A Rule-Based Approach to Real Time Systems

Peijuan Xie

M.Sc. in Computer Applications

1991

A Rule-Based Approach to Real Time Systems

A Dissertation presented in fulfilment

of the requirement for the M.Sc Degree

in Computer Applications

August 1991

Peijuan Xie B.Sc

School of Computer Applications

Dublin City University

Dublin 9.

Supervisor: Mr Michael Ryan.

Declaration.

This dissertation is based on the author's own work. It has not previously been submitted for a degree at any academic institution.

Peijuan Xie

Peijuan Xie
August 1991.

ACKNOWLEDGEMENT

I would like to thank my supervisor, Mr. Michael Ryan for his invaluable advice, patience and encouragement throughout the research for this thesis. I would also like to thank Mr. Garvan McFeeley for his help in building up the interrupt system. Thanks to all the staff in the School of Computer Applications for their great concern and friendship. Finally I wish to thank my husband Jinsong for his continuous encouragement.

ABSTRACT

In this thesis, a progressive refinement approach is proposed for the use of rule-based systems in real time applications. In this approach, the knowledge involved is constructed in a hierarchy of the levels. Part of this knowledge determines the time available for a decision. Progressive refinement allows the system to take corrective action by forming appropriate solutions within time constraints.

An expert system shell written in Prolog is constructed to implement the progressive refinement framework. An English like language is also constructed, which allows the time constrained rules for the system to be specified in a user friendly manner.

The framework can be applied to both discrete and continuous systems. An example is provided of the application of the framework to a discrete system. Experiments were also carried out to investigate the application of the framework to a continuous system, and their results are presented here.

Table of Contents.

Chapter 4 Framework for a Rule-Based Real Time System.

Chapter 1

Introduction

Chapter 1. Introduction

Real-time control systems are being used today in a greater variety of applications than ever before. Both small and large controllers are being used to perform increasingly complex tasks. The complexity is increasing not only in the number of functions to be controlled, but also in the kinds of factors that must be considered before a correct decision can be made. This increasing complexity of the tasks to be controlled has caused considerable interest in employing rule-based techniques for controller applications. Proper application of these techniques can result in more sophisticated control strategies for advanced applications.[R8]

The key feature of a real-time system is its ability to guarantee a response after a fixed time has elapsed. Traditionally, real-time control is implemented by mathematical control algorithms, such as proportional, integral and derivative (PID) control. The majority of the control loops in controllers are PID loops. They can make decisions quickly and accurately in simple situations. Control of process variables, such as temperature, pressure and so on, can be implemented in a single control loop. The parameters of the PID (or other type) of controller are selected initially to achieve both accuracy and a good transient behaviour. In practice, however, due to a variety of reasons, the performance

of the controller may deteriorate and hence tuning of the parameters becomes necessary. A few adaptive and auto-tuning controllers have entered the market also. By the tuning of the controller parameters, the deterioration of the system can be reduced in these controllers.

The rule-based approach offers an attractive alternative to dealing with complex control problems, at least at first sight. The rule-based approach can be described as an attempt to supply a controller with control knowledge expressed as rules and with an efficient inference engine able to apply these rules in a real time environment. The rule-based system can be used on the supervisory level, acting in the same way as an experienced control engineer. Recently, in some real-time control systems, a rule-based controller has been used in the control loop instead of the traditional control algorithm. However, the critical response time constraints of real-time system applications distinguish the rule-based real-time systems from more traditional rule-based systems (expert systems). Special hardware and software techniques have been used in modern rule-based real-time controllers to meet real-time constraints.

Examples of existing real time rule-based systems include G2 , PICON , and others. PICON was developed by Moore and others at LISP Machines Incorporated. It features a dual processor architecture where the symbolic manipulation resides on a LAMBDA LISP machine while the numerical processing needed for real-time analysis is performed by a Motorola 68010 processor. G2 was developed by Gensym and is available for a variety of computers ranging from powerful symbolic computing work-stations to general purpose microcomputers such as the IBM-PC. The symbolic manipulation is performed in a dialect of Common Lisp. The slowing down caused by garbage collection, which is a major problem in using Lisp, has been overcome according to Gensym, and as a result G2 is regarded as being more powerful than PICON.

Both PICON and G2 attempt to deal with the strict response time requirements of the control environment by partitioning the task between a symbolic processor to deal with non time-critical aspects and special hardware for the time critical component .

The approach in this thesis does not involve a partitioning of the task between different hardware systems. Instead, the rules which deal with the behaviour of the system are partitioned in a way which guarantees an appropriate response even in time-critical situations. This partitioning allows the system to progressively refine its response

to a given situation until the time appropriate for that situation has expired whereupon the response is output. The accuracy of the system's response depends on the time available to calculate it. In this respect the system can be thought of as mimicking an aspect of the problem solving behaviour of humans. A similar idea is implemented in the HEXSCON SYSTEM (Stanford Research Institute 1986). This system is used in military applications, in particular, dealing with multiple incoming missiles.

In this paper, we present a rule-based architecture for the control problem. Knowledge is constructed into several levels. At each level, an appropriate response can be obtained. Each successive level gives a more precise response, and also needs more time. Our goal is to develop a problem solver that, when it cannot find the optimal solution due to lack of time, will progressively generate acceptable solutions that meet the deadlines and the user's needs.

Chapter 2

Review of Rule-Based Real Time Systems

Chapter 2. Review of Rule-Based Real Time Systems

Implementation of a control system conventionally uses special purpose hardware or fast processors and a real-time control algorithm. But as these systems become more and more complicated, the difficulty of dealing with them using such traditional methods increases rapidly. Meanwhile, the field of Artificial Intelligence has produced tools and techniques applicable to the design and implementation of complex control systems. These tools include expert systems. In recent years, expert systems are gradually being used in real-time process control. This chapter reviews a number of approaches to the control problem.

2.1 PID and Adaptive Controllers.

In the early design of real-time process control systems, the control algorithm was often implemented directly in analogue hardware, which was used to implement a feedback loop with proportional, integral, and derivative control. Initial microprocessor based control systems typically implemented PID control using an algorithm. Tasks in the control algorithm usually include:

- sampling the variable being controlled using an analog to digital converter.
- comparing the measured variable y(k) with the desired or set value r(k) to calculate the error e(k) where k is the discrete time.

- using current and previous values of the error e(k) to calculate the output signal to actuator u(k).

- sending the signal to the actuator either in digital form or using a digital to analog converter.

A popular control algorithm is the PID (proportional, integral, and derivative) algorithm which involves calculation of the integral and the derivative of the error signal as well as a few multiplications and additions in order to compute the actuator input u(k),

$$u(k)=K_p*e(k) + K_i*I(k) + K_d*d(k)$$

where I(k) is the numerical integral of e(k) and d(k) is its numerical derivative. K_p,K_i and K_d are parameters which determine the performance of the PID controller. Poor estimation of these parameters will make the PID control system unstable and oscillatory.

In pressure and temperature control, PID control is usually sufficient. The control output may need to be updated only once every few seconds to once every minute. The

computation time required to implement a simple controller such as the PID is usually very small compared to the sampling interval time and hence does not pose any problem.

The parameters of the PID (or other type) controller are calculated initially, but due to a variety of reasons, the behavior of the controller may deteriorate and hence tuning of the parameters becomes necessary. A solution to this is the adaptive controller, where the parameters of a model for the process are estimated on-line. A self-tuning controller contains an identification algorithm that periodically updates model parameters. Implementation of self-tuning control using a microcomputer is possible for a process that is not too fast. But there also many problems associated with the implementation of adaptive control. Firstly, it is quite possible for the adaptive control system to become unstable due to poor parameters estimation. Secondly, it is also possible for a sudden change in the process to cause the performance of the adaptive controller to deteriorate.

2.2 Rule-based controller.

During the last few years the development of rule-based techniques has been one of the basic topics in AI research particularly in the context of expert systems. Figure 1 illustrates the basic concept of a rule-based system. The user supplies facts or

information to the system and receives advice and decisions in response. The rule-based system consists of two main components, the knowledge-base containing the knowledge in the form of rules and the inference engine which draws conclusions.

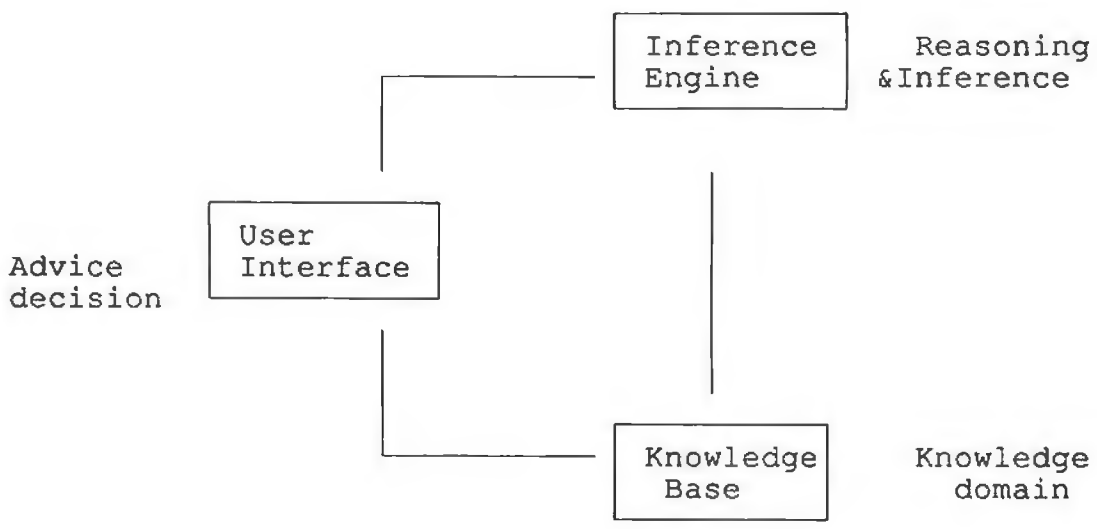


Figure 1. Basic concept of a rule-based system function

There are a lot of papers discussing the advantages ([3],[16],[36]) and the use of the rule-based approach. These include:

1) Knowledge is represented as collections of rules, which gives both a declarative programming style and a modular system where new rules can be added relatively independently.

2) Rules tend to provide an efficient way to categorize a process which is driven by complex and rapidly changing environmental situations.

3) It is possible for a set of rules to specify the reaction of a program without requiring explicit data in the knowledge base about the flow of control.

4) The use of rules also tends to simplify the determination of how a specific conclusion was reached.

These advantages have led to the application of the rule-based approach to a number of domains. Some attempts have been made to apply this approach in the real time domain. The expert system contains knowledge in the form of rules enabling it to diagnose the system periodically and to take appropriate action if deterioration in performance is discovered. But how long will it take to reach a conclusion ?

One definition of "real-time" is: "a strictly limited time period is available in which the system must produce a response to environmental stimuli, no matter what kind of algorithms it employs." An important issue facing the introduction of rule-based technology into real-time applications is the ability of rule-based real-time systems to meet deadlines. Typical approaches to real-time computing assume that a task's priorities, time and other resource needs are completely known in advance and are unrelated to those of other tasks, so that a control component can schedule tasks based on their individual characteristics. If there are more tasks than the system can process within the time limit, the decision about which tasks to ignore is simple and local. It can be based on task priority and the time needed. However, tasks in rule-based applications are interdependent because they search different parts of the solution space to solve related subproblems. Problems arise in this area because the length of the chain of inferences involved in arriving at a conclusion, and the amount of backtracking, can not be determined in advance. There is no way of knowing how long the system will take to arrive at a result. Both hardware and software methods have been used to meet the real-time system time constraints, such as special real-time languages, e.g. "PEARL", separation of symbolic and numeric calculations, development of new strategies of knowledge processing, and so on.

A short overview will be given of current rule-based real-time systems.

PICON

PICON (Process Intelligent Control) was developed by Moore and others at Lisp Machines Incorporated. This package is designed to operate on a Lisp machine interfaced with a conventional distributed control system, where as many as 20,000 measurement points may be accessed. PICON is a development tool for real-time monitoring and diagnosis of process control systems. The general functional capabilities of the system are:

- intelligent alarming, particularly on complex combinations of conditions which require expertise for proper interpretation.

- detection of possibly-significant-events by inference applied to heuristic rules about dynamic process conditions.

- focus inference, in which procedure rules of all priorities and all inference rules are enabled (scanned) for a particular process. In the typical case, a

possibly-significant-event (high priority procedure rule) would trigger a focus on the particular process, thus gathering information required for complete inference around that process.

- diagnosis, a backward chaining inference, triggered by a possibly-significant-event or by operator request. An explanation is then given of the resulting inference path.

PICON uses the LAMBDA machine with two processors running in parallel. A Lisp processor is used for the expert system, while real-time data access and certain low-level inference tasks are performed by a 68010 processor. The 68010 uses an integral multibus to communicate with the distributed process control system.

To increase computational efficiency, PICON employs two processors running in parallel. They proposed an inference strategy, which is supposed to imitate a human experts problem-solving approach. An expert process operator, during normal plant operation, will scan key process information. This is for purposes of monitoring control performance and detecting problems which may not cause explicit alarms. In PICON, the

same approach is modelled by applying heuristic rules about dynamic process conditions to detect possibly-significant-events (high priority procedure rule), which then trigger a focus on the process unit.

G2

The G2 system developed by Gensym is considerably more powerful than PICON. It serves as an environment for development and implementation of real-time expert control systems as well as for simulation of complex, distributed, process control or communication networks. One of the main features of G2 is that it is available for a variety of computers ranging from powerful symbolic computing workstations to general purpose microcomputers such as the IBM PC. The symbolic manipulation is performed in a dialect of common Lisp. The slowing down caused by garbage collection, which was a major problem in using Lisp, has been overcome according to Gensym. Speeds of response at the order of a few milliseconds are possible with G2 on appropriate hardware according to the developers. Another major feature is the ability to distribute knowledge and to have several real-time expert systems work together. At the heart of G2 is a real-time scheduler that allows reasoning about time-dependent events and variables. While

the system in itself does not include any expertise for tuning of controllers or diagnosing sources of problems, it offers the user the best available environment to represent such expertise in the form of a real-time expert system. G2, like PICON, communicates with various available distributed process control systems and hence the host computer does not perform the control function but acts as an active supervisor which can interfere to change the control strategy([R10]).

HEXSCON

HEXSCON (Hybrid EXpert System CONtroller) is a hybrid expert system for real-time control applications. It is developed by SRI(Stanford Research Institute) International, and mainly intended to deal with control problems encountered in military and advanced industrial applications. HEXSCON is claimed to include 1) a capacity of 5000 rules in a microcomputer system with 512k memory, 2) a response time of 10 ms to 100ms, 3) the ability to handle many objects (about 1000), and 4) the ability to continue functioning despite a lot of uncertainty.

In HEXSCON, there are three main features involved in improving the real-time performance of the system. Firstly, conventional logic controllers and knowledge-based

techniques are combined in the system. Thus many real-time operational decisions, particularly the simpler ones, can be made by conventional logic controllers quite adequately and rapidly. Secondly, a sort of inference strategy, called progressive reasoning, is adopted in the system. By using this strategy, the reasoning processes are divided into several levels. More sophisticated solution can be obtained at a higher level than at a lower level, but it needs more time. The system always tries to go to the level which is as high as that allowed by the time available, and therefore, the solution will be the best within that time period. Thirdly, knowledge used in the HEXSCON is represented in the form of rules, and all rules are compiled into a more compact form for use with the inference engine. Therefore, the execution time of the rules may be speeded up.

HEXSCON is implemented in PASCAL. The knowledge-base management software and the English-like knowledge base can be in a large machine, while the "compiled" knowledge, inference engine and conventional logic can be in a microcomputer.

SUMMARY

From above examples, we can see that different hardware and software approaches are used to provide a real-time capability in rule-based systems. These approaches include:

1) The adaptation of existing techniques to improve the real-time performance of the systems, such as the use of conventional algorithms and controllers in the HEXSCON system.

2) The use of specialized hardware and optimized software. For instance, in the PICON system, a specialized Lisp processor is used for the expert system and a 68010 processor used for low level processing. In G2 system, response time can be a few milliseconds by using appropriate hardware. In HEXSCON all the rules are compiled, so that they can be executed faster by the processor.

3) Some low level processing systems are separated from the expert system, and they are capable of concurrent execution. For example, two processors are used in the PICON. Only one of the machines or processors is for the expert system, and the other is used for intelligent communication or low level processing operations.

4) Some parallel techniques are adopted, G2 can distribute knowledge and let several real-time expert systems work together. HEXSCON can reason about multiple objects.

These approaches are often supported by an attempt to classify or priorities the rules used by the system. For example, in PICON, rules are assigned priorities when they are entered into the knowledge base. In HEXSCON, the knowledge was divided into two categories: conceptual knowledge and operational knowledge.

In this thesis, we propose a method which divides rules into several levels. An appropriate decision can be obtained at each level within a certain time. Each higher level will give a more precise response than a lower level, and also need more time. This approach allows the system to obtain the best possible decision within the time available. Unlike other systems which take a broadly similar approach, such as HEXSCON ([8],[11]), the time available to deal with changing situations is determined by rules in the system itself.

Chapter 3

Choice of Implementation

Chapter 3. Choice of Implementation

3.1 Rule-Based Expert System Languages and Tools.

Expert systems are computer systems where the knowledge which specifies what the system should do is kept in a separate knowledge base rather than being buried in a mixture of data structures and procedures. A more correct general term is 'knowledge based systems' - the expression 'expert system' implies that the knowledge relates to that of an expert, and that the system deals with the problem in a way analogous to that which an expert would use, but it has become common practice to ignore this distinction. In most systems, the knowledge is specified as a set of rules, which is stored explicitly in the system in the 'knowledge base'. Such systems are also called 'rule-based systems'. A rule-based system is divided into four main components, as illustrated in figure 3.1:

1) a knowledge base
2) an inference engine
3) a user interface
4) current state

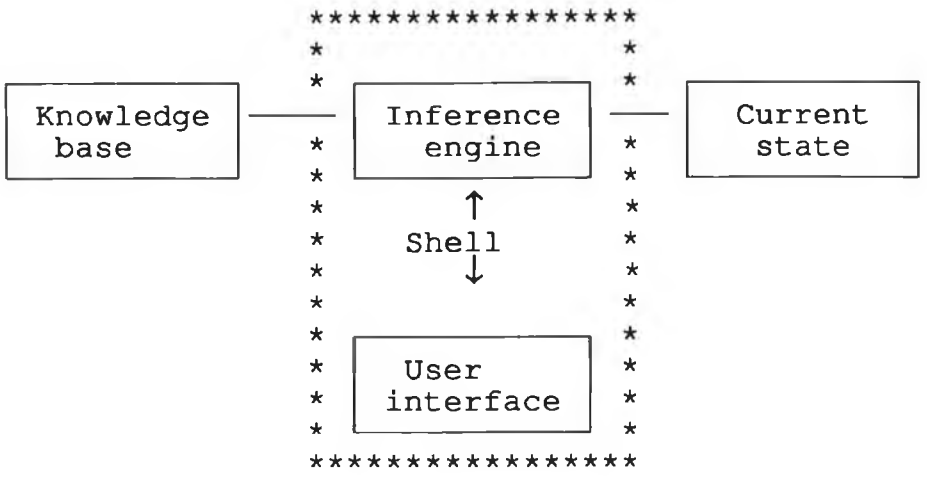


Figure 3.1 Structure of rule-based system.

The knowledge-base comprises the knowledge that is specific to the domain of application, including such things as simple facts about the domain, rules that describe relations or phenomena in the domain, and heuristics and ideas for solving problems in this domain. The language of if-then rules is the most popular formalism for representing this knowledge. An inference engine knows how to actively use the knowledge in the knowledge base and the current state of the world. The inference engine usually uses one of two strategies to do its work, either working forward from the facts about the current state of the world, known as 'forward chaining', or working backwards from a question, known as 'backward chaining'. A mixed strategy is also possible. Most

inference engines can cooperate with the user interface to provide an explanation of how a result was obtained, usually by giving the sequence of rules used. The user interface caters for smooth communication between the user and the system, and also provides the user with an insight into the problem-solving process carried out by the inference engine.

The inference engine and the user interface can be viewed as one module, usually called an expert system shell, or simply a shell for brevity. The knowledge base clearly depends on the application, but the shell is, in principle at least, domain independent. Thus a rational way of developing expert systems for several applications consists of developing a shell that can be used universally, and then plugging in a new knowledge base for each application. Of course, all the knowledge bases will have to conform to the same formalism that is 'understood' by the shell. So one approach to developing a rule-based expert system is to use an existing shell. In practice, however, it is fairly usual for the shell to be tailored to the needs of the application. Rather then using a shell, it is also possible to develop a system using a high level language. This will involve writing the inference engine and user interface but gives complete control over the behavior of the system. For a rule-based system which deals with real time situations such control over the behavior of the system is necessary, and is not adequately catered for in the available

shells. This thesis therefore considers developing such a system using a high level language.

The high level languages that can be used to build expert systems include the principal languages of artificial intelligence such as Lisp or Prolog, and conventional language like C, Ada, Pascal, Basic etc. It is a lot easier to develop rule-based systems in symbolic manipulation languages such as Lisp or Prolog. Conventional languages are oriented toward numerical processing while symbolic languages are oriented toward symbol manipulation. In addition, logic based languages such as Prolog include built in deductive capabilities.

Implementations of expert systems have been done using a variety of different languages. In the real time domain, PICON uses Lisp, running on a Lambda Lisp machine, to implement the rule-based system. G2 is also implemented in Lisp. The major problem in using Lisp is the need for garbage collection which results in the slowing down of the system. In HEXSCON, Pascal was chosen to implement the system, because it simplified construction of large programs, produced compact, fast code in the target environment, could be easily converted to Ada later for military applications and was well known by the developers.

In this thesis, Prolog was chosen. Prolog has many advantages as an application language for rule-based system. We will give a brief discussion of advantages of Prolog in the rest of this chapter.

3.2 Prolog.

Prolog was developed around 1970 by Alain Colmerauer and his associates at the University of Marseilles. Prolog (PROgramming in LOGic) implements a simplified version of predicate calculus based on Horn clauses. So Prolog is a logic programming language. The basis of Prolog is true logic programming using controlled, logical inferences. This makes Prolog well suited for many applications that require simulation of intelligence, including expert systems development, deductive data bases, language processing, robotics control, planning systems, and design applications. Prolog does away with familiar programming concepts such as "goto", and "do-for" and instead incorporates features required by intelligent programs such as advanced pattern matching, generalized record structures, list manipulation, assertional data bases, and depth-first search based on back-chaining.

Prolog has many advantages as an application language for expert systems, primarily due to three major features of the language: rule-based programming, built-in pattern matching, and backtracking execution.

3.2.1 Rule-Based Programming.

The Prolog language allows rules and facts to be expressed easily, and so provides a language in which the domain knowledge and the facts about the current state of the world can be readily represented. The symbolic nature of Prolog together with the declarative reading of Prolog clauses ensures that a flexible quasi-natural language interface can be easily supported.

In Prolog, a fact such as "mary is a child of patric " can be represented as "child(mary, patric)", where child is called the predicate or functor and the mary or patric are called the arguments. The arguments can be atoms, integers, variables or indeed other facts. See[1].

A rule consists of a head and a body. The body is made up of sub-goals which have to be proved true in order for the rule to be proved true. For example, the logical statement:

For all X and Y,

Y is a child of X if

X is a parent of Y.

can be translated easily into the formalism of Prolog.

child(Y,X) :-parent(X,Y).

The body of the rule is also called the condition part or the right - hand side of the rule. The head of the rule is also called the conclusion part or the left - hand side of the rule. If the condition **parent(X,Y)** is true then a logical consequence of this is **child(Y,X).** A clause is the name given collectively to both facts and rules. A predicate can also be defined by a group of clauses.

Prolog specifies known facts and relationships about a particular problem domain using the languages's symbolic representations of objects and the relationships between objects, thus creating clauses. Clauses are implications , and they make up the program with conclusions being stated first. Prolog expresses facts, rules, and relations in a fairly natural form which, in turn, produces clear, concise programs.

3.2.2 Built-in Pattern Matching and Backtracking.

Prolog has an inference mechanism, based on the resolution (Robinson 1965) rule of inference, which is built into the language and this inference mechanism can easily be used on the rules constructed by the developer. Prolog uses a special strategy for resolution theorem proving called SLD, which incorporates matching (roughly equivalent to unification), instantiation and backtracking. A variable is said to be instantiated when the object for which that variable stands is known. A variable is not instantiated when what the variable stands for is not yet known. The object in this case is usually an atom, string, integer or a structure. In practical programming terms instantiation means that the object is assigned perhaps temporarily to this variable. It is temporarily assigned because it can become uninstantiated during backtracking.

With matching, the general rules to decide whether two terms, S and T, match are as follows:

1) If S and T are constants then S and T match only if they are the same object.

2) If S is an uninstantiated variable and T is anything, then they match, and S is instantiated to T. Conversely, if T is an uninstantiated variable and S is not, then T is instantiated to S.

3) If S and T are structures then they match only if

a) S and T have the same principal functor, and

b) all their corresponding components match.

The resulting instantiation is determined by the matching of the components.

So given two terms, we say that they match if:

1) they are identical, or

2) the variable in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.

The following example from [R1] is an illustration. The terms **date(D, M, 1983)** and **date(D1, may, Y1)** match. One instantiation that makes both terms identical is:

1) D is instantiated to **D1**

2) M is instantiated to **may**

3) Y1 is instantiated to **1983**

This instantiation is more compactly written in the familiar form in which Prolog outputs results:

D = D1

M = may

Y1 = 1983

On the other hand, the terms **date(D, M, 1983)** and **date(D1, M1, 1444)** do not match, nor do the terms **date(X, Y, Z)** and **point(X, Y, Z).**

The inference mechanism in prolog is perhaps best explained using an example. Assume the following database where '%' represents comments.

```
% 1 john is a thief
  thief(john).

% 2 mary likes food
  likes(mary, food).

% 3 mary likes wine
  likes(mary, wine).

% 4 john like X if X likes wine
  likes(john, X) :- likes(X,wine).

% 5 X may steal Y if X is a thief and X likes Y and Y is valuable.
  may_steal(X, Y) :- thief(X), likes(X,Y), valuable(Y).
```

In response to the question "what may john steal" i.e. may_steal(john, X)? Prolog proceeds as follows:

1. First it searches through the database (top down) until it finds a fact or a rule to match the query. It finds it in the form of clause 5 which is a rule, marks this place in the database and X in the rule becomes instantiated to john. It then attempts to solve the subgoals on the right hand side of the rule in order left to right starting with thief(john) as X has been instantiated to john from the original query.

2. It initiates the search for the goal thief(john) from the top of the database and finds the fact thief(john). Prolog marks this place in the database also. It then attempts to satisfy the second goal in clause 5 which is effectively likes(john, Y).

3. The goal likes(john, Y) matches with the head of a rule (clause 4), the Y in the goal shares with the X in the head of clause 4, and both remain uninstantiated. To satisfy this rule, Prolog attempts to find a solution to the likes(X, wine) in clause 4.

4. The goal succeeds because it matches with **likes(mary, wine)** (clause 3) with X being instantiated to mary in clause 4 and Y being instantiated to mary in the second goal of clause 5 because X and Y share.

5. Having solved the first two goals in clause 5 it now attempts to solve the third and last goal which is effectively valuable(mary). But there is no fact to match this in the database and no rule to try and establish it so Prolog backtracks to try and find alternative solutions. During backtracking all variables which were previously instantiated become uninstantiated.

6. Prolog has kept track of all the places in the database were it has found solutions. It starts by trying to find an alternative solution to second goal likes(X, Y) in clause 5 which causes clause 4 to backtrack. But this too fails as likes(mary, wine) is the only fact that matches the right hand side of clause 4.

7. It then backtracks further to try and resatisfy thief(X) but this also fails causing the whole of clause 5 to fail. Since Prolog can find no other fact or rule to match the original query, the query fails and Prolog returns with the answer "no".

3.2.3 Declarative and Procedural Semantics of Prolog.

The meaning of a Prolog program can be regarded from the point of view of a) Its declarative semantics. b) Its procedural semantics.

Declarative Semantics

The declarative semantics is concerned with the relations defined by the program without considering how these relations are brought about. Prolog is not a purely declarative language since to use it properly, it is necessary to take account of the way in which it operates, and it also contains purely procedural constructs such as 'cut'.

Procedural Semantics

The procedural semantics determines how the output is obtained; that is, how the relations are actually evaluated by the Prolog system. Because Prolog uses a specific deterministic approach (SLD resolution), the result can be affected by non-declarative aspects of the programm such as the order of statements. Prolog also contains statements which help to control the inference engine. Perhaps the most important of these is 'cut', which is used to prevent backtracking.

Difference

The difference between these two meanings can be seen by example.

Consider a clause

mother(X,Y):- parent(X,Y), female(X).

The declarative reading of this clause is:

X is Y's mother if X is Y's parent and X is female.

The procedural reading of this clause is:

To solve problem "X is Y's mother", first solve the subproblem "X is Y's parent" and then the subproblem "X is female".

Thus the difference between the declarative readings and the procedural ones is that the latter do not only define the logical relations between the head of the clause and the goals in the body, but also the order in which the goals are processed.

Advantage and Problems

The advantage of Prolog's declarative semantics is Prolog expresses facts,rules and relations in a more natural form which, in turn, produces clear, concise programs. Also,it encourages the programmer to consider to a certain extent, the declarative meaning of programs relatively independent of their procedural meaning. The executional details is left to the greatest possible extent to the Prolog system itself. This ability of Prolog is considered to be one of its specific advantages distinguishing it from conventional languages.

This declarative approach indeed often makes programming in Prolog easier than in typical procedurally oriented programming languages. Unfortunately, however, the declarative approach is not always sufficient. In practice, the programmer should know how Prolog systems execute a program. Because the procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program, different order of clause and goals will cause program result variations, although they have the same declarative meaning.

Consider an example(from [R1]):

```
predecessor(X,Z):- parent(X,Z).

predecessor(X,Z):- parent(X,Y),
                   predecessor(Y,Z).
```

by swapping goals and clauses of the above example, we obtain:

```
Predecessor(X,Z):- predecessor(X,Y),
                   parent(X,Z).
predecessor(X,Z):- parent(X,Z).
```

The two versions of the program have the same declarative meaning, but not the same procedural meaning.

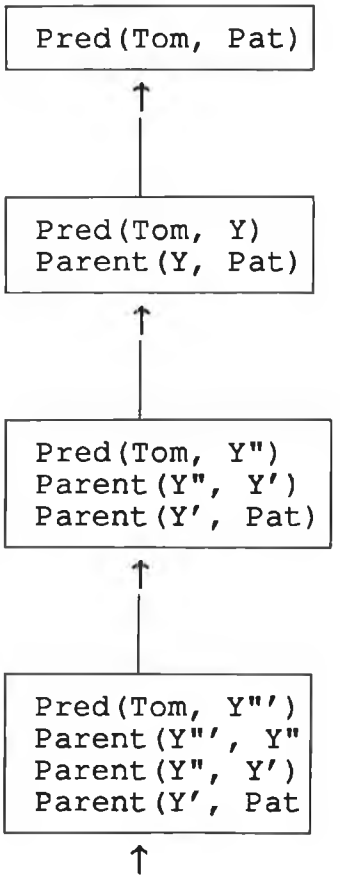
Suppose there are facts :

```
Parent(Tom, Bob)
Parent(Bob, Pat)
```

If we ask question whether Tom is a Predecessor of Pat using the two variations of the Predecessor relation above, we get different result. The first version answers 'Yes' while the Second causes a system crash. Figure 3.2.1 shows the corresponding traces for second program.

```
Predecessor(X,Z):-
 Predecessor(X,Y),
 Parent(Y,Z).

Predecessor(X,Z
 Parent(X,Z)
```



This example shows Prolog trying to solve a problem in such a way that a solution is never reached, although a solution exists. Due to the changes of ordering of clauses and goals, the system enter into an infinite sequence of recursive calls which eventually leads to a stack overflow. From the above example, we can see, although Prolog system has built-in procedure to execute program, the programmer should not ignore the procedural semantics when he or she concentrates on the declarative semantics of program. Nevertheless, the declarative style of thinking is characteristic of logic programming, with the procedural aspects ignored to the extent that is permitted by practical constraints.

3.2.4 Depth-First Search Strategy.

As the above examples illustrate, Prolog uses a depth first search strategy in looking for the chain of logical inferences which links the conclusion to the starting assumptions. This mechanism is not the same as the basically non-deterministic approach of the purely logical formalism, and can lead to different results, usually because of non-termination of the search. In effect, the search continues forever down one path of the search tree, and never encounters the solution, which may be readily accessible but on a different path.

Care must be taken in writing the program to avoid this possibility, and thus the programmer must have a clear understanding of the procedural semantics of the system. In real time applications, it is obviously essential to avoid non termination of the computation, and in addition there is a requirement to be able to anticipate the computation time required in making conclusions. This time requirement is perhaps the major problem in the application of Prolog in a real time environment, where the time for a given response is usually constrained.

3.2.5 Other Features

Prolog supports recursion e.g:

```
append([], L, L).
append([X|L1], L2, [X1|L3]) :- append(L1, L2, L3).
```

It is very adept at handling lists. Lists are a very useful data type which are common to both Lisp and Prolog. They can contain various different data types including other lists. In Prolog, the list is either empty or consists of a head and a tail [H|T]. It is basically a linked collection of nodes, with each node containing two pointers, one pointing to the value of that node, the other pointing to the rest of the list that is to the next node. A

special pointer 'NIL' indicates that the rest of the list is empty. In matching [H|T] with a non empty list, H is matched with the first, or head, element in the list. T is matched with the remainder of the list.

Using recursion it becomes easy to implement relationships such as 'member', 'append' and the other typical list operations. For example, the 'member' relationship "X is a member of the list L" can be writing:

```
member(X,[X|T]).          % X is a member of any list whose head is X.
member(X,[_|T]):-
  member(X,T).            % X is a member of any list if it is a member of that
                          list's tail.
```

This relationship then works as follows:

e.g. (1) member(a,[])

fails because [] can not match with [X|T] or [_|T]

(2) member(a,[a,b,c])

succeeds on matching with X,[X|T]

(3) member(a,[b,a,c])

succeeds after matching of a,[b,a,c] with X,[_|T] followed by match of a,[a,c] with X,[X|T]

Many operations on lists, such as list membership, concatenation, adding an item, deleting an item, sublist ect, are built into some versions of Prolog.

Another useful feature in Prolog is its ability to alter the structure of it's own programs during execution. This is done using the evaluable predicates (evaluable predicates are predicates which are built into the language) retract and assert (and variations on these predicates). The retract predicate allows you to remove a named predicate from the database, while assert allows you to add a new or changed predicate to a database. This ability to make run time changes to the program makes Prolog more flexible than static languages such as C or Pascal.

All these features of Prolog has made Prolog very popular as an AI programming tool.

3.3 Arity Prolog.

Prolog is readily available on both large IBM systems and a wide variety of personal computers. Before choosing a Prolog version, these factors should be considered:

1) The graphical capabilities for the user interface.

2) The database capabilities both from a programming point of view and for the manipulation of the flexible vocabulary.

3) The development environment.

We choose Arity Prolog as our language because it provides a standard Prolog language base together with some useful enhancement features.

Special features

Arity Prolog provides a number of control operators which help define the structure of a program. These include standard Prolog control operators, such as repeat-fail loops, recursion, and cut, and expanded Arity-Prolog control operators, like if-then-else constructs, snip and the case control operator.

Dialog boxes and windows

Prolog provides the programmable features of dialog boxes, windows and pop-down menus which can be used to form a powerful graphical user interface.

Language Interface

Embedded C in Arity-Prolog provides us with added flexibility in our programming tasks. In this thesis, a data acquisition signal processing system is used to interface digital and analog signals to the computer. The required driver software is written in 'C'. It is very efficient and effective.

Others

Arity Prolog provides excellent database features both at the programming level and at a low level. The low level features allow the flexible vocabulary to be manipulated quickly using specialized database manipulation and search facilities. It provides a good development environment in the form of an interpreter which incorporates a sophisticated debugger and good editing facilities. It also provides a good compiler to produce an executable version of a program developed in the interpreter. The compiler also detects and optimises tail recursion.

3.4 Some Problems with Prolog.

Prolog also has a number of disadvantages which have to be overcome in constructing the system.

As we said before, in a depth first search each particular branch in a tree is followed downward from left to right until the original goal is proved to be true or all the possible solutions are investigated. This method has the advantage that it is economical in the use of working memory and can be easily programmed using a stack, but has the disadvantage that for a large program database the search can be very time consuming if the solution lies to the right hand side of the tree, even though it is quite near the root.

As processing can take a long time, it is important to be able to interrupt to deal with a real time event. There is no interrupt service in prolog, but it is possible to provide one using the 'C' interface and BIOS(Basic Input/Output System) interrupt functions.

A bug was discovered in the Arity Prolog compliler, which does not handle the 'restart' predicate correctly. As a result, the system was developed using the interpreter.

Overall Prolog was seen to be the most suitable language with which to implement the framework here as it matched most closely the main features required for a rule-based real time system.

Chapter 4

Framework for a

Rule-Based Real Time System

CHAPTER 4 Framework For A Rule-Based Real Time System.

In the control strategy typically used with rule-based systems the time required to arrive at an output is indeterminate. It is therefore necessary to include in the control strategy for a real time system, an element which guarantees a response within a given time. In the framework put forward here, this is done on the basis of progressive refinement. This is based on analogy with human response patterns. In a situation where very little time is available a fast, approximate output is given, effectively a reflex response. If more time is available, a more accurate response is provided.

The framework involves structuring the knowledge concerning the system in a hierarchy of levels. At one level, relatively simple rules guarantee an approximate response almost immediately. Successive levels require further time, and provide a progressively improved response. Special control rules determine the time available in a given situation, and are used to cut off the reasoning process. At this stage the result from the last completed level of the hierarchy is output.

The expert system shell constructed is designed to facilitate partitioning the control problem into multiple levels, and can be applied to both discrete systems of the type usually dealt with by logic controllers and also to continuous systems.

4.1 Functional Description.

The framework put forward here is an experimental rule-based real time control system, which intends to deal with the control problem in a general purpose way. This system can be used for various cases by the addition of appropriate knowledge.

The elements of a rule-based system are shown in Figure 3.1. By separating the knowledge base from the inference engine and user interface, the system provides a special purpose tool designed for certain types of applications in which the user need only supply the knowledge base. The framework developed here can be used as a tool to create a rule-based real time control system. It is implemented in the PC environment, and constructed in a modular fashion.

The framework has two main components,

(1) A rule input system, which supports the definition of the inputs and outputs to the system, and the entry of rules in an English like language. This is written in Prolog.

(2) A monitoring and control system, which applies these rules to the system. This is written in Prolog and 'C'.

The framework includes a load_rules module, which translates English like If-Then structure rules into a Prolog form. Some examples are shown in Figure 4.1 and Figure 4.2. In figure 4.1, 6 if/then format rules are extracted from the example of controlling the manufacture of the contact lens developed later, which can be entered into the framework. The rules of prolog format translated from the format in figure 4.1 are shown in figure 4.2. With the load_rules module, a user without Prolog experience can use the framework to create his own rule-based real time system easily by adding the rules in a quasi-natural language format. The module "load_rules" includes two modules called user_input and user_output which enable a user to build up a description of the input variables needed and of the output variables desired for an application.

```
1: if
            angle >= 5,    % if stage is in
            angle =< 6,      mould input,and
            moudles = 1,     mould is available
            arm1 =  1,       mould input arm is
   then                      ready
            armin1 is 1;    then do input mould
2:          armin1 is 0;    otherwise do nothing%

3: if
            angle >= 3,    % if the input mould
            angle =< 4,      action is finished,
            armf1 = 1,       then retract arm,
   then                      otherwise do nothing %
            armout1 is 1;
4:          armout1 is 0;

5: if
            angle >= 9,     % if stage is mould
            angle =< 10,      remove,and arm is
            mouldins = 1,     ready,
            arm3 =1,         then remove mould,
   then                      otherwise do nothing%
            armin3 is 1;
6:          armin3 is 0;
```

The English-like format of the rules. Figure 4.1

Note:
A rule like:
```
if
    angle >= 5,
    angle =< 6,
    moulds = 1,
    arm1 = 1,
then
    armin1 is 1;
else
    armin1 is 0;
```
is broken into two rules like (1),(2), shown in figure 4.1, before it is entered into system.

```
armin1(moulds,angle,arm1,1):-
        angle >= 5,
        angle =< 6,
        moulds = 1,
        arm1 = 1.

armin1(moulds,angle,arm1,0).

armout1(armf1,angle,1):-
        angle >= 3,
        angle =< 4,
        armf1 = 1.

armout1(armf1,angle,0).

armin3(mouldins,angle,arm3,1):-
         angle >= 9,
         angle =< 10,
         mouldins =1,
         arm3 =1.

armin3(mouldins,angle,arm3,0).
```

The translated Prolog format rules Figure 4.2

To implement progressive refinement inference, rules are categorized into several levels. There are more rules contained in the higher levels, which allow the inference engine draw a more precise conclusion. Also it needs more time to reach its result. The inference engine uses Prolog's built-in backward chaining to search the problem space. To cut off the searching process when time runs out, an interrupt is employed. This interrupt is generated by an independent process running on the system under BIOS functions, and interfaced to the Arity Prolog system via 'C'.

The framework is constructed in a modular fashion in order to allow easy construction and maintenance of the shell.

4.1.1 The Modular Structure of the Framework.

Modularity and structured design are modern concepts of computer systems design to which considerable attention has been drawn over the last decade or so (notably by Jackson and Davis[R16],[R17]). A good programming language should support these concepts. One of the advantages of using Arity Prolog to develop the framework is that the system could be constructed and tested in modules. The main modules which make up the framework program can be seen in Figure 4.3. Modularity in program

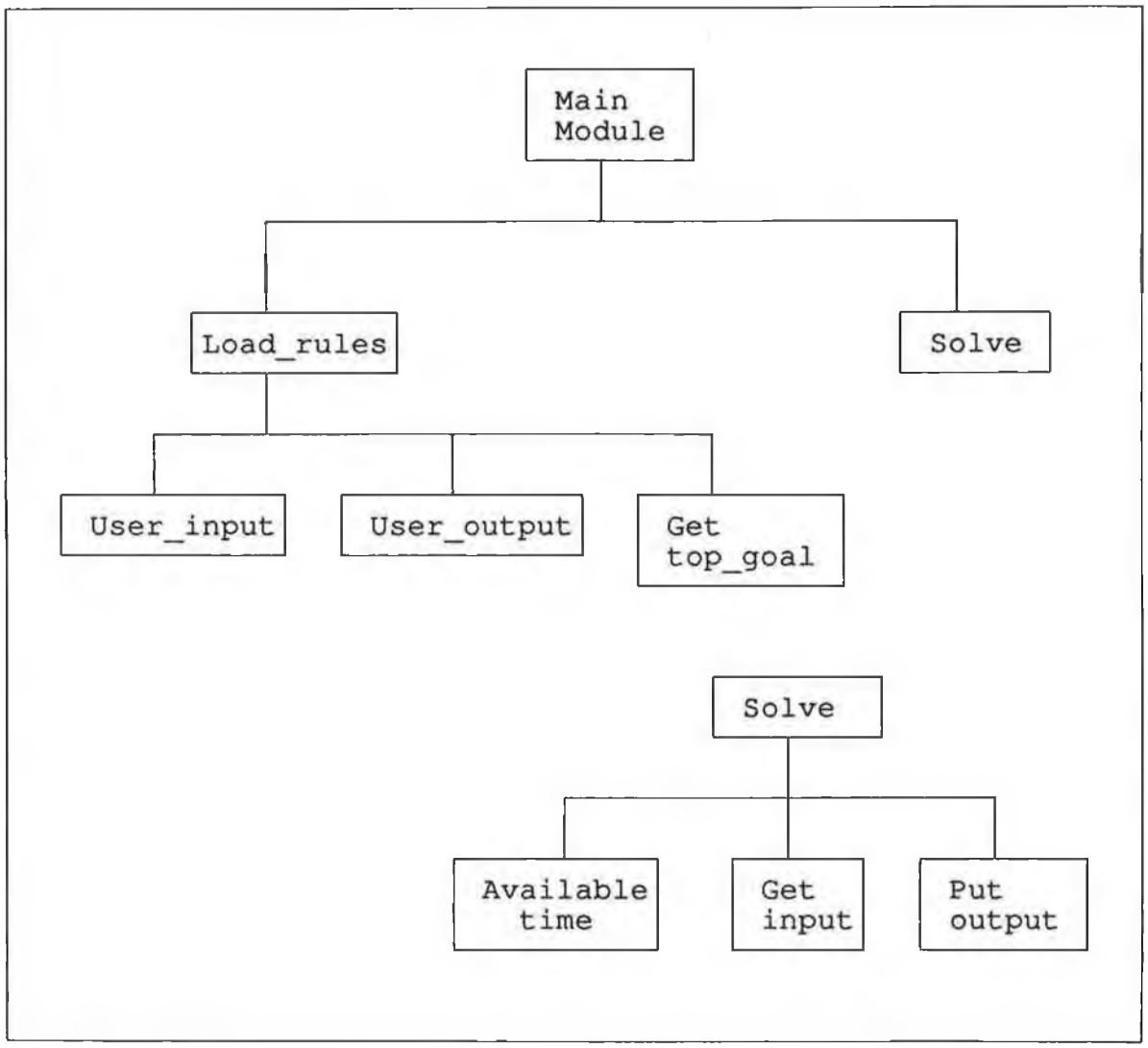


Figure 4.3 The modular structure of the framework

development is important as it reduces the problems of maintainability and debugging. Modularity also increases the readability of programs and leads to more structured programs. Prolog is a highly modularised programming language in that predicates are completely independent pieces of code. As there are no global variable in Prolog it also possesses the added advantage of having good data hiding features [R16] as predicates normally communicate through calls to each other and parameter passing.

The load_rules,user_input and user_output are the first created,tested and debugged modules in the framework. These were subsequently amended to reflect new ideas which arose from testing the system by a small example. The changes to the system are carried out with ease because of the modularity of the system. The other modules which represent additional tools for rule-based system development are added to the system at a later stage. These modules are created independently, and they also can use some predicates developed in the construction of the earlier three modules.

Although Prolog supports program modularity very well, Prolog programs can be difficult to debug because of the backtracking mechanism which is incorporated in the language. From a debugging point of view, modularity is especially important, as it allows one to localize the errors which occur and set the debugger to be activated in that module. This is especially important when a program is a large piece of software.

The modules of the user_input,the user_output and the load_rules are used to define the inputs and outputs of a particular application. The solve module controls the running of the system and gives an appropriate response by progressive refinement. Other modules aid the user to create a rule-based system meeting the user's needs. All these modules are described in detail in later sections.

4.2 The User Interface and Functions.

The user interface of any computer system is used to present information to the user of the system and to gather information from the user. "It has been estimated that half of any decent expert system should be devoted to communicating with the user..."[R18]. The problem which exists is that the users of the framework might have limited experience in the use of computers or Prolog language. So the functions of the tools provided by the shell must be inherently obvious. David Tong says [R19] " the success of an expert system often depends on the acceptance of the end users..." and that " too often the end user interface is neglected at the prototype stage. While end user details may not be of paramount importance at that time, establishing the basic end user requirements will help avoid a later switch in shells".

Arity Prolog has proved to be an excellent choice for developing this interface as it provides facilities to construct customized dialogue boxes which present information to the user of the system in a clear and easily assimilated form. The user interface in the case of the framework is a flexible interactive quasi-natural language interface. It can be easily used by inexperienced computer users to set up the rules appropriate for the application without help. David Tong also has views on this last point. He maintains that "...Knowledge base maintenance is best conducted by the expert himself who is likely to be inexperienced in knowledge engineering. The ease of use of the shell goes far in making this possible and without extensive training of the expert".[R19]

The user interface in the framework presents users with a conversational environment and demonstrates to users how the format for entering the information needed by system functions should be used.

4.2.1 Specification of the System.

The expert system shell containing the inference engine and user interface is designed for certain application areas. The framework is planned to deal primarily with systems where a significant time constraint exists, such as in control of discrete or continuous system. Usually, the rule-based system is required to make decisions depending on sampled data. So the first step for the framework is to set up the specification of the inputs and the outputs of the system being controlled. These are done by the modules 'user_input' and 'user_output'. The two functions can be found within 'shel.ari' and 'shel1.ari'.

The user interface provides a user interactive environment to create the input names, output names list and the parameters list.

Input List.

Before adding rule base to framework, the user should describe how many input variables the system needs and allocate names to each of these inputs. Also, the inputs can be analog and digital inputs. They are entered at different prompt. The input variable names list can be set up interactively by the user at the prompt displayed by user interface, which is shown in Figure 4.3 below.

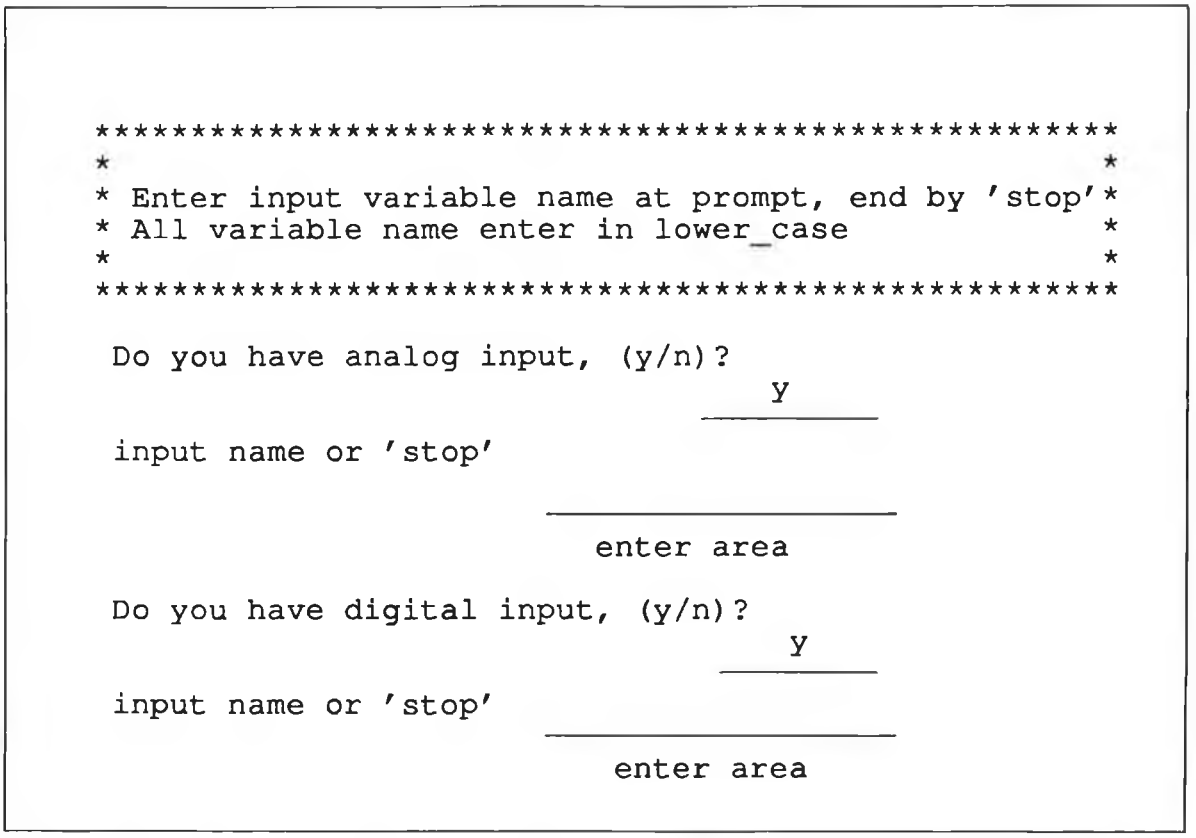


Figure 4.3

All names are entered in lower case, and finished by carriage return. The input names list set up will be displayed to user at the end. The interface gives a user the chance to change it. (See Figure 4.4).

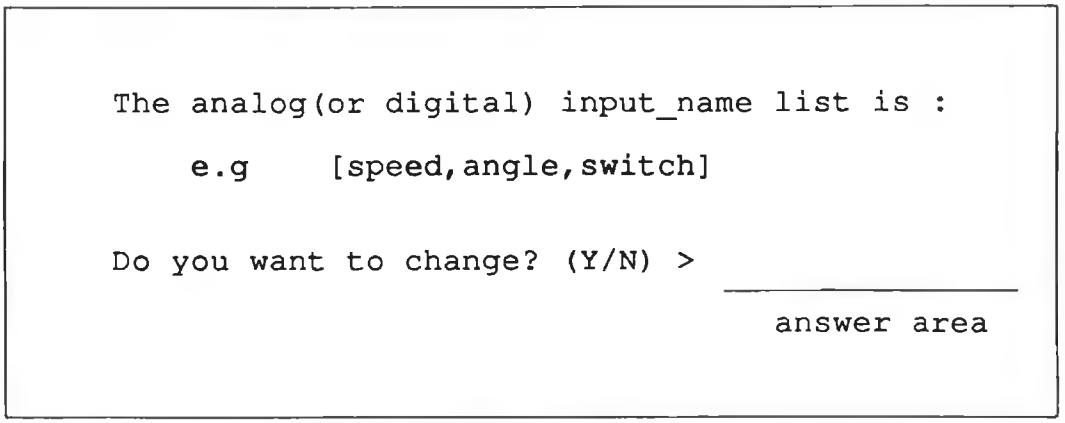


Figure 4.4

Output and Intermediate_output List.

After creating input variable names list, the user will be asked to enter output variable names of the particular problem. The user interface can be seen in Figure 4.5. The output of the system depends on certain input data, these data consist of the parameters of the output. When a output variable name is entered, the parameters of the output will be asked for interactively. The parameters list of the output is then set up See Figure 4.6.

```
**************************************************
*                                                *
* Enter output variable name at prompt,          *
*                      end by 'stop'.            *
*                                                *
**************************************************

Output name or 'stop' >force    (e.g)


The input is 'force',

If it is correct, press 'return'

            otherwise press 'n' >
```

Figure 4.5.

```
    ******************************************
    *                                        *
    * Enter the parameter of 'force' at prompt *
    *                    stop by 'end'       *
    *                                        *
    ******************************************


    Parameter or 'end' >

    The parameters list of 'force' is [Vspeed,Vswitch]
    Do you wanted to change? (Y/N) >
```

Figure 4.6

To simplify the description of the relationship between inputs and outputs, a third type of variable can be used. These 'intermediate' variables are treated syntactically like output variables. However the data of a intermediate variable won't be output from the system, but it may be referred to by output or other intermediate variables. The intermediate variable names list is created at the prompt shown in Figure 4.7. The parameters list is set up in the same user interface shown in Figure 4.6.

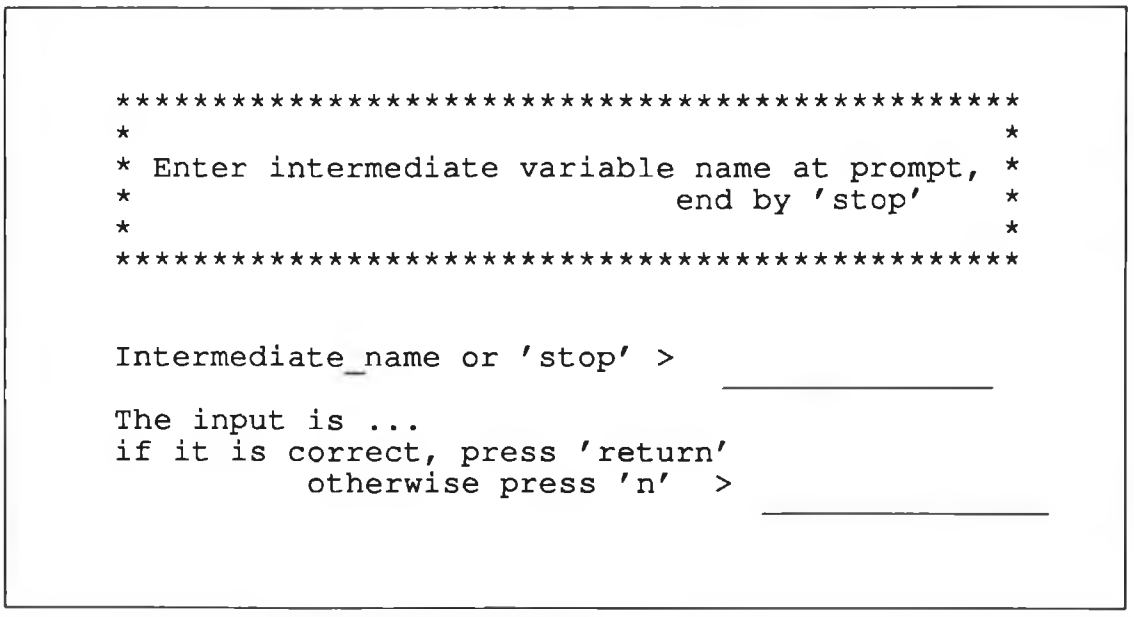


Figure 4.7

All these lists can be seen and changed at the user prompt. The parameter variable name will be translated into internal representation so as to facilitate the rules

construction. For example, speed is the parameter given by the user, it is translated into the Prolog variable name format 'Vspeed'. See Figure 4.6.

These lists set up are saved into database and used in building up the rules in the system.

4.2.2. Knowledge Representation.

The central part of any expert system is the knowledge base. Design of the knowledge base hinges on the knowledge representation. A good format allows efficient information acquisition and translation of internal representation. It facilitates search and inference. There are many different types of knowledge base representation mechanisms including frames, object oriented mechanisms and 'If_Then' rule structure.

In rule-based expert systems, the knowledge base contains the domain knowledge needed to solve problems is coded in the form of rules. As we discussed in Chapter 2, the rule-based approach has the advantages of being easy to read and to follow while some of the more complicated paradigms such as object-oriented expert systems are not so easy to comprehend. Also, it has a modular nature and some similarity to the human cognitive process. As a result rules are the most popular form of knowledge representation, especially for PC-based expert system. In the area of control system, the

problems needed to deal with is that the system makes a response to the changed situation of the system being controlled. The rule-based approach is especially useful for encoding information about cause and effect relationships. So the rule-based approach is a natural selection in this framework.

The representation chosen for the user interface also uses if/then rules. The syntax of the rule description is chosen to be as like English as possible, hence the interface is easy to use. Each rule contains conditions and action separated by key word 'if' and 'then'. The conditions are either made up of clauses joined together by conjunctions, or empty. The empty condition happens when there is a rule like:

if X > 0,

then Y is 1,

else Y is 2;

This rule is broken into two rules before it is entered into system. They are

(1) if X > 0,

then Y is 1;

(2) if

then Y is 2;

The rule (2) condition is empty. This facilitates translation into Prolog rules.

Each clause is made up of three "lexical items", that is variable name, operator and value. The general format for a clause is

name op val[ec]

where ec stands for an end of clause. It is a comma and a carriage return. The comma also means conjunction with the next clause.(See Figure 4.8).

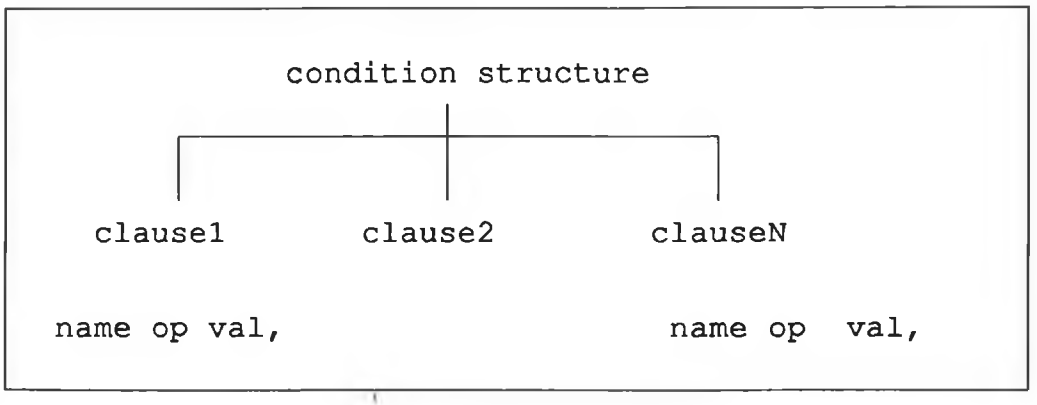


Figure 4.8

The action to be carried out is also described by a clause, which has the same structure as shown in Figure 4.8. The operator in an action clause is 'is' and assigns a value to a variable, while the operator in a condition clause is the comparison symbol, such as (>,>=,<,=<),etc. The rule structure can be seen in Figure 4.9.

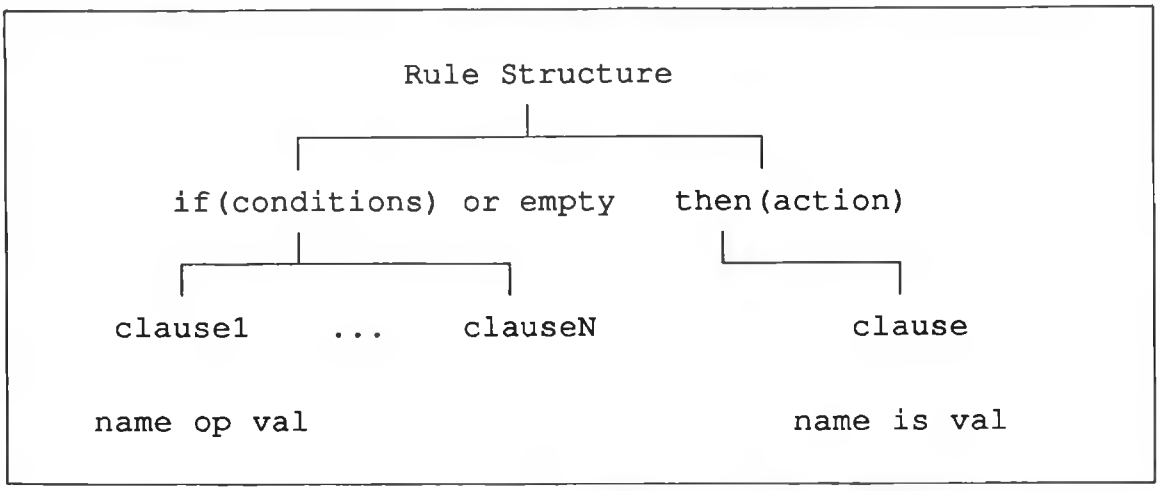


Figure 4.9

Many language constructs have "an inherently recursive structure that can be defined by context-free grammars".[R20] The if/then structure described above is recursive in it's definition and is defined by a context-free grammar. It is therefore possible to construct an efficient parser that determines if a statement is syntactically well formed. The parser which is constructed as a result of the above syntax will be discussed in section 4.3.

4.2.3 Response Time.

Unlike other knowledge-based systems, the time during which the system described here must give a response is decided by rules in the system. Users can define their own

rules for available time depending on their application requirements and add them to the system. The rules determining available time are represented in the same format as the other rules in the system. The value of the action of a time rule could be a constant or a simple arithmetical equation.

4.2.4. Data Acquisition.

The input and output signals are routed via a data acquisition system which provides 16 channels of digital input, 16 channels of digital output, 32 channels of analogue input, and 2 channels of analogue output. The procedures which deal with this system are written in 'C'. The input data is represented as attribute-value pair in the system. For example, the input data of speed is 20, it is translated into the form av(speed,20). Then this input data form is stored in the working storage in this format. The interface of the Prolog program with these procedures can be seen in process.c in Appendix A.

4.3 The Structure of the Knowledge Base.

The knowledge base of an expert system is usually expressed in terms of rules. Humans find it easy to think in terms of if/then rules and thus it is a suitable paradigm on which to base the framework. As the framework is implemented in Prolog, the choice of an 'if/then' structure is also convenient for implementation.

In order to support the inference engine, their knowledge is constructed into the hierarchy of levels. In the rule format, there is a identifier to specify which level the rule is in.

It has been attempted in the course of the framework's construction to present the rule base to the user in an English like format, while translating these rules into Prolog to exploit its powerful inference mechanism of resolution. The Prolog format of the rules is completely hidden from the user who, with the aid of the rule editor, enters the rules using an English like language format. Translating these rules into Prolog rules exploits the symbolic nature of Prolog, allowing the words entered by the user to have meaning by themselves. A rule structure is inherent in the Prolog language and this also allows easy translation of English-like rules. The built-in idea of 'is' and comparison predicates also greatly alleviates the problem of translating these English rules into Prolog rules.

4.3.1 Rule Format and Translation.

To use the framework to create user's own rule-based system to control continuous or discrete system , a user only need to add his rule base. It is easy for the user to construct rules in the if/then structure.

```
if
 condition1,
 condition2,              (head)
  ....
 conditionN,
then
 action;                  (body)
```

The framework provides the user with an interactive environment to enter the rules. The head and the body of the rule are indicated by the key words 'if' and 'then'. Both the head and the body of the rules consist of clauses which have the format of the English-like natural language described earlier.

There are two type of rule:

- 'leaf' rules which directly affect the output of the systems.
- 'node' rules which set up intermediate results.

e.g.

```
rule1:  if
          angle > 25,
          velocity is high, (parameter is velocity)
        then
          force is 8;

rule2: if
          velocity > 0,
          velocity < 2.5,
        then
            velocity is high;
```

Where rule1 is a leaf rule, rule2 is a 'node' rule which is referred by rule1. These two type of rules are distinguished by the input and output specification list of the system set

up earlier. The variable names of the clauses in the head of the rule must be either in the input-names list or intermediate-names list set up earlier. The variable names of the clause in the body of the rule must be in the output-names list or intermediate-names list set up before. Otherwise, the user will be told that the clause is incorrect. The user interface can be seen in Figure 4.10 below.

```
*************************************************************
*                                                           *
* Enter rule at prompt, end by 'eof'.                       *
* Each rule is entered in the format below                  *
*                                                           *
*      Prompt >                                             *
*                                                           *
* rule or eof >if                                           *
*   condition >speed > 2,           %no space after'>',     *
*   condition >condition2,          and before','%          *
*      .....                        %input in low_case.     *
*                                                           *
*   condition >conditionN,          %only one space         *
*   condition >then                  between two items%     *
*      action >force is 4;          %rule ended by ';'%     *
* rule or eof >eof                                          *
*                                                           *
*************************************************************
```

Figure 4.10.

The rules are entered by the user in the format shown in Figure 4.10. In order to distinguish between full stop '.' and decimal point, the rule is ended by semicolon ';'. The rule format shown in above should be strictly followed. For example, there is only one space between two items. Otherwise the rule will not be recognized properly by the load_rules predicate.

Before the rules can be activated in the expert system they must be translated into their equivalent Prolog form. The load_rules predicate translates these English-like rules into Prolog form rules while it reads these English-like rules. Before entering the rules, the user should notify the system which level the rules are in. This is done as shown below:

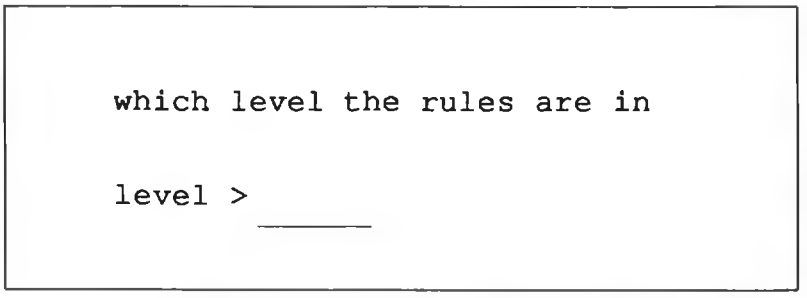


Figure 4.11

After entering all n level rules, the 'stop' is used to end this level rules. Then the user is started at this Prompt again and enter the next level rules. Each rule starts with key word 'if'. After that, the load_rules processes the condition's clauses iteratively. First, it checks if the variable name in a clause is legal name which exists in the input or intermediate name list. Then if the name is a input name, the prolog variable name of this input name is 'V + input-name'. For example, a clause is speed > 20, it will be translated into the Prolog form like that:

```
Vspeed > 20,
```

If the name is a intermediate name, the parameter list of the intermediate variable is retrieved. The clause is translated into the head of the node rule by adding variable parameter to parameter list . For example, there is a clause: temperature > 20. 'temperature' is in the intermediate names list, the parameter list of the name is [Vswitch]. it is translated into the Prolog form is:

```
temperature(Vswitch,Vtemperature),
Vtemperature > 20,
```

When the system reads in 'then', it starts to process the body of the rule. The body

of the rule only contains one clause. It could be a leaf rule directly affecting the output of the system or a node rule. The process is the same. By adding the value and level number into the parameter list retrieved, the Prolog form constructs the head of the rule. For example, the action is force is 1; It is translated into the Prolog form is: force(0,Vspeed,1):-. The example of the English like rules can been seen in Figure 4.11. Their Prolog form is shown in Figure 4.12.

4.3.2. The Structure of Rule Base.

In order to support the progressive refinement inference engine, knowledge is constructed into several levels. In the each higher level, it contains more rules than lower level. It will search out a more precise result and also take longer to reach it. Because there is no way of knowing how long it will take to reach a response in the rule-based system, it is important for the system to guarantee a consistent response before the time has expired. This is implemented by constructing knowledge a hierarchy of levels. The first level(0-level) which is also called panic level, will give a fast, approximate output, effectively a reflex response. Each successive level will need more time to give a output. The number of levels is decided by the user. The rules at each level are identified by the first parameter of the rules.

Because the knowledge base is separated from inference engine, there is need of a high level predicate which starts the knowledge base. Since it is not known what is being controlled, the framework will seek to solve a generic predicate called top_goal. The top_goal is generated depending on output list by the module get_goal. It retrieves the output variable's parameter list and constructs the body of the top_goal.

For example,

the output list is [force],

the parameter of force is [Vspeed,Vswitch],

the top_goal is : top_goal(N):-

```
av(speed,Vspeed),
av(switch,Vswitch),
force (N,Vspeed,Vswitch).
```

In the top_goal, first it gets input value which is stored in the working storage, then calls a leaf-rule force (N,Vspeed,Vswitch). Because the goal is the same in each level, the parameter of the top_goal 'N' is used to indicate calling different level rules. So the first rule in the knowledge base now is top_goal(N), the structure can be seen in Figure 4.13.

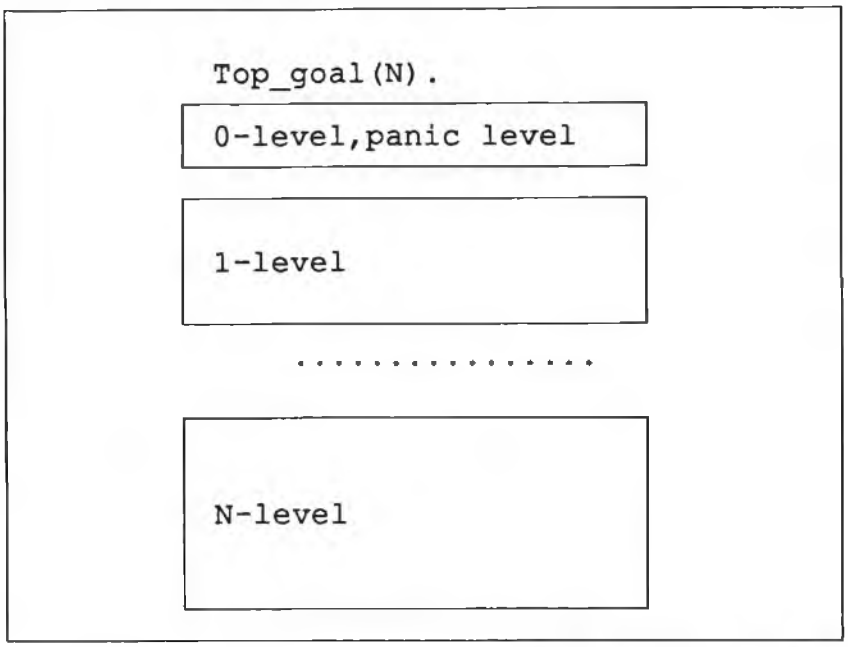


Figure 4.13.

4.4 Inference Engine.

The important issue facing the introduction of rule-based approach to real time application is the ability of the rule-based system to meet deadlines. Many efforts have been made in hardware and software discussed before. The real time ability of the framework is mainly improved by progressive refinement.

4.4.1 Progressive Refinement.

The reasoning model implemented is progressive refinement. In order to implement this inference strategy, knowledge is divided into several levels. The structure is shown in Figure 4.13. Assuming that a new piece of information comes into the system, the system will analyze its data and compare with previous data, then determine how much time is available before an action must be taken. After that, the system starts inferring from the default level, in which it sets up the default 'panic' response ready for output. At the end of each of level, the system updates the result in the output buffer. When time has run out, an interrupt signal is generated, and the system will stop progressive refinement processing, and outputs the result in the output buffer. This progressive reasoning method means that the problem solver, to meet its deadlines, could make a rough pass at solving the problem at default level, then use any remaining time to incremental refine this solution at still higher levels. The output will range from a rough approximation to a precise response depending on the time, but will be self-consistent and represent an appropriate response given the time constraints.

4.4.2 Interrupt.

In order to cut off the progressive refinement process when time runs out, an interrupt mechanism is required. The interrupt function is implemented by using the BIOS interrupt functions, in particular 'Ctrl-Break'.

The first software level in the computer is the BIOS (Basic Input/Output System). This software forms the lowest-level machine with which user normally will deal. It insulates higher levels of software from possible hardware changes in the computer and provides a defined set of services as a base for higher software levels. Since Arity Prolog doesn't have interface with BIOS, the interrupt handling program is written in Microsoft C.

The control flow of the framework is as follow:

1) sample the information from the system being controlled.

2) call the rule deciding the available time, and set real time clock.

3) start progressive refinement processing. The output buffer will be changed at the end of each level.

4) when time is up or no rules are left, the result up to date is output.

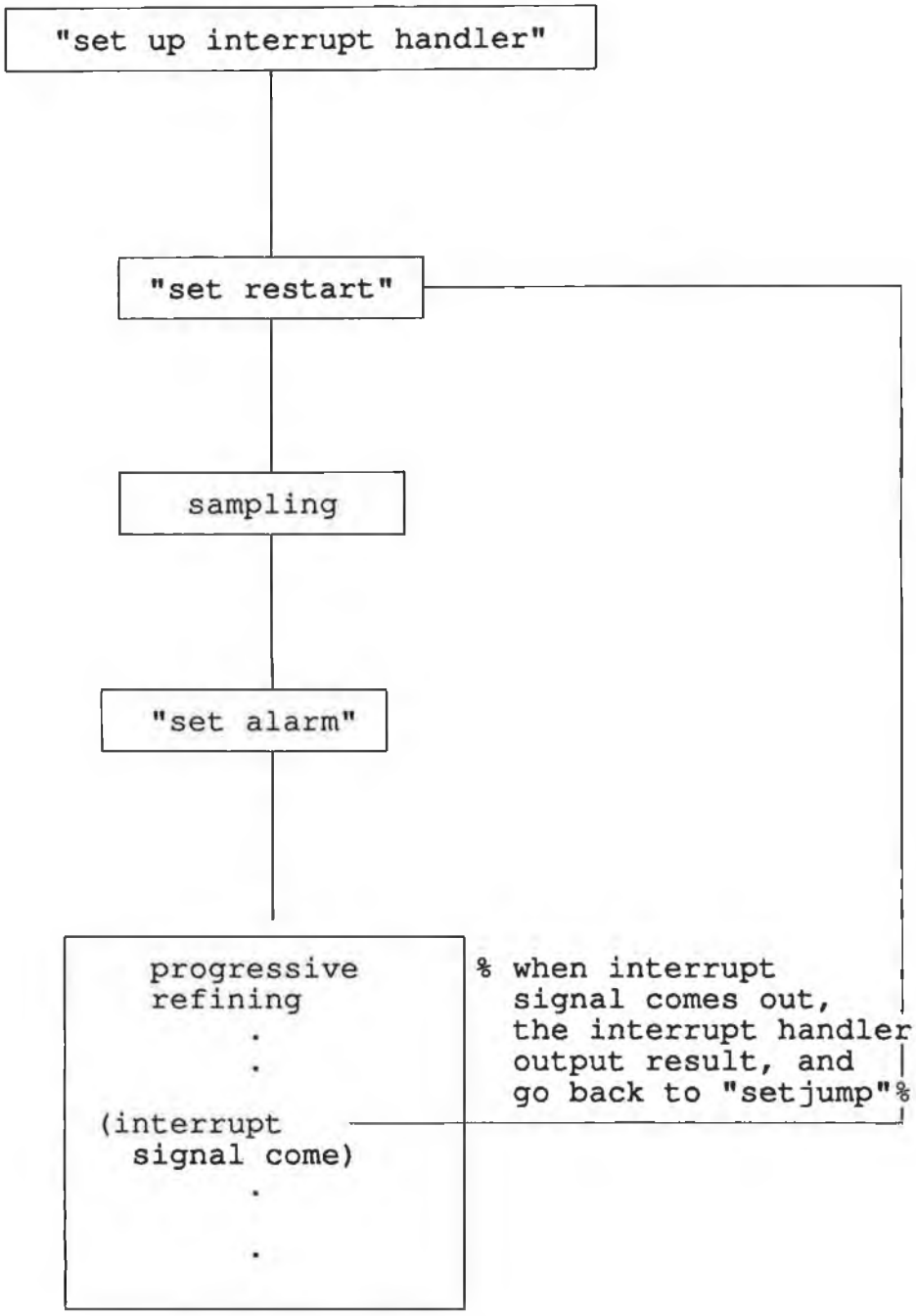
5) go back to the step 1).

Depending on this control character, the functions required of the interrupt mechanism are:

1) stop the progressive refinement program and output the result in the buffer.

2) go back to step 1) above.

To achieve these functions, the Dos interrupt mechanism for allowing the user to break into a running program is used by a C language program which generates an artificial Ctrl-Break interrupt when time runs out. This C program also uses the timer interrupt in the computer to determine how much time has elapsed. In Arity Prolog the restart point for a Ctrl-Break interrupt can be specified using the 'restart' predicate, and so cause the main program to restart from a chosen point. The interrupt is generated when time runs out, and the program then continues from restart, causing the above functions to happen. As a result, the interrupt mechanism works in this way:


"set up interrupt handler"
"set restart"
sampling
"set alarm"
progressive
refining
.
.
(interrupt
signal come)
.
.
% when interrupt
signal comes out,
the interrupt handler
output result, and
go back to "setjump"%

4.4.3 Backward Chaining.

The inference and control mechanisms are designed to manipulate the knowledge built in the knowledge base. Most rule-based systems use either backward chaining, forward chaining, or a mixture of both. Generally, backward chaining systems are most commonly used in consultation systems and for diagnostic and monitoring problems. They are good for solving structured selection types of problems.

The purpose of using backward chaining is that Prolog has a built-in backward chaining inference engine, like automatic backtracking, which can be used to partially implement the framework. The disadvantage of backward chaining is that backward chaining facilitates a depth-first search, while forward chaining is good for a breadth-first search. With the depth-first search, the length of the chain of inferences involved in arriving at a conclusion and the amount of backtracking can not be determined in advance. There is no way of knowing how long the system will take to arrive at a result. With this problem, the progressive inference will guarantee a response before time allowed runs out.

The inference mechanism of the framework works as follows:

1. After sampling input data, the system calls the time rule to determine how much time is available before an action must be taken. Call the function "set alarm" to set up the real time clock.

2. The inference using backward chaining (goal-direct-search) attempts to infer a output value for a specified parameter by testing rules.

3. It starts searching the rule base for the top_goal(0) at panic level. It will set the results to the output buffer. This level will guarantee a quickest response even when the available time is a very little.

4. Then it goes down to the second level top_goal(1). If the time runs out, the output buffer will give out, the system goes back to first step. Otherwise, at the end of the level, the result got from this level will replace the former result in the output buffer.

5. If there is still time left, the system goes to next level, until no rules are left or time is out. The output buffer will give out, then the system goes back to first step.

4.5 Summary.

The framework is a rule_based system development tool for real time systems. It provides the means to support a flexible English-like natural language interface. The users can enter their specification of the system and knowledge base in interactive environment. The if/then English-like rules entered by the users are translated into Prolog rules. The ability of real time of the system is mainly improved by progressive refinement which dividing knowledge into several levels. It can be used to create a rule-based real time system which will guarantee an appropriate response to meet time constraints.

Chapter 5

Testing the System

Chapter 5 Testing the System.

The purpose of this chapter is to show how the real time rule-based framework described in the previous chapter can be tested for discrete applications. For this purpose three sample sets of rules have been set up. A test rig has also been constructed which provides switches and lights as inputs and outputs for connection to the data acquisition system. A number of variable voltage sources are also available.

In setting up the rules, all input and output names are defined at the start of the system specification. The order in which they are defined determines the physical input output channel to which each one is assigned. Any intermediate variables used in calculations are also defined at the start of the program. The variables on which both intermediate and output variables depend are also identified to help avoid errors.

For testing purposes it is necessary to check that the time-based progressive reasoning works correctly. This is difficult to do with a small set of rules, as such systems of rules execute so quickly that the final output is available before time runs out. To test this aspect of the system, a special predicate was included in the Prolog file generated by the translator, as described below.

To give a more comprehensive test of the system, a large set of rules describing a spincast machine for manufacturing contact lenses was also set up.

5.1 Example 1.

This example monitors the input voltage and uses it to determine the time available. Three levels of rules are used.

At Level I, the output appears on light 1 and corresponds to the setting of switch 1. Lights 2 and 3 are turned off.

At Level II, the output appears on light 2 and corresponds to the setting of switch 2. Lights 1 and 3 are turned off.

At Level III, the output appears on light 3 and corresponds to the setting of switch 3. Lights 1 and 2 are switched off.

The time available is based on the input voltage, the higher the voltage, the less time is available.

The original rules, and their translation into Prolog, are given in Appendix B.1.
When these rules are translated and run, they work so quickly that they always have time to complete Level III, outputting the appropriate value on Light 3.
To test the time behaviour, we modify the Prolog version of these rules to include a delay at each level, as described in the next example.

5.2 Example 2.

To test the time behaviour of the system, it is necessary to ensure that the Level 2 and Level 3 rules are not fully executed before the available time runs out. This would normally require large and complex rule sets for Levels 1 and 2, so to avoid this, a special extra predicate is added by hand to the Prolog version of the Level 1 and Level 2 rules, to ensure that each of these levels takes two seconds to complete. The rules used are the same as in Example 1.

The extra predicate uses the Arity 'time' function, which gives the current time in the system to 0.01 seconds.

It is set up as follows

```
time(X),          %records current real time in X.
-------           %arbitrary sequence of instructions
-------           ...
waitsecs(X,2),    %waits for 2 seconds after time X.
```

The Arity predicate **time/1** returns a functor of the form **time(H,M,S,Hs)** where the parameters represent the current hours, minutes, seconds and hundredths of seconds on the system real time clock.

The definition of 'waitsecs' is :

```
waitsecs(time(H1,M1,S1,Hs1),S) :-
        repeat,
        time(time(H2,M2,S2,Hs2)),
        S1 is (H2 - H1)*3600 + (M2 - M1)*60 + S2 - S1,
        S1 >= S.
```

By inserting this predicate manually in the Prolog code generated by the translator for a particular level, the time required to complete this level can be stretched arbitrarily, depending on the value of the parameter S.

The rules set up for time in the example are

Voltage < 4 Volts	:	Time available 6 seconds.
Voltage >=4, <8	:	Time available 3 seconds.
Voltage >= 8 Volts	:	Time available 0 seconds.

When this system is executed the effects of the different levels can be seen by changing the input voltage, thereby allowing more or less time.

5.3 Example 3 : Controlling a Machine for the Manufacture of Contact Lenses.

This example shows the construction of a set of rules to deal with a reasonably complex industrial machine. It tests the capability of the translator program, and demonstrates the functionality of the system for this type of problem.

This machine manufactures contact lenses, creating the inner lens surface curve by spinning the liquid monomer while applying ultra violet light to cause it to polymerise. The outer lens surface is determined by the mould into which the monomer is poured. An individual disposable mould is used for each lens.

The machine consists of a rotating table containing a number of mould spinners. As the table rotates, each mould spinner is brought past a number of stations in turn, where various operations are carried out. These include:

1) Loading the mould in the spinner.
2) Injecting the appropriate amount of liquid monomer into the spinning mould.
3) Curing the monomer under ultra violet lamps.
4) Removing the mould and lens from the machine.

After removal from the spincast machine the mould and lens go through a sequence of further processes, including removal of the lens from the mould, disposal of the mould, testing of the lens, and packaging and labelling of the lens.

The operation of the spincast machine must achieve a high level of accuracy. Any deviations from the correct performance, which could effect the accuracy of the lenses produced or damage the machine, must be acted on immediately. The critical machine settings include:

1) Mould spinner speed — if this is incorrect, the wrong curvature and hence power will be result.

2) Table rotation speed — this determines the time spent under the ultra violet lamps and hence the degree of curing .

3) Monomer quantity — the correct amount must be injected.

4) Gas pressure and flow — gas jets are used to ensure an even dispersal of monomer within the mould.

In addition, a number of critical faults can damage the machine unless immediate action is taken. These include:

1) Injection of Monomer when no mould is present.

2) Attempting to insert mould when previous mould has not been ejected.

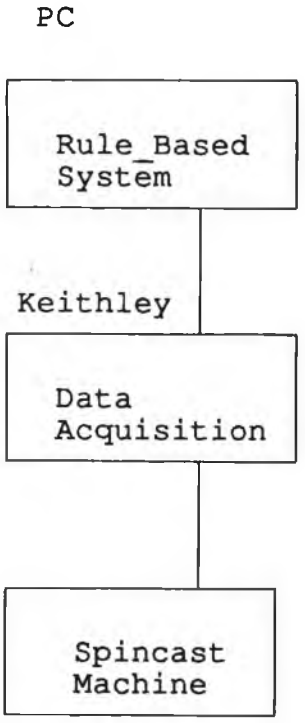
Other fault conditions include attempting to insert mould when no mould is available, and failure of lamps or other parts on the system.

Operator control of the machine provides for setting the different rotation rates depending on the lens type and power.

The rule_based control system will function as a process monitoring, and alarm device, continually monitoring all critical process variables to ensure that they remain within specified tolerance levels. The particular stage of the operation is determined by the angle through which the table has rotated.

The system will give a message to alert the operator whenever attention is required (eg. moulds needed, or gas pressure low), thus both improving quality control and easing the operator load.

The rule_based controller will also sequence and control the various process steps, such as input mould, do monomer injection and remove mould. The signals required for this purpose are provided through a data acquisition system manufactured by Keithley Ltd., which provides 16 digital inputs, 16 digital outputs, 32 analogue inputs and 2 analogue outputs. The configuration of the rule_based control system for spincast machine can be seen below.



The Implementation of the System.

In this implementation, the spincast machine is replaced by a system of switches, voltage sources, and indicators which allow the output to be monitored and the various inputs to be set at will. The framework built before is used to create the rule-based control system. This involves identifying the appropriate rules, inputting these using the input facilities of the framework, and then allowing the system to function via the data acquisition system.

The inputs and the outputs used are described below:

Input variables.

1) Analogue inputs:

angle	In order to identify the current position of the table, an angle between 0 and 360 degrees is available through the data acquisition system. The current stage of the process is determined by stage = angle mod(360/N), where N is the number of spinners in the table, typically 10. Input as a voltage from 0 to 10V.
tablerate	Indicates spin rate of the table. Input as a voltage from 0 to 10V.
spinners	Indicates spin rate of the spinners. Input as a voltage from 0 to 10V.

2) Digital inputs:

moulds	Indicates if a mould is available for insertion in a spinner. 1 if yes, 0 if no.
arm1	Indicates if mould input arm is ready, that is fully retracted. 1 if yes, 0 if no.
armf1	Indicates if mould input arm is fully inserted. 1 if yes, 0 if no.
arm2	Indicates if monomer injector is ready, that is fully retracted, 1 if yes, 0 if no.
armf2	Indicates if monomer injector arm is fully inserted. 1 if yes, 0 if no.
arm3	Indicates if mould remove arm is ready, that is fully retracted. 1 if yes, 0 if no.
armf3	Indicates if mould remove arm is fully inserted. 1 if yes, 0 if no.
monomerl	Indicates if monomer level is satisfactory. 1 if right, 0 if not.
gasp	Gas pressure is used to ensure an even dispersal of monomer within the mould, incorrect gas pressure will affect the precision of the contact lens. 1 if correct, 0 if incorrect.

gasf	Gas flow is also used to ensure an even dispersal of monomer within the mould, incorrect gas flow will also affect the precision of the contact lens. 1 if correct, 0 if incorrect.
mouldins	Checks if mould is in spinner at remove stage. 1 if yes, 0 if no.
type1	Used to indicate lens type, in range 0 to 7
type2	Used to indicate lens type
type3	Used to indicate lens type

Output variable.

Digital outputs

armin1	Causes insertion of the mould input arm.
	1 means to do inserting action, 0 means to do nothing.
armout1	Causes retraction of the mould input arm.
	1 means to do retracting action, 0 means to do nothing.
armin2	Causes insertion of monomer inject arm.
	1 means to do inserting action, 0 means to do nothing.
armout2	Causes retraction of the monomer inject arm.
	1 means to do retracting action, 0 means to do nothing.
armin3	Causes insertion of the mould removal arm.
	1 means to do inserting action, 0 means to do nothing.
armout3	Causes retraction of the mould remove arm.
	1 means to do retracting action, 0 means to do nothing.
alarm	Indicates a fault is detected.
	1 if yes, 0 if no.

3) Operator outputs

opmes Operator message, tells operator what the fault detected is.

Constructing the Rule Base

Rule-based control of the operation of the machine can be implemented in three levels.

Level 1:

Checks for failure of a part of the mechanism and sets up the fail safe outputs.

Level 2:

Checks for operational state and carries out appropriate action.

Level 3:

Checks for consistently between the process parameters such as lens type, power and the various spin rates.

In order to preserve the smooth operation of the machine, it must provide the appropriate output rapidly. If the limited time available runs out before an exact

conclusion is reached, the result from the previous level is used as output. The time available depends on the rotation rate of the table, and is set up by the time determining rules. In general, the appropriate action must be taken within 0.5^0 of travel of the table. If the frequency of rotation is f rotations per minute, then the time available in seconds is:

$$T = 60/720f = 1/12f$$

where f is the table rotation rate in degrees per second.

Level 1 Rules:

The rule_based controller first detects a fault condition, whenever the process variables are sampled. When this happens, the machine is halted, alarm is on, and a message is displayed in the screen indicating the source of the problem. These fault conditions include:

- When the state of the spincast machine is in mould input, there is no mould available.

- When the state of the spincast machine is in monomer injection, there is no monomer available.

- When the state of the spincast machine is in mould remove, there is no mould in the spinner.

If one of these things happens, the system should stop the machine immediately. One of the English_like rules can be seen below.

```
eg.   if
          angle = 1,                      %stage is= mouldinput.
          moulds = 0,                     %mould is not available.
      then
          opmes is no_mould_available;    %operator message.
          alarm is 1;                     %alarm is on.
```

The results of the first level is that alarm is on, operator_message is displayed and all others are off.

Level 2 Rules:

When all parameters, like table_rate, spinner_speed, gas pressure, and gas flow so on, are set up, the spincast machine is started up as normal. In each cycle time, the machine's normal operation are inputting mould, injecting monomer, passing to UV lamps

automatically, and removing mould. All the operations are controlled by the rule_based controller. After checking fault condition, if the spincast machine works normally, time should still be available. The reasoning of the system will go down to second level and check the state of spincast machine, then take relevant actions. These include:

- When the state of the spincast machine is in mould input, check if the mould is available, and input_mould's machine is ready. Then the mould is input.

- When the state of the spincast machine is in monomer injection, check if monomer is available, and the machine for injecting is ready, then the monomer is injected.

- When the state of the spincast machine is in mould remove, check if there is a mould in the spinner, and the machine for removing the mould is ready. Then mould is removed.

These actions maintain the smooth working of the machine.

Level 3 Rules:

At level 3, the system is supposed to have enough time to give an accurate response. The control of the spincast machine now includes taking into account the various consistency rules as well as checking the parameters of the machine, like gas pressure, gas flow, and lens type, to guarantee the accuracy of the lenses produced.

Different lens types allow different spinning speeds and table rotation rates. These variables should normally remain within an acceptable tolerance range. Parameters outside an acceptable tolerance range cause the controller to halt the spincast machine, sound its alarm, and display a message indicating the problem. An example of the relation between the lens type and the spin and rotation rates is given below.

The lens type constrains the Power and Monomer types, which in turn constrain the spin and rotation rates.

Type	Power	Monomer type	spin	table
A1	0-20	Clear,Tint,Aqua,Blue,Brown,Yellow	2-10V	4-10V
A2	0-20	Clear	2-10V	4-5V
A3	0- 9	Tint	2-5.6V	5-6V
A4	0-12	Tint	2-6.8V	5-6V
A5	1-6	Tint	2.4-6.4V	5-6V
A6	0-5	Tint,Aqua,Blue	2-4V	5-8V

The rules used in level 3 also provide for the smooth running of the machine as in level 2. An example of some level 3 rules is given below. All rules in the rule base are listed in Appendix B.

eg.

```
if
        checkss = 1,                        %if spin rotation rate is not
then                                                correct, alarm is on,
        alarm is on;                                tell operator fault message.
if
        checkss = 1,
then
        opmes is spinner_speed_incorrect;
if
        lenstype = a1,
        spinner_speed >= 2V,
        spinner_speed =< 10V,
then
       checkss is correct;
```

After adding the rule base, the specification of the system's input and output variables and the rule for available time within framework, a rule-based control system for spincast machine is created. The control system starts by sampling the information. It then activates the time-rule and gets available time. Depending on the information

sampled, the system starts searching from level 1 using backward chaining. If there is still time left at the end of the level, it will change the result and go down to next level until time runs out and interrupt signal comes out or there is no rule left to do. At that time, the result is output.

The rules given in Appendix B.2 were constructed to describe the system. They were translated and executed. Various fault and other conditions were set up on the test rig, and the system performed to specification.

Conclusion.

The framework can be successfully applied to control of a process involving discrete steps. The description of the process using rules is easer and more intuitive to set up than with conventional systems for this purpose, such as the ladder diagrams and other notations used with programmable logic controllers. The multi-level structure of the system guarantees an appropriate response even when the calculations needed for an exact response can not be carried out in the time available. This may arise due to the inherent complexity of the rules being used, or alternatively due to the processor being heavily loaded with other tasks, as might occur in a Unix or similar environment.

Chapter 6

Progressing Refinement with Continuous Systems

Chapter 6 Progressing Refinement with Continuous Systems.

An investigation was carried out of the possibility of applying the progressive refinement approach to the control of a continuous system.

The approach envisaged was based on the idea of using a progressively refined search to identify the control value that should be output. In such a system, the different levels would essentially try out the different possible outputs at different levels of graininess. If little time were available, only a small selection of outputs spread over the range would be investigated. With more time, a larger selection of possible outputs would be investigated, and so on. This approach was seen as analogous to human response characteristics, which can vary from extremely rapid but crude in reflex response situations to extremely precise when time permits.

To determine the relative merits of the different forces investigated at any level, it is basically necessary to determine their effect upon the system in terms of the resulting distance of the system from its stable state. To do this involves simulating the system being controlled. This corresponds to using the differential equations describing the system directly, and is simpler than integrating them, which can often require specialists in control theory.

As an accurate computer simulation of a particular dynamic system was already available, it was decided to carry out experiments to investigate the feasibility of the progressive refinement approach before attempting to construct the appropriate system of rules. Despite using a number of different approaches for measuring distance from stability, it was not found possible to identify a search strategy which would maintain the system in its region of stability. No basis was therefore available for the construction of a set of rules to deal with this problem. This was a dissappointing result, which indicates that a more formal analysis based on mathematical control theory is desirable in dealing with the progressive refinement approach to continuous systems. Such a mathematical analysis is outside the scope of this thesis.

6.1 Controlling A Cart with A Rigid Pendulum.

The continuous system chosen for investigation was 'Cart with Inverted Pendulum', in which the objective is to move the cart so as to maintain a rigid rod fixed to it by a hinge at its lower end in an upright position. The reason for the choice of this system rather than a simpler one was the availability of an accurate simulation programme, which could be used for investigating the feasibility of the approach with this type of system. Such systems are of course far from the domain usually dealt with using a rule based approach.

The system is made up as follows (Figure 1):

The system consists of (1) a cart moving along a line on a monorail of limited length, (2) a pendulum hinged to the cart so as to rotate through 360^0 in the plane containing the line, and (3) a means of driving the cart involving a d.c. motor, a pulley-belt transmission system and a d.c. power amplifier.

Under the assumptions that the pendulum is a rigid body and that the driving force is proportional to the input voltage to the amplifier and is directly applied to the cart without any delay, a four-dimensional vector x whose components are the position of the cart r, the angle of the pendulum a , and their respective velocities r' and a', i.e.

$$x=(r,a,r',a') \qquad (1)$$

can be considered as the state of this system, and the input voltage u to the d.c. power amplifier can be considered as the system input. The origin of the cart position r is the centre of the range where it can move and the origin of the pendulum angle a is the upright position.

Assuming that the friction of the cart is proportional only to the velocity r' and the friction generating the pivot axis is proportional to the angular velocity a' of the pendulum, the following non-linear differential equations are obtained:

$$(M_0 + M_1)r'' + M_1 L\cos A\, A'' = -Fr' + M_1 L A'^2 \sin A + Gu \qquad (2)$$

$$M_1 L\cos A\, r'' + (J + M_1 L^2)A'' = -CA' + M_1 Lg\sin A$$

which describe the dynamics of the system. The definitions and the values of the parameters M_0,F,G,M_1,L,J,C and g in eqns. (2) are listed in Table 1 with other parameters of the system. Equations (2) are rewritten as the ordinary differential equation

$$x' = f(x,u) \qquad (3)$$

describing the system dynamics, where

$$f_1 = x_3 \quad f_2 = x_4$$

$$f_3 = (x_2)(a_{32}\sin x_2 \cos x_2 + a_{33}x_3 + a_{34}\cos x_2 x_4 + a_{35}\sin x 2x_4 + b_3 u)$$

$$f_4 = (x_2)(a_{42}\sin x_2 + a_{43}\cos x_2 x_3 + a_{44}x_4 + a_{45}\sin x_2 \cos x_2 x_4 + b_4 \cos x_2 u) \qquad (3')$$

$$(x_2) = (1 + \sin^2 x_2)^{-1}$$

In eqn. (3'), f_i denotes the ith component of f(x,u) and the parameters a_{ij}, b_i and are listed in Table 1.

The system considered is subjected to the restrictions

$$r < r_0\,, \quad u =< u_0 \qquad (4)$$

where $r = r_0$ corresponds to both ends of the monorail and u_0 denotes the maximum possible input value to the amplifier without any saturation, those limit values also being listed in Table 1.[R27]

Cart	Mass	M_0	0.48	K_g
	Friction constant	F	3.83	K_g/s
	Gain constant	G	8.41	N/V
	Region of cart movement	$\pm r_0$	$\pm$0.5	m
	Maximum input voltage	$\pm u_0$	$\pm$0.7	V
Pendulum	Mass	M_1	0.16	K_g
	Length between the axis and center of gravity	L	0.25	m
	Moment of inertia about the center of gravity	J	0.0043	K_g-m
	Friction constant	C	0.00218	K_g-m /s
	Acceleration of gravity	g	9.8	m/s

Table 1. Parameters of cart-pendulum system.

The control problem with the cart-pendulum system is to drive the pendulum from the pendent position to the upright position and to keep the pendulum in that position. The problem of synthesizing such a control system can be divided into the following problems. The first problem is how to keep the pendulum in its upright position and the cart in its central position, that is, how to regulate the system at its origin x = 0 so that a stable zone may be created around the origin which is inherently unstable. In short, the problem is to design a stabilizer which may stabilize the inherently unstable origin of the system. The second problem is to drive the system state from the natural stable state to the stable zone which is generated in the neighborhood of the origin by the above stabilizer.

Classical control theory has been successful in dealing with both these problems. The stabilizer in the paper [R27] using feed-back control design is very successful in

controlling the system. The authors designed a linear feedback controller combined with a state observer. Since the possible input value is limited as (4), the actual input applied to the system is as

$$u = -K\mathbf{x}, \quad \text{if } K\mathbf{x} =< u_0$$
$$= -\text{sign}(K\mathbf{x})u_0, \quad \text{if } K\mathbf{x} > u_0 \qquad (5)$$

One of the most satisfactory selection of parameters is :

Feedback law: K=(-7.05 V/m, -13.8 V/rad, -5.05 V/(m/s), -2.56 V/(rad/s)). (6)

The mathematical analysis on which the solution given in [R27] is based is of a fairly advanced nature, and not easy to derive for anyone lacking expertise in control theory.

How might the inherent intuitive simplicity of the progressive refinement approach be applied without using this mathematical analysis?

A plausible approach is to construct a model , or simulation of the system, a task which is typically much simpler than carrying out the mathematical analysis of the systems behaviour, and can be compared to writing down the differential equations describing the system rather than solving them. This model can then be used on a trial

and error basis to find a control input which improves the state of the system, basically by selecting a limited set of trial inputs, identifying those which give the best improvement, and then trying again with a refinement of the input set. This process is repeated until the time available for a decision has run out, whereupon the best response discovered to date is output. The time available for a decision is itself a variable, depending chiefly on the velocities in the system. This approach would appear to mimic quite well the human response, which typically involves a reflex, rough response if time is very pressing, while a very precise response is made if plenty of time is available.

In order to investigate the feasibility of this approach, the pattern of forces used in controlling the system using the classical linear feedback theory was logged, and is given in Table 1. On the basis of this table it seemed that the approach should work, and that by searching it would be possible to identify the required pattern. In order to apply the approach, the existing simulation program was used. Each step in the simulation corresponded to a time interval of 0.05 seconds. Because a software simulation is being used, there was no real time constraint on the calculations that could be done to determine the next input to the simulation. So to check the feasibility of the approach, approximately 1000 different forces were checked on each iteration in a search to find a force which would improve the state of the system.

Changes in the state of the system were measured using three different metrics.

1) $X2^2 + X4^2$ X2- angle, X4- angle rate.

2) $X2^2$

3) $X4^2$

4) $(X2 + X4)^2$

No matter which metric was used, it proved to be impossible to identify pattern of forces which would maintain the system in a stable region.

Table 1.

initial angle=1(degree) u_0= 0.8

time	force	position	angle	velocity	ang. vel.
0.05	0.000	0.000	0.997	0.000	0.000
0.10	0.312	-0.000	1.037	-0.002	0.028
0.15	-0.223	0.005	0.272	0.205	-0.532
0.20	0.098	0.010	-0.354	-0.005	0.055
0.25	-0.089	0.012	-0.496	0.063	-0.143
0.30	0.025	0.013	-0.594	-0.015	0.061
0.35	-0.040	0.012	-0.534	0.007	-0.014
0.40	0.001	0.012	-0.472	-0.021	0.053
0.45	-0.021	0.011	-0.370	-0.013	0.021
0.50	-0.005	0.010	-0.281	-0.022	0.040
0.55	-0.012	0.009	-0.191	-0.019	0.024
0.60	-0.006	0.008	-0.118	-0.021	0.027
0.65	-0.008	0.007	-0.055	-0.019	0.018
0.70	-0.005	0.006	-0.005	-0.018	0.017
0.75	-0.005	0.006	0.034	-0.016	0.011
0.80	-0.004	0.005	0.062	-0.015	0.009
0.85	-0.004	0.004	0.083	-0.013	0.006
0.90	-0.003	0.003	0.096	-0.012	0.004
0.95	-0.002	0.003	0.103	-0.010	0.002

In every cycle of the system, it was essential that the new position would have a smaller error than the previous one, otherwise the system would be regarded as having failed. A sample of the results obtained with different error metrics was as follows(see table 2):

error function1:

$x2^2 + x4^2$ x2 - angle, x4 - angle rate.

Table 2.

time	force	position	angle	velocity	ang.vel	metrics	
0.0	0.0	0.000	0.017	0.000	0.000	0.000305	%intial state%
0.05	0.014	0.000	0.017	0.008	0.001	0.000	%situation is better%
0.10	0.021	0.002	0.017	0.024	-0.000	0.000	%situation is worse%
0.15	0.030	0.008	0.016	0.048	-0.005	0.000	%situation is better%
.							
.							
.							
1.40	-0.360	-2.404	0.189	-0.832	0.994	1.025	%situation is worse%
1.45	-0.360	-3.616	2.496	-0.826	-7.299	59.501	%situation is worse%

out of control!

When there is no force to improve the state among the 1000 forces, the force which makes the new position have a smallest error is chosen. The message "situation is worse" indicates that there is no optimal force in the 1000 forces. Using this alternative choice, after 1 second the system is still out of control. This experiment is done in the very simple initial state. Also, it presumes that the computer is very efficient, which can calculate and compare 1000 times within 0.05. But from the Results, it can be seen that it was not found possible to guarantee the dynamic stability of the system using this approach.

At first sight it seems paradoxical that the forces determined by the classical theory, which achieve dynamic stability, were not identified by the search. However, the actual forces used are to seven digits of precision rather than the three given in the table. Experiments using the forces given by the classical theory rounded to three digits failed to achieve control.

It would appear therefore that in order to achieve dynamic stability forces must be specified to a level of precision which makes them impracticable to identify on a trial and error basis, and that a mathematical analysis is therefore required. Alternatively, a better error metric might lead to stability. It was felt that further investigation of this matter was not appropriate here.

Chapter 7

Conclusions and FutherWork

Chapter 7 The Conclusions and Further Work

7.1 Conclusion.

In the area of knowledge-based real time control system, the main problem is ensuring that the control system meets the time constraints. In this thesis, we have laid out an approach for real time problem solving. It is based on the rule-based approach having a sophisticated control component that can constrain its problem-solving activities, so as to ensure a response in the time available. The time constraints themselves can be input as part of the system.

The approach put forward here involves applying progressive refinement to the rule-based system. The key aspects of this approach are:

1) The criterion for successful real time control behaviour for rule-based systems should be to try to develop the best solution to the overall problem which satisfies the time constraints.

2) A real time rule-based problem solver must be able to reason about its criteria for an acceptable solution, if the best solution is not obtainable within the available time.

In order to support this idea, the rules are constructed into a hierarchy of levels. The more precise solution will be obtained in the higher level, but also needs more time. When time is short, a quick and appropriate response can be guaranteed at the panic level.

The progressive reason strategy of the framework presented here guarantee an approximate response, even then available time is very limited.

A control language is provided in which users can express their rules in an 'if/then' English like format, which is then translated from this format to Prolog. The users can easily add rules and change the specification of the system through the user interface. The knowledge base is translated into Prolog for actual execution. No knowledge of Prolog is required by the users.

The examples have shown that the framework can be successfully used in discrete system control. The time constraints and user's needs are met perfectly. They have also shown that the system controlled on the basis of a mathematical analysis of their control characteristics, which is usually implemented in feedback control, is quite difficult to control by rule-based approach.

7.2 Further Work.

The system seems reasonably complete with regard to dealing with discrete systems. Application of the system to a broader range of real world problems is desirable and would probably give rise to some refinements.

The application of this type of system to control the continuous system needs further investigation. It is suggested that a detailed analysis of this approach using mathematical control theory should be carried out.

Bibliography

Bibliography

[1] Prolog Programming for Artificial Intelligence by Ivan Bratko.

[2] Building Expert Systems in Prolog by Dennis Merritt (Springer-Verlag).

[3] Expert Systems Tools & Applications by Paul Harmon, Rex Maus and William Morrissey.

[4] Advisor an Expert System Shell Written in Prolog by Paul Powell.

[5] Autonomous Vehicles as Real-Time Expert Systems by John Hallam, Dai research Paper No. 332, Department of A.I.uni. of Edinburgh.

[6] Approximate Processing in Real-Time Problem Solving by Victor R. Lesser, Edmund H. Durfee and Jasmina Pavlin, COINS Technical Report 86-126, Dec. 1987.

[7] Computer Architectures for Real-Time Knowledge-Based Control by Hany K. Eldeib, Department of Electrical and Computer Engineering George Mason University, 1989 American control conf. vol3.

[8] Hexscon: A Hybrid Microcomputer-Based Expert System for Real-Time Control Applications, M. Lattimer Wright.

[9] A Real-Time Expert System for Process Contrl, by Robert L. Moore, and others, Proceedings of the IEEE Ist Conf. on Artificial Intelligence Applications,1984.

[10] An Architecture for Real-Time Rule-Based Control by David A. Handdian and Robert F. Stengel, Princeton University, Departmetn of mechanical & Aerospace Engineering Princton, New Jersey 08544.

[11] An Expert System for Real-Time Control by M.lattimer wright, Milton W. Green, and others. SRI International.

[12] An Expert System Shell for Analysis of Real Time Signals by Tong Wei-Guang and Zhao Lin_Liang IFAC AI in Real-Time Control, Shenyang, PRC, 1989.

[13] Realization of Expert System Based Feedback Control by Darl-Erik Arzen, Department of Atomatic Control, Institute of Technology, Sweden.

[14] Expert system and Artifical Intelligent by Nathan Goldenthal.

[15] Issues of Real-Time Expert Systems by Haihong Dai Terry J. Anderson Fabian C. Monds Dept. of Information Systems Uni. of Ulster at Jordanstown Newtownabbey, Co. Antrim Bt37oQB N.Ireland.

[16] Expert systems Principles and Programming by Joseph Giarratano, Ph.D Universi of Houston-Clear Lake and Gary Riley NASA-Johnson Space Center PWS_KENT Publishing Company Boston.

[16] System Development by Michael Jackson. Prentice-Hall international series in Computer Science. ISBN 0-13-880328-5.

[17] Systems analysis and Design. A structured approach by William S. Davis. Addison and Wesley. ISBN 0-201-10272-4.

[18] Crystal by Ruth Aallsgrove. Personal computer owrld November 1988 p172-p175.

[19] PC Expert Sytem prototyping and beyond by David W. Tong IEEE 1987 p184-p188.

[20] Compilers pringciples, Techniques and Tools by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman SBN 0-201-10194-7.

[21] The Rule-Based Distillation Decision and Control System by Z. Song, J.C. Gao and C. H. Zhou, IFAC AI in Real-Time Control,Shenyang, PRC, 1989.

[22] Asynchronous Methods for Expert Systems in Real-Time Applications by Th. Beck and R. J. Lauber, Institute of Control Engineering and Industrial Automation of the Uni. of Stuttgart, Pfaffenwaldring 47, d-7000 Stuttgart 80, FRG.

[23] A Hardware Accelerator for Real-Time System Diagnosis by Razvo Nakajima, IBM 826.

[24] Dynamic System Configuration for Distributed Real-Time Systems by J. Magee, J. Kramer Imperial 1018.

[25] Specifying Meta-Level Architectures for Rule-Based Systems, Kaisers/Autern 916.

[26] A Real-Time Language with a Schedulability Analyzer, Toronto 995.

[27] Control of Unstable Mechanical System, Control of Pendulum by SHOZO MORI, HIROYOSHI XISHIHARA and KATSUHISA FURUTA.

[28] Dos Programmer Reference by Terry R. Dettmann.

[29] Sysmbolic Computation, Prolog by example by Helder Coelho Jose C.Cotta.

[30] Application of Expert Fuzzy controller in the penicillin fermentation processes By E-Hui Xu, Guo-Hua Xu and Shi-Liang Zhang. IFAC AI in Real-Time Control, Shenyang PRC, 1989.

[31] Implementation of very large Prolog-Based Knowledge Bases on Data Flow Architectures by Keki B. Irani and Yi-Fong Shin.

[32] Selective Depth-First Search in Prolog by Martha Palmer, Donp, Mckay L.M.Norton, Lyneiteherschman, M.W. Freeman.

[33] The Real-Time Expter Thomas. J. Laffey.

[34] The Rule-Based Distillation Decision and Control System by Z. Song, J.C. Gao and C. H. Zhou.

[35] The IBM PC BIOS by Brett Glazs April 1989, Byte.

[36] Introduction to expert systems by Peter Jackson.

[37] Rule-based object-oriented approach for modelling real-time systems by Ghaly and N Prabhakaran, School of Computer Science, Florida International University.

Appendix A

Program Listings

```
% main modular %


:- public main/0.
:- public restart/0.

:- visible recorda/3.
:- visible write/1.
:- visible nl/0.
:- visible recorded/3.
:- visible instance/2.
:- visible replace/2.
:- visible is/2.
:- visible '>'/2.
:- visible '>='/2.
:- visible '=<'/2.
:- visible '<'/2.
:- visible '='/2.
:- visible '\=='/2.
:- visible '=\='/2.

:- extrn initialise_keithley/0:c('_initialise_keithley').
:- extrn getdata/2:c('_getdata').
:- extrn getdatd/2:c('_getdatd').
:- extrn getnum/4:c('_getnum').
:- extrn writebuf/2:c('_writebuf').
:- extrn isr_setup/0:c('_isr_setup').
:- extrn set_timer/1:c('_set_timer').
:- extrn isr_remove/0:c('_isr_remove').
:- extrn load_rules/0.
:- extrn conc/3.
:- extrn gettime/0.
```

framework.ari

```
main:-
    greeting,
    isr_setup,                          % set interrupt handler %
    repeat,
    write('  Enter load(rule),monitoring,time(rule for available)'),nl,
    write('                              or quit at the prompt'),nl,
    write('>'),
    read(X),
    do(X),
    X == quit,
    isr_remove.                         % remove interrupt handler %


restart:-
    set_timer(0),
    isr_remove,
    nl,
    write('hello'),nl,
    isr_setup,
    solve_rules.

greeting:-
    write('          **********************************'),nl,
    write('          *                                *'),nl,
    write('          *   Welcome to the framework     *'),nl,
    write('          *    for rule-based system       *'),nl,
    write('          *                                *'),nl,
    write('          **********************************'),nl,
    nl,
    write('                    Superviser---Michael Ryan'),nl,
    write('                    Author---Peijuan Xie'),nl, nl,nl.


do(load):- load_rules,!.

do(monitoring):-solve,!.

do(time):- gettime,!.

do(quit).
```

```
do(X):- write(X),
        write(' is not a legal command'),nl,
        fail.


/* start monitoring */

solve:-
        write('Please enter rule_file name >'),
        read(F),
        write('How many levels do you have ?'),
        read(Lev),
        Lev1 is Lev - 1,
        reconsult(F),
        recorda(result,[],Ref),
        recorded(inname,Input,_),
        initial(Input),
         initialise_keithley,
        solve_rules.


solve_rules:-
        recorded(result,Test,Ref),
        ifthen(Test \== [],                    % test the first time %
              write_out(Test)),
        repeat,
         replace(Ref,[]),
         [!abolish(ap/2),
         ctr_set(1,0),                         % initialise counter 1 for input%
         do_input,
         call(break(T)),                       % available time %
         set_timer(T),                                 % set real time clock %
         refine(Ref,Lev1),                     % progressive refining %
         instance(Ref,Out),
         write_out(Out)!],
         set_timer(0),                         % turn off timer %
         fail.
```

```
refine(Ref,Lev):-
        ctr_set(0,0),
        repeat,
         ctr_inc(0,N),
         [!call(top_goal(N,Xn)),
         replace(Ref,Xn)!],
        N == Lev.

initial([X|Tail]):-
       assertz(av(X,0)),
       assertz(ap(X,0)),
       initial(Tail).

initial([]).

do_input:-
        recorded(screen,S,_),
        do_inputs(S),
        recorded(analog,A,_),
        do_inputa(A),
        recorded(digital,D,_),
        ifthen(D \== [],
              (getdatd(Va,Vb),
                do_inputd(0,Va,Vb,D))).


do_inputa([X|Tail]):-                    % anaolg input %
         ctr_inc(1,N),                   % select channel %
        getdata(V,N),
        call(av(X,V1)),
        asserta(ap(X,V1)),
        retract(av(X,V1)),
        assertz(av(X,V)),
        do_inputa(Tail).

do_inputa([]).

do_inputd(_,_,_,[]).
                                         % digital input %
```

```
do_inputd(P,Va,Vb,[X|Tail]):-
        P1 is P + 1,
         getnum(P1,Va,Vb,V),
        nl,write(V),nl,
        call(av(X,V1)),
        asserta(ap(X,V1)),
        retract(av(X,V1)),
        asserta(av(X,V)),
        do_inputd(P1,Va,Vb,Tail).

do_inputs([]).

do_inputs([X|Tail]):-
    write('Please enter  '),
    write(X),
    write(':'),
    read(V),
    call(av(X,V1)),
    asserta(ap(X,V1)),
    retract(av(X,V1)),
    asserta(av(X,V)),
    do_inputs(Tail).

write_out(L):-
        conc([out],L,L1),
        Stru=..L1,
        writebuf(Stru,A),
        ifthen(recorded(message,M,_),
              (write(M),
               eraseall(message))).
```

loadrule.ari

```
% load rule base %


:- public load_rules/0.
:- public member/2.
:- extrn conc/3.
:- extrn user_input/0.
:- extrn user_output/0.
:- extrn get_goal/2.


load_rules:-

   write('          ********************************************'),nl,
   write('          *                                          *'),nl,
   write('          * % The predicate translates if_then rule *'),nl,
   write('          *    to Prolog_rule %                      *'),nl,
   write('          *                                          *'),nl,
   write('          * Does the file to save prolg_rules exist *'),nl,
   write('          * already or new file ?                    *'),nl,
   write('          * The old file is the rule_file that you   *'),nl,
   write('          * want to add more rules to it.            *'),nl,
   write('          *                                          *'),nl,
   write('          ********************************************'),nl,
   nl,
   write('old/new >'),
   read_line(0,Id),
   write('The file name >'),
   read_line(0,F),
   atom_string(F1,F),
   ifthenelse( Id == $new$,
               create(H,F1),
               open(H,F1,a)),!,
   get_names(Id,A),                    % get input names and output names %
                                       %    of the system %
```

```
   write('How many levels do you have ? >'),
   read_line(0,L),
   string_term(L,Lev1),
   Lev is Lev1 - 1,!,
   ifthen(Id == $new$,
            get_goal(H,Lev)),              %  get top_goal as start rule in the %
                                      %    rule base %

   ifthen(A == $y$,
         get_goal(H,Lev)),
   specify,
   nl,nl,
   load_kb(H).

get_names(Id,A):-
   Id == $new$,
   A = $n$,
   user_input,
   nl,
   user_output,
   save,
!.

get_names(Id,A):-
   Id == $old$,
   recorded(inname,I,_),
   write('The input name list is '),
   write(I),nl,
   recorded(outname,O,_),
   write('The output and intermediate name list is '), nl,
   write(O),nl,
   write('The output name list is'),nl,
   recorded(output,Out,_),
   write(Out),
   write('Do you want to rewrite input,output names?,(y/n)>'),
   read_line(0,A),
   ifthen(A == $y$,
         (write('chang "input","output","intermediate" or"all">'),nl,
         read_line(0,L),
         do_it(L))).
```

```
do_it(L):-
   L == $input$,
   eraseall(inname),
   eraseall(analog),
   eraseall(digital),
   eraseall(screen),
   user_input,
   save,
   nl.

do_it(L):-
   L == $output$,
   recorded(output,Out,_),
   del(Out),
   eraseall(output),
   user_output,
   save.

do_it(L):-
   L == $intermediate$,
   recorded(outname,Out,_),
   del(Out),
   eraseall(outname),
   user_output,
   save,
   nl.

do_it(L):-
   L == $all$,
   eraseall(inname),
   recorded(outname,Out,_),
   del(Out),
   eraseall(outname),
   recorded(output,Out1,_),
   del(Out1),
   eraseall(output),
   user_input,
   nl,
   user_output,
   save.
```

loadrule.ari

```
del([X|T]):-
   eraseall(X),
   del(T).

del([]).



specify:-

   write('          ************************************************************'),nl,
   write('          *                                                          *'),nl,
   write('          * enter rule at prompt,end by "eof"                        *'), nl,
   write('          * each rule is entered in the format below                 *'),nl,
   write('          *                                                          *'),nl,
   write('          *       prompt >                                           *'),nl,
   write('          *                                                          *'),nl,
   write('          *   rule or eof >if(return)                                *'),nl,
   write('          *     condition >speed > 2,     % no space after ">",      *'),nl,
   write('          *     condition >condition2,           and before ",".     *'),nl,
   write('          *      ....        ....    ,    % input in low_case.       *'),nl,
   write('          *     condition >conditionN,    % only one space           *'),nl,
   write('          *     condition >then(return)   % between two item.        *'),nl,
   write('          *       action >force is 4;    % rule ended by  ;  .      *'),nl,
   write('          *   rule or eof >eof(return)                               *'),nl,
   write('          *                                                          *'),nl,
   write('          ************************************************************'),nl,
   nl.
```

loadrule.ari

```
load_kb(H):-
    repeat,
    write('write("rule"/"eof)">'),
    read_line(0,X),
    [!ifthen(X == $rule$,
          (write('Which level are the rules in ?'),nl,nl,
           write('level >'),
           read_line(0,N1),
           atom_string(N,N1),
           get_rule(N,H)))!],

   X == $eof$,
   close(H).

get_rule(N,H):-
    repeat,
      write(N),
      write('_level rule'),nl,
      write('   or "stop">'),
      read_line(0,X),
      [!get_one(H,N,X)!],
    X == $stop$.


get_one(H,N,X):-
    X == $if$,
    recordz(user,X,_),
    process(N,Lif,Lthen),
    ask_user(Ans),                % ask user for confirmation %
    write_rule(Ans,H,Lif,Lthen),!.

get_one(_,_,$stop$):- !.

get_one(H,N,X):-
   write(X),
   write('is not a legal input try again.'),nl.
```

loadrule.ari

```
process(N,Lif,Lthen):-
   recorda(if,[],Ref),
   recorda(then,[],Ref1),
   repeat,
    write('  condition >'),
    read_line(0,L),
    [!ifthen(L \== $then$,
          process_if(N,Ref,L))!],
   L == $then$,
   recordz(user,L,_),
   write('      action >'),
   read_line(0,LL),
   process_then(N,Ref1,LL),
   instance(Ref,Lif),
   instance(Ref1,Lthen).


process_if(_,_,$$):-
    write('blank line, please try again'),nl,!.


process_if(N,Ref,L):-
    recorded(inname,Inlist,_),
    string_search($ $,L,P),
    substring(L,0,P,Attr),
    atom_string(Term,Attr),
    member(Term,Inlist),
    recordz(user,L,_),
    concat($V$,Attr,Attr1),
    instance(Ref,Table),
    string_search($,$,L,S),
    S1 is S - P,
    substring(L,P,S1,Rest),
    concat(Attr1,Rest,L1),
    conc(Table,[L1],Table1),
    replace(Ref,Table1),!.
```

```
process_if(N,Ref,L):-
    recorded(outname,Outlist,_),
    string_search($ $,L,P),
    substring(L,0,P,Attr),
    concat($V$,Attr,Attr1),
    atom_string(Term1,Attr1),
    atom_string(Term,Attr),
    member(Term,Outlist),
    recordz(user,L,_),
    recorded(Term,Head,_),
    conc([N],Head,Head1),
    conc([Term],Head1,Head2),
    conc(Head2,[Term1],Head3),
    Headn =.. Head3,
    string_term(H_str,Headn),
    instance(Ref,Table),
    conc(Table,[H_str],Table1),
    replace(Ref,Table1),
    string_search($,$,L,P1),
    P2 is P1 - P ,
    substring(L,P,P2,Str),
    concat(Attr1,Str,Str1),
    instance(Ref,Tab),
    conc(Tab,[Str1],Tab1),
    replace(Ref,Tab1),!.


process_if(N,Ref,L):-
    write(L),
    write('is not a legal name, '),nl,
    write('please try again'),nl.

process_then(N,Ref,$$):-
    write('blank line,please try again'), nl,
    write('    action >'),
    read_line(0,X),
    process_then(N,Ref,X),!.
```

loadrule.ari

```
process_then(N,Ref,L):-
    [!string_search($ $,L,P),
    substring(L,0,P,Attr),
    atom_string(Term,Attr),
    recorded(outname,Outlist,_)!],
    member(Term,Outlist),
    recordz(user,L,_),
    recorded(Term,Tab,_),
    conc([N],Tab,Tab1),
    conc([Term],Tab1,Tab2),
    string_search($is$,L,P1),
    N1 is P1 + 3,
    string_search($;$,L,P2),
    N2 is P2 - N1,
    substring(L,N1,N2,Val),
    string_term(Val,Tval),
    ifthenelse((number(Tval);(atom(Tval)),

                conc(Tab2,[Tval],Tab3),

                (concat($V$,Attr,Vattr),
                  atom_string(Tv,Vattr),
                  conc(Tab2,[Tv],Tab3),
                  Pp is P1-1,
                  Nn is P2-Pp,
                  substring(L,Pp,Nn,Nval),
                  concat(Vattr,Nval,Vn),
                  recorded(if,Ex,Rif),
                  conc(Ex,[Vn],Ex1),
                  replace(Rif,Ex1))),

    Stru=..Tab3,
    string_term(Str,Stru),
    replace(Ref,[Str]),!.
```

loadrule.ari

```
process_then(N,Ref,L):-
    [!string_search($ $,L,P),
    substring(L,0,P,Attr),
    atom_string(T,Attr),
    recorded(mes,M,_)!],
    member(T,M),
    recordz(user,L,_),
    recorded(T,Tab,_),
    conc([N],Tab,Tab1),
    conc([T],Tab1,Tab2),
    string_search($is$,L,P1),
    K is P1 + 3,
    string_search($;$,L,P2),
    K1 is P2 - K,
    substring(L,K,K1,Val),
    conc(Tab2,[Val],Tab3),
    Stru=..Tab3,
    string_term(Str,Stru),
    replace(Ref,[Str]),!.



process_then(N,Ref,L):-
    write(L),
    write('is not a legal name,please try again'),nl,
    write('      action >'),
    read_line(0,X),
    process_then(N,Ref,X).

ask_user(Ans):-
    write('last rule is :'),nl,
    list_all,
    write('Is the rule correct , answer(y/n)> '),
    read_line(0,Ans),
    eraseall(user).
```

loadrule.ari

```
list_all:-
    recorded(user,X,_),
    write(X), nl,
    fail.

list_all.


write_rule(Ans,H,[],[X]):-
    Ans == $y$,
    write(H,X),
    write(H,' .'),
    nl(H),
    nl(H).


write_rule(Ans,H,L1,L2):-
    Ans == $y$,
    write_head(H,L2),
    write_body(H,L1).

write_rule(Ans,_,_,_):-
   Ans == $n$,
   write('last rule is incorrect,enter again'),nl.

write_head(H,[X]):-
   write(H,X),
   write(H,' :-'),
   nl(H).


write_body(H,[X|T]):-
   tab(H,3),
   write(H,X),
   ifthen( T \== [],
           (write(H,' ,'),
           nl(H))),
   write_body(H,T).
```

```
write_body(H,[]):-
   write(H,'.'),
   nl(H),
   nl(H).

member(X,[X|Tail]).

member(X,[Head|Tail]):-
  member(X,Tail).
```

shel.ari

```
% specification of the input variable of the system %

:- public user_input/0.
:- public conc/3.
:- visible conc/3.

user_input:-
      write('*************************************************************'),nl,
      write('*                                                           *'),nl,
      write('* Enter input variable name at prompt, end by "stop"  *'),nl,
      write('* All variable name enter in lower_case                   *'),nl,
      write('*                                                           *'),nl,
      write('*************************************************************'),nl,
      nl,
      recorda(inname,[],Ref),
      do_analog(Ref),
      do_digital(Ref),
      do_screen(Ref).


do_analog(Ref):-
      repeat,
      recorda(analog,[],Ref1),
      write('Do you have analog input,(y/n)?'),
      read_line(0,A),
      [!ifthenelse(A == $y$,
              (get_name(Ref1),
               ask_user(Ref1,analog,Ans)),
              Ans = $n$) !],
      Ans == $n$,
      instance(Ref,In),
      instance(Ref1,An),
      conc(In,An,Nin),
      replace(Ref,Nin).
```

```
do_digital(Ref):-
      repeat,
      recorda(digital,[],Ref1),
      write('Do you have digital input,(y/n)?'),
      read_line(0,A),
      [!ifthenelse(A == $y$,
                  (get_name(Ref1),
                   ask_user(Ref1,digital,Ans)),
                  Ans = $n$) !],
      Ans == $n$,
      instance(Ref,In),
      instance(Ref1,An),
      conc(In,An,Nin),
      replace(Ref,Nin).

do_screen(Ref):-
      repeat,
      recorda(screen,[],Ref1),
      write('Do you have screen input,(y/n)?'),
      read_line(0,A),
      [!ifthenelse(A == $y$,
                  (get_name(Ref1),
                   ask_user(Ref1,screen,Ans)),
                  Ans = $n$) !],
      Ans == $n$,
      instance(Ref,In),
      instance(Ref1,An),
      conc(In,An,Nin),
      replace(Ref,Nin).

ask_user(Ref,A,Ans):-
      instance(Ref,I),
      write(A),
      write(' input name list is '),
      write(I),nl,
      write('Do you want to change? (y/n) >'),
      read_line(0,Ans),
      ifthen(Ans == $y$,
            eraseall(A)).
```

```
get_name(Ref):-
      repeat,
       nl,
       write('input name or"stop">'),
       read_line(0,X),
       [lifthen(X \== $stop$,
                (instance(Ref,Inputp),
                 atom_string(A,X),
                 conc([A],Inputp,Inputn),
                 replace(Ref,Inputn)))!],
      X == $stop$.


conc([],L,L).

conc(L,[],L).

conc([X|L1],L2,[X|L3]):-
   conc(L1,L2,L3).
```

```
% specification of the output of the system %


:- public user_output/0.
:- extrn conc/3.

user_output:-
      write('*****************************************************'),nl,
      write('*                                                   *'),nl,
      write('* Enter output  variable name at prompt,end by "stop"    *'),nl,
      write('* All variable name enter in lower_case                  *') ,nl,
      write('*                                                   *'),nl,
      write('*****************************************************'),nl,
      nl,
      recorda(output,[],Ref),
      recorda(outname,[],Ref1),
      repeat,
       nl,
       write('outputname or "stop">'),
       read_line(0,X),nl,
       write('The input is "'),
       write(X),
       write('"'),nl,
       write('if it is correct,press "return"'),nl,
       write('           otherwise press "n">'),
       read_line(0,Ans),nl,nl,
       [!ifthen((Ans == $$,
                 X \== $stop$),
                 get_list(Ref1,X))!],
      X == $stop$,
      instance(Ref1,T),
      replace(Ref,T),
      write('*************************************************************'),nl,
      write('*                                                           *'),nl,
      write('* enter intermediate variable name at prompt,end by "stop"  *'),nl,
      write('*                                                           *'),nl,
      write('*************************************************************'),nl,
      nl,nl,
```

```
      repeat,
       write('intermediate_name or "stop"> '),
       read_line(0,Y),nl,
       write('The input is "'),
       write(Y),
       write(""),nl,
       write('if it is correct,press "return"  '),nl,
       write('            otherwise press "n">'),
       read_line(0,Ans1),nl,nl,
      Ans == $$,
     [ifthen((Ans1 == $$,
              Y \== $stop$),
              get_list(Ref1,Y))!],
      Y == $stop$,
      recorda(mes,[],Rm),
      write('**************************************************'),nl,
      write('*                                                *'),nl,
      write('* Is any message output from the system?         *'),nl,
      write('* If there is, please enter which name it is,    *'),nl,
      write('* this variable only will give message in the    *'),nl,      %   get   message
variable%
      write('* screen. Don"t enter the name again in later    *'),nl,
      write('* Otherwise, enter "no".                         *'),nl,
      write('*                                                *'),nl,
      write('**************************************************'),nl,
      write('>'),
      read_line(0,Mes),
      keep(Mes,Rm).

get_list(_,$stop$):-!.

get_list(Ref,X):-
     instance(Ref,Old),
     atom_string(A,X),
     conc([A],Old,New),
     replace(Ref,New),
```

```
    repeat,
     [!ask_parameter(A),
     recorded(A,P,_),nl,
     write('The parameter of"'),
     write(A),
     write('" is "'),
     write(P),
     write('"'),nl,
     write(' Do you want to change? (y/n) >'),
     read_line(0,Ans),nl,nl,
     ifthen(Ans == $y$,
            eraseall(A))!],
    Ans == $n$.


ask_parameter(A):-
    write('*****************************************'),nl,
    write('enter the  parameter of"'),
    write(A),
    write('"at prompt '),
    nl,
    write('stop by "end" '),nl,
    write('*****************************************'),nl,
    recorda(A,[],Ref),
    repeat,
     write('parameter or "end">'),
     read_line(0,X),
     [!ifthen(X \== $end$,
            (concat($V$,X,Str),
             atom_string(Y,Str),
              instance(Ref,Outp),
              conc([Y],Outp,Outn),
              replace(Ref,Outn)))!],
    X==$end$.
```

```
keep($no$,_).

keep(M,R):-
  atom_string(A,M),
  replace(R,[A]),
  recorded(output,X,R1),
  del(A,X,X1),
  replace(R1,X1),
  recorded(outname,Y,R2),
  del(A,Y,Y1),
  replace(R2,Y1).

del(X,[X|Tail],Tail).

del(X,[Y|Tail],[Y|Tail1]):-
  del(X,Tail,Tail1).

conc([],L,L).

conc([X|L1],L2,[X|L3]):-
  conc(L1,L2,L3).
```

getgoal.ari

```
% get top goal %


:- public get_goal/2.
:- extrn conc/3.

get_goal(H,Lev):-
  recorded(output,Out,_),
  recorded(inname,In,_),
  ctr_set(0,0),
  repeat,
   ctr_inc(0,N),
   Stru=..['top_goal',N,'Xn'],
   string_term(Str,Stru),
   write(H,Str),
   write(H,' :-'),
   nl(H),
   tab(H,10),
   Strr=..[recorda,path,[],'Ref'],
   string_term(Stri,Strr),
   write(H,Stri),
   write(H,' ,'),
   nl(H),
   [!do_input(H,In)!],
   [!construct_goal(H,N,Out)!],
   tab(H,10),
   Strup=..[recorded,path,'Xn','_'],
   string_term(Strp,Strup),
   write(H,Strp),
   recorded(mes,Ms,_),
   ifthenelse(Ms == [],
              write(H,' .'),
              (write(H,','),nl(H),
               do_mes(H,N,Ms))),
   nl(H),
   nl(H),
  N == Lev.
```

getgoal.ari

```
construct_goal(H,N,[X|Tail]):-
  recorded(X,Par,_),
  conc([N],Par,Par1),
  atom_string(X,Xx),
  concat($V$,Xx,Xy),
  concat($S$,Xx,Zz),
  concat($C$,Xx,Uu),
  atom_string(Y,Xy),
  atom_string(Z,Zz),
  atom_string(U,Uu),
  conc(Par1,[Y],Par2),
  conc([X],Par2,Par3),
  Stru=..Par3,
  string_term(Str,Stru),
  tab(H,10),
  write(H,Str),
  write(H,' ,'),
  nl(H),
  tab(H,10),
  Struu=..[instance,'Ref',U],
  string_term(Strr,Struu),
  write(H,Strr),
  write(H,' ,'),
  nl(H),
  tab(H,10),
  Struy=..[conc,U,[Y],Z],
  string_term(Stry,Struy),
  write(H,Stry),
  write(H,' ,'),nl(H),
  tab(H,10),
  Struz=..[replace,'Ref',Z],
  string_term(Strz,Struz),
  write(H,Strz),
  write(H,' ,'),
  nl(H),
  construct_goal(H,N,Tail).
```

```
construct_goal(_,_,[]).

do_input(H,[X|Tail]):-
  atom_string(X,Y)2,
  concat($V$,Y,Y1),
  atom_string(X1,Y1),
  M = av(X,X1),
  string_term(M1,M),
  tab(H,10),
  write(H,M1),
  write(H,' ,'),
  nl(H),
  do_input(H,Tail).

do_input(_,[]).

do_mes(H,N,[M]):-
  recorded(M,S,_),
  conc([N],S,S1),
  conc([M],S1,S2),
  atom_string(M,M1),
  concat($V$,M1,M2),
  atom_string(A,M2),
  conc(S2,[A],S3),
  Struu=..S3,
  string_term(Stri,Struu),
  tab(H,10),
  write(H,Stri),
  write(H,','),
  nl(H),
  Stru=..[recorda,message,A,'_'],
  string_term(Str,Stru),
  tab(H,10),
  write(H,Str),
  write(H,','),
  nl(H),
  tab(H,10),
  write(H,$nl.$),
  nl(H).
```

getgoal.ari

```
construct_goal(_,_,[]).

do_input(H,[X|Tail]):-
  atom_string(X,Y)2,
  concat($V$,Y,Y1),
  atom_string(X1,Y1),
  M = av(X,X1),
  string_term(M1,M),
  tab(H,10),
  write(H,M1),
  write(H,' ,'),
  nl(H),
  do_input(H,Tail).

do_input(_,[]).

do_mes(H,N,[M]):-
  recorded(M,S,_),
  conc([N],S,S1),
  conc([M],S1,S2),
  atom_string(M,M1),
  concat($V$,M1,M2),
  atom_string(A,M2),
  conc(S2,[A],S3),
  Struu=..S3,
  string_term(Stri,Struu),
  tab(H,10),
  write(H,Stri),
  write(H,','),
  nl(H),
  Stru=..[recorda,message,A,'_'],
  string_term(Str,Stru),
  tab(H,10),
  write(H,Str),
  write(H,','),
  nl(H),
  tab(H,10),
  write(H,$nl.$),
  nl(H).
```

interrupt.ari

```
% load time rule %


:- public gettime/0.
:- extrn conc/3.
:- extrn member/2.



gettime:-
        write('*****************************************'),nl,
        write('*    Please write the rule of           *'),nl,
        write('*    deciding available time.           *'),nl,
        write('* If the data needed is previous input *'),nl,
        write('* the variable name is p+inputname      *'),nl,
        write('* e.g:   the function needs speed"s    *'),nl,
        write('* previous data,the variable name for  *'),nl,
        write('* the data is "pspeed".                 *'),nl,
        write('*****************************************'),nl,nl,
        write('enter rule at prompt,end by stop'),nl,
        repeat,
         write('rule or "stop" >'),
         read_line(0,L),
         [!ifthen(L \== $stop$,
                  do_rest(L))!],
        L == $stop$.

do_rest(H):-
        H \== $if$,
        write('ilegeal input try again').


do_rest(H):-
        recorda(sent,$$,Ref),
        repeat,
          write('condition >'),
          read_line(0,L),
          [!ifthen(L \== $then$,
                 (string_search($ $,L,P),
```

```
            substring(L,0,P,S),
            check(S,Id),
            concat($V$,S,S1),
            atom_string(A1,S1),
            ifthenelse(Id == $new$,
                       (atom_string(A,S),
                        conc([A],[A1],B),
                        conc([av],B,B1)),
                       (string_length(S,Ls),
                        Ls1 is Ls -1,
                        substring(S,1,Ls1,Ps),
                        atom_string(A,Ps),
                        conc([A],[A1],B),
                        conc([ap],B,B1))),
            Stru=..B1,
            string_term(Stri,Stru),
            instance(Ref,Part),
            concat([Part,Stri,$,$],Part1),
            string_search($,$,L,P1),
            Lp1 is P1 -P+1,
            substring(L,P,Lp1,In),
            concat(S1,In,Si),
            concat(Part1,Si,Part2),
            replace(Ref,Part2)))!],
 L == $then$,
write('action >'),
read_line(0,X),
string_search($is$,X,Pos),
string_search($;$,X,Pos1),
Len is Pos1 - Pos,
substring(X,Pos,Len,Y),
concat([$Clock $,Y,$.$],Last),
instance(Ref,Con),
concat([$break(Clock):- $,Con,Last],R),
string_term(R,Tr),
assertz(Tr).
```

```
check(S,Id):-
    recorded(inname,I,_),
    atom_string(A,S),
    member(A,I),
    Id = $new$.

check(S,Id):-
    string_length(S,L),
    L1 is L-1,
    substring(S,1,L1,S1),
    atom_string(A,S1),
    member(A,I),
    Id = $old$.

check(S,_):-
   write('ilegeal input name, try again'),nl,
   do_rest(S).
```

process.c

```
/* interface of input/output */


#include <stdio.h>
#include "apctype.h"

#define maskh 0x0fff
#define motherboard 6
#define ready 0x7f

static int buf[20],array[20];

/* function power */

int power(i)
  int i;
  {
    int j,v;
    v=1;
    for (j=1; j <= i; j++)
     {
       v*=2;
     }
     return(v);
  }


/* intialise keithley */

initialise_keithley()

{
        char *global_gain,*strobe;
       global_gain=(char far *) 0xcff9000a;
        *global_gain=0;
        strobe=(char far *) 0xcff9000d;
       *strobe=64;
       return(SUCCESS);
}
```

process.c

```
/*  sampling */

getdata(z,c)
 reftype z,c ;
{
        int word,wait,num;
        int *ad_value;
        char  *slot,*channel, *ad_converter;
        slot=(char far *)0xcff80001;
        channel=(char far *)0xcff8000a;
       ad_converter=(char far *)0xcff90008;
        ad_value= (int far *)0xcff80002;
        getint_c(c,&num);
        *channel= num;                  /* select channel*/
        *slot=motherboard;
       wait = 1000;
       while (wait > 0)
       {
         wait--;
       }
        *ad_converter=0;                /*  start ad_converter*/
       wait=100;
        while (*ad_converter != ready && wait > 0)
       {
         wait--;
         }
        word=*ad_value & maskh;
        putint_c(word,z);
        return (SUCCESS);
}

/* input digital data */

getdatd(a,b)
reftype a,b;
{
   char *porta,*portb;
   int i,j;
   porta=(char far *)0xcff80006;
   portb=(char far *)0xcff80007;
   i=*porta;
   j=*portb;
```

```
   putint_c(i,a);
   putint_c(j,b);
   return(SUCCESS);
}


getnum(p,va,vb,bit)
reftype p,va,vb,bit;
{
  int a,b,i,j,val;
  getint_c(p,&i);
  getint_c(va,&a);
  getint_c(vb,&b);
  if(i <= 8)
  {
   i=i-1;
   j=power(i);
   i=j;
   val=a&i;
  }
  else
  {
   i=i-9;
   j=power(i);
   i=j;
   val=b&i;
   }
   putint_c(val,bit);
   return(SUCCESS);
}


/* write result to buffer */

writebuf(stru,arg)
reftype stru,arg;
{
  int i,j,l,m,k,out1;
```

```
  char *out;
  out=(char far *)0xcff80008;
  getfunctor_c(stru,array,&l,&j);
  for (i=1; i <= j; i++)
 {
  getfuncarg_c(stru,i,&arg);
  getint_c(arg,&m);
  buf[i]=m;
 }
  out1=0;
  for (i=0; i <= j-1; i++)
   if (buf[i+1] == 1)
     {
       out1+=power(i);

     }
  printf("%d\n",out1);
  *out=out1;
  return (SUCCESS);


}
```

pei.c

```
/*
  File PEI.c

  Contains functions for interrupt setup and handling

*/

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <stdlib.h>
#include <setjmp.h>
#include "apctype.h"

/*
  give prototypes
*/
int  isr_setup(void);
int  isr_remove(void);
int  set_timer(int);

/*
  declare variables
*/
void (interrupt far *old_int8)(); /* pointer to old interrupt */

long    ticks=0L;  /* declare ticks left */
int time =0;    /* intermediate storage */

void (interrupt far *int1b)();  /* pointer to control break handler */

extern getint_c(int,int *);

/*
  function to set alarm call
  parameter: number of ticks to wait, 18.2 per second

*/
```

```
int set_timer(reftype parm)
  {
   printf("setting timer\n");
   getint_c(parm,&time);
   ticks=(long)time;
   printf("ticks %ld\n",(long)ticks);

   return 1;
  }


/*
 New interrupt vector to handle hardware clock interrupt

*/
void interrupt far newint8(void)
{
        _enable(); /* allow other interrupts to happen */
        (*old_int8)(); /* call old int vector */
        if (--ticks == 0L) /* time left ? */
            _chain_intr( (void(interrupt far *)())int1b);
           /* jump back up program */
}


/*
  function to set up the interrupt service routine

*/
int isr_setup(void)
  {
   _disable();  /* disable interrupts  */
   old_int8 = _dos_getvect(0x8); /* get old interrupt address */
   int1b    = _dos_getvect(0x1b); /* get control break address */
   /* now replace it with my own */
   _dos_setvect(0x8,(void (interrupt far *) ())newint8);
   _enable(); /* all done re enable the interrupts */
   puts("\nInstalled Successfully\n");  /* tell user its A ok  */
   return(1);
  }
```

```
/*
  function to remove isr.....if you dont do this then
  the machine is non serviceable !

*/
int isr_remove(void)
   {
    _disable();  /* disable interrupts  */

    _dos_setvect(0x8,(void (interrupt far *) ())old_int8); /* reset old vector */
    _enable(); /* all done re enable the interrupts */
    puts("\nDe-Installed Successfully\n");  /* tell user its A ok  */
    return (1);
   }
```

Appendix B

A List of the Rule Base

Appendix B.1 The Rule Base of Example1 and Example2.

Example 1 A simple example using switchs and lights. The time available depends on the input voltage. Because the rules execute so rapidly, the level 3 rules are always reached.

Input.

analog:

volt	0V to 10V.

digital:

switch1	1 or 0.
switch2	1 or 0.
switch3	1 or 0.

Output.

digital:

light1	1 or 0.
light2	1 or 0.
light3	1 or 0.

Time rules.

```
if
  volt < 4,
then
  time is 6(s);
if
  volt >= 4,
  volt =< 8,
then
  time is 3(s);
if
  volt >= 8,
then
  time is 0;
```

Appendix B.1 The Rule Base of Example1 and Example2.

level 1:

```
1)    if
        switch1 = 1,
      then
        light1 is 1;
2)    if
        switch1 = 0,
      then
        light1 is 0;

3)    light2 is 0;

4)    light3 is 0;
```

level 2:

```
1)    if
        switch2 = 1,
      then
        light2 is 1;
2)    if
        switch2 = 0,
      then
        light2 is 0;

3)    light1 is 0;

4)    light3 is 0;
```

Appendix B.1 The Rule Base of Example1 and Example2.

level 3:

1) if
   switch3 = 1,
   then
   light3 is 1;

2) if
   switch3 = 0,
   then
   light3 is 0;

3) light1 is 0;

4) light2 is 0;

Example 1 Translated Prolog version rules.

```
top_goal(0,Xn) :-
        recorda(path,[],Ref) ,
        av(volt,Vvolt) ,
        av(switch3,Vswitch3) ,
        av(switch2,Vswitch2) ,
        av(switch1,Vswitch1) ,
        light3(0,Vswitch3,Vlight3) ,
        instance(Ref,Clight3) ,
        conc(Clight3,[Vlight3],Slight3) ,
        replace(Ref,Slight3) ,
        light2(0,Vswitch2,Vlight2) ,
        instance(Ref,Clight2) ,
        conc(Clight2,[Vlight2],Slight2) ,
        replace(Ref,Slight2) ,
        light1(0,Vswitch1,Vlight1) ,
        instance(Ref,Clight1) ,
        conc(Clight1,[Vlight1],Slight1) ,
        replace(Ref,Slight1) ,
        recorded(path,Xn,_) .
```

Appendix B.1 The Rule Base of Example1 and Example2.

```
top_goal(1,Xn) :-
        recorda(path,[],Ref) ,
        av(volt,Vvolt) ,
        av(switch3,Vswitch3) ,
        av(switch2,Vswitch2) ,
        av(switch1,Vswitch1) ,
        light3(1,Vswitch3,Vlight3) ,
        instance(Ref,Clight3) ,
        conc(Clight3,[Vlight3],Slight3) ,
        replace(Ref,Slight3) ,
        light2(1,Vswitch2,Vlight2) ,
        instance(Ref,Clight2) ,
        conc(Clight2,[Vlight2],Slight2) ,
        replace(Ref,Slight2) ,
        light1(1,Vswitch1,Vlight1) ,
        instance(Ref,Clight1) ,
        conc(Clight1,[Vlight1],Slight1) ,
        replace(Ref,Slight1) ,
        recorded(path,Xn,_) .

top_goal(2,Xn) :-
        recorda(path,[],Ref) ,
        av(volt,Vvolt) ,
        av(switch3,Vswitch3) ,
        av(switch2,Vswitch2) ,
        av(switch1,Vswitch1) ,
        light3(2,Vswitch3,Vlight3) ,
        instance(Ref,Clight3) ,
        conc(Clight3,[Vlight3],Slight3) ,
        replace(Ref,Slight3) ,
        light2(2,Vswitch2,Vlight2) ,
        instance(Ref,Clight2) ,
        conc(Clight2,[Vlight2],Slight2) ,
        replace(Ref,Slight2) ,
        light1(2,Vswitch1,Vlight1) ,
        instance(Ref,Clight1) ,
        conc(Clight1,[Vlight1],Slight1) ,
        replace(Ref,Slight1) ,
        recorded(path,Xn,_) .
```

Appendix B.1 The Rule Base of Example1 and Example2.

```
light1(0,Vswitch1,1) :-
   Vswitch1 = 1.

light1(0,Vswitch1,0) :-
   Vswitch1 = 0.

light2(0,Vswitch2,0) .

light3(0,Vswitch3,0) .

light2(1,Vswitch2,1) :-
   Vswitch2 = 1.

light2(1,Vswitch2,0) :-
   Vswitch2 is 0.

light1(1,Vswitch1,0) .

light3(1,Vswitch3,0) .

light3(2,Vswitch3,1) :-
   Vswitch3 is 1.

light3(2,Vswitch3,0) :-
   Vswitch3 is 0.

light1(2,Vswitch1,0) .

light2(2,Vswitch2,0) .
```

Appendix B.1 The Rule Base of Example1 and Example2.

Example 2. The 'If/Then' English like rules are the same as example 1. To test the time behaviour of the system, a extra predicate is added manully to the Prolog version of the level 1 and level 2, to delay the runing time of the rule base.
The level which determines the output can be changed by altering the input voltage and hence the time available.

```
top_goal(0,Xn) :-
        time(X),
        recorda(path,[],Ref) ,
        av(volt,Vvolt) ,
        av(switch3,Vswitch3) ,
        av(switch2,Vswitch2) ,
        av(switch1,Vswitch1) ,
        light3(0,Vswitch3,Vlight3) ,
        instance(Ref,Clight3) ,
        conc(Clight3,[Vlight3],Slight3) ,
        replace(Ref,Slight3) ,
        light2(0,Vswitch2,Vlight2) ,
        instance(Ref,Clight2) ,
        conc(Clight2,[Vlight2],Slight2) ,
        replace(Ref,Slight2) ,
        light1(0,Vswitch1,Vlight1) ,
        instance(Ref,Clight1) ,
        conc(Clight1,[Vlight1],Slight1) ,
        replace(Ref,Slight1) ,
        recorded(path,Xn,_) ,
        waitsecs(X,2).

top_goal(1,Xn) :-
        time(X),
        recorda(path,[],Ref) ,
        av(volt,Vvolt) ,
        av(switch3,Vswitch3) ,
        av(switch2,Vswitch2) ,
        av(switch1,Vswitch1) ,
        light3(1,Vswitch3,Vlight3) ,
        instance(Ref,Clight3) ,
```

Appendix B.1 The Rule Base of Example1 and Example2.

```
        conc(Clight3,[Vlight3],Slight3) ,
        replace(Ref,Slight3) ,
        light2(1,Vswitch2,Vlight2) ,
        instance(Ref,Clight2) ,
        conc(Clight2,[Vlight2],Slight2) ,
        replace(Ref,Slight2) ,
        light1(1,Vswitch1,Vlight1) ,
        instance(Ref,Clight1) ,
        conc(Clight1,[Vlight1],Slight1) ,
        replace(Ref,Slight1) ,
        recorded(path,Xn,_) ,
        waitsecs(X,2).


top_goal(2,Xn) :-
        recorda(path,[],Ref) ,
        av(volt,Vvolt) ,
        av(switch3,Vswitch3) ,
        av(switch2,Vswitch2) ,
        av(switch1,Vswitch1) ,
        light3(2,Vswitch3,Vlight3) ,
        instance(Ref,Clight3) ,
        conc(Clight3,[Vlight3],Slight3) ,
        replace(Ref,Slight3) ,
        light2(2,Vswitch2,Vlight2) ,
        instance(Ref,Clight2) ,
        conc(Clight2,[Vlight2],Slight2) ,
        replace(Ref,Slight2) ,
        light1(2,Vswitch1,Vlight1) ,
        instance(Ref,Clight1) ,
        conc(Clight1,[Vlight1],Slight1) ,
        replace(Ref,Slight1) ,
        recorded(path,Xn,_) .
```

Appendix B.1 The Rule Base of Example1 and Example2.

```
light1(0,Vswitch1,1) :-
   Vswitch1 = 1.

light1(0,Vswitch1,0) :-
   Vswitch1 = 0.

light2(0,Vswitch2,0) .

light3(0,Vswitch3,0) .

light2(1,Vswitch2,1) :-
   Vswitch2 = 1.

light2(1,Vswitch2,0) :-
   Vswitch2 is 0.

light1(1,Vswitch1,0) .

light3(1,Vswitch3,0) .

light3(2,Vswitch3,1) :-
   Vswitch3 is 1.

light3(2,Vswitch3,0) :-
   Vswitch3 is 0.

light1(2,Vswitch1,0) .

light2(2,Vswitch2,0) .


waitsecs(time(H1,M1,S1,Hs1),S):-
   repeat,
   time(time(H2,M2,S2,Hs2)),
   S1 is (H2 - H1)*3600 + (M2-M1)*60 + S2 - S1,
   S1 >= S.
```

Appendix B.2 : The Rule Base of controlling a machine for the manufacture of contact lenses.

Part 1 is the specification of input,output,intermediate variable and relative parameters of the system. All these variables are compiled to an internal format, and saved in an internal database. A rule deciding time available is written in 'if/then' format and included in part 1. The rule will be compiled to the Prolog format by module "gettime", and saved in an internal database.

Part 2 is the rule base in which the rules are written in 'if/then' format, and constructed in three levels. The 'if/then' format rule base are compiled to the Prolog format rule base by module "loadrule", and saved in a rule file. The Prolog format rule base is shown in part 3.

Part 1:

Specification.

Input variables:

Analog:

angle	rotation angle of the table.
tablerate	table rotation rate.
spinners	spinner speed.

Digital:

arm1	state of the mould input arm.
armf1	fully in of the mould input arm.
arm2	state of the monomer inject arm.
armf2	fully in of the monomer inject arm.
arm3	state of the mould remove arm.
armf3	fully in of the monomer inject arm.
moulds	availability of mould.
monomerl	availability of monomer.
mouldins	mould in spinner.
gasp	gas pressure.
gasf	gas flow.
type1,2,3	lens type.

Appendix B.2

Output varibles:
Digital:

armin1	insert mould input arm. parameter: angle, moulds, arm1.
armout1	withdraw mould input arm. parameter: angle, armf1.
armin2	insert monomer inject arm. parameter: angle, monomerl, arm2.
armout2	withdraw monomer inject arm. parameter: angle, armf2.
armin3	insert mould remove arm. parameter: angle, mouldins, arm3.
armout3	withdraw mould remove arm parameter: angle, armf3.
alarm	warn signal. parameter: lenstype, spinners, tablerate, gasp, gasf.

Operator:

opmes	warn information. parameter: gasp, gasf, moulds, monomerl, mouldins, lenstype, spinners, tablerate.

Intermediate varibles:

checkss	check spinner speed. parameter: lenstype, spinners.
checktr	check table rate. parameter: lenstype, tablerate.
lenstype	lens type. parameter: type1,2,3.

Rule for deciding time available.

```
if
tablerate >= 0,
then
time is 1 / 12*tablerate;

if
tablerate < 0,
then
time is 1 / 12*(-tablerate);
```

Appendix B.2 The 'if/then' format rule base

Part 2 The rule base written in 'if/then' format.

There is 5 stages of the table:

mould input,	angle >= 5V, =< 6V.
monomer inject,	angle >= 7V, =< 8V.
mould remove,	angle >= 9V, =< 10V.
mould fully in,	angle >= 3V, =< 4V.
injection done,	angle >= 1V, =< 2V.

level 1:

```
1: if
       angle >= 5,                          %stage of mould input
       angle =< 6,
       moulds = 0,                          %no mould available
   then
       opmes is no_mould_supply;
2: if
       angle >= 7,                          %stage of monomer inject
       angle =< 8,
       monomerl = 0,                        %no monomer available
   then
       opmes is no_monomer;
3: if
       angle >= 9,                          %stage of mould remove
       angle =< 10,
       mouldins = 0,                        %no mould in spinner
   then
       opmes is no_mould_remove;

4: opmes is ok;
5: alarm is 1;
6: armin1 is 0;                             %other output is off
7: armout1 is 0;
8: armin2 is 0;
9: armout2 is 0;
10: armin3 is 0;
11: armout3 is 0;
```

Appendix B.2 The 'If/then' format rule base

level 2:

```
1:  if
        angle >= 5,                    %if stage is mould input,
        angle =< 6,                    %   mould is available,
        moulds = 1,                    %   mould input arm is ready,
        arm1 = 1,                      % then input mould.
    then                               % otherwise do nothing.
        armin1 is 1;

2:  armin1 is 0;

3:  if
        angle >= 3,                    % if stage is mould input done,
        angle =< 4,                    %    arm fully in,
        armf1 = 1,                     % then retract arm.
    then                               % otherwise do nothing.
        armout1 is 1;

4:  armout1 is 0;

5:  if
        angle >= 7,                    % if stage is monomer inject,
        angle =< 8,                    %    monomer is available,
        monomerl = 1,                  %    machine arm is ready,
        arm2 = 1,                      % then inject monomer.
    then                               % otherwise do nothing.
        armin2 is 1;

6:  armin2 is 0;

7:  if
        angle >= 1,                    % if stage is injection done,
        angle =< 2,                    %    machine arm fullu in,
        armf2 = 1,                     % then retract arm.
    then                               % otherwise do nothing.
        armout2 is 1;

8:  armout2 is 0;
```

Appendix B.2 The 'If/then' format rule base

```
9:  if
        angle >= 9,                    % if stage is mould remove,
        angle =< 10,                   %    mould is in spinner,
        mouldins = 1,                  %    machine arm is ready,
        arm3 = 1,                      % then remove mould.
    then                               % otherwise do nothing.
        armin3 is 1;

10:   armin3 is 0;


11: if
        angle >= 3,                    % if stage is arm fully in,
        angle =< 4,                    % then remove arm,
        armf3 = 1,                     % otherwise do nothing.
    then
        armout3 is 1;

12:  armout3 is 0;

13: alarm is 0;
14: opmes is ok;
```

Appendix B.2 The 'if/then' format rule base

level 3:

```
1: if
       checkss = 0,                    %check spinner speed
   then
       alarm is 1;

2: if
       checktr = 0,                    %check table rotation
   then
       alarm is 1;

3: if
       gasp = 0,                       %check gas pressure
   then
       alarm is 1;

4: if
       gasf = 0,                       %check gas flow
   then
       alarm is 1;

5: alarm is 0;

6: if
       lenstype = a1,
       spinners >= 2,
       spinners =< 10,
   then
       checkss is 1;


7: if
       lenstype = a2,
       spinners >= 2,
       spinners =< 10,
   then
       checkss is 1;
```

Appendix B.2 The 'if/then' format rule base

```
 8:  if
         lenstype = a3,
         spinners >= 2,
         spinners =< 5.6,
     then
         checkss is 1;

 9:  if
         lenstype = a4,
         spinners >= 2,
         spinners =< 6.8,
     then
         checkss is 1;

10:  if
         lenstype = a5,
         spinners >= 2.4,
         spinners =< 6.4,
     then
         checkss is 1;

11:  if
         lenstype = a6,
         spinners >= 2,
         spinners =< 4,
     then
         checkss is 1;


12:    checkss is 0;
```

Appendix B.2 The 'if/then' format rule base

```
13: if
        lenstype = a1,
        tablerate >= 4,
        tablerate =< 10,
    then
        checktr is 1;


14:  if
        lenstype = a2,
        tablerate >= 4,
        tablerate =< 5,
    then
        checktr is 1;

15:  if
        lenstype = a3,
        tablerate >= 5,
        tablerate =< 6,
    then
        checktr is 1;

16:  if
        lenstype = a4,
        tablerate >= 5,
        tablerate =< 6,
    then
        checktr is 1;

17:  if
        lenstype = a5,
        tablerate >= 5,
        tablerate =< 6,
    then
        checktr is 1;
```

Appendix B.2 The 'if/then' format rule base

```
18:   if
          lenstype = a6,
          tablerate >= 5,
          tablerate =< 8,
      then
          checktr is 1;

19:   checktr is 0;


20:   if
        checkss = 0,
      then
        opmes is spinner_speed_incorrect;

21:   if
        checktr = 0,
      then
        opmes is table_rate_incorrect;

22:  if
        gasp = 0,
     then
        opmes is gas_pressure_incorrect;

23:   if
        gasf = 0,
      then
        opmes is gas_flow_incorrect;



24:       opmes is ok;
```

Appendix B.2 The 'if/then' format rule base

```
25:  if
        angle >= 5,
        angle =< 6,
        moulds = 1,
        arm1 = 1,
     then
        armin1 is 1;

26:  armin1 is 0;

27:  if
        angle >= 3,
        angle =< 4,
        armf1 = 1,
     then
        armout1 is 1;

28:  armout1 is 0;

29:  if
        angle >= 7,
        angle =< 8,
        monomerl = 1,
        arm2 = 1,
     then
        armin2 is 1;

30:  armin2 is 0;

31:  if
        angle >= 1,
        angle =< 2,
        armf2 = 1,
     then
        armout2 is 1;

32:  armout2 is 0;
```

Appendix B.2 The 'if/then' format rule base

```
33: if
        angle >= 9,
        angle =< 10,
        mouldins = 1,
        arm3 = 1,
    then
        armin3 is 1;

34:  armin3 is 0;


35: if
        angle >= 3,
        angle =< 4,
        armf3 = 1,
    then
        armout3 is 1;

36: armout3 is 0;

37: if
        type1 = 0,
        type2 = 0,
        type3 = 1,
    then
        lenstype is a1;

38: if
        type1 = 0,
        type2 = 1,
        type3 = 0,
    then
        lenstype is a2;
```

Appendix B.2 The 'If/then' format rule base

```
39: if
        type1 = 0,
        type2 = 1,
        type3 = 1,
    then
        lenstype is a3;

40: if
        type1 = 1,
        type2 = 0,
        type3 = 0,
    then
        lenstype is a4;

41: if
        type1 = 1,
        type2 = 0,
        type3 = 1,
    then
        lenstype is a5;

42: if
        type1 = 1,
        type2 = 1,
        type3 = 0,
    then
        lenstype is a6;
```

Appendix B.2 The compiled rule base

```
top_goal(0,Xn) :-
        recorda(path,[],Ref) ,
        av(spinners,Vspinners) ,
        av(tablerate,Vtablerate) ,
        av(angle,Vangle) ,
        av(type3,Vtype3) ,
        av(type2,Vtype2) ,
        av(type1,Vtype1) ,
        av(gasf,Vgasf) ,
        av(gasp,Vgasp) ,
        av(mouldins,Vmouldins) ,
        av(monomerl,Vmonomerl) ,
        av(moulds,Vmoulds) ,
        av(armf3,Varmf3) ,
        av(arm3,Varm3) ,
        av(armf2,Varmf2) ,
        av(arm2,Varm2) ,
        av(armf1,Varmf1) ,
        av(arm1,Varm1) ,
        alarm(0,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,Valarm) ,
        instance(Ref,Calarm) ,
        conc(Calarm,[Valarm],Salarm) ,
        replace(Ref,Salarm) ,
        armout3(0,Varmf3,Vangle,Varmout3) ,
        instance(Ref,Carmout3) ,
        conc(Carmout3,[Varmout3],Sarmout3) ,
        replace(Ref,Sarmout3) ,
        armin3(0,Varm3,Vmouldins,Vangle,Varmin3) ,
        instance(Ref,Carmin3) ,
        conc(Carmin3,[Varmin3],Sarmin3) ,
        replace(Ref,Sarmin3) ,
        armout2(0,Varmf2,Vangle,Varmout2) ,
        instance(Ref,Carmout2) ,
        conc(Carmout2,[Varmout2],Sarmout2) ,
        replace(Ref,Sarmout2) ,
        armin2(0,Varm2,Vmonomerl,Vangle,Varmin2) ,
        instance(Ref,Carmin2) ,
        conc(Carmin2,[Varmin2],Sarmin2) ,
        replace(Ref,Sarmin2) ,
```

Appendix B.2 The compiled rule base

```
        armout1(0,Varmf1,Vangle,Varmout1) ,
        instance(Ref,Carmout1) ,
        conc(Carmout1,[Varmout1],Sarmout1) ,
        replace(Ref,Sarmout1) ,
        armin1(0,Varm1,Vmoulds,Vangle,Varmin1) ,
        instance(Ref,Carmin1) ,
        conc(Carmin1,[Varmin1],Sarmin1) ,
        replace(Ref,Sarmin1) ,
        recorded(path,Xn,_),

opmes(0,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
Vopmes),
        recorda(message,Vopmes,_),
        nl.


top_goal(1,Xn) :-
        recorda(path,[],Ref) ,
        av(spinners,Vspinners) ,
        av(tablerate,Vtablerate) ,
        av(angle,Vangle) ,
        av(type3,Vtype3) ,
        av(type2,Vtype2) ,
        av(type1,Vtype1) ,
        av(gasf,Vgasf) ,
        av(gasp,Vgasp) ,
        av(mouldins,Vmouldins) ,
        av(monomerl,Vmonomerl) ,
        av(moulds,Vmoulds) ,
        av(armf3,Varmf3) ,
        av(arm3,Varm3) ,
        av(armf2,Varmf2) ,
        av(arm2,Varm2) ,
        av(armf1,Varmf1) ,
        av(arm1,Varm1) ,
        alarm(1,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,Valarm) ,
        instance(Ref,Calarm) ,
        conc(Calarm,[Valarm],Salarm) ,
        replace(Ref,Salarm) ,
        armout3(1,Varmf3,Vangle,Varmout3) ,
```

Appendix B.2 The compiled rule base

```
        instance(Ref,Carmout3) ,
        conc(Carmout3,[Varmout3],Sarmout3) ,
        replace(Ref,Sarmout3) ,
        armin3(1,Varm3,Vmouldins,Vangle,Varmin3) ,
        instance(Ref,Carmin3) ,
        conc(Carmin3,[Varmin3],Sarmin3) ,
        replace(Ref,Sarmin3) ,
        armout2(1,Varmf2,Vangle,Varmout2) ,
        instance(Ref,Carmout2) ,
        conc(Carmout2,[Varmout2],Sarmout2) ,
        replace(Ref,Sarmout2) ,
        armin2(1,Varm2,Vmonomerl,Vangle,Varmin2) ,
        instance(Ref,Carmin2) ,
        conc(Carmin2,[Varmin2],Sarmin2) ,
        replace(Ref,Sarmin2) ,
        armout1(1,Varmf1,Vangle,Varmout1) ,
        instance(Ref,Carmout1) ,
        conc(Carmout1,[Varmout1],Sarmout1) ,
        replace(Ref,Sarmout1) ,
        armin1(1,Varm1,Vmoulds,Vangle,Varmin1) ,
        instance(Ref,Carmin1) ,
        conc(Carmin1,[Varmin1],Sarmin1) ,
        replace(Ref,Sarmin1) ,
        recorded(path,Xn,_),

opmes(1,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
Vopmes),
        recorda(message,Vopmes,_),
        nl.
```

Appendix B.2 The compiled rule base

```
top_goal(2,Xn) :-
        recorda(path,[],Ref) ,
        av(spinners,Vspinners) ,
        av(tablerate,Vtablerate) ,
        av(angle,Vangle) ,
        av(type3,Vtype3) ,
        av(type2,Vtype2) ,
        av(type1,Vtype1) ,
        av(gasf,Vgasf) ,
        av(gasp,Vgasp) ,
        av(mouldins,Vmouldins) ,
        av(monomerl,Vmonomerl) ,
        av(moulds,Vmoulds) ,
        av(armf3,Varmf3) ,
        av(arm3,Varm3) ,
        av(armf2,Varmf2) ,
        av(arm2,Varm2) ,
        av(armf1,Varmf1) ,
        av(arm1,Varm1) ,
        alarm(2,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,Valarm) ,
        instance(Ref,Calarm) ,
        conc(Calarm,[Valarm],Salarm) ,
        replace(Ref,Salarm) ,
        armout3(2,Varmf3,Vangle,Varmout3) ,
        instance(Ref,Carmout3) ,
        conc(Carmout3,[Varmout3],Sarmout3) ,
        replace(Ref,Sarmout3) ,
        armin3(2,Varm3,Vmouldins,Vangle,Varmin3) ,
        instance(Ref,Carmin3) ,
        conc(Carmin3,[Varmin3],Sarmin3) ,
        replace(Ref,Sarmin3) ,
        armout2(2,Varmf2,Vangle,Varmout2) ,
        instance(Ref,Carmout2) ,
        conc(Carmout2,[Varmout2],Sarmout2) ,
        replace(Ref,Sarmout2) ,
        armin2(2,Varm2,Vmonomerl,Vangle,Varmin2) ,
        instance(Ref,Carmin2) ,
        conc(Carmin2,[Varmin2],Sarmin2) ,
        replace(Ref,Sarmin2) ,
```

Appendix B.2 The compiled rule base

```
        armout1(2,Varmf1,Vangle,Varmout1) ,
        instance(Ref,Carmout1) ,
        conc(Carmout1,[Varmout1],Sarmout1) ,
        replace(Ref,Sarmout1) ,
        armin1(2,Varm1,Vmoulds,Vangle,Varmin1) ,
        instance(Ref,Carmin1) ,
        conc(Carmin1,[Varmin1],Sarmin1) ,
        replace(Ref,Sarmin1) ,
        recorded(path,Xn,_),

opmes(2,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
Vopmes),
        recorda(message,Vopmes,_),
        nl.


opmes(0,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
no_mould_supply) :-
   Vangle >= 5 ,
   Vangle =< 6 ,
   Vmoulds = 0.

opmes(0,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
no_monomer) :-
   Vangle >= 7 ,
   Vangle =< 8 ,
   Vmonomerl = 0.

opmes(0,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
no_mould_remove) :-
   Vangle >= 9 ,
   Vangle =< 10 ,
   Vmouldins = 0.

opmes(0,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,ok)
.
```

Appendix B.2 The compiled rule base

```
alarm(0,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,1) .

armin1(0,Varm1,Vmoulds,Vangle,0) .

armout1(0,Varmf1,Vangle,0) .

armin2(0,Varm2,Vmonomerl,Vangle,0) .

armout2(0,Varmf2,Vangle,0) .

armin3(0,Varm3,Vmouldins,Vangle,0) .

armout3(0,Varmf3,Vangle,0) .

armin1(1,Varm1,Vmoulds,Vangle,1) :-
   Vangle >= 5 ,
   Vangle =< 6 ,
   Vmoulds = 1 ,
   Varm1 = 1.

armin1(1,Varm1,Vmoulds,Vangle,0) .

armout1(1,Varmf1,Vangle,1) :-
   Vangle >= 3 ,
   Vangle =< 4 ,
   Varmf1 = 1.

armout1(1,Varmf1,Vangle,0) .

armin2(1,Varm2,Vmonomerl,Vangle,1) :-
   Vangle >= 7 ,
   Vangle =< 8 ,
   Vmonomerl = 1 ,
   Varm2 = 1.

armin2(1,Varm2,Vmonomerl,Vangle,0) .
```

Appendix B.2 The compiled rule base

```
armin3(1,Varm3,Vmouldins,Vangle,1) :-
   Vangle >= 9 ,
   Vangle =< 10 ,
   Vmouldins = 1 ,
   Varm3 = 1.

armin3(1,Varm3,Vmouldins,Vangle,0) .

armout3(1,Varmf3,Vangle,1) :-
   Vangle >= 3 ,
   Vangle =< 4 ,
   Varmf3 = 1.

armout3(1,Varmf3,Vangle,0) .

alarm(1,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,0) .

opmes(1,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,ok)
.

alarm(2,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,1) :-
   checkss(2,Vspinners,Vlenstype,Vcheckss) ,
   Vcheckss = 0.

alarm(2,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,1) :-
   checktr(2,Vtablerate,Vlenstype,Vchecktr) ,
   Vchecktr = 0.

alarm(2,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,1) :-
   Vgasp = 0.

alarm(2,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,1) :-
   Vgasf = 0.

alarm(2,Vgasf,Vgasp,Vtablerate,Vspinners,Vlenstype,0) .
```

Appendix B.2 The compiled rule base

```
checkss(2,Vspinners,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a1 ,
   Vspinners >= 2 ,
   Vspinners =< 10.

checkss(2,Vspinners,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a2 ,
   Vspinners >= 2 ,
   Vspinners =< 10.

checkss(2,Vspinners,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a3 ,
   Vspinners >= 2 ,
   Vspinners =< 5.6.

checkss(2,Vspinners,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a4 ,
   Vspinners >= 2 ,
   Vspinners =< 6.8.

checkss(2,Vspinners,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a5 ,
   Vspinners >= 2.4 ,
   Vspinners =< 6.4.

checkss(2,Vspinners,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a6 ,
   Vspinners >= 2 ,
   Vspinners =< 4.

checkss(2,Vspinners,Vlenstype,0) .
```

Appendix B.2 The compiled rule base

```
checktr(2,Vtablerate,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a1 ,
   Vtablerate >= 4 ,
   Vtablerate =< 10.

checktr(2,Vtablerate,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a2 ,
   Vtablerate >= 4 ,
   Vtablerate =< 5.

checktr(2,Vtablerate,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a3 ,
   Vtablerate >= 5 ,
   Vtablerate =< 6.

checktr(2,Vtablerate,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a4 ,
   Vtablerate >= 5 ,
   Vtablerate =< 6.

checktr(2,Vtablerate,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a5 ,
   Vtablerate >= 5 ,
   Vtablerate =< 6.

checktr(2,Vtablerate,Vlenstype,1) :-
   lenstype(2,Vtype3,Vtype2,Vtype1,Vlenstype) ,
   Vlenstype = a6 ,
   Vtablerate >= 5 ,
   Vtablerate =< 8.

checktr(2,Vtablerate,Vlenstype,0) .
```

Appendix B.2 The compiled rule base

```
opmes(2,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
spinner_speed_incorrect) :-
   checkss(2,Vspinners,Vlenstype,Vcheckss) ,
   Vcheckss = 0.

opmes(2,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
table_rate_incorrect) :-
   checktr(2,Vtablerate,Vlenstype,Vchecktr) ,
   Vchecktr = 0.

opmes(2,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
gas_pressure_incorrect) :-
   Vgasp = 0.

opmes(2,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,
gas_flow_incorrect) :-
   Vgasf = 0.

opmes(2,Vtablerate,Vspinners,Vlenstype,Vmouldins,Vmonomerl,Vmoulds,Vgasf,Vgasp,ok)
.

armin1(2,Varm1,Vmoulds,Vangle,1) :-
   Vangle >= 5 ,
   Vangle =< 6 ,
   Vmoulds = 1 ,
   Varm1 = 1.

armin1(2,Varm1,Vmoulds,Vangle,0) .

armout1(2,Varmf1,Vangle,1) :-
   Vangle >= 3 ,
   Vangle =< 4 ,
   Varmf1 = 1.

armout1(2,Varmf1,Vangle,0) .
```

Appendix B.2 The compiled rule base

```
armin2(2,Varm2,Vmonomerl,Vangle,1) :-
   Vangle >= 7 ,
   Vangle =< 8 ,
   Vmonomerl = 1 ,
   Varm2 = 1.

armin2(2,Varm2,Vmonomerl,Vangle,0) .

armout2(2,Varmf2,Vangle,1) :-
   Vangle >= 1 ,
   Vangle =< 2 ,
   Varmf2 = 1.

armout2(2,Varmf2,Vangle,0) .

armin3(2,Varm3,Vmouldins,Vangle,1) :-
   Vangle >= 9 ,
   Vangle =< 10 ,
   Vmouldins = 1 ,
   Varm3 = 1.

armin3(2,Varm3,Vmouldins,Vangle,0) .

armout3(2,Varmf3,Vangle,1) :-
   Vangle >= 3 ,
   Vangle =< 4 ,
   Varmf3 = 1.

armout3(2,Varmf3,Vangle,0) .


lenstype(2,Vtype3,Vtype2,Vtype1,a1) :-
   Vtype1 = 0 ,
   Vtype2 = 0 ,
   Vtype3 = 1,!.

lenstype(2,Vtype3,Vtype2,Vtype1,a2) :-
   Vtype1 = 0 ,
   Vtype2 = 1 ,
   Vtype3 = 0,!.
```

Appendix B.2 The compiled rule base

```
lenstype(2,Vtype3,Vtype2,Vtype1,a3) :-
   Vtype1 = 0 ,
   Vtype2 = 1 ,
   Vtype3 = 1,!.

lenstype(2,Vtype3,Vtype2,Vtype1,a4) :-
   Vtype1 = 1 ,
   Vtype2 = 0 ,
   Vtype3 = 0,!.

lenstype(2,Vtype3,Vtype2,Vtype1,a5) :-
   Vtype1 = 1 ,
   Vtype2 = 0 ,
   Vtype3 = 1,!.

lenstype(2,Vtype3,Vtype2,Vtype1,a6) :-
   Vtype1 = 1 ,
   Vtype2 = 1 ,
   Vtype3 = 0.
```