figure A5.1 Slave/Master Communication

The slave waits to receive a command from the master. The command consists of a predicate (name) followed by input parameters. This command is then executed by the slave, whilst the master awaits its completion. During command execution various forms of interaction between master and slave may take place. When all interaction for the command is complete, the slave informs the master and sends him any results.

Initialisation The Asynchronous adapters at both computers need to be set to the same communications parameters. Without this the computers could not communicate, since data

would be lost by the computer expecting the slower communications rate. The parameters used here are :

9600 baud (960 char/second)
1 stop bit
no parity
8 bit words

When characters are being sent at this speed, it may occur that the receiver is not processing them fast enough - in which case some characters may be lost. To overcome this problem a simple version of the ^Q/^S protocol is used.

When the sender passes a character to the receiver, no more characters are passed across until the receiver echoes back a ^Q indicating that he is ready to receive the next character.

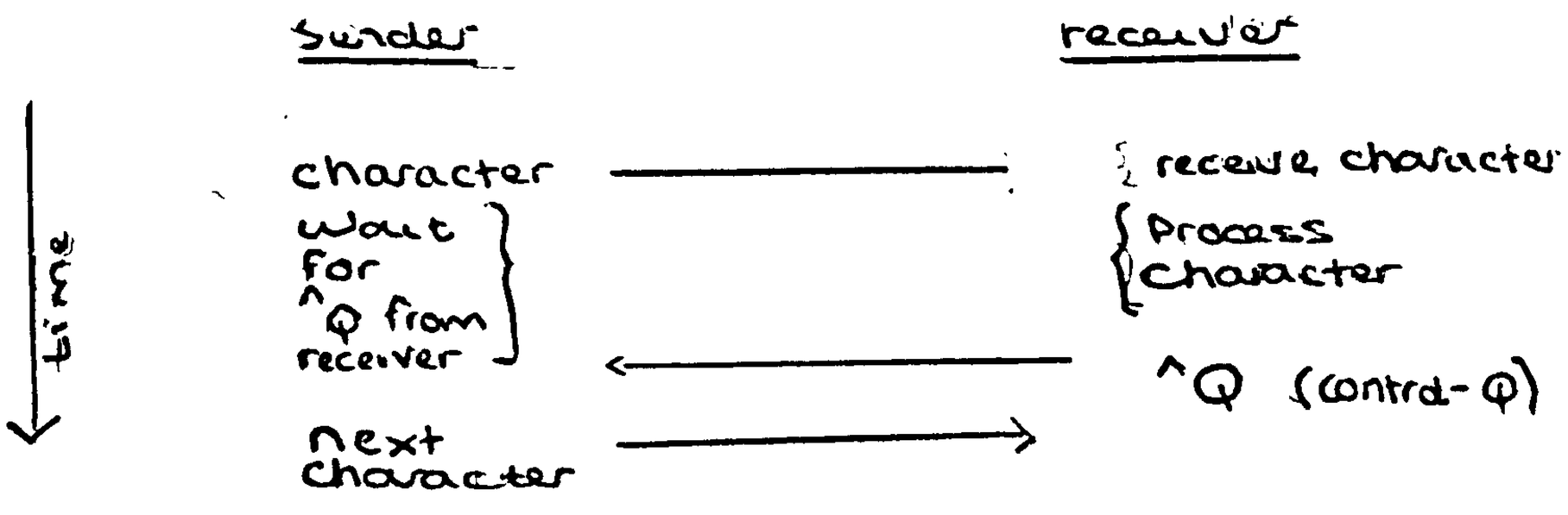


Figure A5.2 Simplified Handshaking

(note that in normal ^Q/^S protocol the receiver sends a ^S to stop the sender. Here that ^S is implicit after each character is sent. This is necessary here because the transmission rate is so fast data waiting at a port would be

overwritten by new data arriving. Hence we call this the 'implicit ^S protocol').

The implicit ^S protocol is not the only form of handshaking used. It has also proved necessary to coordinate the two communicating programs so that they are ready to send or receive at precisely the correct moment. We call this the 'coordinating protocol'.

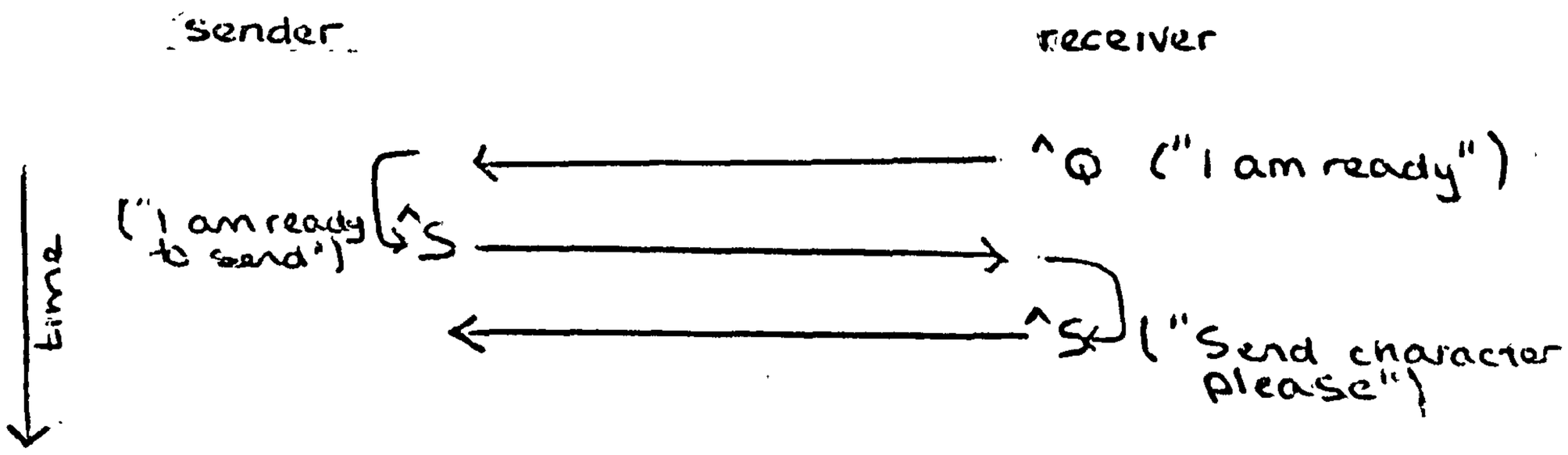


Figure A5.3 The Coordinating Protocol

PROLOG Communication with Assembler

The version of PROLOG used at Warwick (in common with most other implementations) includes a mechanism for calling machine language code from within PROLOG (ESI, 1983). This machine code is executed via the PROLOG system predicate 'external_code'. This requires three arguments: an operation number, an input parameter list and an output parameter list. Each list may contain up to eight parameters which may be PROLOG atoms or integers. Passing lists or other structures is not permitted.The operation number specifies which of a number of operations provided by the machine language module is to be used.

Parameters are passed between PROLOG and the external code module by means of a parameter area in the users data segment. The parameter area consists of 51 bytes, which are used as follows (ESI, 1983):

Table A5.1 The PROLOG-1/Assembler Common Area

Address	Name	Storage	Use
0	op	1 byte	operation number (0-255)
1	incnt	1 byte	number of input arguments

2	outcnt	1 byte	number of output arguments
3	in	8 words	up to 8 input argument values
19	out	8 words	up to 8 output argument values
35	intag	8 bytes	types of input arguments
43	outtag	8 bytes	types of output arguments

APPENDIX 6 - LISTING OF THE PROLOG AND ASSEMBLER LINKING ROUTINES

PROLOG Remote Link Routines

```
/* remote communications predictes for use with link3.exe :

   command    : wait for a command and execute it
   commsend(X) : send a command X & interact
   commend : master ends communication
   commmsg(X) : master sends string X
   readri(X) : remote read integer
   readra(X) : remote read atom
   readrl(X) : remote read list
   writera(X) : remote write atom
   writeri(X) : remote write integer
   comlsend(X) : send a list X
   comlsnd(X) : send a list X to be printed
   comlrec(X) : receive a list X
   readi(M) : receive an integer M
   reada(M) : receive an atom M
   nlr : remote write new line                    */

command :- manaut(m),external_code(12,[],[X]),callo(X).
command :- external_code(12,[],[X]),comcall(X).

comcall(1000).
comcall(X) :-callo(X),!,command.

callo(X) :- call(X).

reada(X) :- external_code(12,[],[X]),!.

commsend(X) :- external_code(7,[X],[]),
          remparams(X),
          repeat,
          external_code(11,[],P),
          remote_list_request(P),!.

remote_list_request([500]) :-
                read(X),
                 comlsnd(X),!,fail.
remote_list_request(_).

commend :- external_code(1,[],[]).

commmsg(X) :- external_code(2,[],[]),
          outputr(X),
          external_code(4,[],[]).
outputr([]):-!.
outputr([X|L]) :- external_code(3,[X],[]),
          outputr(L).

readri(X) :- external_code(5,[],[X]),!.

readra(X) :- external_code(6,[],[P]),
          name(P,[_|Y]),
```

```
          name(X,Y),!.

readrl(Z) :- external_code(14,[],[]),
```

```
            repeat,
            external_code(13,[],X),
            smth(X),
            X==[],
            retract(remote_list(Z)),
            assert(remote_list([])).

writera(X) :- external_code(7,[X],[]),!.

writeri(X) :- external_code(8,[X],[]),!.

nlr :-      external_code(2,[],[]),
      external_code(3,[13],[]),
      external_code(3,[10],[]),
      external_code(4,[],[]).


mem(X,[X|_]).
mem(X,[_|L]):- mem(X,L).

comlsend(X) :-
      trace,
            external_code(14,[],[]),
            mem(M,Q),
            comati(M,N),
            external_code(N,[M],[]),
            fail.
comlsend(_):-   external_code(10,[],[]).

 comlsnd(X) :-
            mem(M,Q),
            comati(M,N),
            external_code(N,[M],[]),
            fail.
 comlsnd(_):-   external_code(10,[],[]).

readi(X) :- external_code(15,[],[X]),!.

remote_list([]).

comlrec(Z) :-
            external_code(11,[],[]),
            repeat,
            external_code(13,[],X),
            smth(X),
            X==[],
            retract(remote_list(Z)),
            assert(remote_list([])).

smth([]):-!.
smth([X]) :- retractb(remote_list(Y)),
            assert(remote_list([X|Y])),!.

retractb(X) :- retract(X),!.

comati(M,7) :- atom(M),!.
comati(_,8) :- !.


rev([],[]) :- !.
rev([X|L],Y) :-
      rev(L,L1),append(L1,[X],Y).
```

```
append([],X,X).
append([X|L1],L2,[X|L3]) :-
          append(L1,L2,L3).

?-consult(automan).
```

Assembler Remote Link Interface

```
; this file controls the PROLOG communications protocol
; for use  over the RS-232 link

    intg equ  0
    atom equ  4
    black     equ  0
    blue equ  1
    red  equ  2
    white     equ  3
    commprt   equ  3f8h
    mctl equ  3fch
    istat     equ  mctl+1

ex      segment
assume    cs:ex,ds:ex
main proc far
op   db   0
incnt     db   0
outcnt    db   0
in   dw   8 dup      (?)
out  dw      8 dup   (?)
intag   db       8 dup   (?)
outtag  db       8 dup   (?)

assume    cs:ex,ds:ex

     mov  bl,[op]
     xor  bh,bh
     add  bx,bx
     mov  bx,table[bx]
     jmp  bx

init macro

     mov  ax,0e3h          ;set up for 9600 baud,1 stop bit,no
     int  14h         ;parity, 8-bit words
     mov  dx,mctl          ;point at modem control register
     mov  al,3        ;force DTR_ and RTS_ low
     out  dx,al
     endm

cstl macro                 ; read typed character on keyboard
     push dx
     mov  ah,06
     mov  dl,0ffh
     int  21h
     pop  dx
     endm

prm  macro      p1,p2,p3
     cmp  al,p1
     jb   p2
     push ax
     mov  ah,06
     mov  dl,p3
     int  21h
     pop  ax
     sub  al,p1
     jmp  pr10
     endm

table:     dw   extc0
```

```
        dw    extc1
        dw    extc2
        dw    extc3
        dw    extc4
        dw    extc5
        dw    extc6
        dw    extc7
        dw    extc8
        dw    extc9
        dw    extc10
        dw    commpg
        dw    commp3
        dw    extc13
        dw    extc14
        dw    extc15

flag            dw    0
buff            dw    0
bf      dw      0
point           dw    0
count           dw    0
no      db      0
ass     db      0
tot     db      0
multer          db    0
pri0    db      0
pri1    db      1
pri2    db      2
pri3    db      3
pri4    db      4
pri5    db      5
pri6    db      6
pri7    db      7
pri8    db      8
pri9    db      9
recsend         db    0


area:           db    256 dup (?)

extc0:          init                    ;sending an atom
        mov     recsend,0       ;sending (not a remote read)
        mov     count,0
        mov     point,offset area
        call    fbuff           ;read atom into memory
ext04:          mov   dx,commprt        ;point at comms base port
                call  csts
        mov     ax,bf
        cmp     flag,0
        je      ext01
        mov     dx,commprt
        call    ocmott          ;send lst. char after co-ord
h/shake
ext02:          call  csts      ;get next character
        mov     ax,bf
        cmp     flag,0          ;end of atom ?
        je      ext01
                call  comout            ;no. send character to serial
device
        jmp     ext02
ext01:          call  comsts            ;serial character received ?
        jz      ext02
        call    comin           ;yes. get it from serial port
        call    com         ;check that its ^Q (h/shake)
```

```
        mov  al,"Q"-40h         ;indicate end of atom - send ^Q
        call comot
        cmp  recsend,0
        jne  comjmp             ;just executed remote read
        ret

extc1:  mov  al,"T"-40h         ;end of predicate comms.
        mov  dx,commprt
        call comott
        ret

extc2:  mov  al,"P"-40h         ;start of typed message
        mov  dx,commprt
        call comott
        ret


extc3:  init                    ;send character from in
        mov  count,0            ;after co-ordinating handshake
        mov  point,offset area
        mov  dx,commprt
        mov  ax,in
        call comott
        ret

extc4:  mov  al,"R"-40h         ;end of typed message
        mov  dx,commprt
        call comott
        ret

extc5:  mov  al,"I"-40h         ;read remote integer
        mov  dx,commprt
        call comott
        call rdint              ;get integer from serial port
        ret

extc6:  mov  al,"A"-40h         ;read remote atom
        mov  dx,commprt
        call comott
        jmp  commp2             ;get atom and move it about

extc7:  mov  al,"X"-40h         ;send an atom
        mov  dx,commprt
        call comott
        jmp  extc0

comjmp: jmp  commpg

extc8:  mov  al,"W"-40h         ;write integer remote
        mov  dx,commprt
        call comott
        mov  ax,in
        mov  dx,commprt
        call comott             ;send integer at in to serial port
        ret

commp3: init                    ;read atom for out
        mov  count,0
        mov  point,offset area
        mov  dx,commprt
        call hshakf
        call hshakg
commsr: call comsts
```

```
      jz    commsr
      call  comin
      call  co
      cmp   count,10  ;end of a list ?
      jne   commp2          ;no
      mov   outcnt,0  ;yes
      ret
commp2:   init
      mov   no,1
      mov   count,0
      mov   point,offset area
            mov  dx,commprt     ;point at comms base port
      call  hshakf
            call hshakg
comml2:   call  comsts          ;get next character
      jz    comml2
      call  comin           ;store next character, or send
      call  coo         ;completed atom to out
      cmp   count,0
      jne   comml3          ;end of atom
      call  hshake          ;implicit ^S
      jmp   comml2
comml3:   ret

commsg:   init              ;receive typed message
      mov   count,0
      mov   point,offset area
      mov   dx,commprt
      call  hshakf
      call  hshakg
commsp:   call  comsts
      jz    commsp
      call  comin
      call  co
      cmp   count,4         ;have we received end of message ?
      jne   commsq
      jmp   commpg
commsq:   jmp   commsg

commpg:   init            ;standard comms receive
      mov   count,0         ;tests ctrl chars to jump to
correct code
      mov   point,offset area
            mov  dx,commprt     ;point at comms base port
      call  hshakf
            call hshakg
commlp:   call  comsts          ;serial character received ?
      jz    commlp
      call  comin           ;yes. get it from serial port
      call  co          ;test it
      cmp   count,2         ;end of predicate comms. ?
      jne   commlr
      mov   outcnt,0  ;yes
      mov   out,0
      ret
commlr:   cmp   count,3           ;typed message coming ?
      je    commsg          ;yes
      cmp   count,5         ;remote integer read request ?
      je    sendif          ;yes
      cmp   count,6         ;remote read atom request
      je    senda           ;yes
      cmp   count,7         ;atom being sent ?
```

```
     je   commpg            ;yes
     cmp  count,8           ;integer being sent ?
     je   wintj             ;yes
     cmp  count,14  ;remote read request list ?
     je   sendl             ;yes
     cmp  count,10  ;end of list ?
     je   commpg            ;yes
     cmp  count,9           ;list arriving ?
     jne  commlq
     ret                ;yes
commlq:   cmp  count,0          ;end of atom ?
     jne  commpg            ;yes
     call hshake            ;no - get next char
     jmp  commlp

wintj:    call wint
     jmp  commpg

sendl:    mov  outcnt,1  ;leave link2.exe to read
     mov  outtag,intg    ;list via prolog
     mov  out,500
     ret

sendif:   jmp  sendi

senda:    init                ;send requested atom
     mov  no,1
     mov  count,0
     mov  point,offset area
     inc  point
sendal:   cstl                ;read keyboard
     jz   sendal
     push dx          ;echo keyboard
     mov  dl,al
     int  21h
     pop  dx
     cmp  al,13             ;end of atom ?
     jz   sendad            ;yes
     mov  di,point  ;no - store character
     mov  [di],al
     inc  no
     inc  point
     jmp  sendal            ;get next character
sendad:   push dx           ;send atom
     mov  dl,10
     int  21h
     pop  dx
          mov  point,offset area
     mov  di,point
     mov  al,no
     mov  [di],al
     mov  buff,1
     xor  ah,ah
     mov  flag,ax
     mov  point,offset area
     mov  recsend,1
     jmp  ext04

Sendi:    init                ;send requested integer
     mov  no,0
     mov  count,0
     mov  point,offset area
Sendil:   cstl                ;read keyboard
     jz   sendil            ;echo keyboard
```

```
      push dx
      mov  dl,al
      int  21h
      pop  dx
      cmp  al,13              ;was character <CR> ?
      jz   sendid             ;yes compile integer and send
      mov  di,point   ;no - store character
      mov  [di],al
      inc  no
      inc  point
      jmp  sendil             ;get next character
sendid:   push dx             ;convert characters typed in
      mov  dl,10              ;to a single integer (1 byte)
      int  21h
      pop  dx
           mov  point, offset area
      mov  tot,0
snd5:      cmp  no,3
      jne  snd1
      mov  multer,100
      jmp  snd4
snd1:      cmp  no,2
      jne  snd2
      mov  multer,10
      jmp  snd4
snd2:      cmp  no,1
      jne  snd3
      mov  multer,1
snd4:      mov  di,point
           mov  al,[di]
      mov  ass,al
      sub  ass,48
      mov  al,multer
      mul  ass
      add  tot,al
      inc  point
      dec  no
      jmp  snd5
snd3:      mov  al,tot                    ;send integer to serial
port
      mov  dx,commprt
      call comott
      jmp  commpg

extc9:     init                      ;send a list control code
      mov  al,"L"-40h
      mov  dx,commprt
      call comott
      ret

extc10:    init                      ;end of 'sent list' control
code
      mov  al,"M"-40h
      mov  dx,commprt
      call comott
      ret

extc14:    mov  al,"N"-40h                ;read request list
control code
      mov  dx,commprt
      call comott
      ret

extc13:    init                      ;get
```

```
        mov  point,offset area
        mov  dx,commprt
        call hshakf
        call hshakg
xtl3sr: call comsts
        jz   xtl3sr
        call comin
        call co
        cmp  count,8          ;integer coming ?
        je   tcl314           ;yes
        cmp  count,10  ;end of list ?
        jne  xtcl32           ;no
        mov  outcnt,0  ;yes
        ret
xtcl32: init                  ;atom coming -
        mov  no,1        ;compile and send to out
        mov  count,0
        mov  point,offset area
             mov  dx,commprt      ;point at comms base port
        call hshakf
             call hshakg
tcl312: call comsts
        jz   tcl312
        call comin
        call coo
        cmp  count,0
        jne  tcl313
        call hshake
        jmp  tcl312
tcl313: ret
tcl314: call rdint            ;integer coming - send to out
        ret

extcl5: init                ;read integer without request
        mov  count,0
        mov  point,offset area
        mov  dx,commprt
        call hshakf
        call hshakg
tcl5sr: call comsts           ;read over control characters
        jz   tcl5sr
        call comin
        call co
        call rdint            ;get integer from serial port
        ret

main endp

;check for control characters arriving  to
;interpret what message type is coming.
;if no control character - type on screen

co   proc near
     push dx
     mov  ah,06
     mov  dl,al
     cmp  al,"Q"-40h
     je   co20
     cmp  al,"T"-40h
     jne  co5
     mov  count,2
```

```
        jmp   co3
co5:    cmp   al,"P"-40h
        jne   co6
        mov   count,3
        jmp   co3
co6:    cmp   al,"R"-40h
        jne   co7
        mov   count,4
        jmp   co3
co7:    cmp   al,"I"-40h
        jne   co8
        mov   count,5
        jmp   co3
co20:           jmp   co2
co8:    cmp   al,"A"-40h
        jne   co9
        mov   count,6
        jmp   co3
co9:    cmp   al,"X"-40h
        jne   c10
        mov   count,7
        jmp   co3
c10:    cmp   al,"W"-40h
        jne   c11
        mov   count,8
        jmp   co3

c11:    cmp   al,"L"-40h
        jne   c12
        mov   count,9
        jmp   co3
c12:    cmp   al,"M"-40h
        jne   c13
        mov   count,10
        jmp   co3
c13:    cmp   al,"N"-40h
        jne   co4
        mov   count,14
        jmp   co3
co4:    cmp   dl,10
        je    c15
        int   21h
        jmp   co3
c15:    int   21h
        mov   dl,13
        int   21h
        jmp   co3
co2:    mov   count,1
        mov   dl,32
        int   21h
co3:    pop   dx
        ret
co      endp

;mov integer to out

cooi    proc near
        push dx
        xor   ah,ah
        mov   out,ax
        mov   outtag,intg
        mov   outcnt,1
        pop   dx
        ret
```

```
cooi endp

;compile an arriving atom or
;indicate end of arriving list

coo  proc near
     push dx
     cmp  al,"Q"-40h
     je   cool
     cmp  al,"M"-40h      ;end of arriving list ?
     je   coo3       ;yes
     inc  no
     inc  point
     mov  di,point
     mov  [di],al
     jmp  coo2
coo3:     mov  outcnt,0
     mov  count,1
     ret
     jmp  coo2
cool:     mov  di,offset area
     mov  al,no
     mov  [di],al
     mov  bx,offset area
     mov  out,bx
     mov  outtag,atom
     mov  outcnt,1
     mov  count,1
coo2:     pop  dx
     ret
coo  endp

;get integer from serial port and type onto screen

wint proc near
     push dx
     init
     mov  count,0
     mov  point,offset area
     mov  dx,commprt
     call hshakf
     call hshakg
wintlp:   call comsts
     jz   wintlp
     call comin
     call convint                ;convert al to characters
     mov  ah,06
     mov  dl,32
     int  21h
     pop  dx
     ret
wint endp

;convert arriving integer inro a set of
;characters, for example
;123 --> '1' '2' '3'
;for printing on screen

convint   proc near
     push dx
     mov  ah,06
```

```
        cmp   al,100
        jb    cv1
        mov   no,3
        jmp   cv3
cv1:    cmp   al,10
        jb    cv2
        mov   no,2
        jmp   cv4
cv2:    mov   no,1
        jmp   cv5
cv3:    mov   pri0,0
        mov   pri1,100
        mov   pri2,200
        mov   pri3,255
        mov   pri4,255
        mov   pri5,255
        mov   pri6,255
        mov   pri7,255
        mov   pri8,255
        mov   pri9,255
        call  pr
cv4:    mov   pri0,0
        mov   pri1,10
        mov   pri2,20
        mov   pri3,30
        mov   pri4,40
        mov   pri5,50
        mov   pri6,60
        mov   pri7,70
        mov   pri8,80
        mov   pri9,90
        call  pr
cv5:    mov   pri0,0
        mov   pri1,1
        mov   pri2,2
        mov   pri3,3
        mov   pri4,4
        mov   pri5,5
        mov   pri6,6
        mov   pri7,7
        mov   pri8,8
        mov   pri9,9
        call  pr
        pop   dx
        ret
convint   endp

;print integer on screen

pr      proc near
        push dx
pr9:    prm   pri9,pr8,57
pr8:    prm   pri8,pr7,56
pr7:    prm   pri7,pr6,55
pr6:    prm   pri6,pr5,54
pr5:    prm   pri5,pr4,53
pr4:    prm   pri4,pr3,52
pr3:    prm   pri3,pr2,51
pr2:    prm   pri2,pr1,50
pr1:    prm   pri1,pr0,49
pr0:    prm   pri0,pr10,48
pr10:         pop   dx
```

```
        ret
pr      endp

;read an integer from serial port
;and send to out

rdint       proc near
        init
        mov   count,0
        mov   point,offset area
        mov   dx,commprt
        call  hshakf
        call  hshakg
rdmlp:      call  comsts
        jz    rdmlp
        call  comin
        call  cooi
        ret
rdint       endp

;implicit ^S protocol

hshake      proc near
        push  ax
hshak2:     call  comsts
        test  al,20h
        jz    hshak2
        mov   al,"Q"-40h
        mov   dx,commprt
        out   dx,al
        pop   ax
        ret
hshake      endp

;hshakf and hshakg : co-ordinating protocol

hshakf      proc near
        push  ax
              call  hshake
boing:      call  comsts
        jz    boing
        call  comin
        and   al,7fh
        cmp   al,"S"-40h
        jnz   boing
        pop   ax
        ret
hshakf      endp

hshakg      proc near
        push  ax
hwt:    call  comsts
        test  al,20h
        jz    hwt
        mov   al,"S"-40h
        mov   dx,commprt
        out   dx,al
        pop   ax
        ret
hshakg      endp

;confirm end of atom
```

```
com   proc near
      push dx
      mov  ah,06
      mov  dl,al
      cmp  al,"Q"-40h
      je   coml
      int  21h
coml:      pop      dx
      ret
com   endp

; fill input argument atom ready
; for sending over serial port

fbuff      proc near
      push dx
      mov  si,in
      mov  buff,0
      mov  cl,(es:[si])
      xor  ch,ch
      mov  di,offset area
iter:      mov  al,(es:[si])
      mov  [di],al
      inc  di
      inc  si
      inc  buff
      cmp  buff,cx
      je   leave
      jmp  iter
leave:     mov  buff,1
      mov  flag,cx
      pop  dx
fbuff      endp

;get next character of atom from
;buffer for sending over serial port

csts proc near
      mov  cx,flag
      cmp  buff,cx
      jg       cstse
      push dx
      inc  point
      mov  di,point
      dec  di
      mov  al,[di]
      inc  buff
      pop  dx
      jmp cstsf
cstse:     mov  flag,0
cstsf:     mov  bf,ax
csts endp

comsts     proc near
           add  dx,6          ;modem input status routine
      in   al,dx              ;get modem status in ah
      mov  ah,al
      nop
      dec  dx            ;acces must be min 2 microsecs apart
      in   al,dx              ;get line status in al
      sub  dx,5          ;leave dx pointing at data port
```

```
        test al,1       ;return nonzero if character received
        ret
comsts  endp

comin   proc near
        call comsts             ;serial input routine
        jz  comin
        in  al,dx
        ret
comin   endp

;send a character,
;with co-ordinating protocol

comott  proc near
        push ax         ;serial output routine
waits:  call comin          ;hang for ctrl-Q
        and al,7fh
        cmp al,"Q"-40h
        jnz waits
comou3: call comsts
        test al,20h
        jz  comou3
        mov al,"S"-40h
        mov dx,commprt
        out dx,al
waitr:  call comin
        and al,7fh
        cmp al,"S"-40h
        jnz waitr
comou4: call comsts
        test al,20h             ;transmitter ready for another char
?
        jz  comou4
        pop ax          ;yes. send it
        out dx,al
        ret
comott  endp

;send a character with co-ordinating protocol

ocmott  proc near
        push ax         ;serial output routine
awits:  call comin
        cmp al,"Q"-40h
        jne awits
ocmou3: call comsts
        test al,20h
        jz  ocmou3
        mov al,"S"-40h
        mov dx,commprt
        out dx,al
awitr:  call comin
        cmp al,"S"-40h
        jnz waitr
ocmou4: call comsts
        test al,20h             ;transmitter ready for another char
?
        jz  ocmou4
        pop ax          ;yes. send it
        out dx,al
        ret
ocmott  endp
```

```
;send a character without co-ordinating protocol

comout    proc near
          push ax          ;serial output routine
waitq:    call comin            ;hang for ctrl-Q
     and  al,7fh
     cmp  al,"Q"-40h
     jnz  waitq
comou2:   call comsts
     test al,20h          ;transmitter ready for another char
?
     jz   comou2
     pop  ax         ;yes. send it
     out  dx,al
     ret
comout    endp

;if port ready, send character in al

comot     proc near
     push ax
como2:    call comsts
     test al,20h
     jz   como2
     pop  ax
     out  dx,al
     ret
comot     endp

ex   ends

sseg segment   stack
     dw   80   dup  (?)
sseg ends
     end
```

Modifications to The Simulation Engine

```
    /* FIRST SECTION deals with modifications needed for
       dynamic scheduling problems.
        Schedules are arranged in the 'new_schedule'
       predicate which therefore is where the expert
       interface is placed.                          */

    c_phase :-
          clock(X),
          current_queue_is(CQ),
          new_schedule(CQ,X),prev_cl_time(X),!.

    current_queue_is(Q) :-
              pclock(PC),
              retract(qusize(QQ,1)),assert(qusize(QQ,0)),
              QQ=..[Q|_],
              remve_f_event(Q),!.

 remve_f_event(Q) :-
              clock(X),
              f_event(Q,T),
              T=<X,
              retract(f_event(Q,T)),
              chnge_demand_satis,fail.
 remve_f_event(_).

 chnge_demand_satis :-
              retract(demand_satis(P)),
              PP is P+1,
              assert(demand_satis(PP)),!.

 prev_cl_time(X) :-
              retractall(pclock(_)),
              assert(pclock(X)).

 new_schedule(CQ,X) :-
              pclock(PX),
              f_event(E,T),
              T=<X,T>PX,
              form_new_schedule(CQ,X).
 new_schedule(_,_).

 /* SECOND SECTION deals with the modifications needed
    for using the graphics package.       */

 record(end).
 record(_) :-
          clock(X),
          XX is fix(X),
          qusize(Q,1),
          picnum(Q,Num),
          retract(prevpic(P,PN)),
          assert(prevpic(Num,XX)),
          Numn is Num+14,
          Nump is P+14,
          external_code(Nump,[XX,1],[]),
          external_code(Numn,[XX,2],[]),!.

    /*
    record(_) :-
          clock(X),
          XX is fix(X),
```

```
        qusize(Q,1),
        picnum(Q,Num),
        retractall(prevpic(_,_)),
        assert(prevpic(Num,XX)),
        external_code(Num,[XX],[]),!.
*/
/* external_code procedure nos. for different pictures */

picnum(a(agv),11).
picnum(b(agv),1).
picnum(c(agv),2).
picnum(d(agv),3).
picnum(e(agv),4).
picnum(f(agv),5).
picnum(g(agv),6).
picnum(h(agv),7).
picnum(i(agv),8).
picnum(j(agv),9).
picnum(k(agv),10).

initialise :-
        d_base_initialise,
        external_code(0,[0],[]),!.

/* THIRD SECTION deals with the EXPERT SYSTEM INTERFACE.
   Co-ordinated by 'form_new_schedule' predicate :

   prepare_params: find the next problem for which
     expert is needed.

   pre_interface: allow user interaction to facilitate
     possible modification of expert behaviour.

   post_interface: inform user of experts advice, in the
     experts language. Allow requests for explanation
     of experts decision making process.

   restore_pic: restore results screen to its state
     of immediately before expert consultation.

   translate_expert: translate experts recommendation into
     a schedule for use by the simulation. */

prepare_params(X,LQ,LR) :-
            event(end,ET),
            retractall(event(_,_)),
            assert(event(end,ET)),
            repeat,
            f_et(E,X),
            chvstr(E),
            retract(vislist(LQ)),
            retract(vishead(LR)),
            assert(vislist([])),
            assert(vishead([])).

pre_interface(LQ,LR) :-
            (offtrace,show_demand([LR|LQ]));(
            external_code(12,[],[]),
            output(" demand at nodes :"),
            tab(1),write_list([LR|LQ]),nl,
            output(" do you wish to interact (y/n) ? "),
            (get(110);(
```

```
            commsend(choose),
            commsend(method),readi(Z),
            trcheck(Z))))).


post_interface(OUT) :-
            (offtrace,commsend(r));(
            output(" visit nodes thus : "),
            write_list(OUT),nl,
            output(" do you wish to know why (y/n) ? "),
            explan_interface,
            external_code(13,[],[])).

restore_pic :-
            prevpic(Num,XX),
            external_code(Num,[XX],[]).

translate_expert(OUT) :-
            assert(bout(OUT)),
            retract(bout([H|T])),
            assert(bout(T)),
            conout(H,T),
            retract(boutl(BL)),
            assert(boutl([])),
            part2(BL),!.

trcheck(5) :-
            assert(offtrace),
            external_code(13,[],[]),
            restore_pic,
            asserta(restore_pic).
trcheck(Z) :-
            assert(paramr(1,Z)),
            commsend(database).

form_new_schedule(CQ,X) :-
            prepare_params(X,LQ,LR),
            pre_interface(LQ,LR),
            assert(paramr(1,CQ)),
            assert(paramr(2,[LR|LQ])),
            demand_satis(R),
            assert(paramr(3,R)),
            commsend(intel),
            comlrec(OUT),
            post_interface(OUT),
            restore_pic,
            translate_expert(OUT),!.

/* FOURTH SECTION deals with simple data processing
   routines for use by the expert system interface */

/* moveon(Y) : move on the entity that has just completed
event Y.
                This involves moving & testing attributes as
well */

moveon(Y):-actqu(Y,X1),
            Y=..[_|Y1],assert(entities(Y1)),
            queue_select(Y,X1,X2),mem(X,X2),
            attrib_mod(Y,X),qu_siz_inc(X),fail.
moveon(Y):-actqu(Y,X),select_last(X,X1),
            mem(MX,X1),
            check_and_move(MX,Y),fail.
moveon(Y):-retractall(entities(_)),fail.
```

```
moveon(ja(agv)) :- retractall(demand_satis(_)),
          assert(demand_satis(0)).
moveon(_).

/* fin outputs end of simulation message
   & halts master-slave communications */

fin :- output("End of
simulation"),nl,external_code(42,[],[]),!.

explan_interface :- get(110),commsend(r).
explan_interface :-
          external_code(12,[],[]),commsend(how).

write_list([]).
write_list([X|L]) :-
          write(X),
          tab(1),
          write_list(L).

/* show_demand prints black boxes where demand exists */

show_demand(X) :-
          mem(M,X),
          Z=..[M,agv],
          picnum(Z,N),
          NN is N+14,
          clock(Y),
          external_code(NN,[Y,3],[]),fail.
show_demand(_).

part2(BR) :-
     revse(BR,BL),
     push,
     mem(MX,BL),
     time(MX,TL),
     assert(event(MX,TL)),rclock(TL),fail.
part2(_) :- pop,!.

revse(L1,L2) :- revzap(L1,[],L2).
revzap([X|L],L2,L3) :- revzap(L,[X|L2],L3).
revzap([],L,L).

push :-
     clock(X),assert(kcolc(X)),!.

pop :-
     retractall(clock(X)),
     retract(kcolc(X)),
     assert(clock(X)),!.

rclock(TL) :-
     retract(clock(X)),
     assert(clock(TL)),!.

conout(H,[T1|T]) :-
          conv(H,T1,ACT),
          retract(boutl(L)),
          conc([ACT],L,ACTL),
          assert(boutl(ACTL)),!,fail.
conout(_,[]) :- !.

/* translation information for interpretation of
```

```
   experts advice */

conv(a,b,ab(agv)).
conv(b,c,bc(agv)).
conv(b,k,bk(agv)).
conv(c,d,cd(agv)).
conv(d,e,de(agv)).
conv(d,k,dk(agv)).
conv(e,f,ef(agv)).
conv(f,g,fg(agv)).
conv(g,k,gk(agv)).
conv(g,h,gh(agv)).
conv(h,i,hi(agv)).
conv(i,j,ij(agv)).
conv(j,a,ja(agv)).
conv(k,j,kj(agv)).
conv(k,h,kh(agv)).

f_et([E],X) :-
     f_event(E,T),
     T =< X.
f_et([],_) :- !.

chvstr([]):-
     retractall(vishead(_)),
     retract(vislist([H|T])),
     assert(vislist(T)),
     assert(vishead(H)).
chvstr(E):-
      retract(vislist(VL)),
     conc(E,VL,EVL),
     assert(vislist(EVL)),!,fail.

/* initial database for changes */

prevpic(11,0).
pclock(0).
boutl([]).
vislist([]).
vishead([]).
demand_satis(0).

remparams(choose).
remparams(method) :-
          retract(paramr(1,X)),
          writeri(X).
remparams(database) :-
          retract(paramr(1,X)),
          writeri(X).
remparams(intel) :-
          retract(paramr(1,X)),
          writera(X),
          retract(paramr(2,Y)),
          comlsend(Y),
          retract(paramr(3,Z)),
          writeri(Z).

remparams(r).
remparams(how).
remparams(method).

?-reconsult(prvink4).
```

Linked Version of The General AGV Expert

```
/* FIRST SECTION : simple data processing predicates
   used by the rest of the expert system          */

member(E,[E|_]).
member(E,[_|T]):-member( E , T ).

output(X) :- commmsg(X),!.
output([]):-!.
output([X|L]):-put(X),output(L).

/* min(L,M) M is smallest element of L
   max(L,M) M is largest  element of L */

min(L,M) :-
     member(M,L),sthan(M,L).
max(L,M) :-
     member(M,L),lthan(M,L).

sthan(M,L) :-
     member(MM,L),M>MM,!,fail.
sthan(_,_).
lthan(M,L) :-
     member(MM,L),M<MM,!,fail.
lthan(_,_).

/* sthanl(S,L1,L2) : L2 is a list of the elements
                     of L1 smaller than S     */

sthanl(St,[],[]).
sthanl(St,[H|T],[H|R]) :- H<St,sthanl(St,T,R).
sthanl(St,[_|T],R) :- sthanl(St,T,R).

difference(_,[],[]) :- !.
difference(X,[Y|L],Z):-member(Y,X),difference(X,L,Z).
difference(X,[Y|L],[Y|Z]):-
not(member(Y,X)),difference(X,L,Z).

/* ascii(L1,L2) : find L2, list of ascii values of list L1
   char(L1,L2)  : find L2, list of chars of ascii nos. in L1

   convass : converts first two parameters to ascii
   assconv : converts first two parameters to character form
*/

ascii([],[]).
ascii([N1|N2],[MN|N3]) :-
          name(N1,[MN]),ascii(N2,N3).
ascii(N,NA) :-
          atomic(N),name(N,NA).

char([],[]).
char([N1|N2],[MN|N3]) :-
          name(MN,[N1]),char(N2,N3).
char(N,NA) :-
          atomic(N),name(NA,[N]).

convass(NL,N,ANL,AN) :-
          ascii(NL,ANL),ascii(N,[AN]),!.
assconv(NL, N ,ANL,AN) :-
          char(NL,ANL),char(N,AN),!.

conc([],X,X) :- !.
conc([A|B],C,[A|D]) :- conc(B,C,D).
```

```
/* SECOND SECTION deals with user interaction facilities */

/* database(N) : requests user for the extra information
          required for method no. N.          */

database :-
     readi(X),
     database(X),
     commend.

database(1).
database(2) :-
     output(" maximum distance : "),
     readri(M),
     assert(max(M)).
database(3) :-
     output(" start of route : "),
     readrl(S),
     assert(start(S)).
database(4) :-
     database(2),
     database(3).

/* choose : allows user to specify required method no. */

choose :-
   output(" type required method no. : "),
   readri(X),
   retractall(method(_)),
   assert(method(X)),commend.

curr_distl(X) :- curr_dist(X),!.

curr_dist(0).

/* THIRD SECTION deals with the expert system
   explanation facility */

/* retpred (& retpar) access the trace of
   heuristic calls */

 retpar(A,B,C,D) :-
          retract(tparams( A,B,C,D)),!.
retpred(X,A,B,C,D) :-
          retract(pred(X)),retpar(A,B,C,D).

/* how predicate executes explanation facility
   using the trace of heuristic calls */

how :-
     retpred(X,A,B,C,D),
        text(X,A,B,C,D),
     output(" and then..."),nlr,
     readri(Y),
     fail.
how :- output(" I did a depth first AND-OR tree search "),
     nlr,rr,commend.

/* explanation texts associated with each of the
   heuristics */
```

```
text(intel2,A,B,C,D) :-output("I tried heuristic 1"),nlr.
text(intel3,A,B,C,D) :-output("I tried heuristic 2"),nlr.
text(intel1,A,B,C,D) :-output("I tried heuristic 3"),nlr.
text(intel4,A,B,C,D) :-output("I tried heuristic 4"),nlr.
text(intel5,A,B,C,D) :-output("I tried heuristic 5"),nlr.


/* enquiry_save(P,A,B,C,D) adds parameters of newly
   envoked heuristic P to the trace */

enquiry_save(P,A,B,C,D) :-
          assertz(pred(P)),
          assertz(tparams(A,B,C,D)),!.

/* FOURTH SECTION deals with search mechanism and
   heuristics of the expert system        */

/* intel(S,N,P) : controls expert process, ie.

   1. process the problem parameters
   2. envoke the 'sh' predicate.          */

intel :-
      reada(S),
      comlrec(N),
      readi(Q),
      intelo(S,N,P,Q),
      commend,
      comlsend(P).

intelo(S,N,P,Q) :-
      return_no(NO),
      Q>=NO,
      intel(S,[a],P1),rr,
      enquiry_save(intel4,S,N,Q,P1),
      difference(P1,N,NP1),
      intel(a,NP1,P2),
      ((P2==[],P=P1);
       (P2=[_|P3],conc(P1,P3,P))).
intelo(S,N,P,Q) :-
      enquiry_save(intel5,S,N,Q,0),
      intel(S,N,P).

intel(_,[],[]).
intel(S,N,P) :-
           intel2(S,N,P).
intel(S,N,P) :-
           member(M,N),difference([M],N,NM),
           sh(S,NM,M,P),
           enquiry_save(intel1,S,NM,M,P).

intel2(Z,N1,P ) :-
      convass(N1,Z,N,Y),
      member(M,N),M<Y,sthan1(Y,N,N2),max(N2,E),difference([E]
,N,NE),
      assconv(NE,E,ANE,AE),!,
      enquiry_save(intel2,Z,ANE,AE,P),
      sh(Z,ANE,AE,P).
intel2(S,N1,P ) :-
      convass(N1,S,N,Y),
      max(N,E),difference([E],N,NE),
      assconv(NE,E,ANE,AE),!,
      enquiry_save(intel3,S,ANE,AE,P),
      sh(S,ANE,AE,P).
```

```
/* sh (s_route,retr,distance,path,pat_t,check_distance,end)
:
   envoke the required search method to find the best route.
    Method envoked is coded 1 to 4 as the first parameter of
   pat_t :

   1 = normal exhaustive depth first search
   2 = exhaustive depth first search with max. distance
       cut off point
   3 = exhaustive depth first search with specified start
   4 = exhaustive depth first search with specified start
       and max. distance.                              */

sh(Source,_,Dest,_):-
    path(Source,Dest,Path),
    distance(Path,Dist),
    assert(route(Path,Dist)),
    fail.

sh(_,List,_,_):-
    route(Path,_),
    member(M,List),
    not member(M,Path),
    retract(route(Path,_)),
    fail.

sh(_,List,_,_):-
    s_route(X,D),
    D<X,
    retr(D),
    fail.

sh(_,List,_,Path):-
    smallest_dist(X),
    routel(Path,X).


s_route(X,D) :-
    retract(route(N,D)),
    assert(routel(N,D)),
    smallest_dist(X).
```

```
retr(D):- retract(smallest_dist(_)),
    asserta(smallest_dist(D)),!.

distance([_|[]],0).
```

```
distance([X,Y|Z],Dist):-
    arc(X,Y,D1),
    distance([Y|Z],D2),
    Dist is D1+D2.


path(Source,Dest,Path):-
    method(X),
    pat_t(X,Source,Dest,Path,[]).

pat_t(1,Point,Point,[Point],_).
pat_t(1,New_source,Dest,[New_source|Rest],Points_so_far) :-
    arc(New_source,Next,_),
    not member(Next,Points_so_far),
```

```
    pat_t(1,Next,Dest,Rest,[Next|Points_so_far]).

pat_t(2,Point,Point,[Point],_).
pat_t(2,New_source,Dest,[New_source|Rest],Points_so_far) :-
    arc(New_source,Next,D),
    not(member(Next,Points_so_far)),
    curr_dist1(CD),CDD is CD+D,
    asserta(curr_dist(CDD)),
    check_distance(CDD),
      pat_t(2,Next,Dest,Rest,[Next|Points_so_far]).

pat_t(3,Source,Dest,Path,LL) :-
    start(L),
    end(L,End),
    pat_t(1,End,Dest,[_|Path1],LL),
    conc(L,Path1,Path).

pat_t(4,Source,Dest,Path,LL) :-
    start(L),
    end(L,End),
    pat_t(2,End,Dest,[_|Path1],LL),
    conc(L,Path1,Path).

end([X|[]],X) :-!.
end([_|L],X) :- end(L,X).

check_distance(CDD) :-
    max(M),
    CDD=<M.
check_distance(_) :-
    retract(curr_dist(_)),!,fail.

/* FIFTH SECTION deals with initial conditions & reseting */

/* reset system after a consultation */
```

```
rr:- retractall(routel(_,_)),
     retractall(route(_,_)),
     retractall(max(_)),
     retractall(method(_)),assert(method(1)),
     retractall(start(_)),
     retractall(curr_dist(_)),assert(curr_dist(0)),

retractall(smallest_dist(_)),asserta(smallest_dist(16383)),
     retractall(tparams(_,_,_,_)),retractall(pred(_)).

r :-rr,commend,!.

method :- method(Z), chm(Z), commend, writeri(Z).

chm(5) :-
     retract(method(_)),assert(method(1)).
chm(_).

/* default method is no. 1. */

smallest_dist(16383).
method(1).
return_no(4).
```

```
/* PROBLEM SPECIFIC map information */

arc(a,b,7).
arc(b,c,10).
arc(b,k,5).
arc(c,d,4).
arc(d,e,6).
arc(d,k,10).
arc(e,f,3).
arc(f,g,14).
arc(g,k,4).
arc(g,h,11).
arc(h,i,3).
arc(i,j,5).
arc(j,a,4).
arc(k,j,7).
arc(k,h,13).


?-reconsult(prlink3).
```

```
(*    CALCEVENTS :
      ----------

      this writes PROLOG f_event database (for use by
simvel.pro)
      in file PASPRO
*)

PROGRAM CALCEVENTS(INPUT,PASPRO,OUTPUT) ;

(*    EVENTNAME is the type whose enumerated values are the
      possible events.                                    *)

TYPE
      EVENTNAME = (B,C,D,E,F,G,H,I,J,K) ;

VAR
      PASPRO : TEXT ;
      EVENT  : EVENTNAME;
      TIMES : INTEGER;
      RANSEED,LENGTH,MEAN : REAL;

(*    WRITENAMES : write next event/time (demand) to PASPRO  *
)

PROCEDURE WRITENAMES(EV:EVENTNAME; TIME:INTEGER);
BEGIN
      WRITE(PASPRO,'f_event(');
      CASE EV OF
            B:    WRITE(PASPRO,'b');
            C:  WRITE(PASPRO,'c');
            D:    WRITE(PASPRO,'d');
            E:    WRITE(PASPRO,'e');
            F:    WRITE(PASPRO,'f');
            G:    WRITE(PASPRO,'g');
            H:    WRITE(PASPRO,'h');
            I:    WRITE(PASPRO,'i');
            J:  WRITE(PASPRO,'j');
            K:    WRITE(PASPRO,'k')
      END;
      WRITE(PASPRO,',');
      WRITELN(PASPRO,TIME:3,').')
END;

(*    RANDNO : random number generator    *)

FUNCTION RANDNO(X:REAL) : REAL;
CONST
      A = 123.0 ;
      M = 256.0 ;
VAR
      R,R1 : REAL;
      RF : INTEGER;
BEGIN
      R1 := A*X;
      R := R1/M;
      RF := TRUNC(R);
      R := RF*M;
      RANDNO := R1-R
END;
```

```
(*    CONVEV : convert a random number into a
      random event (called by FEVENT)          *)

PROCEDURE CONVEV(RAN:REAL; VAR EVN:EVENTNAME);
BEGIN
      IF RAN <= 0.1 THEN EVN :=B ELSE
      IF RAN <= 0.2 THEN EVN :=C ELSE
      IF RAN <= 0.3 THEN EVN :=D ELSE
      IF RAN <= 0.4 THEN EVN :=E ELSE
      IF RAN <= 0.5 THEN EVN :=F ELSE
      IF RAN <= 0.6 THEN EVN :=G ELSE
      IF RAN <= 0.7 THEN EVN :=H ELSE
      IF RAN <= 0.8 THEN EVN :=I ELSE
      IF RAN <= 0.9 THEN EVN :=J ELSE
      EVN := K
END;

(*    FEVENT : calculate next random event/time (demand)
request    *)

PROCEDURE FEVENT(VAR RA1:REAL; VAR EVE:EVENTNAME; VAR
TIM1:INTEGER);
CONST
      M = 256.0;
VAR
      RA2,RA3,RA4,TIM2,TIM3 : REAL;
BEGIN
      RA2 := RANDNO(RA1);
      RA1 := RANDNO(RA2);
      RA3 := RA1/M;
      RA4 := RA2/M;
      CONVEV(RA4,EVE);
      TIM2 := MEAN*LN(1/RA3);
      TIM3 := TIM1+TIM2;
      TIM1 := TRUNC(TIM3)
END;


(*    MAIN PROGRAM    *)
(*    ---- -------    *)

BEGIN

      REWRITE(PASPRO);
      TIMES := 0;

(*    read in problem data :
      1. random number seed.
      2. interarrival time (time between arrivals should
                            follow negative-exponential dist.,
                            because we assume a poisson stream).
      3. simulation length.                                  *)
```

```
WRITE(OUTPUT,'TYPE RANDOM SEED : ');
READ(RANSEED);
WRITE(OUTPUT,'TYPE MEAN ARRIVAL TIME : ');
READ(MEAN);
WRITE(OUTPUT,'TYPE LENGTH OF SIMULATION : ');
READ(LENGTH);

(*   write the 'start of simulation' event.   *)

     WRITELN(PASPRO,'f_event(a,0).');

(*   WHILE simulation time has not expired
         calculate next demand and add to database.     *)

     WHILE TIMES<LENGTH DO
          BEGIN
           FEVENT(RANSEED,EVENT,TIMES);
           WRITENAMES(EVENT,TIMES)
          END;
     WRITELN(PASPRO)
END.
```

APPENDIX 8 - LISTING OF THE FORTRAN LINKING ROUTINES

```
C
C
C THIS SECTION CONTAINS THE FORTRAN SUBROUTINES NEEDED FOR
C INTERFACING THE SIMULATION WITH THE ASSEMBLER
C SIMULATION-EXPERT INTERFACE
C
      SUBROUTINE WRRA
C
      INTEGER INTG2
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1)
,CHAR(17)),
     1    (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      OP=7
      CALL PRLINK
      RETURN
      END
C
C
      SUBROUTINE WRRI
C
      INTEGER INTG2
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),
CHAR(17)),
     1    (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      OP = 8
      CALL PRLINK
      RETURN
      END
C
C
      SUBROUTINE RDA
C
      INTEGER INTG2
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1)
,CHAR(17)),
     1    (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      OP=12
      CALL PRLINK
      RETURN
      END
C
C
      SUBROUTINE RDI
C
      INTEGER INTG2
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1)
,CHAR(17)),
     1    (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      OP=15
```

```
      RETURN
      END
C
C
      SUBROUTINE CEND
C
      INTEGER INTG2
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1)
,CHAR(17)),
     1     (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      OP=1
      CALL PRLINK
      RETURN
      END
C
C
      SUBROUTINE CSEND
C
      INTEGER INTG2
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      INTEGER*1 INTP(20),ATM1(20),ATM2(20),ATM3(20)
      COMMON/PAR/INTP,ATM1,ATM2,ATM3
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1)
,CHAR(17)),
     1     (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      OP=7
      CALL PRLINK
      CALL RPRAMS
123   CALL CLRC
      OP=11
      CALL PRLINK
      IF (OINTG.NE.50) GOTO 990
      CALL CLRC
      CALL RERQ
      GOTO 123
990   RETURN
      END
C
C
      SUBROUTINE ENDS
C
      INTEGER INTG2
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1)
,CHAR(17)),
     1     (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      OP=16
      CALL PRLINK
      RETURN
      END
C
C
      SUBROUTINE CLRC
C
      INTEGER INTG2
```

```
      INTEGER*1 OP, INTG,ELIST,ATOM(15),CHAR(120),OINTG
      COMMON/ATA/CHAR
      EQUIVALENCE (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1)
,CHAR(17)),
     1     (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      DO 10 I=1,120
10    CHAR(I) = 0
      RETURN
      END
C
C
C THE NEXT SET OF SUBROUTINES ALLOW PARAMETERS TO BE SENT TO
C THE EXPERT
C
C
C FATM(N,T) FILLS THE ATOM PARAMETER ARRAY ATMn
(n=N=1,...,3)
C WITH THE ATOM (TEXT STRING) T
C
      SUBROUTINE FATM(N,T)
C
      INTEGER*1 N,T(20),I
      INTEGER INTG2
      INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120),OINTG
      INTEGER*1 INTP(20),ATM1(20),ATM2(20),ATM3(20)
      COMMON/PAR/INTP,ATM1,ATM2,ATM3
      COMMON/ ATA/CHAR
      EQUIVALENCE
(OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
     1  (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      GOTO (101,102,103),N
C
 101 DO 100 I=1,20
      ATM1(I) = T(I)
 100 CONTINUE
      GOTO 990
 102 DO 200 I=1,20
      ATM2(I) = T(I)
 200 CONTINUE
      GOTO 990
 103 DO 300 I=1,20
      ATM3(I) = T(I)
 300 CONTINUE
 990 RETURN
      END
C
C FCHAR FILLS THE COMMON CHAR ARRAY WITH AN ATOM READY FOR
INPUT
C TO THE REMOTE EXPERT
C
      SUBROUTINE FCHAR(N)
C
      INTEGER*1 I,N
      INTEGER INTG2
      INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120),OINTG
      INTEGER*1 INTP(20),ATM1(20),ATM2(20),ATM3(20)
      COMMON/PAR/INTP,ATM1,ATM2,ATM3
      COMMON/ ATA/CHAR
      EQUIVALENCE
(OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
     1  (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
```

```
      GOTO (101,201,301),N
C
 101 DO 100 I=1,20
      CHAR(I+1) = ATM1(I)
 100 CONTINUE
      GOTO 990
 201 DO 200 I=1,20
      CHAR(I+1) = ATM2(I)
 200 CONTINUE
      GOTO 990
 301 DO 300 I=1,20
      CHAR(I+1) = ATM3(I)
 300 CONTINUE
 990 RETURN
      END
C
C TATOM IS USED BY THE PROBLEM DEPENDANT RPRAMS TO FIND
WHICH COMMAND
C HAS JUST BEEN REQUESTED (IN ORDER TO DISCOVER WHAT
PARAMETERS
C NEED TO BE SENT
C
      SUBROUTINE TATOM(M,FLG)
C
      INTEGER*1 I,M(20),FLG,SEMIC,II
      INTEGER INTG2
      INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120),OINTG
      INTEGER*1 INTP(20),ATM1(20),ATM2(20),ATM3(20)
      COMMON/PAR/INTP,ATM1,ATM2,ATM3
      COMMON/ ATA/CHAR
      EQUIVALENCE
 (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
     1  (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      SEMIC = 59
C
      DO 100 I=1,20
      II=I+1
      IF (M(I).EQ.SEMIC) GOTO 200
      IF (M(I).NE.CHAR(II)) GOTO 300
      IF (I.EQ.20) GOTO 300
  100 CONTINUE
C THE ATOM HAS BEEN FOUND. MARK THE FLAG
  200 FLG = 1
      GOTO 400
C THE ATOM HAS NOT BEEN FOUND
  300 FLG = 0
  400 RETURN
      END
C
C RERQ SENDS A PROLOG STYLE LIST FROM THE SCREEN AND SENDS
IT TO THE REMOTE
C EXPERT (EQUIV. TO THE PROLOG remote_read_request)
C
      SUBROUTINE RERQ
C
      INTEGER*1
COMMA,RLST(80),BRAC,IFLG,II,IL,J,JJ,K,NUM,NB,SEMIC
      INTEGER INTG2
      INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120)
      INTEGER*1 INTP(20),ATM1(20),ATM2(20),ATM3(20)
      COMMON/PAR/INTP,ATM1,ATM2,ATM3
      COMMON/ ATA/CHAR
      EQUIVALENCE
 (OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
```

```
C
C READ IN THE LIST
      READ(5,100) RLST
  100 FORMAT(80A1)
C SEND THE LIST
      CALL LSND(RLST)
      RETURN
      END
C
C
C THIS SUBROUTINE SENDS A LIST (REPRESENTED IN CHARACTER
FORM IN RLST)
C TO THE REMOTE PROLOG EXPERT
C
      SUBROUTINE LSND(RLST)
C
      INTEGER*1
COMMA,RLST(80),BRAC,IFLG,II,IL,J,JJ,K,NUM,NB,SEMIC
      INTEGER INTG2
      INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120)
      INTEGER*1 INTP(20),ATM1(20),ATM2(20),ATM3(20)
      COMMON/PAR/INTP,ATM1,ATM2,ATM3
      COMMON/ ATA/CHAR
      EQUIVALENCE
(OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
     1  (ELIST,CHAR(2)),
     2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
C
      COMMA = 44
      BRAC = 93
      SEMIC = 59
      IFLG = 1
C PTR TO START OF NEXT LIST ELEMENT
      II = 2
C REVERSE LIST ORDER
      CALL REVL(RLST)
C
C
      DO 200 I=2,79
      IF (RLST(I).EQ.COMMA) GOTO 300
      IF (RLST(I).EQ.BRAC) GOTO 300
      IF (RLST(I).GT.57) IFLG = 0
      IF (RLST(I).LT.48) IFLG = 0
      GOTO 201
C
C END OF LIST ELEMENT
  300 IL = I-1
C TEST FOR INTEGER
      IF (IFLG.EQ.1) GOTO 400
      JJ = 2
      DO 350 J=II,IL
      CHAR(JJ) = RLST(J)
      JJ = JJ+1
  350 CONTINUE
      CHAR(JJ) = SEMIC
      CALL WRRA
      CALL CLRC
      IFLG = 1
      GOTO 199
C INTEGER
  400 NB = IL-II+1
      NUM = 0
      DO 360 K=II,IL
  360 RLST(K) = RLST(K)-48
```

```
      GOTO (361,362,363),NB
  363 NUM = NUM + (RLST(II)*100)
      II = II+1
  362 NUM = NUM + (RLST(II)*10)
      II = II+1
  361 NUM = NUM + RLST(II)
      INTG = NUM
      CALL WRRI
      CALL CLRC
      IFLG = 1
  199 II = I+1
      IF (RLST(I).EQ.BRAC) GOTO 990
  201 CONTINUE
  200 CONTINUE
C
C
  990 OP = 10
      CALL PRLINK
      RETURN
      END
C
C
C THIS SUBROUTINE IS USED BY RERQ TO REVERSE A PROLOG LIST
C SO THAT ITS ELEMENTS ARE SENT IN THE CORRECT ORDER
C
      SUBROUTINE REVL(LR)
C
      INTEGER*1
LR(80),RL(80),LBRAC,RBRAC,COMMA,COUNT,J,JB,I,JBI,II,P
C
      LBRAC = 91
      RBRAC = 93
      COMMA = 44
C COUNT SIZE OF THE INPUT LIST
      DO 100 I=1,80
      IF (LR(I).EQ.RBRAC) GOTO 110
  100 CONTINUE
  110 COUNT = I
C
      JB = 2
C FIND THE NEXT RIGHT MOST COMMA
      II = 0
      DO 200 I=1,COUNT
      IF (LR(COUNT-I).EQ.COMMA) GOTO 250
      IF (LR(COUNT-I).EQ.LBRAC) GOTO 250
      II = II+1
      GOTO 290

  250 CONTINUE
      JBI = JB+II-1
      P = 0
      DO 300 J=JB,JBI
      P = P+1
      RL(J) = LR(COUNT-I+P)
  300 CONTINUE
      RL(JB+II) = COMMA
      JB = JB+II+1
      II = 0
  290 CONTINUE
  200 CONTINUE
C INSERT BRACKETS
      RL(1) = LBRAC
      RL(COUNT) = RBRAC
C COPY BACK REVERSED LIST INTO LR
```

```
      DO 400 I=1,80
      LR(I) = RL(I)
  400 CONTINUE
      RETURN
      END
C
C
C THIS SUBROUTINE TRANSLATES THE FORTRAN REPRESENTATION
C OF A PROLOG LIST INTO A (PROLOG) CHARACTER REPRESENTATION
C OF THE LIST
C
      SUBROUTINE TRANL(CPL)
C
      INTEGER*1 CPL(80),COMMA,IM,I,TEMP,ICH,ICPL,ENDL
      INTEGER*1 LBRAC,RBRAC,SEMIC
      INTEGER*1 PLIST(50),PLAT(30,18)
      COMMON/PROL/PLIST,PLAT
C
      LBRAC = 91
      RBRAC = 93
      SEMIC = 59
      ENDL = -100
      COMMA = 44
      CPL(1) = LBRAC
      ICPL = 2
C
C
      DO 100 I=1,50
      IF (I.EQ.50) GOTO 900
      IF (PLIST(I).EQ.ENDL) GOTO 900
C TEST FOR INTEGER VS. ATOM
      IF (PLIST(I).LT.0) GOTO 200
C INTEGER
      IM = 1
      IF (PLIST(I).GE.10) IM = IM+1
      IF  (PLIST(I).GE.100) IM = IM+1
      TEMP = PLIST(I)
      GOTO (110,120,130),IM
  130 ICH = TEMP/100
      TEMP = TEMP - (ICH*100)
      CPL(ICPL) = ICH+48
      ICPL = ICPL+1
  120 ICH = TEMP/10
      TEMP = TEMP - (ICH*10)
      CPL(ICPL) = ICH+48
      ICPL = ICPL+1
  110 CPL(ICPL) = TEMP+48
      CPL(ICPL+1) = COMMA
      ICPL = ICPL+2
      GOTO 90
C ATOM
  200 TEMP = -PLIST(I)
      DO 150 J=1,18
      IF (PLAT(TEMP,J).EQ.SEMIC) GOTO 160
      CPL(ICPL) = PLAT(TEMP,J)
      ICPL = ICPL+1
  150 CONTINUE
  160 CPL(ICPL) = COMMA
      ICPL = ICPL+1
  90  CONTINUE
  100 CONTINUE
C
```

```
 900 CPL(ICPL-1) = RBRAC
     RETURN
     END
C
C
C THIS SUBROUTINE SENDS A LIST (IN THE FORTRAN
REPRESENTATION)
C TO THE REMOTE EXPERT (IN THE PROLOG REPRESENTATION)
C
     SUBROUTINE LSEND
C
     INTEGER*1 LST(80)
     INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120)
     COMMON/ ATA/CHAR
     EQUIVALENCE
(OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
    1  (ELIST,CHAR(2)),
    2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
     CALL TRANL(LST)
     OP = 9
     CALL PRLINK
     CALL CLRC
     CALL LSND(LST)
     RETURN
     END
C
C
C THIS SUBROUTINE RECEIVES A LIST FROM THE REMOTE EXPERT
C AND STORES IT IN THE MANNER APPROPRIATE FOR THE FORTRAN
C USE OF PROLOG LISTS (THIS ROUTINE IS EQUIVALENT
C TO PROLOGS comlrec )
C
     SUBROUTINE RDL
C
     INTEGER*1 PLIPTR,PLAPTR,PLCPTR,LEND,SEMIC,ATPTR
     INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120)
     INTEGER*1 PLIST(50),PLAT(30,18)
     COMMON/PROL/PLIST,PLAT
     COMMON/ ATA/CHAR
     EQUIVALENCE
(OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
    1  (ELIST,CHAR(2)),
    2  (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
C POINTERS FOR FORTRAN LIST DATA STRUCTURES
     PLIPTR = 1
     PLAPTR =1
     PLCPTR = 1
     LEND = -100
     SEMIC = 59
C
     CALL CLRC
     OP = 11
     CALL PRLINK
C
C READ IN NEXT ELEMENT OF LIST
 10  CALL CLRC
     OP = 13
     CALL PRLINK
C TEST FOR END OF LIST
     IF (CHAR(17).LT.0) GOTO 400
C TEST FOR ATOM
     IF (CHAR(18).NE.0) GOTO 300
C INTEGER
```

```
      PLIST(PLIPTR) = CHAR(17)
      PLIPTR = PLIPTR+1
      GOTO 10
C ATOM
  300 ATPTR = 1
  310 IF (ATOM(ATPTR).EQ.SEMIC) GOTO 350
      PLAT(PLAPTR,PLCPTR) = ATOM(ATPTR)
      PLCPTR = PLCPTR+1
      ATPTR = ATPTR+1
      GOTO 310
  350 PLAT(PLAPTR,PLCPTR) = ATOM(ATPTR)
      PLCPTR = 1
      PLIST(PLIPTR) = -PLAPTR
      PLAPTR = PLAPTR+1
      PLIPTR = PLIPTR+1
      GOTO 10
C END OF LIST
  400 continue
      PLIST(PLIPTR) = LEND

      RETURN
      END
C
C
C THIS SUBROUTINE CAUSES A COMMAND TO BE PLACED IN THE
C COMMON CHAR ARRAY WHEN I=IMESS. IT IS CALLED BY THE
C PROBLEM DEPENDANT CFILL SUBROUTINE
C
      SUBROUTINE STORE(IMESS,I,M)
C
      INTEGER*2 I,IMESS
      INTEGER*1 M(40),SEMIC,CHAR(120),J,JJ
      COMMON/ATA/CHAR
C
      SEMIC = 59
C
      IF (IMESS.NE.I) GOTO 200
      DO 100 J=1,40
      JJ = J+1
      CHAR(JJ) = M(J)
      IF (M(J).EQ.SEMIC) GOTO 200
  100 CONTINUE
C
  200 RETURN
      END
C
C
C STORES AN ATOM IN THE LIST ARRAY PLAT WHEN I=IMESS
C P IS THE POINTER TO THE PLAT ARRAY
C
      SUBROUTINE STOREL(IMESS,I,M,P)
C
      INTEGER*2 I,IMESS,P
      INTEGER*1 M(40),SEMIC,PLIST(50),PLAT(30,18),J
      COMMON/PROL/PLIST,PLAT
C
      SEMIC = 59
C
      IF (IMESS.NE.I) GOTO 200
      DO 100 J=1,40
      PLAT(P,J) = M(J)
      IF (M(J).EQ.SEMIC) GOTO 200
  100 CONTINUE
```

```
C
 200 RETURN
     END
C
C
     subroutine incomm
     include 'lsim'
c
     if (ipo.eq.0) goto 15
     ipo = 1
     op = 17
     call prlink
     call clrc
15   continue
     return
     end
c
c
```

```
      NAME LNK
CGROUP     GROUP      CODE
      PUBLIC     prlink
DGROUP     GROUP      ATA,KINSKI
      INCLUDE    DAT.ASM
CODE SEGMENT     BYTE PUBLIC     'CODE'
      ASSUME     CS:CGROUP,DS:DGROUP

init macro
      mov  ax,0e3h          ;set up for 9600 baud,1 stop bit,no
      int  14h         ;parity, 8-bit words
      mov  dx,mctl          ;point at modem control register
      mov  al,3        ;force DTR_ and RTS_ low
      out  dx,al
      endm

cstl macro
      push dx
      mov  ah,06
      mov  dl,0ffh
      int  21h
      pop  dx
      endm

prm  macro      p1,p2,p3
      cmp  al,p1
      jb   p2
      push ax
      mov  ah,06
      mov  dl,p3
      int  21h
      pop  ax
      sub  al,p1
      jmp  pr10
      endm

mchar      macro      source
      mov  al,source
      cmp  al,";"           ;have we reached end of atom ?
      je   enat        ;yes - store character count
      inc  count            ;no - save this character
      mov  [bx],al
      inc  bx
           endm

mchar2     macro      source
      mov  al,source
      cmp  al,";"           ;have we reached end of atom ?
      je   enat2            ;yes - store character count
      inc  count            ;no - save this character
      mov  [bx],al
      inc  bx
           endm

mrch macro      dest,dl
      cmp  ax,count  ;have we transfered all the characters ?
      je   comm16           ;yes - end of atom
      inc  count            ;no - transfer the next character
      push dx
      mov  dl,[bx]
      mov  dest,dl
```

```
        mov   dl,59
        pop   dx
        inc   bx
        endm

mrch2   macro   dest,dl
        cmp   ax,count  ;have we transfered all the characters ?
        je    comm18        ;yes - end of atom
        inc   count         ;no - transfer the next character
        push  dx
        mov   dl,[bx]
        mov   dest,dl
        mov   dl,59
        pop   dx
        inc   bx
        endm

mrch3   macro   dest,dl
        cmp   ax,count  ;have we transfered all the characters ?
        je    comm21        ;yes - end of atom
        inc   count         ;no - transfer the next character
        push  dx
        mov   dl,[bx]
        mov   dest,dl
        mov   dl,59
        pop   dx
        inc   bx
        endm

mrch4   macro   dest,dl
        cmp   ax,count  ;have we transfered all the characters ?
        je    tcl315        ;yes - end of atom
        inc   count         ;no - transfer the next character
        push  dx
        mov   dl,[bx]
        mov   dest,dl
        mov   dl,59
        pop   dx
        inc   bx
        endm

mrch5   macro   dest,dl
        cmp   ax,count  ;have we transfered all the characters ?
        je    tcl316        ;yes - end of atom
        inc   count         ;no - transfer the next character
        push  dx
        mov   dl,[bx]
        mov   dest,dl
        mov   dl,59
        pop   dx
        inc   bx
        endm

mrch6   macro   dest,dl
        cmp   ax,count  ;have we transfered all the characters ?
        je    tcl313        ;yes -  end of atom
        inc   count         ;no - transfer the next character
        push  dx
        mov   dl,[bx]
        mov   dest,dl
        mov   dl,59
        pop   dx
```

```
inc  bx
endm

intg equ  0
atom equ  4
black     equ  0
blue equ  1
red  equ  2
white     equ  3
commprt   equ  3f8h
mctl equ  3fch
istat     equ  mctl+1

prlink:   nop

          push bp
     push dx
     push ax
     mov  al,char
     mov  op,al              ;load external code no in  p
     cmp  op,3         ;do we have an input character ?
     jne  tstintg            ;no - test for integer
intin:    mov  al,char+1 ;yes - load it into in
     xor  ah,ah
     mov  in,ax
     call tbljmp             ;jump to correct section of code
     pop  ax
     pop  dx
     pop  bp
     ret
tstintg:cmp    op,8         ;do we have an input integer ?
     jne  tstatm             ;no - test for atom
     jmp  intin
;mov ah,char+1 ;yes - load it into in
;mov al,char+2
;mov in,ax
;call     tbljmp             ;jump to correct section of code
;pop bp
;ret
tstatm:   cmp  op,7          ;have we an input atom ?
     je   tsll
     call tbljmp             ;o - jump to correct section of
code
     pop  ax
     pop  dx
     pop  bp
     ret
                        ;yes - store in standard (indirect) way
tsll:     push count              ;save registers
     push ax
     push bx
     push di
     mov  count,1
     mov  bx,offset atombf ;point to atom storage area
     mov  in,bx
     inc  bx
     mchar2    char+1
     mchar2    char+2
     mchar2    char+3
     mchar2    char+4
     jmp  en3
enat2:    jmp  enat
```

```
en3:  mchar2      char+5
      mchar2      char+6
      mchar2      char+7
      mchar       char+8
      mchar       char+9
      mchar       char+10
      mchar       char+11
      mchar       char+12
      mchar       char+13
      mchar       char+14
      mchar       char+15
      mchar       char+16
enat:       mov   bx,offset atombf ; end of  atom, store
character count
      mov   ax,count
      mov   [bx],al
      pop   di          ;restore registers, etc.
      pop   bx
      pop   ax
      pop   count
      call  tbljmp
tet:  pop   ax
      pop   dx
      pop   bp
      ret

atombf:     db    16 dup (?)

area:       db    256 dup (?)

cmpm macro      a,b
      cmp   op,a
      je    b
      endm

tbljmp      proc near

            cmpm 0,zxtc0
      cmpm 1,zxtc1
      cmpm 2,zxtc2
      cmpm 3,zxtc3
      cmpm 4,zxtc4
      cmpm 5,zxtc5
      cmpm 6,zxtc6
      cmpm 7,zxtc7
      cmpm 8,zxtc8
      cmpm 9,zxtc9
      cmpm 10,zxtc10
      cmpm 11,zommpg
      cmpm 12,zommp3
      cmpm 13,zxtc13
      cmpm 14,zxtc14
      cmpm 15,zxtc15
      cmpm 16,zxtc16
      cmpm 17,zxtc17

zxtc0:      jmp   extc0
zxtc1:      jmp   extc1
zxtc3:      jmp   extc3

zxtc2:      jmp   extc2
zxtc4:      jmp   extc4
```

```
zxtc5:     jmp  extc5
zxtc6:     jmp  extc6
zxtc7:     jmp  extc7
zxtc8:     jmp  extc8
zxtc9:     jmp  extc9
zxtcl0:    jmp  extcl0
zommpg:    jmp  commpg
zommp3:    jmp  commp3
zxtcl3:    jmp  extcl3
zxtcl4:    jmp  extcl4
zxtcl5:    jmp  extcl5
zxtcl6:    jmp  extcl6
zxtcl7:    jmp  extcl7
→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→
```

APPENDIX 10 - LISTINGS OF THE QUEUE SIMULATION AND 'EXPERT'

The Queue Simulation

```
      program queue
include 'quesim'
include 'lsim'
call setsys
call vclass(cust,150,'c$',0,14)
call vset(pool,'pool$',1,16,-1,0,0,0)
call vset(qu  ,'arrival-queue$',1,10,15,0,-1,13)
call vset(sv  ,'server-area$',  1,30,22,0,-1,12)
call vset(temp,'exit$',1,40,9,0,1,11)
call ventit(config,'configuration$',3,10)
call vhist(hist,'time_in_system $',5,42,12,31,0,100,20)
c
c
      ipo = 1
      call setatt(config,1,20)
      call setatt(config,2,30)
      call setatt(config,3,2)
      call vload(cust,1,150,pool)
      call schedl(1,5,ihead(pool))
      call recon
      call formsc
c
c
c
10    call advanc(ievent,itime,iele)
      if(ievent.eq.1)call arrive(iele)
      if(ievent.eq.2)call leave(iele)
      if(ievent.eq.999)goto 990
      call starts
      call incomm
15    intp(1) = isize(qu)
      call cfill(1)
      call csend
      call clrc
      call rdi
      call setatt(config,3,ointg)
      goto 10
990   stop
      end
c
c
      subroutine arrive(iele)
      include 'quesim'
      call timeon(iele)
      call movexy(iele,pool,qu)
      rval=iatt(config,1)
      itim=ineg(rval,1)
      call schedl(1,itim,ihead(pool))
      return
      end
c
c
      subroutine starts
      include 'quesim'
10    if(isize(qu).eq.0)goto 990
      if(isize(sv).ge.iatt(config,3))goto 990
      iele=ihead(qu)
      call movexy(iele,qu,sv)
      rval=iatt(config,2)
      itim=ineg(rval,2)
      call schedl(2,itim,iele)
      goto 10
```

```
990   return
      end
c
c
      subroutine leave(iele)
      include 'quesim'
      call record(iele,hist)
      call movexy(iele,sv,temp)
      call movexy(iele,temp,pool)
      return
      end
c
c
      subroutine formti(itime)
      call iform(1,28,0,itime,26)
      return
      end
c
c
      subroutine formsc
      include 'quesim'
      call tform(1,22,0,'time=$',27)
      call fill(1,11,1,15,10,32)
      return
      end
c
c
      subroutine ownint
      include 'quesim'
      call clear
      do 5 i=1,20
5     call lsnoff(i)
      call lsnon(5)
      call fill(5,8,2,60,22,32)
      call rect(5,7,1,61,23,96)
      call tform(5,10,4,'MENU OF SIMULATION OPTIONS  $',12)
      call tform(5,10,6,'0:CONTINUE THE SIMULATION   $',12)
      call tform(5,10,7,'1:END SIMULATION            $',12)
      call sform(5,10,20,24)
      call inputi(5,10,20,'WHAT OPTION PLEASE ?$',9,1)
      if (1.le.0) goto 990
      if (1.gt.1) goto 990
      call clrc
      call ends
      stop
990   call lsnoff(5)
      call lsnon(1)
      call reform
      return
      end
C
C
C RPRAMS IS A PROBLEM DEPENDANT SUBROUTINE WHICH SENDS OUT
TO THE EXPERT
C THE PARAMETERS IT NEEDS
C
      SUBROUTINE RPRAMS
C
      INTEGER INTG2
      INTEGER*1 OP,INTG,ELIST,ATOM(15),CHAR(120),OINTG
      INTEGER*1 INTP(20),ATM1(20),ATM2(20),ATM3(20),FLG
      COMMON/PAR/INTP,ATM1,ATM2,ATM3
      COMMON/ ATA/CHAR
```

```
      EQUIVALENCE
(OP,CHAR(1)),(INTG,CHAR(2)),(ATOM(1),CHAR(17)),
     1   (ELIST,CHAR(2)),
     2   (INTG2,CHAR(2)),(OINTG,CHAR(17))
C
      CALL TATOM('control;',FLG)
      IF (FLG.EQ.1) GOTO 10
      GOTO 990
C SEND THE PARAMETERS FOR control
 10   INTG = INTP(1)
      CALL WRRI
 990  RETURN
      END
C
C
C CFILL IS A PROBLEM DEPENDANT SUBROUTINE THAT CAUSES THE
C COMMON CHAR ARRAY TO BE FILLED WITH THE COMMAND WHOSE
C INDEX IS IMESS
C
      SUBROUTINE CFILL(IMESS)
C
      INTEGER*2 IMESS
C
      CALL STORE(IMESS,1,'control;')
      RETURN
      END
```

The Queue 'Expert'

```
noservs(5).

control :-
     readi(I),
     control(I,J),
     commend,
     writeri(J).

control(I,J) :- I>4, J=5,chnservs(J).
control(I,J) :- I>3, J=4,chnservs(J).
control(I,J) :- I>2, J=3,chnservs(J).
control(I,J) :- I>1, J=2,chnservs(J).
control(I,J) :- I>0, J=1,chnservs(J).
control(I,1) :- chnservs(1).

chnservs(J) :- retract(noservs(_)),
          assert(noservs(J)).

?-consult(prlink4).
```

APPENDIX 11 - LISTING OF THE LORRY SIMULATION AND 'EXPERT'

The Lorry Simulation

```
 PROGRAM LORRY
include 'simdef'
include 'lsim'
ipo = 1
call setsys
 CALL VCLASS(MERCH,150,'merch$',1,9)
 CALL VCLASS(NCB,150,'ncb-v$',1,14)
 CALL VCLASS(TRAIN,50,'train$',1,10)
 CALL VENTIT(WPOOL,'wpool$',1,19)
 CALL VENTIT(LPOOL,'lpool$',1,19)
 CALL VENTIT(ARATES,'rates$',7,19)
 CALL VSET(MPOOL,'mpool$',20,1,-5,0,0,19)
 CALL VSET(QWIN,'weighin.q$',1,1,5,0,-1,12)
 CALL VSET(WIN,'weigh*$',1,10,10,0,-1,13)
 CALL VSET(QMERL,'lorry.q$',1,46,17,0,-1,14)
 CALL VSET(MLOAD,'loader$',1,55,23,0,-1,15)
 CALL VSET(QWOUT,'weighout-qu$',1,22,15,0,1,16)
 CALL VSET(WOUT,'bridge$',1,16,10,0,1,30)
 CALL VSET(NPOOL,'npool$',20,1,-5,0,0,19)
 CALL VSET(OWORK,'owork$',1,34,20,0,-1,29)
 CALL VSET(TPOOL,'tpool$',20,72,-5,0,0,19)
 CALL VSET(QTRAL,'train.q$',1,70,17,0,-1,28)
 CALL VSET(TLOAD,'/loader$',1,61,23,0,-1,27)
call vset(lout,'lorry$',1,1,21,6,0,30)
call vset(lexit,'lorry$',1,-10,21,0,1,28)
call vset(texit,'train$',1,90,22,-1,0,29)
call setatt(wpool,1,2)
call setatt(lpool,1,2)
call setatt(arates,1,20)
call setatt(arates,2,15)
call setatt(arates,3,50)
call setatt(arates,4,3)
call setatt(arates,5,4)
call setatt(arates,6,10)
call setatt(arates,7,25)
C
C     INITIALISE MODEL
C
      call vload(merch,1,150,mpool)
      call vload(ncb,1,150,npool)
      call vload(train,1,50,tpool)
       CALL SCHEDL(8,10,IHEAD(MPOOL))
       CALL SCHEDL(9,5,IHEAD(NPOOL))
       CALL SCHEDL(7,25,IHEAD(TPOOL))
      CALL FORMSC
C
C      SIMULATION MODEL
C
1000  CALL ADVANC(IEVENT,ITIME,IELE)
      call expint
      if(ievent.eq.999)goto 999
c
       GO TO (1,1,3,3,5,6,7,8,9,10),IEVENT
C
C     WEIGHIN ENDS
C
1     CALL ENDWIN(IELE)
      CALL STMLOA
      CALL STWIN
      CALL STWOUT
      GO TO 1000
c
```

```
c     weighout ends
c
3      CALL ENDOUT(IELE)
       CALL STWIN
       CALL STWOUT
       GO TO 1000
c
c     train unloading ends
c
5      CALL ENDTLO(IELE)
       CALL STTLOA
       CALL STMLOA
       GO TO 1000
c
c     merchant loading ends
c
6      CALL ENDMLO(IELE)
      CALL STTLOA
       CALL STMLOA
       CALL STWOUT
       GO TO 1000
c
c     train arrives
c
7      CALL ARRT(IELE)
       CALL STTLOA
       GO TO 1000
c
c     merchant arrives
c
8      CALL ARRM(IELE)
       CALL STWIN
       GO TO 1000
c
c     coal lorry arrives
c
9      CALL ARRN(IELE)
       CALL STWIN
       GO TO 1000
c
c     coal lorry other work finished
c
10     CALL ENDOW(IELE)
       CALL STWOUT
       GO TO 1000
c
c
999   stop
       END
C
C     END OF MAIN PROGRAM
C
       SUBROUTINE ARRT(IELE)
      include 'simdef'
      call movexv(iele,tpool,qtral)
      call setatt(iele,1,3)
       RVAL=IATT(ARATES,3)
       ITIM=INEG(RVAL,1)
      IF(ISIZE(TPOOL).LE.0)GOTO 990
       CALL SCHEDL(7,ITIM,IHEAD(TPOOL))
990    RETURN
       END
```

```
C
C
      SUBROUTINE ARRM(IELE)
     include 'simdef'
     call movexy(iele,mpool,qwin)
     call setatt(iele,1,2)
      RVAL=IATT(ARATES,2)
      ITIM=INEG(RVAL,2)
     IF(ISIZE(MPOOL).LE.0)GOTO 990
      CALL SCHEDL(8,ITIM,IHEAD(MPOOL))
990   RETURN
      END
C
C
      SUBROUTINE ARRN(IELE)
     include 'simdef'
     call movexy(iele,npool,qwin)
     call setatt(iele,1,1)
      RVAL=IATT(ARATES,1)
      ITIM=INEG(RVAL,3)
     IF(ISIZE(NPOOL).LE.0)GOTO 990
      CALL SCHEDL(9,ITIM,IHEAD(NPOOL))
990   RETURN
      END
C
C
      SUBROUTINE STWIN
     include 'simdef'
10   ILOAD=ISIZE(WIN)+ISIZE(WOUT)
     IF(ILOAD.GE.IATT(WPOOL,1)) GO TO 990
     IF(ISIZE(QWIN).LE.0) GO TO 990
      K=IHEAD(QWIN)
     call movexy(k,qwin,win)
      ITIM=IATT(ARATES,4)
     call schedl(1,ITIM,K)
     goto 10
990   RETURN
      END
C
C
       SUBROUTINE endwin(iele)
     include 'simdef'
     if (iatt(iele,1).eq.2) call movexy(iele,win,qmerl)
     if (iatt(iele,1).eq.1) call stow(iele)
      RETURN
      END
C
C
      SUBROUTINE STOW(iele)
     include 'simdef'
     call movexy(iele,win,owork)
     itim=irand(15,25,4)
     call schedl(10,itim,iele)
      RETURN
      END
c
c
     subroutine endow(iele)
     include 'simdef'
     call movexy(iele,owork,qwout)
     return
     end
```

```
C
C
      SUBROUTINE STWOUT
      include 'simdef'
10    ILOAD=ISIZE(WIN)+ISIZE(WOUT)
      IF(ILOAD.GE.IATT(WPOOL,1)) GO TO 990
      IF(ISIZE(QWOUT).LE.0) GO TO 990
      K=IHEAD(QWOUT)
      call movexy(k,qwout,wout)
      ITIM=IATT(ARATES,5)
      CALL SCHEDL(3,ITIM,K)
      GOTO 10
990   RETURN
      END
C
C
      SUBROUTINE ENDOUT(IELE)
      include 'simdef'
      call movexy(iele,wout,lout)
      call moveyx(iele,lout,lexit)
      IF(IATT(IELE,1).EQ.1)       CALL
moveyx(iele,lexit,npool)
      IF(IATT(IELE,1).EQ.2)       CALL
moveyx(iele,lexit,mpool)
      RETURN
      END
C
C
      SUBROUTINE STMLOA
      include 'simdef'
10    ILOAD=ISIZE(MLOAD)+ISIZE(TLOAD)
      IF(ILOAD.GE.IATT(LPOOL,1)) GO TO 990
      IF(ISIZE(QMERL).LE.0) GO TO 990
      K=IHEAD(QMERL)
      call movexy(k,qmerl,mload)
      RVAL=IATT(ARATES,6)
      ITIM=INEG(RVAL,5)
      CALL SCHEDL(6,ITIM,K)
      GOTO 10
990   RETURN
      END
C
C
      SUBROUTINE ENDMLO(IELE)
      include 'simdef'
      call movexy(iele,mload,qwout)
      RETURN
      END
C
C
      SUBROUTINE STTLOA
      include 'simdef'
10    ILOAD=ISIZE(MLOAD)+ISIZE(TLOAD)
      IF(ILOAD.GE.IATT(LPOOL,1)) GO TO 990
      IF(ISIZE(QTRAL).LE.0) GO TO 990
      K=IHEAD(QTRAL)
      call movexy(k,qtral,tload)
      RVAL=IATT(ARATES,7)
      ITIM=INEG(RVAL,6)
      CALL SCHEDL(5,ITIM,K)
      GOTO 10
990   RETURN
      END
C
```

```
C
      SUBROUTINE ENDTLO(IELE)
     include 'simdef'
     call movexy(iele,tload,texit)
     call moveyx(iele,texit,tpool)
      RETURN
      END
C
C
      SUBROUTINE FORMTI(ITIME)
     include 'simdef'
      CALL IFORM(1,34,0,ITIME,19)
      RETURN
      END
C
C
      SUBROUTINE FORMSC
      CALL TFORM(1,28,0,'TIME =$',19)
     call fill(1,10,1,39,3,32)
     call rect(1,54,5,65,11,96)
      RETURN
      END
C
C
      SUBROUTINE OWNINT
     include 'simdef'
     call clear
     do 5 i=1,20
5    call lsnoff(i)
     call lsnon(5)
     call fill(5,8,2,60,22,32)
     call rect(5,7,1,61,23,96)
     CALL  TFORM(5,10,4,'MENU OF SIMULATION OPTIONS  $',12)
     CALL  TFORM(5,10,6,'0:CONTINUE THE SIMULATION   $',12)
     CALL  TFORM(5,10,7,'1:ARRIVAL RATE (NCB LORRIES)$',12)
     CALL  TFORM(5,10,8,'2:ARRIVAL RATE (MERCHANTS  )$',12)
     CALL  TFORM(5,10,9,'3:ARRIVAL RATE (TRAINS     )$',12)
     CALL TFORM(5,10,10,'4:WEIGH IN TIME    (CONST  )$',12)
     CALL TFORM(5,10,11,'5:WEIGH OUT TIME   (CONST  )$',12)
     CALL TFORM(5,10,12,'6:MERCHANT LOADING TIME     $',12)
     CALL TFORM(5,10,13,'7:TRAIN UNLOADING  TIME     $',12)
     CALL TFORM(5,10,14,'8:NO OF WEIGHBRIDGES        $',12)
     CALL TFORM(5,10,15,'9:NO OF LOADERS             $',12)
     CALL TFORM(5,10,16,'10:RANDOMIZE                $',12)
     CALL TFORM(5,10,17,'11:SIMULATION SPEED (0-200) $',12)
     call tform(5,10,18,'12:END SIMULATION/EXPERT    $',12)
C
C
10   DO 20 I=1,7
     J=IATT(ARATES,I)
     K=I+6
20   CALL IFORM(5,40,K,J,9)
     CALL IFORM(5,40,14,IATT(WPOOL,1),9)
     CALL IFORM(5,40,15,IATT(LPOOL,1),9)
     CALL IFORM(5,40,17,ISPEED(IDUM),9)
C
     CALL SFORM(5,10,20,24)
     CALL INPUTI(5,10,20,'WHAT OPTION PLEASE ?$',9,L)
     IF(L.LE.0)GOTO 990
     IF(L.GT.12)GOTO 990
     if(L.eq.12) goto 801
     CALL SFORM(5,34,20,14)
```

```
      CALL INPUTI(5,34,20,'VALUE = $',9,KK)
      IF(KK.LT.0)GOTO 990
      IF(KK.GT.1000)GOTO 990
      IF((L.GE.1).AND.(L.LE.7))CALL SETATT(ARATES,L,KK)
      IF(L.EQ.8) CALL SETATT(WPOOL,1,KK)
      IF(L.EQ.9) CALL SETATT(LPOOL,1,KK)
      IF(L.EQ.10)GOTO 800
      IF(L.EQ.11)call speed(KK)
      GOTO 10
801   call clrc
      call ends

      stop
800   DO 810 JJ=1,KK
      DO 810 JK=1,6
810   R=RNDS(JK)
      GOTO 10
990      CALL LSNOFF(5)
      CALL LSNON(1)
      CALL REFORM
         RETURN
         END
C
C
      subroutine rprams
      integer*1 flg
      include 'lsim'
      call tatom('control;',flg)
      if (flg.eq.1) goto 10
      goto 990
10    call lsend
990   continue
      return
      end
c
c
      subroutine cfill(imess)
      integer*2 imess
      call store(imess,1,'control;')
      return
      end
c
c
      subroutine expint
      integer db
      include 'simdef'
      include 'lsim'
      call incomm
      plist(1) = isize(qwin)
      plist(2) = isize(qwout)
      plist(3) = isize(qtral)
      plist(4) = isize(qmerl)
      plist(5) = endl
      call clrc
      call cfill(1)
      call csend
      call clrc
      call rdl
      db = plist(1)
      call setatt(wpool,1,db)
      db = plist(2)
      call setatt(lpool,1,db)
      return
      end
```

The Lorry Expert

```
control :-
        comlrec(L),
        control(L,M),
        commend,
        comlsend(M).

control([A,B,C,D],[E,F]) :-
              conw([A,B],E),
              conl([C,D],F).

conw([A,B],4) :- X is A+B,X>4.
conw([A,B],3) :- X is A+B,X>3.
conw([A,B],2) :- X is A+B,X>2.
conw(_,1).

conl([A,B],4) :- X is A+B,X>4.
conl([A,B],3) :- X is A+B,X>3.
conl([A,B],2) :- X is A+B,X>2.
conl(_,1).
```

```
?-consult(prlink4).
```

APPENDIX 12 - LISTINGS OF THE PROCEDURAL AGV SIMULATION AND THE SPECIFIC AGV EXPERT

The AGV Simulation

```
      program agv
      include 'agvsim'
      common/bl/iti,imi(9),itime,indxx,iprev
      include 'lsim'
      call setsys
      ipo = 1
      call ventit(gap,' $',0,16)
      call deftrk(track(1),10,10,1,0,11,16,1,2)
      call deftrk(track(2),20,10,1,0,11,16,2,3)
      call deftrk(track(3),30,10,1,0,11,16,3,4)
      call deftrk(track(4),40,10,1,0,11,16,4,5)
      call deftrk(track(5),50,10,0,1,6,16,5,6)
      call deftrk(track(6),50,15,-1,0,11,16,6,7)
      call deftrk(track(7),40,15,-1,0,11,16,7,8)
      call deftrk(track(8),30,15,-1,0,11,16,8,9)
      call deftrk(track(9),20,15,-1,0,11,16,9,10)
      call deftrk(track(10),10,15,0,-1,6,16,10,1)
      call deftrk(track(11),20,10,0,1,6,16,2,9)
      call deftrk(track(12),30,10,0,1,6,16,3,8)
      call deftrk(track(13),40,10,0,1,6,16,4,7)
      call defagv(agv(1),'1$',5,65)
      call defagv(agv(2),'2$',5,65)
      call place(agv(1),track(1),1)
      call place(agv(2),track(6),1)
      call vclass(part,150,'p$',2,20)
      call vset(pool,' $',1,-5,20,0,0,95)
      call vset(arrvq,'load/unload area$',1,11,20,-1,0,95)
      call vset(dept,'exit$',1,10,24,0,-1,95)
      call vset(npl,' $',1,-5,24,0,0,95)
      call vload(part,1,150,pool)
      call schedl(2,5,ihead(pool))
      call defmct
      call defmcb
      call reform
      do 10 i=1,2
10    call requst(agv(i))
      ievent = 0
      iiff = 0
1000  call loadpt
      call leave
      call uload
      if(iiff.ne.1) goto 12
      iiff = 0
      goto 11
12    call tranou
11    call aload
      call tranin
      call mcload
      call chkmov(agv(1))
      call chkmov(agv(2))
      iprev = ievent
      call advanc(ievent,itime,iele)
      call iform(1,2,1,ievent,7)
```

```
      if(ievent.eq.1)goto 1
      if(ievent.eq.2)goto 2
      if(ievent.eq.3)goto 3
      if(ievent.eq.4)goto 4
      if(ievent.eq.5)goto 5
      if(ievent.eq.6)goto 6
      if(ievent.eq.7)goto 7
      if(ievent.eq.8)goto 8
      if(ievent.eq.999)goto 999
```

```
      if(ievent.eq.9)goto 9
      goto 1000
1     call endmov(iele)
      call chkmov(iele)
      call uload
      goto 1000
2     call arrp(iele)
      call loadpt
      goto 1000
3     call setatt(iele,1,1)
      call aload
      call loadpt
      goto 1000
4     call transc(iele)
      goto 1000
5     call setatt(iele,1,4)
      call mcload
      goto 1000
6     iiff = 1
      call setatt(iele,1,5)
      call tranou
      goto 1000
7     i=iatt(iele,2)
      call vaddla(iele,ou(i))
      call vdelet(iele,pr(i))
      call uload
      goto 1000
8     ipart = iatt(iele,12)
      call setdsp(iele,65)
      call vaddla(ipart,ou(10))
      call vaddla(ipart,dept)
      call vdelet(ipart,ou(10))
      call movexy(ipart,dept,npl)
      call movexy(ipart,npl,pool)
      call setatt(iele,12,0)
      call setatt(iele,1,1)
      goto 1000
9     call endmov(iele)
      call chkmov(iele)
      call uload
      goto 1000
999   stop
      end
c
c
```

```
      subroutine defmct
      include 'agvsim'
      do 100 i=1,5
      iy=10
      ix=i*10
      ixp=ix-1
      iyp=iy-1
      call vset(in(i),' $',1,ixp,iyp,0,0,0)
      ixp=ix
      iyp=iy-2
      call vset(pr(i),' $',1,ixp,iyp,0,0,0)
      ixp=ix+1
      iyp=iy-1
      call vset(ou(i),' $',1,ixp,iyp,0,0,0)
100   continue
      return
      end
```

```
c
c
      subroutine defmcb
      include 'agvsim'
      do 100 i=6,10
      iy=15
      ix=110-i*10
      ixp=ix-1
      iyp=iy+1
      call vset(in(i),' $',1,ixp,iyp,0,0,0)
      ixp=ix
      iyp=iy+2
      if(i.eq.10)goto 50
      call vset(pr(i),' $',1,ixp,iyp,0,0,0)
50    ixp=ix+1
      iyp=iy+1
      call vset(ou(i),' $',1,ixp,iyp,0,0,0)
100   continue
      return
      end
c
c
      subroutine formti(itime)
      call iform(1,34,0,itime,19)
      return
      end
c
c
      subroutine formsc
      include 'agvsim'
      call tform(1,28,0,'time=$',19)
      do 200 iy=10,18,8
      do 200 ix=10,50,10
      ix1=ix-2
      ix2=ix+2
      iy1=iy-3
      iy2=iy
200   call fill(1,ix1,iy1,ix2,iy2,32)
      do 100 i=1,13
100   call dsptrk(track(i))
      return
      end
c
c
      subroutine arrp(iele)
      include 'agvsim'
      call movexy(iele,pool,arrvq)
      itim=ineg(150.0,1)
      call setatt(iele,1,0)
      call schedl(2,itim,ihead(pool))
      return
      end
c
c
      subroutine loadpt
      include 'agvsim'
      if(isize(arrvq).eq.0)goto 990
      if(isize(in(10)).gt.0)goto 990
      iele=ihead(arrvq)
      call vbehea(arrvq)
      call vaddla(iele,in(10))
      call schedl(3,3,iele)
```

```
990   return
      end
c
c
      subroutine tranou
      common/bl/iti,imi(9),itime,indxx,iprev
      include 'agvsim'
      indxx = 0
      do 900 i=1,9
      if(isize(pr(i)).eq.0)goto 900
      if(isize(ou(i)).gt.0)goto 900
      ient=ihead(pr(i))
      if(iatt(ient,1).ne.5)goto 900
      iti = itime
      imi(indxx+1) = i
      indxx = indxx +1
      call schedl(7,1,ient)
      call setatt(ient,1,6)
      call setatt(ient,2,i)
900   continue
990   return
      end
c
c
      subroutine aload
      include 'agvsim'
      ient=iempty(track(10),1)
      if(ient.eq.gap)goto 990
      if(isize(in(10)).eq.0)goto 990
      iagv=ient
      ipos=iatt(iagv,12)
      if(ipos.gt.0)goto 990
      ipart=ihead(in(10))
      if(iatt(ipart,1).ne.1)goto 990
      call setatt(ipart,1,2)
      call vbehea(in(10))
      call setatt(iagv,1,3)
      call setatt(iagv,12,ipart)
      call schedl(4,6,iagv)
      call setdsp(iagv,98)
990   return
      end
c
c
      subroutine transc(iele)
      include 'agvsim'
      call setatt(iele,1,1)
      ient=iatt(iele,12)
      if (iatt(ient,1).eq.9) goto 900
      call setatt(ient,1,3)
      inn = iempty(track(10),1)
      if (inn.eq.iele) goto 900
      call requst(iele)
 900  return
      end
c
c
      subroutine tranin
      include 'agvsim'
      do 900 i=1,9
      ient=iempty(track(i),1)
      if(ient.eq.gap)goto 900
```

```
      if(isize(in(i)).gt.0)goto 900
      iagv=ient
      ipart=iatt(iagv,12)
      if(ipart.eq.0)goto 900
      if(iatt(ipart,1).ne.3)goto 900
      call schedl(5,3,ipart)
      call setatt(iagv,12,0)
      call vaddla(ipart,in(i))
      call setdsp(iagv,65)
900   continue
      ient = iempty(track(10),1)
      if(ient.eq.gap)goto 990
      iagv = ient
      ipart = iatt(iagv,12)
      if(ipart.eq.0) goto 990
      if (iatt(ipart,1).ne.9) goto 990
      call setatt(ipart,1,0)
      call schedl(8,5,iagv)
990   continue
      call uload
      return
      end
c
c
      subroutine mcload
      include 'agvsim'
      do 900 i=1,9
      if(isize(pr(i)).gt.0)goto 900
      if(isize(in(i)).eq.0)goto 900
      ient=ihead(in(i))
      if(iatt(ient,1).ne.4)goto 900
      call vdelet(ient,in(i))
      call vaddla(ient,pr(i))
      call schedl(6,43,ient)
900   continue
      return
      end
c
c
      subroutine uload
      include 'agvsim'
      do 900 i=1,9
      if(isize(ou(i)).eq.0)goto 900
      iagv=iempty(track(i),1)
      if(iagv.eq.gap)goto 900
      if(iatt(iagv,12).ne.0) goto 900
      ient=ihead(ou(i))
      if(iatt(ient,1).ne.6)goto 900
      call setatt(ient,1,9)
      call vbehea(ou(i))
      call setatt(iagv,1,3)
      call setatt(iagv,12,ient)
      call schedl(4,6,iagv)
      call setdsp(iagv,98)
900   continue
      return
      end
c
c
      subroutine leave
      include 'agvsim'
      iagv=iempty(track(10),1)
```

```
      if(iagv.eq.gap)goto 990
      ipart=iatt(iagv,12)
      if(ipart.eq.0)goto 990
      if(iatt(ipart,1).ne.6)goto 990
      call vaddla(ipart,pool)
990   return
      end
c
c
c
c
      subroutine ownint
c     call clear
c     do 5 i=1,20
c5    call lsnoff(i)
c     call lsnon(5)
c     call fill(5,8,2,60,22,32)
c     call rect(5,7,1,61,23,96)
c     call tform(5,10,4,'MENU OF SIMULATION OPTIONS  $',12)
c     call tform(5,10,6,'0:CONTINUE THE SIMULATION   $',12)
c     call tform(5,10,7,'1:END SIMULATION            $',12)
c     call sform(5,10,20,24)
c     call inputi(5,10,20,'WHAT OPTION PLEASE ?$',9,1)
c     if (1.le.0) goto 990
c     if (1.gt.1) goto 990
      call clrc
      call ends
      stop
c990 call lsnoff(5)
c     call lsnon(1)
c     call reform
      return
      end
c
c
      subroutine expint(iele,node,j)
      include 'agvsim'
      include 'lsim'
      call incomm
      if (iele.eq.agv(1)) plist(1) = 1
      if (iele.eq.agv(2)) plist(1) = 2
      plist(2) = node
      if (plist(1).eq.2) goto 400
      iag = agv(2)
      itemp = iatt(iag,2)
      do 408 iil = 1,13
      if (track(iil).eq.itemp) goto 409
408   continue
409   plist(3) = iil
      plist(4) = iatt(iag,10) +1
      plist(5) = 5
      plist(5) = iatt(iag,3)
      iag = agv(1)
      if (iatt(iag,12).ne.0) plist(6) = 1
      if (iatt(iag,12).eq.0) plist(6) = 0
      goto 500
400   iag = agv(1)
      itemp = iatt(iag,2)
      do 308 iil = 1,13
      if (track(iil).eq.itemp) goto 309
308   continue
309   plist(3) = iil
```

```
      plist(4) = iatt(iag,10) + 1
      plist(5) = 5
      plist(5) = iatt(iag,3)
      iag = agv(2)
      if (iatt(iag,12).ne.0) plist(6) = 1
      if (iatt(iag,12).eq.0) plist(6) = 0
500   plist(7) = 0
      jj = 8
      do 100 i=1,9
      if (isize(in(i)).eq.0) goto 100
      plist(jj) = i
      jj = jj+1
100   continue
      plist(jj) = 0
      jj = jj+1
      do 200 i=1,9
      if (isize(pr(i)).eq.0) goto 200
      plist(jj) = i
      jj = jj+1
200   continue
      plist(jj) = 0
      jj = jj+1
      do 300 i=1,9
      if (isize(ou(i)).eq.0) goto 300
      plist(jj) = i
      jj = jj+1
300   continue
      if (isize(in(10)).ne.0) plist(jj) = 1
      if (isize(in(10)).eq.0) plist(jj) = 0
      plist(jj+1) = endl
      call clrc
      call cfill(1)
      call csend
      call clrc
      call rdl
      j = plist(1)
      return
      end
c
c
      subroutine cfill(imess)
      integer*2 imess
      call store(imess,1,'control;')
      return
      end
c
c
      subroutine rprams
      integer*1 flg
      include 'lsim'
      call tatom('control;',flg)
      if (flg.eq.1) goto 10
      goto 990
10    call lsend
990   continue
      return
      end
c
c
```

APPENDIX 13 - LISTINGS OF THE LORRY SIMULATION (WITH LEARNING INTERFACE) AND THE LEARNING EXPERT (LOGICS 1 & 2)

The Lorry Simulation

```
      PROGRAM LORRY
        common/ctype/icot
        include 'simdef'
        include 'lsim'
        icot = 1
        ipo = 1
       call setsys
      CALL VCLASS(MERCH,150,'merch$',1,9)
      CALL VCLASS(NCB,150,'ncb-v$',1,14)
      CALL VCLASS(TRAIN,50,'train$',1,10)
      CALL VENTIT(WPOOL,'wpool$',1,19)
      CALL VENTIT(LPOOL,'lpool$',1,19)
      CALL VENTIT(ARATES,'rates$',7,19)
      CALL VSET(MPOOL,'mpool$',20,1,-5,0,0,19)
      CALL VSET(QWIN,'weighin.q$',1,1,5,0,-1,12)
      CALL VSET(WIN,'weigh*$',1,10,10,0,-1,13)
      CALL VSET(QMERL,'lorry.q$',1,46,17,0,-1,14)
      CALL VSET(MLOAD,'loader$',1,55,23,0,-1,15)
      CALL VSET(QWOUT,'weighout-qu$',1,22,15,0,1,16)
      CALL VSET(WOUT,'bridge$',1,16,10,0,1,30)
      CALL VSET(NPOOL,'npool$',20,1,-5,0,0,19)
      CALL VSET(OWORK,'owork$',1,34,20,0,-1,29)
      CALL VSET(TPOOL,'tpool$',20,72,-5,0,0,19)
      CALL VSET(QTRAL,'train.q$',1,70,17,0,-1,28)
      CALL VSET(TLOAD,'/loader$',1,61,23,0,-1,27)
        call vset(lout,'lorry$',1,1,21,6,0,30)
        call vset(lexit,'lorry$',1,-10,21,0,1,28)
        call vset(texit,'train$',1,90,22,-1,0,29)
        call setatt(wpool,1,2)
        call setatt(lpool,1,2)
        call setatt(arates,1,20)
        call setatt(arates,2,15)
        call setatt(arates,3,50)
        call setatt(arates,4,3)
        call setatt(arates,5,4)
        call setatt(arates,6,10)
        call setatt(arates,7,25)
C
C     INITIALISE MODEL
C
        call vload(merch,1,150,mpool)
        call vload(ncb,1,150,npool)
        call vload(train,1,50,tpool)
       CALL SCHEDL(8,10,IHEAD(MPOOL))
       CALL SCHEDL(9,5,IHEAD(NPOOL))
       CALL SCHEDL(7,25,IHEAD(TPOOL))
        CALL FORMSC
C
C     SIMULATION MODEL
C
1000  CALL ADVANC(IEVENT,ITIME,IELE)
        call expint
        if(ievent.eq.999)goto 999
c
      GO TO (1,1,3,3,5,6,7,8,9,10),IEVENT
C
C     WEIGHIN ENDS
C
1     CALL ENDWIN(IELE)
      CALL STMLOA
      CALL STWIN
      CALL STWOUT
```

```
      GO TO 1000
c
c       weighout ends
c
3     CALL ENDOUT(IELE)
      CALL STWIN
      CALL STWOUT
      GO TO 1000
c
c       train unloading ends
c
5     CALL ENDTLO(IELE)
      CALL STTLOA
      CALL STMLOA
      GO TO 1000
c
c       merchant loading ends
c
6     CALL ENDMLO(IELE)
        CALL STTLOA
      CALL STMLOA
      CALL STWOUT
      GO TO 1000
c
c       train arrives
c
7     CALL ARRT(IELE)
      CALL STTLOA
      GO TO 1000
c
c       merchant arrives
c
8     CALL ARRM(IELE)
      CALL STWIN
      GO TO 1000
c
c       coal lorry arrives
c
9     CALL ARRN(IELE)
      CALL STWIN
      GO TO 1000
c
c       coal lorry other work finished
c
10    CALL ENDOW(IELE)
      CALL STWOUT
      GO TO 1000
c
c
999     stop
      END
C
C     END OF MAIN PROGRAM
C
      SUBROUTINE ARRT(IELE)
        include 'simdef'
        call movexy(iele,tpool,qtral)
        call setatt(iele,1,3)
      RVAL=IATT(ARATES,3)
      ITIM=INEG(RVAL,1)
        IF(ISIZE(TPOOL).LE.0)GOTO 990
      CALL SCHEDL(7,ITIM,IHEAD(TPOOL))
```

```
990     RETURN
        END
C
C
        SUBROUTINE ARRM(IELE)
          include 'simdef'
          call movexy(iele,mpool,qwin)
          call setatt(iele,1,2)
        RVAL=IATT(ARATES,2)
        ITIM=INEG(RVAL,2)
          IF(ISIZE(MPOOL).LE.0)GOTO 990
        CALL SCHEDL(8,ITIM,IHEAD(MPOOL))
990     RETURN
        END
C
C
        SUBROUTINE ARRN(IELE)
          include 'simdef'
          call movexy(iele,npool,qwin)
          call setatt(iele,1,1)
        RVAL=IATT(ARATES,1)
        ITIM=INEG(RVAL,3)
          IF(ISIZE(NPOOL).LE.0)GOTO 990
        CALL SCHEDL(9,ITIM,IHEAD(NPOOL))
990     RETURN
        END
C
C
        SUBROUTINE STWIN
          include 'simdef'
10        ILOAD=ISIZE(WIN)+ISIZE(WOUT)
          IF(ILOAD.GE.IATT(WPOOL,1)) GO TO 990
          IF(ISIZE(QWIN).LE.0) GO TO 990
        K=IHEAD(QWIN)
          call movexy(k,qwin,win)
        ITIM=IATT(ARATES,4)
          call schedl(1,ITIM,K)
          goto 10
990     RETURN
        END
C
C
            SUBROUTINE endwin(iele)
          include 'simdef'
          if (iatt(iele,1).eq.2) call movexy(iele,win,qmerl)
          if (iatt(iele,1).eq.1) call stow(iele)
        RETURN
        END
C
C
        SUBROUTINE STOW(iele)
          include 'simdef'
          call movexy(iele,win,owork)
          itim=irand(15,25,4)
          call schedl(10,itim,iele)
        RETURN
        END
c
c
          subroutine endow(iele)
          include 'simdef'
          call movexy(iele,owork,qwout)
```

```
        return
        end
C
C
      SUBROUTINE STWOUT
        include 'simdef'
10      ILOAD=ISIZE(WIN)+ISIZE(WOUT)
      IF(ILOAD.GE.IATT(WPOOL,1)) GO TO 990
      IF(ISIZE(QWOUT).LE.0) GO TO 990
      K=IHEAD(QWOUT)
        call movexy(k,qwout,wout)
      ITIM=IATT(ARATES,5)
      CALL SCHEDL(3,ITIM,K)
        GOTO 10
990   RETURN
      END
C
C
      SUBROUTINE ENDOUT(IELE)
        include 'simdef'
        call movexy(iele,wout,lout)
        call moveyx(iele,lout,lexit)
      IF(IATT(IELE,1).EQ.1)       CALL moveyx(iele,lexit,npool)
      IF(IATT(IELE,1).EQ.2)       CALL moveyx(iele,lexit,mpool)
      RETURN
      END
C
C
      SUBROUTINE STMLOA
        include 'simdef'
10    ILOAD=ISIZE(MLOAD)+ISIZE(TLOAD)
      IF(ILOAD.GE.IATT(LPOOL,1)) GO TO 990
      IF(ISIZE(QMERL).LE.0) GO TO 990
      K=IHEAD(QMERL)
        call movexy(k,qmerl,mload)
      RVAL=IATT(ARATES,6)
      ITIM=INEG(RVAL,5)
      CALL SCHEDL(6,ITIM,K)
        GOTO 10
990   RETURN
      END
C
C
      SUBROUTINE ENDMLO(IELE)
        include 'simdef'
        call movexy(iele,mload,qwout)
      RETURN
      END
C
C
      SUBROUTINE STTLOA
        include 'simdef'
10    ILOAD=ISIZE(MLOAD)+ISIZE(TLOAD)
      IF(ILOAD.GE.IATT(LPOOL,1)) GO TO 990
      IF(ISIZE(QTRAL).LE.0) GO TO 990
      K=IHEAD(QTRAL)
        call movexy(k,qtral,tload)
      RVAL=IATT(ARATES,7)
      ITIM=INEG(RVAL,6)
      CALL SCHEDL(5,ITIM,K)
        GOTO 10
990   RETURN
```

```
      END
C
C
      SUBROUTINE ENDTLO(IELE)
        include 'simdef'
        call movexy(iele,tload,texit)
        call moveyx(iele,texit,tpool)
      RETURN
      END
C
C
      SUBROUTINE FORMTI(ITIME)
        include 'simdef'
      CALL IFORM(1,34,0,ITIME,19)
      RETURN
      END
C
C
      SUBROUTINE FORMSC
      CALL TFORM(1,28,0,'TIME =$',19)
        call fill(1,10,1,39,3,32)
        call rect(1,54,5,65,11,96)
      RETURN
      END
C
C
      SUBROUTINE OWNINT
        include 'simdef'
        common/ctype/icot
        call clear
        do 5 i=1,20
5       call lsnoff(i)
        call lsnon(5)
        call fill(5,8,2,60,22,32)
        call rect(5,7,1,61,23,96)
        CALL  TFORM(5,10,4,'MENU OF SIMULATION OPTIONS  $',12)
        CALL  TFORM(5,10,6,'0:CONTINUE THE SIMULATION   $',12)
        CALL  TFORM(5,10,7,'1:ARRIVAL RATE (NCB LORRIES)$',12)
        CALL  TFORM(5,10,8,'2:ARRIVAL RATE (MERCHANTS  )$',12)
        CALL  TFORM(5,10,9,'3:ARRIVAL RATE (TRAINS     )$',12)
        CALL TFORM(5,10,10,'4:WEIGH IN TIME    (CONST  )$',12)
        CALL TFORM(5,10,11,'5:WEIGH OUT TIME   (CONST  )$',12)
        CALL TFORM(5,10,12,'6:MERCHANT LOADING TIME     $',12)
        CALL TFORM(5,10,13,'7:TRAIN UNLOADING  TIME     $',12)
        CALL TFORM(5,10,14,'8:NO OF WEIGHBRIDGES        $',12)
        CALL TFORM(5,10,15,'9:NO OF LOADERS             $',12)
        CALL TFORM(5,10,16,'10:RANDOMIZE                $',12)
        CALL TFORM(5,10,17,'11:SIMULATION SPEED (0-200) $',12)
        call tform(5,10,18,'12:END SIMULATION/EXPERT    $',12)
        call tform(5,10,19,'13:EXPERT COMMAND NUMBER    $',12)
C
C
10      DO 20 I=1,7
        J=IATT(ARATES,I)
        K=I+6
20      CALL IFORM(5,40,K,J,9)
        CALL IFORM(5,40,14,IATT(WPOOL,1),9)
        CALL IFORM(5,40,15,IATT(LPOOL,1),9)
        CALL IFORM(5,40,17,ISPEED(IDUM),9)
        call iform(5,40,19,icot,9)
C
        CALL SFORM(5,10,20,24)
```

```
      CALL INPUTI(5,10,20,'WHAT OPTION PLEASE ?$',9,L)
      IF(L.LE.0)GOTO 990
      IF(L.GT.13)GOTO 990
      if(L.eq.12) goto 801
      if(L.eq.13) goto 851
      CALL SFORM(5,34,20,14)
      CALL INPUTI(5,34,20,'VALUE = $',9,KK)
      IF(KK.LT.0)GOTO 990
      IF(KK.GT.1000)GOTO 990
      IF((L.GE.1).AND.(L.LE.7))CALL SETATT(ARATES,L,KK)
      IF(L.EQ.8) CALL SETATT(WPOOL,1,KK)
      IF(L.EQ.9) CALL SETATT(LPOOL,1,KK)
      IF(L.EQ.10)GOTO 800
      IF(L.EQ.11)call speed(KK)
      GOTO 10
801   call clrc
      call ends
      stop
851   call sform(5,34,20,14)
      call inputi(5,34,20,'VALUE = $',9,kk)
      if(kk.lt.1)goto 851
      if(kk.gt.3)goto 851
      icot = kk
      goto 10
800   DO 810 JJ=1,KK
      DO 810 JK=1,6
810   R=RNDS(JK)
      GOTO 10
990   CALL LSNOFF(5)
      CALL LSNON(1)
      CALL REFORM
      RETURN
      END
C
C
      subroutine rprams
      integer*1 flg
      include 'lsim'
      call tatom('monitor;',flg)
      if (flg.eq.1) goto 10
      call tatom('control;',flg)
      if (flg.eq.1) goto 10
      goto 990
10    call lsend
990   continue
      return
      end
c
c
      subroutine cfill(imess)
      integer*2 imess
      call store(imess,1,'monitor;')
      call store(imess,2,'control;')
      call store(imess,3,'save;')
      return
      end
c
c
      subroutine expint
      integer db
      common/ctype/icot
      include 'simdef'
```

```
      include 'lsim'
      call incomm
      goto (100,200,300),icot
100   plist(1) = isize(qwin)
      plist(2) = isize(qwout)
      plist(3) = isize(qtral)
      plist(4) = isize(qmerl)
      plist(5) = iatt(wpool,1)
      plist(6) = iatt(lpool,1)
      plist(7) = endl
      call clrc
      call cfill(1)
      call csend
      call clrc
      goto 990
200   plist(1) = isize(qwin)
      plist(2) = isize(qwout)
      plist(3) = isize(qtral)
      plist(4) = isize(qmerl)
      plist(5) = endl
      call clrc
      call cfill(2)
      call csend
      call clrc
      call rdl
      if(plist(1).lt.0) goto 250
      if(plist(2).lt.0) goto 250
      db = plist(1)
      call setatt(wpool,1,db)
      db = plist(2)
      call setatt(lpool,1,db)
      goto 990
250   call inter2
      goto 990
300   call clrc
      call cfill(3)
      call csend
      call clrc
990   return
      end
c
c
      subroutine inter2
c
      common/ctype/icot
c
      icot = 1
      call openw(40,0,79,5,19)
      call scroll(0,1,40,0,79,5,19)
      call tform(0,50,1,'UNKNOWN DOMAIN TO EXPERT $',19)
      CALL tform(0,50,2,'RE-ENTERING MONITOR MODE$',19)
      CALL INPUTT(0,50,3,'TYPE CHAR. FOR OWN : $',19,IIAR)
      call ownint
      call closew
      return
      end
c
c
```

The Learning Expert Logic 1

```
control :-
        comlrec(L),
        control(L,M),
        commend,
        comlsend(M).

control([Q1,Q2,Q3,Q4],[W,L]) :-
        Wsum is Q1+Q2,
        Lsum is Q3+Q4,
        tbound(Wsum,w,W),
        tbound(Lsum,l,L),
        evaluate(Wsum,Lsum,W,L),!.
control([Q1,Q2,Q3,Q4],[W,L]) :-
        Wsum is Q1+Q2,
        Lsum is Q3+Q4,
        tbound(Lsum,l,L),
        not(tbound(Wsum,w,_)),
        min_res(w,Wsum,W),
        evaluate(Wsum,Lsum,W,L),!.
control([Q1,Q2,Q3,Q4],[W,L]) :-
        Wsum is Q1+Q2,
        Lsum is Q3+Q4,
        tbound(Wsum,w,W),
        not(tbound(Lsum,l,_)),
        min_res(l,Lsum,L),
        evaluate(Wsum,Lsum,W,L),!.
control([Q1,Q2,Q3,Q4],[W,L]) :-
        Wsum is Q1+Q2,
        Lsum is Q3+Q4,
        not(tbound(Wsum,w,_)),
        min_res(w,Wsum,W),
        not(tbound(Lsum,l,_)),
        min_res(l,Lsum,L),
        evaluate(Wsum,Lsum,W,L),!.

tbound(S1,V,NV) :-
        compare(V,NV),
        bound(V,NV,UV1,LV1),
        rndnr(UV1,UV),rndnr(LV1,LV),
        S1 =< UV,S1 >= LV,!.

min_res(V,S,M) :-
        member(M,[1,2,3,4,5,6]),
        tbound2(S,V,M),retractall(compare(V,_)),
        assert(compare(V,M)),!.
min_res(_,_,ping):-!.
/*
min_res(V,_,7) :-
        retractall(compare(V,_)),assert(compare(V,7)),!.
*/
tbound2(S,V,M) :-
        bound(V,M,UV1,LV1),
        rndnr(UV1,UV),rndnr(LV1,LV),
        S=<UV,S>=LV,!.

member(X,[X|L]).
member(X,[_|L]) :- member(X,L).

rndnr(In,Out) :-
        Out is fix(In),
```

```
        X is In - Out,
        X<0.5,!.
rndnr(In,Out) :-
        X is fix(In),
        Out is X+1,!.
evaluate(Wsum,Lsum,ping,_).
evaluate(_,_,_,ping).
evaluate(Wsum,Lsum,W,L) :-
        chosen(Wsum,N),
        retract(ceval(w,E,EN)),
        EN1 is EN+1,
        WS1 is E+Wsum+(N*W),
        assert(ceval(w,WS1,EN1)),
        chosen(Lsum,N1),
        retract(ceval(l,E1,EN1)),
        EN11 is EN1+1,
        LS1 is E1+Lsum+(N1*L),
        assert(ceval(l,LS1,EN11)),!.

ceval(w,0,0).
ceval(l,0,0).
/*
chosen(S,6) :- S =< 3.
chosen(S,5) :- S =< 7.
chosen(S,4) :- S =< 9.
chosen(S,3) :- S =< 12.
chosen(S,2) :- S =< 15.
*/
chosen(S,2).

test_expert :- nl,
               output(" The experts average performance constants are :")
               nl,nl,
               output(" For weighbridges :"),
               ceval(w,S,N),R is S/N,tab(1),write(R),nl,
               output(" For loaders      :"),
               ceval(l,Sl,Nl),Rl is Sl/Nl,tab(1),write(Rl),nl,nl.

/*
control([A,B,C,D],[E,F]) :-
               conw([A,B],E),
               conl([C,D],F).

conw([A,B],4) :- X is A+B,X>4.
conw([A,B],3) :- X is A+B,X>3.
conw([A,B],2) :- X is A+B,X>2.
conw(_,1).

conl([A,B],4) :- X is A+B,X>4.
conl([A,B],3) :- X is A+B,X>3.
conl([A,B],2) :- X is A+B,X>2.
conl(_,1).
*/
monitor :-
        comlrec(l),
        monitor(L),
        commend.

monitor([Qwin,Qwout,Qtral,Qmerl,W,L]) :-
        compare(w,W),
        Sum is Qwin+Qwout,
        cupper(w,Sum),
```

```
        clower(w,Sum),
        compare(l,L),
        Suml is Qtral+Qmerl,
        cupper(l,Suml),
        clower(l,Suml),!.
monitor([Qwin,Qwout,Qtral,Qmerl,W,L]) :-
        not(compare(w,W)),
        calc_bound(w),
        new_start(w,W),
        Sum is Qwin+Qwout,
        clower(w,Sum),
        cupper(w,Sum),fail.
monitor([Qwin,Qwout,Qtral,Qmerl,W,L]) :-
        not(compare(l,L)),
        calc_bound(l),
        new_start(l,L),
        Sum is Qwin+Qwout,
        clower(l,Sum),
        cupper(l,Sum),fail.
monitor(_) :- !.

upper(w,0).
lower(w,1000).
upper(l,0).
lower(l,1000).

cupper(V,S) :-
        upper(V,U),S>U,retract(upper(V,U)),
        assert(upper(V,S)),!.
cupper(_,_) :- !.

clower(V,S) :-
        lower(V,L),S<L,retract(lower(V,L)),
        assert(lower(V,S)),!.
clower(_,_) :- !.

compare(w,2).
compare(l,2).

new_start(V,N) :-
        retractall(lower(V,_)),
        retractall(upper(V,_)),
        retractall(compare(V,_)),
        assert(lower(V,1000)),
        assert(upper(V,0)),
        assert(compare(V,N)),!.

first(w,0).
first(w,1).
first(w,2).
first(w,3).
first(w,4).
first(w,5).
first(w,6).

first(l,0).
first(l,1).
first(l,2).
first(l,3).
first(l,4).
first(l,5).
first(l,6).
```

```
calc_bound(V) :-
        compare(V,N),
        retract(first(V,N)),
        upper(V,U),
        lower(V,L),
        assb(V,N,U,L),!.
calc_bound(V) :-
        compare(V,N),
        retract(bound(V,N,U1,L1)),
        Alpha is 0.2,
        upper(V,U),
        lower(V,L),
        U2 is (Alpha*U) + ((1-Alpha)*U1),
        L2 is (Alpha*L) + ((1-Alpha)*L1),
        assb(V,N,U2,L2),!.

assb(V,1,U,_) :- assert(bound(V,1,U,0)),!.
assb(V,N,U,1000) :- assert(bound(V,N,U,0)),!.
assb(V,N,U,L) :- assert(bound(V,N,U,L)),!.

save    :-
        calc_bound(l),
        calc_bound(w),
        savep,
        commend.

savep :-
        output("request to save data received "),nl,
        output("type name of data file :"),read(X),nl,
        save(X),!.
save(X) :-
        tell(X),
        predicate_list(Y),
        member(YM,Y),
        listing(YM),
        fail.
save(X) :- told.

predicate_list([bound]).

alt_d_base :- bound(A,B,_,_),retract(first(A,B)),fail.
alt_d_base.

output([]):-!.
output([X|L]) :- put(X),output(L).

zz:- retract(bound(w,1,U,_)),
     assert(bound(w,1,U,0)).
zz.
zl:- retract(bound(l,1,UU,_)),
     assert(bound(l,1,UU,0)).
zl.

start:-nl,output("This is the coal depot learning expert"),nl,
  output("do you want to run a new or old problem (sn./o.) ? :"),seer
  read(X),
  assert(X),
  (sn;(output("name of saved data file :"),read(Y),
   consult(Y),alt_d_base,zz,zl,
   nl)),
   see('dummy.pro'),nl.
```

```
rest :- retractall(ceval(_,_,_)),
        assert(ceval(1,0,0)),assert(ceval(w,0,0)).

?-spy control(2).
?-spy monitor(1).
?-spy savep.
?-consult(prlink4).

?-start.
```

The Learning Expert Logic 2

```
control :-
        comlrec(L),
        control(L,M),
        commend,
        comlsend(M).

control([Q1,Q2,Q3,Q4],[W,L]) :-
        Wsum is Q1+Q2,
        Lsum is Q3+Q4,
        min_res(w,Wsum,W),
        min_res(l,Lsum,L),
        evaluate(Wsum,Lsum,W,L),!.

tbound(S1,V,NV) :-
        compare(V,NV),
        bound(V,NV,UV1,LV1),
        rndnr(UV1,UV),rndnr(LV1,LV),
        S1 =< UV,S1 >= LV,!.

min_res(V,S,M) :-
        member(M,[1,2,3,4,5,6]),
        tbound2(S,V,M),retractall(compare(V,_)),
        assert(compare(V,M)),!.
min_res(_,_,ping):-!.
/*
min_res(V,_,7) :-
        retractall(compare(V,_)),assert(compare(V,7)),!.
*/
tbound2(S,V,M) :-
        bound(V,M,UV1,LV1),
        rndnr(UV1,UV),rndnr(LV1,LV),
        S=<UV,S>=LV,!.

member(X,[X|L]).
member(X,[_|L]) :- member(X,L).

rndnr(In,Out) :-
        Out is fix(In),
        X is In - Out,
        X<0.5,!.
rndnr(In,Out) :-
        X is fix(In),
        Out is X+1,!.
evaluate(Wsum,Lsum,ping,_).
evaluate(_,_,_,ping).
evaluate(Wsum,Lsum,W,L) :-
        chosen(Wsum,N),
        retract(ceval(w,E,EN)),
        EN1 is EN+1,
        WS1 is E+Wsum+(N*W),
        assert(ceval(w,WS1,EN1)),
        chosen(Lsum,N1),
        retract(ceval(l,E1,EN1)),
        EN11 is EN1+1,
        LS1 is E1+Lsum+(N1*L),
        assert(ceval(l,LS1,EN11)),!.

ceval(w,0,0).
ceval(l,0,0).
/*
```

```
chosen(S,6) :- S =< 3.
chosen(S,5) :- S =< 7.
chosen(S,4) :- S =< 9.
chosen(S,3) :- S =< 12.
chosen(S,2) :- S =< 15.
*/
chosen(S,2).

test_expert :- nl,
               output(" The experts average performance constants are :"),
               nl,nl,
               output(" For weighbridges :"),
               ceval(w,S,N),R is S/N,tab(1),write(R),nl,
               output(" For loaders      :"),
               ceval(l,Sl,Nl),Rl is Sl/Nl,tab(1),write(Rl),nl,nl.

/*
control([A,B,C,D],[E,F]) :-
                  conw([A,B],E),
                  conl([C,D],F).

conw([A,B],4) :- X is A+B,X>4.
conw([A,B],3) :- X is A+B,X>3.
conw([A,B],2) :- X is A+B,X>2.
conw(_,1).

conl([A,B],4) :- X is A+B,X>4.
conl([A,B],3) :- X is A+B,X>3.
conl([A,B],2) :- X is A+B,X>2.
conl(_,1).
*/
monitor :-
         comlrec(L),
         monitor(L),
         commend.

monitor([Qwin,Qwout,Qtral,Qmerl,W,L]) :-
         compare(w,W),
         Sum is Qwin+Qwout,
         cupper(w,Sum),
         clower(w,Sum),
         compare(l,L),
         Suml is Qtral+Qmerl,
         cupper(l,Suml),
         clower(l,Suml),!.
monitor([Qwin,Qwout,Qtral,Qmerl,W,L]) :-
         not(compare(w,W)),
         calc_bound(w),
         new_start(w,W),
         Sum is Qwin+Qwout,
         clower(w,Sum),
         cupper(w,Sum),fail.
monitor([Qwin,Qwout,Qtral,Qmerl,W,L]) :-
         not(compare(l,L)),
         calc_bound(l),
         new_start(l,L),
         Sum is Qwin+Qwout,
         clower(l,Sum),
         cupper(l,Sum),fail.
monitor(_) :- !.

upper(w,0).
```

```
lower(w,1000).
upper(l,0).
lower(l,1000).

cupper(V,S) :-
        upper(V,U),S>U,retract(upper(V,U)),
        assert(upper(V,S)),!.
cupper(_,_) :- !.

clower(V,S) :-
        lower(V,L),S<L,retract(lower(V,L)),
        assert(lower(V,S)),!.
clower(_,_) :- !.

compare(w,2).
compare(l,2).

new_start(V,N) :-
        retractall(lower(V,_)),
        retractall(upper(V,_)),
        retractall(compare(V,_)),
        assert(lower(V,1000)),
        assert(upper(V,0)),
        assert(compare(V,N)),!.

first(w,0).
first(w,1).
first(w,2).
first(w,3).
first(w,4).
first(w,5).
first(w,6).

first(l,0).
first(l,1).
first(l,2).
first(l,3).
first(l,4).
first(l,5).
first(l,6).

calc_bound(V) :-
        compare(V,N),
        retract(first(V,N)),
        upper(V,U),
        lower(V,L),
        assb(V,N,U,L),!.
calc_bound(V) :-
        compare(V,N),
        retract(bound(V,N,U1,L1)),
        Alpha is 0.2,
        upper(V,U),
        lower(V,L),
        U2 is (Alpha*U) + ((1-Alpha)*U1),
        L2 is (Alpha*L) + ((1-Alpha)*L1),
        assb(V,N,U2,L2),!.

assb(V,1,U,_) :- assert(bound(V,1,U,0)),!.
assb(V,N,U,1000) :- assert(bound(V,N,U,0)),!.
assb(V,N,U,L) :- assert(bound(V,N,U,L)),!.

save    :-
```

```
        calc_bound(w),
        calc_bound(l),
        savep,
        commend.

savep :-
        output("request to save data received "),nl,
        output("type name of data file :"),read(X),nl,
        save(X),!.
save(X) :-
        tell(X),
        predicate_list(Y),
        member(YM,Y),
        listing(YM),
        fail.
save(X) :- told.

predicate_list([bound]).

alt_d_base :- bound(A,B,_,_),retract(first(A,B)),fail.
alt_d_base.

output([]):-!.
output([X|L]) :- put(X),output(L).

zz:- retract(bound(w,1,U,_)),
     assert(bound(w,1,U,0)).
zz.
zl:- retract(bound(l,1,UU,_)),
     assert(bound(l,1,UU,0)).
zl.

start:-nl,output("This is the coal depot learning expert"),nl,
  output("do you want to run a new or old problem (sn./o.) ? :"),see
  read(X),
  assert(X),
  (sn;(output("name of saved data file :"),read(Y),
   consult(Y),alt_d_base,zz,zl,
   nl)),
   see('dummy.pro'),nl.
rest:- retractall(ceval(_,_,_)),assert(ceval(1,0,0)),
        assert(ceval(w,0,0)).
?-spy control(2).
?-spy monitor(1).
?-spy savep.

?-consult(prlink4).

?-start.
```

APPENDIX 14 - THE LEARNING EXPERIMENT INSTRUCTIONS AND DATABASES

Instructions to Experiment Volunteers

THE PROBLEM

The simulation is of a coal depot. Merchant and NCB lorries arrive and must be weighed at a set of weighbridges. Once in the depot, the Merchants must be loaded with coal before weighing out via the weighbridges. Loading is done by a set of loaders which must also load trains arriving at the depot. Trains have a higher priority than Merchants for loading.

Two resources are variable by the user in this simulation: Weighbridges and Loaders. The aim of the user is to control the depot with minimum cost. It has been noticed in the past that the cost of one entity queueing for a resource is approximately half the cost of a single increment in the level of that resource. Rates of arrival and duration of events will be varied throughout the simulation. The simulation will stop automatically after 200 time units.
nb. the number of weighbridges and loaders must lie between 1 and 6.

THE EXPERIMENT

Whilst you are using the simulation you will be monitored by another computer. This will collect information and learn how you have decided to control the simulation.

Using this information from you and other people it is hoped that we can answer questions such as:

Can we combine knowledge aquired from different users to gain an improved performance ?

Would one novice using the computer for say 5 periods be superior to the combined knowledge (and therefore performance) of 5 different people for 1 hour each ?

The Law Individual Database

```
bound(w,1,2,0) .
bound(w,2, 0.8000000000E+000, 0.6000000000E+000) .
bound(w,3,5,2) .
bound(w,5,16,2) .
bound(1,5,6,0) .
bound(1,1, 0.7200000000E+001,0) .
bound(1,2, 0.7912000000E+001, 0.1128000000E+001) .
bound(1,4, 0.1064000000E+002, 0.8400000000E+000) .
bound(w,6,26,13) .
bound(1,3, 0.9080000000E+001, 0.2880000000E+001) .
```

The Maths Individual Database

```
bound(w,1,3,0) .
bound(w,2, 0.1200000000E+001, 0.6000000000E+000) .
bound(1,1,3,0) .
bound(w,3, 0.7000000000E+001, 0.4000000000E+001) .
bound(w,4, 0.8616000000E+001, 0.5904000000E+001) .
bound(1,2, 0.4200000000E+001, 0.9600000000E+000) .
bound(1,4, 0.1279200000E+002, 0.5744000000E+001) .
bound(w,5, 0.1115168000E+002, 0.8137600000E+001) .
bound(w,6, 0.1324640000E+002, 0.8316800000E+001) .
bound(1,3, 0.1378080000E+002, 0.3803200000E+001) .
```

The Physics Individual Database

```
bound(w,2,0,0) .

bound(w,1,3,0) .

bound(1,1,5,0) .

bound(1,5,6,6) .

bound(w,3, 0.7000000000E+001, 0.2600000000E+001) .

bound(1,4, 0.7712000000E+001, 0.1360000000E+001) .

bound(w,6, 0.1080000000E+002, 0.4400000000E+001) .

bound(w,5, 0.7800000000E+001, 0.5440000000E+001) .

bound(1,3, 0.1013472000E+002, 0.2844160000E+001) .

bound(w,4, 0.6800000000E+001, 0.3800000000E+001) .

bound(1,2, 0.6729600000E+001, 0.3523200000E+001) .
```

```
bound(w,1,4,0) .
bound(w,2, 0.1200000000E+001, 0.6000000000E+000) .
bound(1,1,4,0) .
bound(w,4,7,5) .
bound(1,2, 0.2640000000E+001, 0.1120000000E+001) .
bound(w,5, 0.6960000000E+001, 0.4640000000E+001) .
bound(w,6,24,11) .
bound(1,4, 0.1241152000E+002, 0.4342080000E+001) .
bound(w,3, 0.1060000000E+002, 0.6080000000E+001) .
bound(1,3, 0.1285312000E+002, 0.4515840000E+001) .
```

```
bound(w,2,0,0) .

bound(1,1,4,0) .

bound(w,1,7,0) .

bound(w,3, 0.5200000000E+001, 0.1400000000E+001) .

bound(1,5,7,4) .

bound(1,4, 0.1460000000E+002, 0.4400000000E+001) .

bound(1,2, 0.8432000000E+001, 0.1968000000E+001) .

bound(w,6,28,7) .

bound(1,3, 0.1519936000E+002, 0.4622080000E+001) .
```

The Combined Database i

```
bound(w,1,3.8,0).
bound(w,2,0.64,0.3286).
bound(w,3,6.96,3.216).
bound(w,4,7.472,4.9013).
bound(w,5,10.7792,5.0544).
bound(w,6,20.40928,8.74336).
bound(l,1,4.64,0).
bound(1,2,5.98272,1.73984).
bound(1,3,12.2096,3.733056).
bound(1,4,11.6311,3.337216).
bound(1,5,6.33,3.333).
```

The Combined Database ii

```
bound(w,2,0,0) .
bound(w,1,7,0) .
bound(w,3, 0.5200000000E+001, 0.1400000000E+001) .
bound(w,6,28,7) .
bound(1,1,3,0) .
bound(1,2, 0.4200000000E+001, 0.9600000000E+000) .
bound(1,4, 100, 0.5744000000E+001) .
bound(1,3, 0.1378080000E+002, 0.3803200000E+001) .
```

The Combined Database iii

```
bound(w,1,2,0) .
bound(w,2, 0.8000000000E+000, 0.6000000000E+000) .
bound(w,3,5,2) .
bound(w,5,16,2) .
bound(w,6,100,13) .
bound(1,1,3,0) .
bound(1,2, 0.4200000000E+001, 0.9600000000E+000) .
bound(1,4, 100, 0.5744000000E+001) .
bound(1,3, 0.1378080000E+002, 0.3803200000E+001) .
```

The Combined Database iv

```
bound(w,2,0,0) .


bound(w,1,7,0) .
bound(w,3, 0.5200000000E+001, 0.1400000000E+001) .
bound(w,6,28,7) .
bound(1,5,100,0) .
bound(1,4, 0.1064000000E+002, 0.8400000000E+000) .
bound(1,1,3,0) .
bound(1,2, 0.4200000000E+001, 0.9600000000E+000) .
bound(1,3, 9.0, 0.3803200000E+001) .
```

The Combined Database v

```
bound(w,2,0,0) .
bound(w,1,7,0) .
bound(w,3, 0.5200000000E+001, 0.1400000000E+001) .
bound(w,6,28,7) .
bound(1,1,3,0) .
bound(1,2, 0.4200000000E+001, 0.9600000000E+000) .
bound(1,4, 100, 0.5744000000E+001) .
bound(1,3, 0.1378080000E+002, 0.3803200000E+001) .
```