# *ARTIFICIAL INTELLIGENCE TECHNIQUES FOR ASSEMBLY PROCESS PLANNING*

*By: Yen Ping Cheung*

*A thesis submitted for the Degree of Doctor of Philosophy*

***Department of Engineering,***

***University of Warwick***

***December 1991***

TABLE OF CONTENTS

*List of Abbreviations*

*List of Figures and Tables*

*Acknowledgements*

*Summary*

Page

**CHAPTER 1. INTRODUCTION**

1.1 Objectives....2
1.2 Motivations for the use of AI techniques in Process Planning....3
1.2.1 Change in Customer Taste....3
1.2.2 The Role of the Modern Engineer....4
1.2.3 The Nature of Process Planning....5
1.2.4 Shortage of Process Planners....6
1.2.5 Problems Associated With Experience....6
1.3 Relationship Between AI Planning And Process Planning....7
1.4 Outline Of Thesis....8

**CHAPTER 2. PROCESS PLANNING : TECHNIQUES & SYSTEMS**

2.1 What Is Process Planning....9
2.2 The Role of The Computer In Process Planning....11
2.3 Approaches of Computer Aided Process Planning....12
2.3.1 The Variant Approach....12
2.3.1.1 The Preparatory Stage....13
2.3.1.2 The Production Stage....16
2.3.2 The Generative Approach....17
2.4 Product Description....21
2.4.1 Coding and Classification....22
2.4.2 Interpreting CAD Data....23
2.4.3 Adding Feature To Design....26
2.4.4 Question/Answer Approach....26

Page

CHAPTER 2. PROCESS PLANNING : TECHNIQUES & SYSTEMS

2.5 Conclusions....27

CHAPTER 3. AI AND THE PLANNING PROBLEM

3.1 The Beginning....31
3.2 AI Planning....34
3.2.1 Importance of Planning....35
3.2.2 What's in a Plan....36
3.2.3 Plan Representation....39
3.2.3.1 State Space Plans....39
3.2.3.2 Representation Of State Space Data Structure....39
3.2.3.3 Action Ordering Plans....42
3.2.4 Plan Generation....42
3.2.4.1 Search Mechanisms....42
3.3 Green's Formulation....46
3.4 Kowalski's System....51
3.5 STRIPS....53
3.6 NOAH....56
3.7 TWEAK....61
3.8 Conclusion....66

CHAPTER 4. AI TECHNIQUES FOR ASSEMBLY PLANNING

4.1 Introduction....68
4.2 Assembly....68
4.2.1 Assembly Sequencing and AI Planning....71
4.3 Representing Knowledge....75
4.3.1 Production Rules....76
4.3.2 Semantic Networks and Frames....78
4.3.3 Non-Monontonic Reasoning....83
4.3.4 Predicate Logic and Prolog....86
4.4 Processing Knowledge....88

Page

CHAPTER 4. AI TECHNIQUES FOR ASSEMBLY PLANNING

4.5 Programming Techniques........................................90
4.5.1 Assembly Planning in Prolog........................................93
4.6 Conclusion........................................95

CHAPTER 5. AN AUTOMATIC ASSEMBLY PLANNER

5.1 Introduction........................................97
5.2 Results of Field Trips........................................97
5.3 Initial Tests........................................98
5.3.1 The Test Planner........................................99
5.4 Approach Of The Automatic Assembly Planner........................................107
5.4.1 Initialisation: Input of Goal States........................................110
5.4.2 Sequencing........................................112
5.4.2.1 Eliminate Redundant Goal States And Conflict Check........................................115
5.4.2.2 Generation Of Star List........................................124
5.4.2.3 Ordering Of The Goal States........................................127
5.4.2.4 Insertion of Star Goals to Main Goal List........................................139
5.4.3 Producing Action List........................................145
5.5 Finding The Pre and Post Conditions........................................150
5.6 Miscellaneous Clauses........................................156
5.7 Ball Point Pen Assembly........................................157
5.8 Summary........................................159

CHAPTER 6. DISCUSSION

6.1 Introduction........................................161
6.2 The AI Planning Situation........................................161
6.3 The Logical Approach........................................165
6.4 The CAPP Situation........................................167

CHAPTER 6. DISCUSSION

6.5 Philosophical Issues ........ 169
6.6 Findings From the Development Of AAP ........ 170
6.6.1 Assembly Constraints ........ 170
6.6.2 Input Description ........ 171
6.6.3 Assembly Space Conflict ........ 172
6.6.4 Level Of Detail ........ 172
6.6.5 Additional Reaction Face ........ 173
6.6.6 Assembly Database ........ 174
6.6.7 Problem Size ........ 174
6.6.8 Improving The Sorting Procedure ........ 175
6.6.9 Optimization Of The AAP Output ........ 176
6.6.10 Other Hardware Platforms ........ 177
6.7 Lessons And Future Systems ........ 178

**CHAPTER 7 CONCLUSION** ........ 184

*References*
*Bibliography*

*Appendix i* *Fundamentals of Theorem Proving and Prolog*
*Appendix ii* *A List of Some Process Planning Systems*
*Appendix iii* *Derivation of Formula in Sorting*
*Appendix iv* *Test Program For Connecting-rod Sub-Assembly*
*Appendix v* *Sample runs for Test Planner*
*Appendix vi* *Program listing For AAP*
*Appendix vii* *Test Runs For AAP*

## LIST OF ABBREVIATIONS

| | |
|---|---|
| *AAP* | *Automatic Assembly Planner* |
| *AI* | *Artificial Intelligence* |
| *CAD* | *Computer Aided Design* |
| *CAM* | *Computer Aided Manufacture* |
| *CAM-I* | *Computer Aided Manufacturing International* |
| *CAPP* | *Computer Aided Process Planning* |
| *CFL* | *Clausal Form Logic* |
| *GPS* | *General Problem Solver* |
| *IGES* | *International Graphics Exchange Specification* |
| *MTC* | *Modal Truth Criterion* |
| *PFA* | *Production Flow Analysis* |
| *TMS* | *Truth Maintenance System* |

LIST OF FIGURES AND TABLES

Page

Figure 2.1 An Operation Plan .......... 14
Figure 2.2 A Simple Matrix With Four Clusters .......... 15
Figure 2.3 Drawing Entity Record .......... 24

Figure 3.1 Sussman's Anomaly .......... 37
Figure 3.2 An Example Of A Tree .......... 43
Figure 3.3 .......... 47
Figure 3.4 A Refutation Graph For Figure 3.3 .......... 50
Figure 3.5 An Example To Illustrate ABSTRIPS .......... 55
Figure 3.6 An Example To Illustrate NOAH .......... 57
Figure 3.7 Levels 1 & 2 Of Example .......... 57
Figure 3.8 Level 3 Before Criticism .......... 58
Figure 3.9 Level 3 - After Criticism .......... 59
Figure 3.10 Level 4 - Before Criticism .......... 59
Figure 3.11 Level 4 - After Criticism By Resolving Conflicts .......... 60
Figure 3.12 Final Plan .......... 60
Figure 3.13 The TWEAK Planner .......... 62
Figure 3.14 Stacking Three Blocks Onto One Block .......... 64
Figure 3.15 An Incorrect Ordering .......... 65

Table 1. Some AI Planners .......... 66

Figure 4.1 Hierarchy Of Assembly Process Planning .......... 70
Figure 4.2 An Example Of a Precedence Network .......... 73
Figure 4.3 A Simple Semantic Net .......... 78
Figure 4.4 A Semantic Net To Illustrate Inheritance .......... 79
Figure 4.5 A Network Fragment .......... 80
Figure 4.6 An Example Of A Specific Frame .......... 81

Figure 5.1 The Piston-Connecting Rod Sub-Assembly .......... 100
Figure 5.2 An Initial State Of Piston-Connecting Rod Sub-Assembly .......... 101
Figure 5.3 Actual Initial State Of Piston-Connecting Rod Sub-Assembly .......... 104
Figure 5.4 Overview of AAP .......... 108
Figure 5.5 Redundant Goal Case .......... 115
Figure 5.6 Conflicting Goals Case .......... 115

Figure 5.7 Difference Between AAP and TWEAK.................................. 119
Figure 5.8 Primary Shaft Sub-Assembly................................................ 120
Figure 5.9 Goal State of Primary Shaft Sub-Assembly ............................ 121
Figure 5.10 Initial State ........................................................................ 123
Figure 5.11 C Clobbers Q....................................................................... 123
Figure 5.12 A Final Plan........................................................................ 124
Figure 5.13 Possible Outcomes For Goals, A & B.................................... 126
Figure 5.14 A Sequence......................................................................... 133
Figure 5.15 A Diagram Of a Simple Assembly........................................ 151
Figure 5.16 Post-Conditions For a Simple Clear State............................... 154
Figure 5.17 Initial State Of a Ball Point Pen Assembly ............................. 157
Figure 5.18 A Possible Goal State For The Ball Point Pen Assembly............. 157

Figure 6.1 Types of Goal Relationships................................................. 175
Figure 6.2 Interactions of An Assembly Process Planning System ................ 178

*I would like to express my deepest gratitude to the following people who have contributed towards this thesis:*

*Dr. A.L.Dowd, my project supervisor for accepting me as her research student, her support, advice and guidance throughout the course of this research;*

*Professor S.K. Bhattacharyya and Mr. John Fielden for accepting as a research student at the Department of Engineering;*

*SERC for providing me with a studentship to pursue this research;*

*Dr. Chris Hinde for teaching me AI and the many refreshing and interesting conversations on the subject;*

*Staff at the gearbox Process Planning Department, Rover, Longbridge for providing me with some useful practical information;*

*My husband, Vincent and my son, James who have both been very supportive of this research;*

*AIAI of Edinburgh University for providing the Edinburgh Prolog and Mr. Alan Hulme for setting up the Prolog environment on the Engineering Department SUNs.*

## SUMMARY

Due to current trends in adopting flexible manufacturing philosophies, there has been a growing interest in applying Artificial Intelligence (AI) techniques to implement these manufacturing strategies. This is because conventional computational methods alone are not sufficient to meet these requirements for more flexibility. This research examines the possibility of applying AI techniques to process planning and also addresses the various problems when implementing such techniques.

In this project AI planning techniques were reviewed and some of these techniques were adopted and later extended to develop an assembly planner to illustrate the feasibility of applying AI techniques to process planning. The focus was on assembly process planning because little work in this area has been reported. Logical decisions like the sequencing of tasks which is a part of the process planning function can be viewed as an AI planning problem.

The prototype Automatic Assembly Planner (AAP) was implemented using Edinburgh Prolog on a SUN workstation. Even though expected assembly sequences were obtained, the major problem facing this approach and perhaps AI applications in general is that of extracting relevant design data for the process planning function as illustrated by the planner. It is also believed that if process planning can be regarded as making logical decisions with the knowledge of company specific data then perhaps AAP has also provided some possible answers as to how human process planners perform their tasks. The same kind of reasoning for deciding the sequence of operations could also be employed for planning different products based on a different set of company data.

AAP has illustrated the potentialities of applying AI techniques to process planning. The complexity of assembly can be tackled by breaking assemblies into sub-goals. The Modal Truth Criterion (MTC) was applied and tested in a real situation. A system for representing the logic of assembly was devised. A redundant goals elimination feature was also added in addition to the MTC in the AAP. Even though the ideal is a generative planner, in practice variant planners are still valid and perhaps closer to manual assembly process planning.

# CHAPTER 1. INTRODUCTION

The needs of manufacturing industry have changed considerably in recent years due to keen competition at home and abroad. To address these changing needs, computers are increasingly used. This has led to the so called *islands of automation* within a manufacturing system where each *island* is an individual department or function in an organisation. There is now a recognised need to link these *islands* together to form a fully integrated system.

Where conventional computational methods have been fully exploited and exhausted, Artificial Intelligence (AI) techniques could be applied to form this vital link between the individual functions of a manufacturing system. One of the main areas of manufacturing, where computational and artificial intelligence techniques could be applied to improve efficiency is process planning.

The process planning task is a combination of both skilled and repetitive/tedious work. It involves generating operation/process sheets on how to manufacture/assemble a component/product given the specifications of the tools to be used and sometimes the length of the times of the jobs. These instruction sheets provide information to operators on how a task is to be performed.

There are two main types of process planning - process planning for machining processes and process planning for assembly. The former type is normally a well defined task where each machining process has its own characteristics, e.g. the speed and feed rate of the process. Most of this information can be found in handbooks or is normally a company's standard information. Thus this is an area of manufacturing where much automation or computerisation has taken place as evident in computer numerical controlled machines. The latter type is not so well defined and is often manually operated. There is also less documentation and also perhaps less *understanding* on the assembly process. As a result, it is an area of manufacturing where computerisation has been rather difficult to implement.

## 1.1 Objectives

The objective of this research is to investigate AI techniques that may be applicable to process planning. In order to achieve this the following are needed:

i) investigate what AI planning techniques are appropriate to assembly process planning as there are a large number of process planning systems for machining [1];

ii) implement an assembly planner using AI planning techniques;

iii) identify limitations of AI techniques, if any, to real applications.

## 1.2 Motivations for the use of AI techniques in Process Planning

### 1.2.1 Change in customer tastes

Today product life cycles have been reduced drastically and in some cases, product life cycles may be even shorter than the development lead time[1]. In order to remain competitive the lead time must be reduced to facilitate the production of new models and to cope quickly and efficiently with additions and modifications. Therefore the whole process from design through to process planning and manufacture must be speeded up. Since the process planning task is the vital link between design and manufacture, computational methods and perhaps AI techniques could be employed to speed up this stage of the production.

Not only have changes in people's requirements made automation of process planning a difficult task but also changes in products and the processes have contributed to this fact. A more flexible approach such as using AI techniques could perhaps be employed to tackle these changes.

[1] Time to design and getting into production by conventional means.

## 1.2.2 The Role of the Modern Engineer

The somewhat traditional definition of the engineer[2] as defined in the Chambers concise dictionary bears evidence to the reluctance of the engineering community in applying AI techniques. This definition reflects the traditional nature of the engineering world and the traditional view that we may have of engineering as a whole, i.e. one of solving problems that are of a highly deterministic nature such as design of electrical networks.

However in addition to these traditional tasks, the modern engineer today has to make decisions e.g. planning or making company decisions at management levels that are of a non-deterministic nature in order to keep up with the keen competition and changing tastes of the consumers. Manufacturing engineers in particular have to take decisions on the types of manufacturing processes and the tools to be used. The decisions made at this stage of the manufacturing process can be classified as the manufacturing strategy. Once the strategy has been decided, detailed tactics such as the types of finishings necessary to achieve the specified tolerances and surface finishes, the inspection processes required at each stage can then be made. Even at this latter stage, further decisions or estimates for those phases that do not directly involve a manufacturing process, e.g. transfer of work from one machine or one area to another have to be made. It can be seen that the

[2] 'one who designs or makes ,or puts to practical use, engines or machinery of any types, including electrical , designs or constructs public works, such as roads , railways, sewers, bridges, etc.

above tasks involve much planning and estimation as well as clerical and low level technical work.

Therefore, new aids such as planning and estimation tools should be available to enhance the engineer's work. Few (process) planning tools exist and those that do are analysis tools rather than synthesis tools. They often require existing plans to aid in plan formation of new products. A more flexible planner that is easily adapted to fit different scenarios is required. The planner described in Chapter 5 illustrates how flexibility may be incorporated by using AI planning techniques.

### 1.2.3 The Nature of Process Planning

Another reason for the slow development of process planning aids is that manual planning relies very much on intuition and is normally done in an unstructured manner. It relies a lot on the engineering judgement and experience of the planner. Therefore little attention has been paid to analysing the methods (which are normally unstructured and heavily dependent on the individual planner's personal judgement) by which the plans are or should be formed. This is also why the mechanisms used in process planning have not been published and/or formally taught to novices as these are often acquired through experience. When experienced planners leave their companies, their skills can be lost by the companies concerned. This has become a recognised fact and one of the solutions to retaining their expertise in an organisation is by designing AI systems.

### 1.2.4 Shortage of Process Planners

Due to the nature of the process planning activity as described above, it is therefore not surprising that there is now a severe shortage of experienced and skilled planners. This is because manual planning requires continual re-education about company standards and policies which takes a long time to learn. Even though academic institutions are trying to shorten the period of training to four or five years, there is still a shortage of experienced process planners today [2].

### 1.2.5 Problems Associated With Experience

It should also be noted that decisions made by relying solely on experience are only approximate decisions and may therefore vary amongst different process planners. This could lead to inconsistencies and recommendations that may be more costly than necessary. Also the experience of process planners may not be directly applicable to new processes and designs. Therefore in situations where planning is often required for new designs, relying solely on the experience of process planners may sometimes not provide the best solution [3].

Therefore with the above reasons this project aims to investigate the possibilities of the application of AI techniques to process planning as an alternative to relying solely on the experience of process planners, i.e. manual systems.

## 1.3 Relationship between AI planning and Process Planning

AI techniques are best at solving problems that are of a non-deterministic[3] nature such as process planning. Planning has been a classic research topic in AI for more than two decades. A plan in AI terminology can be defined as a linear or partially ordered sequence of actions to achieve a desired goal. A possible plan for making a cup of tea could be:

fill a kettle with water

bring the water to boil

put some tea into a cup

pour the water into the cup.

It is believed that whether one is devising a plan for making a cup of tea or a plan to manufacture a product as in process planning, the same kind of reasoning must be employed, i.e. that of achieving an ordering of operations from a set[4] of given operations and constraints. In manufacturing, the operations would be manufacturing processes.

It is also believed that there are similarities in process planning across different domains, e.g. process planning for the machining of components and process planning for the construction of buildings. The former activity

[3] Cases where there are many possibilites to a solution, an example is in medical diagnosis where symptoms could lead to multiple diagnosis.

[4] A set is a collection of physical objects or mathematical concepts.

requires the sequencing of the machining operations for manufacturing those components while the latter involves the sequencing of construction activities for building parts of faciliities [4]. The common feature between these two applications is the logical decisions made in the sequencing of tasks. Thus while the work described in this project originated from the sub-assemblies of the gearbox, some of the work may in principle be applicable to the general problem of process planning.

## 1.4 Outline Of Thesis

Initially, visits were made to the process planning departments of Rover car plants at Cowley and Longbridge to obtain an insight into process planning. The process planning problem in general was analysed and an investigation with regard to relevant AI techniques and how these techniques may be applied to aid the process planning problem was carried out. Chapter 2 describes the process planning task and the approaches that have been used to automate this task as well as some examples of existing process planning systems. Chapter 3 covers AI planning and Chapter 4 describes AI techniques that may be applicable to process planning. A planner based on AI planning techniques was then developed to illustrate the feasibility of applying AI techniques to real world problems such as process planning in the manufacturing domain. Its development is described in Chapter 5. Discussion on the research is found in Chapter 6 and the conclusions are in Chapter 7.

# CHAPTER 2. PROCESS PLANNING:TECHNIQUES & SYSTEMS

## 2.1 What Is Process Planning

The Society of Manufacturing Engineers define process planning as: '*the systematic determination of the methods by which a product is to be manufactured economically and competitively*' [5].

Generally, this involves a series of steps which can roughly be broken down into :

i) Interpretation of the design data which can be in the form of blue prints in the traditional way or on a CAD system.

ii) Determination of the types of operations or processes; requirements of the products such as batch size, raw material properties, dimensional tolerances; tools for making the products, fixtures plus any other special requirements.

iii) Determination of the operation sequence.

iv) Calculation of the individual and overall times of the operations.

v) Production of the operation sheets (or process sheets).

Operation sheets (also known as process plans or route sheets) contain the detailed instructions to make the product, e.g. series of operations, machines and tools required to make the product, etc. These detailed steps dictate the final quality, cost and rate of production.

Therefore process planning is of utmost importance to a manufacturing system and must be completed before detailed costings or further decisions can be made. Once the plan has been drafted all the available information is then passed to the shop floor and production control department where a coherent schedule of the work taking into account the man/machine resources and existing workload can be made. If it is not possible, then the manufacturing engineer may have to revise the strategy to suit the existing situation.

All the above phases of work demand a combination of skilled engineering tasks as well as much clerical and low level technical tasks, i.e. work of both deterministic and non-deterministic nature. To summarise, the engineer or process planner is faced with a considerable amount of decision making, data references and calculations during these phases of work. Very often the nature of this work is tedious, time consuming and error prone. The errors made could be due to the engineer having to make frequent cross references to data on machine capabilities, process data, company standards, etc.

The above description provides an overview of the traditional process planning task in general (i.e. both machining and assembly). As can be seen, process planning is a task that requires both a significant amount of time and experience. It is evident that computers can be used to perform those tedious calculations and data storage that they are so good at. In particular the reasoning involved in the determination of the sequence of operations (see ii. above) is similar to the reasoning in AI planning. The following section describes the involvement of computers in process planning.

## 2.2 The Role of The Computer in Process Planning

In manufacturing great strides have been made in the development of computer applications in design (as seen in the many Computer Aided Design (CAD) Systems available today) and manufacture (Computer Aided Manufacturing Systems) which was not parallelled by process planning. This situation has also made process planning a bottle neck in the manufacturing process when companies attempt to adopt flexible automation in order to keep up with competition. Until recently this was probably due to the limitations of the hardware and software. Thus the use of computers in process planning was not broadly addressed until the beginning of the 1970's. The first system, CAPP (Computer Aided Process Planning) developed by CAM-I (Computer Aided Manufacturing International) was first presented in 1976 at the 1976 NC conference [6]. After that many similar systems were developed to address this need of integration of design with manufacture. In 1991 it was indicated that 20% of the manufacturing industries would utilise an integrated system of Material Requirement Planning and CAPP and that 50% of the process plans used to produce parts or assemblies would be computer generated [5]. Despite the comprehensive list of CAPP systems as shown in Appendix ii, it is currently unclear as to whether these forecasts were or could be realised. As seen from Appendix ii, there has been much research in the area of process planning for machining but little work has been reported in the area of assembly.

## 2.3 Approaches of Computer Aided Process Planning

Early attempts at automating process planning consisted primarily of developing computer-assisted systems for report generation, storage and retrieval of existing plans. Currently there are two main approaches to computer aided process planning; variant and generative.

### 2.3.1 The Variant Approach

This approach is comparable with the manual approach where a process plan for a new part is created by retrieving previous process plans for a similar part and then making the necessary changes to the plan if necessary to cater for the new design. However in comparison with the manual approach, this approach is superior in that the information processing capabilities are increased. Consequently, the time required to *search* through existing plans is greatly reduced if standard procedures and company standards have also been incorporated.

Variant systems store completed plans as well as manufacturing information from which process plans can be obtained. They use library retrieval procedures to find standard plans for similar parts. It is also estimated that when used effectively, these systems can save up to 40% of a process planner's time [7]. The CAPP system mentioned in Section 2.2 is an example of a typical variant type system. In this particular system, previously prepared plans are stored in a database. Whenever a new design is planned, a process plan for a similar design is retrieved and subsequently modified by the process planner to satisfy the requirements of the new design. The Group Technology technique is used to code and classify parts into families. A process plan that is used by a family of components is called a standard plan. These standard plans are stored permanently in the database with a family number as its key. The logic of variant systems is based on the grouping of parts into families. Common manufacturing methods can then be identified for each of these families. A family is represented by a family matrix which includes all the possible members. In general, variant systems have two operational stages:

preparatory stage;
production stage. [8]

#### 2.3.1.1 The Preparatory Stage

Existing components are coded and classified and the grouped into families and a family matrix is also constructed. Process plans already prepared for these existing components are also summarised. Standard plans are then stored in a database indexed by family matrices.

Even though there is no set definition for families, family formation is usually based on manufacturing features. For process planning purposes, all those requiring the same process plans belong to the same family. From a user's point of view, a family could be formed from components having exactly the same process sequence. This means that very little or no modification of the standard plans is necessary. In this case, the family size may be very small. On the other hand, if there are many variations in a family, e.g. grouping all parts requiring a common machine into a family, then much plan modifications would be required.

There are two main types of code for coding the parts, i.e. design code and operation code. The design code is equivalent to the design features of parts. The operation code represents a series of operations that are required. Figure 2.1 shows an example of an operation plan.

| operation code | operation plan |
|---|---|
| 01 join | glue |
| 02 press | check reaction face, press parts together |

**Figure 2.1 An Operation Plan**

The coding of part families can be done manually or by using a computerised approach such as Production Flow Analysis (PFA). PFA was first introduced by J.L. Burbridge to solve the family formation problem for manufacturing cell design. [9,10]. In the PFA technique a large matrix is constructed where each row represents an operation code and each column represents a component. The matrix can be defined as $M_{ij}$ where i represents operation codes and j represents components. So, $M_{ij} = 1$ if component j has operation code i, otherwise $M_{ij} = 0$. The objective is to bring together components that require a similar set of operation codes into clusters. Thus parts that are grouped into a cluster, belong to the same family.

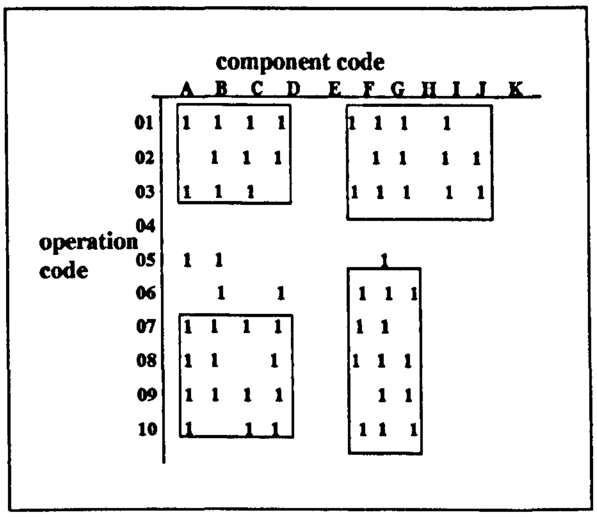

Figure 2.2 A Simple Matrix With Four Clusters

A family of matrix file can be set up based on the clusters. From the above figure there are four clusters and therefore there are four possible families. Each of these families can be given a name and standard plans associated with the particular families concerned can be developed and attached to the families concerned. Associated plans can then be retrieved given say, the family name. Commercial database management systems can be used to set up this structure.

#### 2.3.1.2 The Production Stage

New components are planned at this stage. A new component is first coded and the code is input to the part family search procedure to find the family to which the component belongs. The family number is then used to retrieve a standard plan.

The search procedure is simply a matching of the family matrix with a given code. If a match can be found then the part belongs to the family and the standard plan for the family can be used for that part. The human planner then modifies the plan as necessary to satisfy any new design requirements. New information can also be added to the system but must be managed properly in order to keep the family matrix intact and the database consistent.

CAM-I's CAPP system is a variant system developed by McDonnell Douglas Automation company. It is a database management system written in FORTRAN. The coding scheme for part classification and output format is added by the user. It has a (maximum) 36-digit alphanumeric code.

Recent work on computer aided process planning systems has been focussed on incorporating the experience of process planners in order to eliminate the human process planner from the entire planning function. This approach is known generally as the generative approach.

## 2.3.2 The Generative Approach

In this approach, process plans are synthesized in order to create a process plan for a new component automatically. Plans are derived from the manufacturing data base without human intervention. The input to these systems would be design data. The advantages of such systems over the variant approach are as follow:

i) consistent plans can be generated quickly;

ii) new plans can be made easily;

iii) has potentialities for linking up to other manufacturing facilities such as CAM, manufacturing management, etc.

The key requirement in developing these systems is to encapsulate the logic of process planning into these systems. To provide the 'missing link' between CAD and CAM, the relevant design data must also be clearly and precisely defined for the process planning system. Ideally, a generative system is a turnkey system with all the necessary decision logic. Since the system contains all the information needed for process planning, no preparatory stage is required. However the current definition of generative systems has been relaxed to give a less complete system since the difficult technicalities of the problem have been realised. Thus systems with some kind of built-in decision logic are often called generative systems. There are basically two approaches to developing generative systems, forward and backward process planning. In the former, the approach is that given the initial knowledge of the design and the type of processes, an attempt is made to find a plan for that design. In the latter, using the final product (i.e.

either the machined component or the final assembly) an attempt is made to obtain the sequence of operations to satisfy the initial requirements.

The decision logic is the heart of these generative systems. A common approach is to write manufacturing process capabilities in the form of *IF-THEN* expressions by taking information from handbooks and/or interviewing[10] process planners. Decision tables or decision trees have been used to represent manufacturing knowledge.

In drawing up the decision table, a question is asked at each junction or a decision is made at each junction. An action block is included for each true condition and for a false condition, another branch or process can be directed to the end of the logic block. In computing terms, this is rather like developing a flow-chart for a computer program.

```
e.g.    if a1 then do n1 ..... .
        else if a2 then do ....
        else    stop.
        n1
        procedure n1
        if ....
```

[10] this could involve performing knowledge acquisition as recommended for expert systems design. See Chapter 4 for more details on knowledge acquisition.

Decision tables have long been a popular method of presenting complex engineering data. They can be implemented using general programming languages such as FORTRAN, COBOL, PASCAL, etc. This is rather like writing a computer program except that in this case the application is in process planning. In this approach, the logic of the application is embedded in the flow of the program. Therefore whenever any changes in the decisions occur the whole program has to be reviewed again [11].

Another approach is to use AI techniques since the subject attracted much interest and curiosity due to the applications of AI seen mainly in the form of so-called expert systems[11]. Most of these systems have implemented manufacturing knowledge in the form of *IF condition THEN action* statements based on machining processes which is similar to the decision trees approach. Examples of AI process planning systems can be found in [12,13,14,15].

Probably the first AI based process planning system to be reported in the literature, GARI is an example of a rule-based system. It consists of a knowledge base of approximately fifty production rules[12] for machining components. By assigning weights to different pieces of advice, it attempts to find the best answer. The plan generation technique is through successive refinements where it assumes that all machines are available initially. At each iteration new assertions are produced which may then imply a partial

[11] A piece of computer program that is said to be able to derive conclusions and new information based on existing information - something like decision making.
[12] See Chapter 4 for more on production rules.

ordering. It stops when it has exhausted the list of active rules by the current set of assertions. EXCAP is another production rule based process planning system for the selection of cutting sequences for prismatic parts. TOM, an acronym for *Technostructure of Machining* is a rule-based system for generating machining (*hole*) sequences. Hi-Mapp is another AI process planner for machining which is based on manufacturing knowledge that is stored about machining processes.

Most CAPP systems e.g. [12,13,14] (whether variant or generative) do not make use of AI planning techniques such as those mentioned in Chapter 3. Ad hoc additions of manufacturing knowledge in the form of if-then rules is a common feature of expert systems for process planning. A more organised approach to this is sought. Therefore, this research had concentrated on applying the fundamental AI planning concepts to assembly process planning.

A semi-generative approach has also been suggested when researchers failed to develop a truly generative system [15]. This approach combines both the generative and variant approaches. It contains a degree of decision logic to enable a plan to be synthesized but it is then manually modified before being utilised. Semi-generative systems are attractive from the system developer's point of view as their competitiveness over variant systems is increased and are thus more marketable than variant systems. However the ultimate aim is still to solve the planning problem using the generative approach.

It can be seen from the large number of computer aided process planning systems available today that the manufacturing logic required for process planning is highly variable. Despite being a highly variable task, the nature of the logic of the sequencing of operations remains fairly static irrespective of the type of application. Hence the application of AI planning research to address this problem is possible.

## 2.4 Product Description

In the early days, the designer and manufacturer were the same person. As machines and methods of production (working practices) become more sophisticated the separation between these two roles became greater and greater. Today the designer is mainly responsible for designing a new product and the manufacturing phase is further divided into various stages, i.e. planning on how to make the product and then actually making it.

The design phase can be divided into two main phases:

conceptual phase (when an idea is conceived);
geometrical phase (where ideas are transformed into a design).

Due to the separation of the design and manufacturing tasks, the engineering drawing became the principal form of communication between designers and manufacturers. Today advances in computer graphics made it possible to replace drawings of parts with the electronic form using tools such as CAD. In a manual system, the process planner would study the engineering drawing, interpret it and then provide the manufacturer with a set of instructions on how to fabricate the design. This is still very much the case where automation of process planning has been attempted, i.e. interpretation of designs is still very much a manual process before relevant data can be input to the process planning system. However in order to achieve full integration of the manufacturing system (i.e. CAD and CAM), this interpretation process has to be automated as well. The input format to the process planning system affects the ease of use as well as the capability of the system. A tedious input format would defeat the purpose of automation while inaccurate and insufficient data would reduce the competence of the system. Since this part of the automation process is a vital path to integration, the following sections examine the types of input format available and the types of problems associated with each of them.

### 2.4.1 Coding and Classification

A popular approach of variant systems is to make use of a coding and classification system to categorise parts into families. In other words parts with similar characteristics both from a design and manufacturing point of view are grouped together into families. In order to classify parts into families, a method for classification is necessary. There are two main methods of coding: hierarchical and chain. In the former one, it is usually a numerical code and the meaning of a code is dependent on its previous number, e.g. if the first digit represents a shaft then the second one defines a type of shaft and so on. In the chain type, each digit represents a distinct bit of information independent of its previous digit. Each digit represents a small building block of the complete part, e.g. one digit may represent the type of part and another may represent the dimensions. Examples of coding and classification systems are MICLASS [16] and OPITZ [17].

This approach is not efficient in situations where there is much variety amongst products with very few members per part family (as much time would be spent on creating new families and new process plans). Users of such a system would have to be familiar with the coding scheme. Thus the benefits of coding systems cannot be achieved in generative systems which aim to synthesize plans rather than relying on the information provided by coding systems.

As the code and classification concept originates from machining (assignment of codes to parts and part programs - part program for one member of a part family can be used for other members of the family), this method of part description may not be feasible for assembly process planning which do not require the generation of a part program.

### 2.4.2 Interpreting CAD data

Another attempt at obtaining product description information is by interpreting CAD data directly as a front-end to process planning systems [18]. As there is no single universal CAD package, the danger of this approach is that the techniques involved may only be applicable to the specific CAD package that was used. This means that different *translators* may be needed for each CAD package. As can be seen from the technique described below, the translation process itself can be a tedious process.

The devised algorithm for translating 2-D CAD data by Wang [18] consists of defining nine fields in a record to represent each drawing entity, i.e.

| | | | | | |
|---|---|---|---|---|---|
| Type | line/arc | (x,y) | (x,y) | (x,y) | radius |
| | | start | end | centre | |

**Figure 2.3 Drawing Entity Record**

All line and curve entities are extracted from the CAD data file. These entities are then grouped, i.e. all solid lines into one group and all hidden lines in another. The external contour is then recognised by searching through all the solid lines. The lines and arcs (based on the x-coordinates) are then sorted. The basic idea is to connect the line segments so that the ending point of the nth segment is connected to the starting point of the (n+1)th segment. For example, the rule for recognising a straight cylindrical type of surface for a pair of adjacent points, n, n+1 is:

if $x_n < x_{n+1}$ and $y_n = y_{n+1}$

where $(x_n, x_{n+1})$ are x-coordinates of points n, n+1

$(y_n, y_{n+1})$ are y-coordinates of points n,n+1 respectively.

However this method was based on a 2-D wire frame model and would not work for 3-D models. Also wire frames can sometimes provide ambiguous interpretations (e.g. which is solid and which is not). For accuracy, interpretation of a 3-D model is preferred. This would require even more tedious algorithms which may only be feasible for the CAD system on which they were developed. At a gross level, this means that new algorithms would have to be devised for the (many) different CAD systems available. This defeats the aim of flexibility.

In addition to recognising the surface feature, the relationships between mating pairs of features are also essential in the case of assembly process planning. These kinds of algorithms, in particular the one described above would not be sufficient. Alternatively, CAD data could be converted into a neutral format, e.g. IGES (Initial Graphics Exchange Specification, 1980, NBS USA). With IGES each CAD system requires a pair of IGES pre-processor and IGES post-processor. It was suggested that vendors of CAD systems provide these IGES translators. However the knowledge that is extracted directly from IGES can only provide the topological and geometrical information. Functional and feature information are also needed for the process planning stage. It has also been reported that because different CAD systems may be running on different types of computers, the process of reading the data from the tape of another CAD system may not be as straight forward as anticipated due to the incompatibility of the tape format among different computers [19]. Another disadvantage is that revisions of IGES may not be able to keep up with the advancement of different CAD systems. Hence further discouraging the conversion of CAD data into the IGES format for feature description.

### 2.4.3 Adding Feature to Design

Another possible approach is to specify explicitly the features of the parts during the design stage. Thus enabling the encoding of feature based information with the design. This requires the direct design of parts using features (instead of the conventional method of using lines, arcs, etc.). Such features (in the *ready* form) are then stored in the system's library which could be used for process planning. This kind of system may however require a lot of human interaction such as verification of results. Presently similar CAD systems using a knowledge-based approach are available in the market but their credibility would require further investigation.

These so-called intelligent CAD systems provide a design language for the designers to create new designs. The data generated from these new designs can then be used and/or manipulated in a manufacturing environment.

### 2.4.4 Question/Answer Approach

The simplest alternative is to describe in words the design to the process planning system. In this approach, the aim is to obtain the relationships between parts from answers to a series of questions about the mating of part pairs. This could be very cumbersome and tedious from the user's point of view as it depends on the number and the structure of the questions that are asked by the system. Related work in this area can be found in [20,21,22,23].

## 2.5 Conclusion

The process planning task can be sub-divided into two main parts, i.e. the decision on the types of manufacturing processes to be used and the ordering (or sequencing) of those manufacturing processes. As different manufacturing systems vary in their decisions on the types of manufacturing processes to be used, no one single process planning system can satisfy all of the different manufacturing needs. Hence, in order to achieve the first sub-task of the process planning activity, manufacturing databases pertaining to the individual needs of manufacturing systems have to be created. As suggested in Section 2.3.2 that common knowledge can be found in handbooks such as [24] for machining (unfortunately very little has been written on assembly), but company specific data (such as that on special processes that originate from the company and is possibly a trade secret) has also to be obtained from the company itself. System developers adopting the expert systems approach have suggested various methods of eliciting the relevant knowledge from the process planners. As knowledge elicitation is another major subject by itself and is also appropriate to the generative approach, further comments on the subject are included in Chapter 4.

When using the variant approach, standard plans must be changed when major factory renovation takes place. Also, this approach is of little use if there are many families and very few similarities between components. This is because a lot of time would be spent on adding new families and standard plans thus increasing the time spent in the preparatory stage even further. In this case, the generative approach would be more suitable. The variant approach is more useful if the manufacturing system makes similar or identical components for perhaps different types of assemblies.

The generative approach is recommended when there exist many variations among the types of products made which makes it difficult to code and classify them into a relatively small number of families. However for a moderate number of component families and moderately sized manufacturing systems, the variant approach may well be the most economic automated planning system alternative at the moment as variant systems are better understood than generative systems. Therefore, variant systems are cheaper, easier and faster to develop. On the other hand, if the situation is such that a high degree of flexibility is needed (such as frequent changes in designs) and integration of a CAD system further upstream with a CAM system is desired, then the generative approach is the ultimate solution. As mentioned in Section 2.3.2, very few truly generative systems are available today, so companies adopting this approach should be prepared to spend much time and effort on a technology that at the moment is still developing.

Ideally the front-end of the process planning system should be independent of any representation or specific system. However this vital part of the manufacturing system is still a bottle neck to fully integrated systems at the moment. Incorporation of feature information at an early stage seems to be an attractive approach as such information is already available at that stage partly as specifications of the design and also partly as knowledge held by the designers themselves. This means that designers would have to adopt a different design philosophy, i.e. including engineering features (such as grooves and slots) as well as topological and geometrical data. On the other hand, option of interpreting CAD data could also involve much tedious programming which may have to be changed whenever the CAD system is upgraded or replaced. The controversy of which is the better approach, i.e. directly interpreting CAD data and describing features through independent means (like question and answer type or incorporation of feature information with design) still remains. However the final result of both approaches must be that the method to obtain the information should be simple and easily applied to other situations and at the same time information must not be over-specified.

Sometimes the simplest approach is the best and also the cheapest. The possible extension of the question and answer approach is to deduce a minimum number of questions that can be asked in order to obtain answers to the relationships between mating pairs. This could be classified as a kind of knowledge acquisition exercise and would involve much time spent with a willing process planner at the company concerned. In order to avoid disrupting the normal routine of the company, a plausible arrangement could be to co-ordinate the work of the person eliciting the knowledge with a process planner who is about to retire and whose time and effort is devoted solely to this project. This is because past experience has suggested that no matter how cooperative the expert may be, he/she is bound to be called away on another more *urgent* matter.

At the moment, there are not many options available for interfacing design to process planning. If the process planning system is a variant type then a coding and classification system is probably the best option but it will have to be adapted for assembly purposes. As for generative systems, there is no standard input format and the form of input usually depends on the approach used by the individual process planning system. A question and answer (as in expert systems) type of interface is the most commonly used at the moment. It is believed that in the long term, future design systems should invariably contain some elements of manufacturing data which could be of use further downstream, e.g. process planning.

This research suggests that there is a similarity between the reasoning used in determining the sequence of operations in process planning and AI planning. AI planning techniques could therefore be used as a basis for process planning systems (i.e. solving the second part of the process planning task). The next chapter describes the study of AI and aspects of AI planning that are relevant to process planning.

# CHAPTER 3. AI AND THE PLANNING PROBLEM

## 3.1 The Beginning

Machines that characterised the Industrial Revolution in the 1700s were machines that helped extend and multiply people's physical capabilities. Today, people are confronted by a different category of machines that are said to extend and multiply human mental capabilities, i.e thinking machines. Whilst the former type of machines met much in the way of opposition (such as the Luddite Movement founded in Nottingham in 1811 which perceived these machines as a diabolical danger to the workers of the textile industry), the announcement and creation of the latter types of machines were met with mixed feelings. While humans accept that machines could perform physically better (since birds can fly and humans cannot), it is more difficult to accept machines that equal or even better human mental capacities. The study of these machines which is closely associated with the way the human mind (or brain) works is therefore the prime concern of Artificial Intelligence or AI.

The study of AI has been the subject of philosophers, physicists, mathematicians, computer scientists, etc and has its roots as early as the days of Plato (427-347 B.C.). He recognised the similarity between certain aspects of human thinking and the apparently determined cause-effect behaviour of machines. He believed in a *corpuscular physics* based on fixed and determined rules of cause and effect. Thus if human decision making is based on such interactions of basic particles then the decisions too must be pre-determined.

In more recent times, Ludwig Wittgenstein (1921) described human thought as comprising certain elementary facts in his first major work, Tractatus Logico-Philosophicus. In his model there are propositions about relations between these elementary facts and also certain allowable transformations on these propositions to yield composite propositions. In simple terms, we cannot think what we cannot say; we cannot say, or ought not say what is meaningless in the language that we are speaking. He has made two major points that have a direct bearing on the intellectual roots of AI, i.e. he made a direct link between human thoughts and a formal process that can be described as computation. This was re-stated two decades later in the Church-Turing thesis at the famous Dartmouth conference in 1956. It asserted that if a problem presented to a Turing machine is not solvable by that Turing machine, then it is also not solvable by human thought. This formed the basis of the AI movement and until today the measure of the intelligence of a machine is still the Turing test. An interrogator can ask questions of either the person or the computer but does not know which of them is which. The aim is to determine which of them is the person and which is the machine. If the machine succeeds in fooling the interrogator to believe that it is the person, then it can be concluded that the machine can think. Some people believe that no machine will ever pass the test due to the complexity of the human mind[1].

[1]Kurzweil predicted in his book, 'The age of intelligent machines' that a computer will pass the Turing test between 2020-2070.

There are many definitions to the term AI and it is still very much a topic of debate today. The definition of *artificial* is easily taken care of by *machine* but it is *intelligence* which is difficult to define. This can be illustrated by the fact that so long as a problem remains unsolved and retains its mystique then it is said to require intelligence to solve. However once it is solved and one knows how it works then it does not seem to be any different from any other piece of computer program (therefore it does not require intelligence). Amongst all the definitions of AI, perhaps the most valid definition is: *the study of computer problems that have not yet been solved,* by Minsky since the 1960s. This is certainly the case in AI planning.

From a practical point of view, AI can be defined as:

*The study of how to make computers do things which, at the moment, people are better* [25]

Therefore as a branch of Science, it is concerned with developing concepts and vocabulary to aid the understanding of intelligent behaviour in people. In terms of engineering, it is the task of engineering a technology of thought, i.e. it is concerned with the concept, theory and practice of building intelligent machines (e.g. expert systems, vision systems, etc).

One of the hard and fast results to come out of the first twenty years of AI research is that intelligence requires knowledge. Thus the main difference between conventional computing and AI programming is that in the former the knowledge to solve a problem is implicit in the algorithm whereas in the latter knowledge is explicitly defined (as proposed by Plato and Wittgenstein). Since it is explicitly defined, an AI program is said to be more *flexible* than conventional programs in that new and relevant knowledge can be added to the original programs for new applications. In the case of conventional programs a completely new program would have to be written for different applications.

## 3.2 AI Planning

Much of AI planning research has concentrated on domain independent planners, i.e. developing plan representation and plan generation techniques that are expected to work on a reasonably large variety of problems. In general, the study of AI planning involves the computation of a sequence of operations to perform a specific task before it is actually carried out. For example, the recipe for baking a sponge cake is a plan. Thus a plan consists of a sequence of operations or instructions to perform a task.

Historically, the role of AI planning is based around the automatic generation of plans of actions to be executed by autonomous robots in real-time environments such as nuclear plants or battle fields. These applications reflect the extensive funding by the US military. However the complex sensors that are needed to provide the planners with necessary knowledge of their environments do not actually exist at the moment. Consequently most

AI planners were applied and tested on simple robot manipulation domains such as the *Blocks World.*

In the *Blocks World* there is a set of labelled blocks (usually A, B, C, etc), a table with infinite space on which the blocks rest and a robot which performs simple block manipulations. The only move allowed is either to stack a block onto the top of another block or move a block to the table. The constraint in the system is that a block can only be moved if it and its destination is said to be *clear.* Also the top of each block has only enough room for one other block and the robot can only handle one block at a time. Examples using blocks described in subsequent sections will be based on the Blocks World.

### 3.2.1 Importance of Planning

It is the capability of humans to plan ahead (thus preventing destructive or fatal consequences) that make humans more superior to other beings on earth. The ability to plan is therefore considered to be an intelligent aspect of human behaviour. AI planning research has been linked closely with early attempts at producing general problem solvers such as the GPS (General Problem Solver) [26] as planning can be envisaged as solving a problem prior to its execution. The technique of means-end analysis was first used in the GPS. The fundamental concept of means-end analysis is to detect the differences between the current state and the goal state. Once this has been detected, an operator is then used to reduce this difference with the hope of bringing the problem state closer to its solution.

This technique is said to simulate human problem solving behaviour [27]. Therefore, besides developing planners for military purposes, it is also the intention of AI planning to shed some light into how the human mind works which at the moment is still a mysterious domain.

As mentioned, everyday life planning is an essential aspect of intelligent human behaviour. Apart from preventing fatal consequences, the planning function is also at the heart of a manufacturing system. It is essential for making the system work in the first place. Through ingenious planning, the system is able to manufacture at a competitive cost and thus ensure a place in today's keen market.

## 3.2.2 What's in a Plan

A plan can be one of the possible solutions to a problem. Planning is easy and straight forward if tasks or goals can be decomposed into disjoint (or non-conflicting) sub-goals. So these sub-goals can be achieved in any order or in parallel. Problems occur when there are conflicts between sub-goals. Consider, a simple planning problem consisting of two sub-goals say, $G_1$ and $G_2$. It might be possible to solve it by doing $G_1$ first and then $G_2$. However when doing $G_2$, $G_1$ gets undone. The same situation might also exist by doing $G_2$ first and then $G_1$. A classic example where one sub-goal gets undone by another is best illustrated by Sussman's Anomaly in the following section.

In Sussman's example, there are three blocks, *a*, *b* and *c* as shown in Figure 3.1. The goal consists of two sub-goals, i.e. *stack b onto c* and *stack a onto b*. In the initial situation, *c* is on top of *a* which is on the table while *b* is on the table with no other blocks on top. A computer planning program would consider each sub-goal in turn. In order to achieve the first sub-goal, i.e. *stack b onto c*, this could be done by directly stacking *b* onto *c*. But *a* is underneath both *b* and *c*, therefore the first sub-goal cannot be done first. If the second sub-goal is considered first, then *a* must have no blocks on top. Thus, *c* is unstacked and put onto the table. Once the top of *a* is *cleared* then it can be put onto *b* to achieve the sub-goal. But, the goal requires *c* to be below blocks *b* and *a*. Therefore, it is also not possible to start with the second sub-goal.

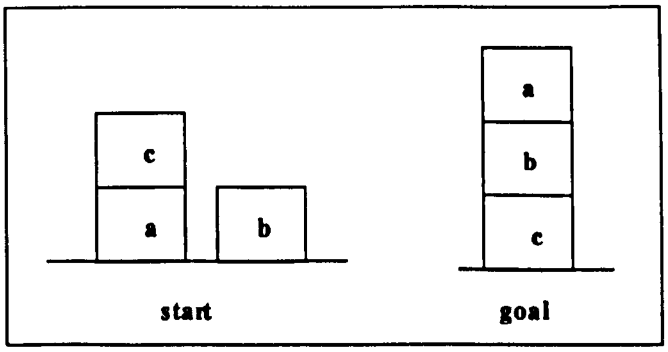

Figure 3.1 Sussman's Anomaly

Planning is difficult when constraints exist which is often the case in the real world. This aspect of the planning problem, conjunctive planning as opposed to disjunctive planning where there are no interactions between sub-goals, has been the centre of much AI research for the past two decades.

Unlike most AI problems, conjunctive planning cannot be solved by just using well known AI methodologies such as divide-and-conquer as illustrated in the example above. In order to tackle this type of problem, the planning problem must be considered as a whole rather than as independent sub-problems or perhaps, to solve G, a bit of $G_1$ has to be done first followed by doing a bit of $G_2$, then finishing $G_1$ and finally finishing $G_2$. However some problems (if they cannot be broken down into sub-problems) may be too large to be solved in practice [28]. To date very little, if any, of this work has shown any impact on real world problems. Instead the work tends to be limited to research laboratories and is based on the Blocks World examples. As research in AI planning has been going on for more than two decades, there is now a rich and growing armory of AI planning systems. The existence of the large number of AI planners today is evidence of the competitive spirit in this field. However little effort has been made to apply these planners to real world problems which were considered as unchallenging and uninteresting.

Due to the large number of AI planners in existence today, any work involving AI planning must begin with a study of some significant AI planners and their characteristics. This is because much can be learnt from these planners and also lessons drawn from them. A formalism for the planning problem is essential for representing it in a computer. This will be described in the next section.

## 3.2.3 Plan Representation

There are basically two main types of plan representation techniques: state space plans and action ordering plans.

### 3.2.3.1 State Space Plans

A problem is often characterised by state descriptions. The situation/condition of the problem at each stage of its solution is defined by a state data structure. A state is a snapshot of the problem at a given point in time. Operators are used to transform the problem from one state into another. At different times, the problem is said to be in different states.

There are three types of states in a state space representation scheme, i.e. initial state, goal state and intermediate state. The initial state is the state at the start of the problem and the goal state is the problem in its final state, i.e. the state of the solution to the problem. The intermediate state is any the state that exists between the initial state and the goal state.

### 3.2.3.2 Representation Of State Space Data Structure

Logical systems can be used to represent the state space data structure. Plato and Aristotle provided the basic approach for computer-based reasoning, i.e. that of devising a way of representing knowledge in symbols and then doing reasoning about that knowledge by manipulating those symbols in various ways. Predicate Calculus is a form of symbolic logic that allows us to express logical concepts (meaning of sentences or utterances) and then manipulate them using various rules to obtain new knowledge from old (i.e. making inferences). This is normally used by logicians, mathematicians and philosophers. Thus the meaning of sentences is a proposition which consists of terms, of which there are two types, predicates and arguments. Predicates are relation names and usually correspond to verbs in sentences and arguments are the objects that are related and usually correspond to nouns. For example, the sentence, Block a is on top of Block b can be represented as:

*on(block_a, block_b).*

Here the relation is *on* and the two objects are: *block_a* and *block_b*. This also suggests that it is true that *block_a* is on top of *block_b*, i.e. it has the value of *true*. Predicate Calculus logic has either the value of true or false. Where the reasoning is certain, i.e. statements are either true or false, the systems involved are called deductive systems. In contrast to deductive systems, reasoning in inductive systems is less than certain, where given that the values of initial statements are true, the conclusions reached are only more or less plausible. In deductive reasoning, given that one is sure of the initial data, one can be equally sure of any results reached by reasoning (i.e. conclusions). The reasoning that is carried out is called a proof or deduction. AI planners are thus deductive systems rather than inductive[2] systems which involve the assignment of probabilities to reasoning steps.

It is also possible to combine individual statements to form compound propositions by the use of logical connectives. These include *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implies* (-->) and *equivalent* ($\Leftrightarrow$). To further extend the system to represent the notion of *some* and *all*, two quantifiers, universal ($\forall$) and existential ($\exists$) are used. $\forall$ is called the universal quantifier because it talks about everything in the universe and $\exists$ is the existential quantifier because it talks about the existence of some objects.

Due to the complexity of Predicate Calculus, it can be very difficult to represent in a computer. The programming language, Prolog, used in modelling the Automatic Assembly Planner (AAP) described in Chapter 5 is based on a simpler version of logic called Clausal Form Logic (CFL). Prolog consists internally of a program that carries out deductions using symbols such as objects and relations to manipulate its set of data (database). Fundamentals of CFL is covered in Appendix i as it is the basis of the language used to implement AAP.

Thus by using a form of logic, the meaning of the state data structures could be represented in a computer. Similarly, the state space representation can also be represented in a computer as a directed graph

[2] Inductive systems are important where much uncertainty exists as in medical diagnosis systems and systems to aid managerial decisions.

where the nodes in a graph describe the states and the arcs describe the application of operators which will transform one state into another.

### 3.2.3.3 Action Ordering Plans

Another form of common plan representation technique is the action-ordering plan representation in which the planner attempts to produce a sequential list of actions which are limited by constraints. The actions are written down in a list and are executed in the order in which they have occurred on the action list. NOAH described in section 3.6 is an example of an action-ordering planner.

## 3.2.4 Plan Generation

The most common approach to the formulation of plans is to search the space in which the problem is represented. The search procedure is probably all the reasoning that is required of the planning system.

### 3.2.4.1 Search Mechanisms

There are two main strategies for searching the space, depth-first and breadth first. In the former, the deepest node in the search space is always selected for expansion. In the latter strategy, the search proceeds by examining all possibilities of a node by going across the search space before proceeding *down* into the next level.

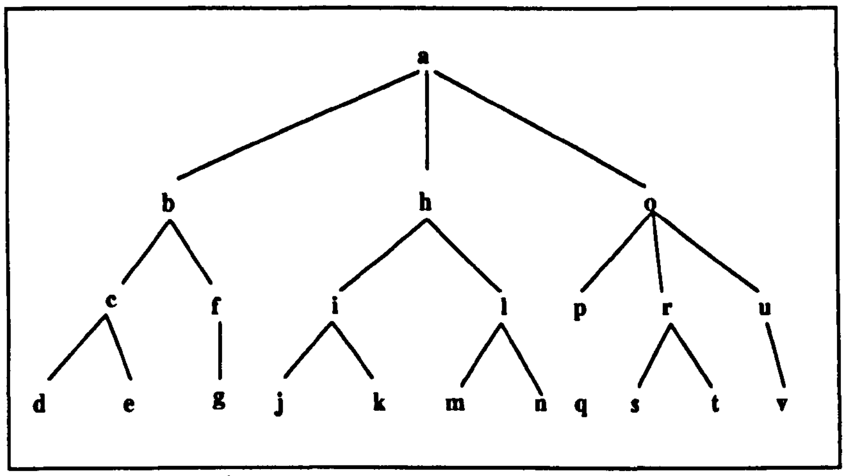

Figure 3.2 An Example of a Tree

From Figure 3.2, the order in which the nodes are visited in a depth-first manner is {a,b,c,d,e....,v}. The ordering of nodes visited in a breadth-first manner is {a,b,h,o,c,f,i,l,p,r,u,d,e,g,j,k,m,n,q,s,t,v}. In principle, both methods are exhaustive in searching for a plan. Sometimes these two extreme search strategies may be infeasible because there may be practical limits on the amount of time and storage available to expand the search space. Therefore variations of these techniques have been suggested. Some of these common variations are depth-first search with backtracking, Means-End-Analysis, Least Commitment Strategy, Dependency-Directed, Opportunistic Search, Meta-level search.

The depth-first search with backtracking is a simple method of considering alternative solutions. The state of the solution at each point where there are alternatives is saved together with the alternative choices. If a failure occurs, the saved state at the last decision point is restored and the next alternative taken (if there are none, backtracking continues over the previous decisions). The programming language Prolog uses this strategy.

In the least commitment strategy, the central idea is to leave partial solutions incompletely specified until the last possible moment. When as much information as possible becomes available, then the ordering is completed such that no conflicts arise (this is also used in the NOAH planner).

Assumptions, alternatives or dependencies can also be added to the set of data and then treated like any other piece of data. Whenever a conflict exists, only the dependent parts are undone leaving unrelated parts intact. This approach is called dependency-directed backtracking.

The focus of the opportunistic search is identified on the basis of the most constrained operation that can be performed, i.e. take the path with the most constraints. The requirements for the solution can be summarized as constraints on the possible solutions. The planner then suspends its operation until further information becomes available where a more definite choice can be made. These planners usually operate with a *blackboard* through which various components can communicate via constraint information.

In meta-level planning, a separate search is made to decide which of the operators is best applied at any point before detailed decisions are made, i.e additional reasoning about the various techniques available for generating the plan.

The search mechanisms described above have evolved over the past two decades and are characteristic of planners developed over that period. The search can be difficult due to the large number of interactions between different states or partial plans. These interactions can lead to a surprising amount of complexity as shown by Gupta and Nau [29] that the problem of finding an optimal plan in the simple Blocks World domain is NP-hard[3]. Also the establishment of the existence of a pre-condition in a partially ordered plan can require exponential computation [30]. Thus planning problems are considered hard and are still a topic of AI research.

If the planner searches through an action-ordering plan then it can add and remove operators at various points in the plan as illustrated by NOAH in Section 3.6. In the state space representation, modification of the plan is only possible at the end of the plan. Here operators are added to the plan by trying out another operator application and operators are removed when backtracking occurs. Thus the kind of reasoning or search involved depends on the representation technique used. In the following sections,

[3] In Computer Science, the economy of an algorithm is measured in terms of the complexity measures of time and memory space. E.g. the bubblesort algorithnm is said to be a quadratic time algorithm. Unreasonable algorithms are said to require super-polynomial or exponential time. There are a class of problems called NP-complete where their lower bounds are conceivable but have exponential upper bounds. An example is the Towers of Hanoi problem. The number of single ring moves produced by the algorithm is $2^N - 1$. So the number of moves for a three ring case is 7 while for a 64 ring problem, it would take more than half a million years if the speed is a million rings per second.

some significant AI planners and their techniques which are relevant to AAP are described.

## 3.3 Green's Formulation

Green [31] first formulated the planning problem using Predicate Calculus and applied a resolution[4] theorem prover to generate plans. His formalism involved a set of assertions[5] that describe the initial state and another set that described the effects of various actions on the states. A situation or state variable was included in each predicate in order to keep track of which facts were true in which state.

The system then attempted to prove that there exists a state in which the required goals are true. The essence of his method is rather like a *fill in the blank* resolution where a correct plan is derived as a side effect of proving the existence of a correct plan. In addition, it is also necessary to state that certain relations are unaffected by the actions (assertions for each relation that is not affected by an action must be included, otherwise the resolution proof fails). These assertions describe what stays the same during an action and are called frame axioms. This technique is often referred to as the Situation Calculus. The following example illustrates how it works.

[4] The resolution principle is described in Appendix i .

[5] An assertion is a statement that is held to be true, i.e. when someone believes it, when someone claims it to be true or when it is a fact in a knowledge base.

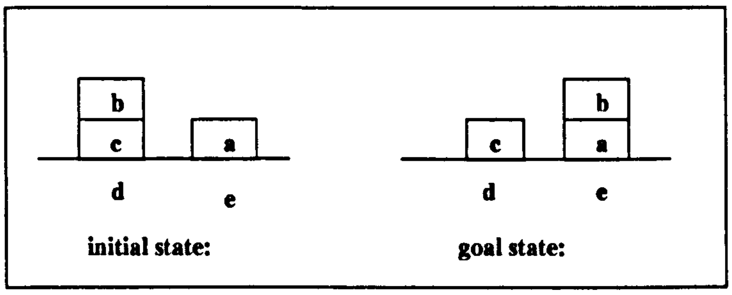

Figure 3.3

From the above figure, the formulas for the initial state are:

| |
|---|
| *on(b,c,s0)* |
| *on(c,d,s0)* |
| *on(a,e,s0)* |
| *clear(b,s0)* |
| *clear(a,s0)* |

The goal state is:

| |
|---|
| *on(c,d,s1)* |
| *on(a,e,s1)* |
| *on(b,a,s1)* |
| *clear(c,s1)* |
| *clear(b,s1)* |

The fact that block *a* is in position *e* can be written as:

*on(a,e,s0)*

where *s0* is the state in which *a* is on *e* is true. In this case *s0* is the initial state and *s1* is the goal state. The action of moving a block from one place to another can be expressed as: *move(x,y,z)* where *x* is moved from position *y* to position *z*. After executing an action in one state, the result is a new state and can be expressed as: *do(action,state)*, i.e. map a state into the one resulting from an action. It is also necessary to indicate that two blocks are different in order to prevent the situations like e.g. *on(a,a)* from occuring. Hence *diff(b,a)* can be used to suggest that *b* is different from *a*. Thus the rule in this problem is:

*clear(x,s) & clear(z,s) & on(x,y,s) & diff(x,z)*
=> *on(x,z,do(move(x,y,z),s)) &*
*clear(x,do(move(x,y,z),s)) &*
*clear(y,do(move(x,y,z),s)).* (Rule 1)

Rule 1 can be read as:

if *x* and *z* are clear and if *x* is on *y* in state *s* and if *x* and *z* are different, then *x* and *y* will be clear and *x* will be on *z* in the state resulting from performing the action of *move(x,y,z)* in the state *s*. The *diff* predicate does not need a state variable because its truth value is independent of state. The refutation graph for this example is given in Figure 3.4.

The main drawback of this approach is the need to write down formulas for each relation that is not affected by an action. These formulas are called frame axioms. A frame axiom for the above example is:

*on(p,q,s) & diff(p,x) => on(p,q,do(move(x,y,z),s)).*

This is to suggest that blocks that are not moved stay in the same position. In the above example, the frame axiom was not used in the resolution procedure but if another sub-goal is added say, *on(c,b)*, it would be necessary to prove that *b* stayed on *a* while putting *c* on *b*. Typically the number of frame axioms required is proportional to the product of the number of relations and the number of operations in the problem. Thus as the problem gets more complicated the number of frame axioms required can be phenomenal and for the sake of manageability one would like to limit the number of frame axioms. The system described in the next section is an attempt to reduce the frame problem.

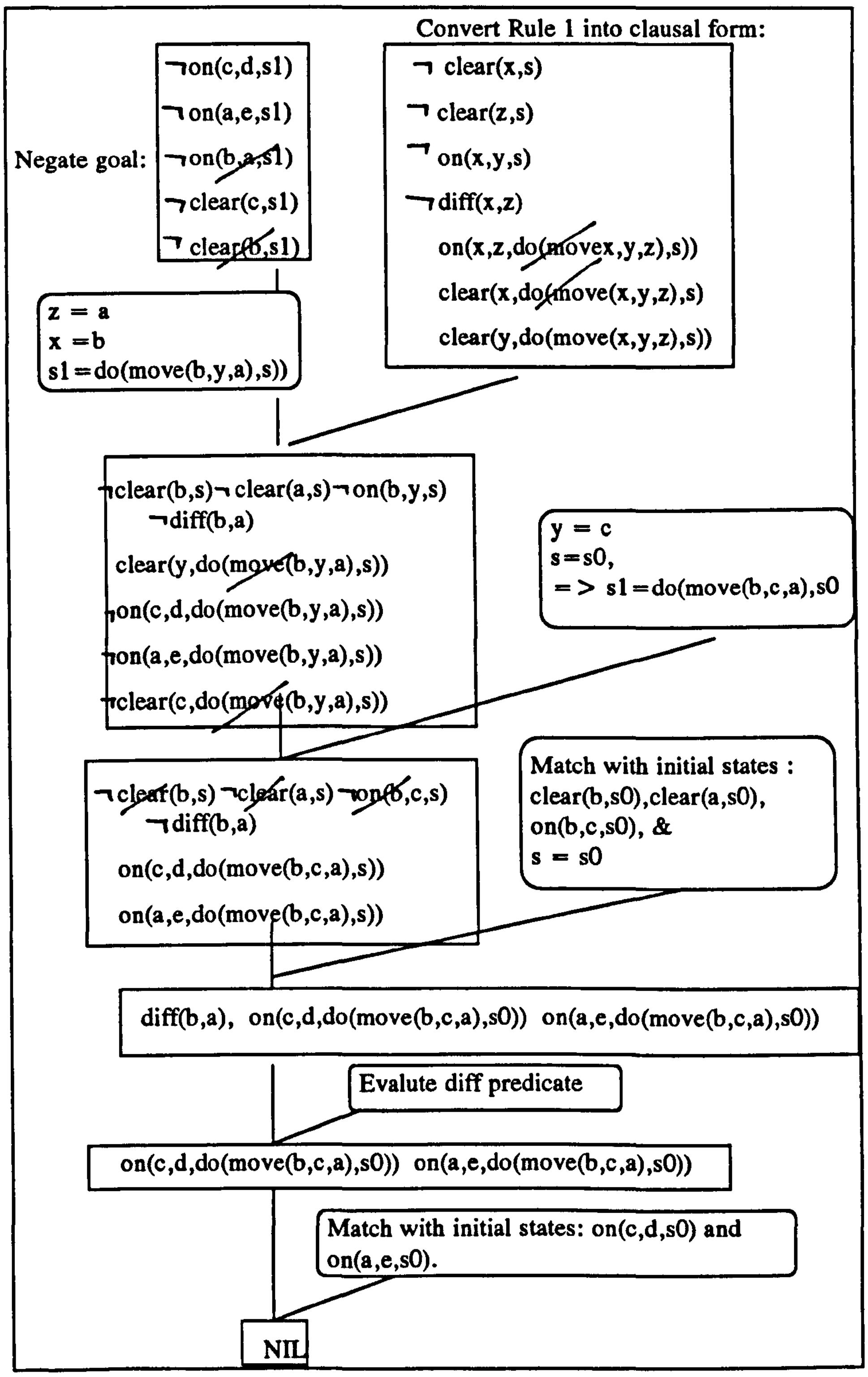
Convert Rule 1 into clausal form:
¬ clear(x,s)
¬ clear(z,s)
¬ on(x,y,s)
¬ diff(x,z)
on(x,z,do(move(x,y,z),s))
clear(x,do(move(x,y,z),s)
clear(y,do(move(x,y,z),s))
Negate goal:
¬on(c,d,s1)
¬on(a,e,s1)
¬on(b,a,s1)
¬clear(c,s1)
¬ clear(b,s1)
z = a
x = b
s1 = do(move(b,y,a),s))
¬clear(b,s)¬ clear(a,s)¬on(b,y,s) ¬diff(b,a)
clear(y,do(move(b,y,a),s))
¬on(c,d,do(move(b,y,a),s))
¬on(a,e,do(move(b,y,a),s))
¬clear(c,do(move(b,y,a),s))
y = c
s = s0,
=> s1 = do(move(b,c,a),s0
¬ clear(b,s) ¬clear(a,s) ¬on(b,c,s) ¬diff(b,a)
on(c,d,do(move(b,c,a),s))
on(a,e,do(move(b,c,a),s))
Match with initial states :
clear(b,s0),clear(a,s0),
on(b,c,s0), &
s = s0
diff(b,a), on(c,d,do(move(b,c,a),s0)) on(a,e,do(move(b,c,a),s0))
Evalute diff predicate
on(c,d,do(move(b,c,a),s0)) on(a,e,do(move(b,c,a),s0))
Match with initial states: on(c,d,s0) and on(a,e,s0).
NIL

Figure 3.4 A Refutation Graph For Figure 3.3

## 3.4 Kowalski's System

Kowalski (1979) offered a different formulation using Predicate Calculus which simplifies the statement of frame axioms [32]. He used a predicate, *holds* to indicate that a given condition holds in a given state. For example, *holds(on(a,c),s))* is the same as Green's *on(a,c,s))*. This means that the number of frame axioms needed is equal to the number of operators. So the initial state for Example 3.3 is:

*poss(sO)*

*holds(on(b,c),sO)*

*holds(on(c,d),sO)*

*holds(on(a,e),sO)*

*holds(clear(b),sO)*

*holds(clear(a),s))*

*poss(sO)* means that state *sO* is possible, i.e. one that can be reached. Effects of the actions (post-conditions) are also specified as follow:

*holds(clear(x),do(trans(x,y,z),s))*
*holds(clear(y),do(trans(x,y,z),s)).*
*holds(on(x,z),do(trans(x,y,z),s))*

Another predicate, *pact* is used to represent the fact that it is possible to perform a given action in a given state, i.e. the pre-conditions of the action match that state description. So, *pact(a,s)* means that it is possible to perform a given action, *a* in a state, *s*. A rule for the action, *move* is:

$$\{holds(clear(x),s) \;\&\; holds(clear(z),s) \;\&\; holds(on(x,y),s) \;\&\; diff(x,z)\} \Rightarrow pact(move(x,y,z),s).$$

It is also necessary to define the pre-conditions, i.e. if a given state is possible and if the pre-conditions of an action are satisfied in that state then the state produced by performing that action is also possible.

So,

$$poss(s) \;\&\; pact(u,s) \Rightarrow poss(do(u,s))$$

Here, only one frame axiom is needed for each action. Hence for the example in the previous section, the frame axiom is :

$$(holds(u,s) \;\&\; diff(u,clear(z)) \;\&\; diff(u,on(x,y))) \Rightarrow holds(u,do(trans(x,y,z),s)).$$

The above formula simply says that if all the terms are different then *clear(z)* and *on(x,y)* will still hold in all the states produced by performing the action, *trans(x,y,z)*. Even though this formalism has reduced the number of frame axioms, there is still a need to define frame axioms.

## 3.5 STRIPS

One of the most significant AI planner to date, STRIPS [33] removed the need for frame axioms. It is adapted from GPS to apply to planning problems. It uses a simple stack based implementation. The system consists of a single stack that contains goals and the operators that have been proposed to satisfy those goals and a database that describes the current situation (of a problem) and a set of pre-conditions, an add-list (containing states that are true after performing the associated action) and a delete-list (containing states that are no more true after performing the action) for each operator.

In this representation, the frame axioms are implicit (in the add and delete lists), so there is no need to define them explicitly as in the previous two planners. Consider the same example in Section 3.3,

The goal is:

*on(b,a),table(a),table(c),clear(c),clear(b).*

Take the first sub-goal, i.e. *on(b,a)* and place it on the stack. So the content of the stack is: *on(b,a).* initially. Other conditions are defined as follows:

| |
|---|
| Pre-conditions of stack: *clear(a), clear(b)* |

| | |
|---|---|
| Add-list: | *on(b,a),clear(c)* |
| Delete list: | *clear(a), on(b,c).* |

The pre-conditions are true in the initial state, so the action on the stack is possible. The database is then updated according to the add and delete lists, i.e.:

*on(b,a),table(a),table(c), clear(c), clear(b).*

This matches the goal state. Hence the plan is to stack *b* onto *a*.

However this approach still could not solve Sussman's Anomaly. Irrespective of which sub-goal is placed onto the stack first, the goal cannot be reached. This is due to the undoing of goals which hampers the progress in generating a plan. At this point, it becomes obvious that a mechanism for backtracking is needed for conjunctive planning. Therefore as it is, STRIPS can lead to redundant actions in a plan or worse re-introducing and trying to satisfy the same goal over and over again such as in the case of Sussman's example. Extension of STRIPS such as ABSTRIPS [34] were then developed to tackle this problem.

ABSTRIPS is an attempt to make STRIPS into a hierarchical planner. Criticality values are assigned to pre-conditions and those that are the easiest to achieve are given the lowest value while those that are 'most difficult' to achieve (such as a major goal predicate) are given the highest value. Pre-conditions with the highest criticality values are considered first, i.e. put onto the stack first during the first level of the planning algorithm. Predicates with the second highest criticality values are considered next and so on until the goal is achieved. Consider the example given in Figure 3.5.

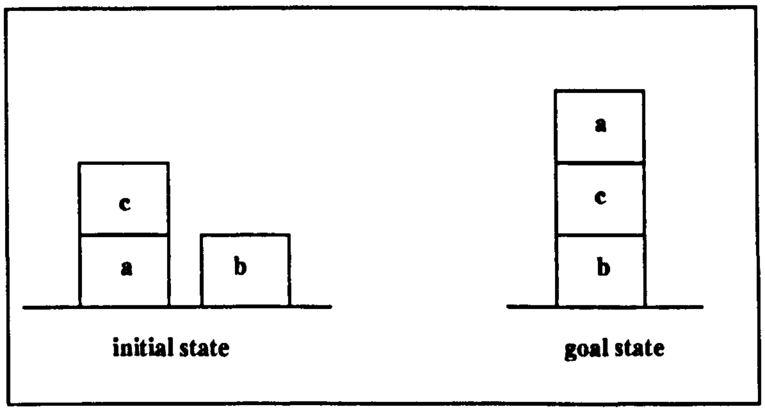

Figure 3.5 An Example to Illustrate ABSTRIPS

From Figure 3.5, the goal is: *on(c,b), on(a,c).* Since goal predicates have the highest criticality values then *on(c,b)* is considered first followed by *on(a,c).* (If *on(a,c)* were to be considered first, the planner would have to backtrack, so for the sake of simplicity the correct sub-goal has been chosen). This is the first level solution. At the second level, the pre-conditions of *on(c,b)* i.e. *clear(b)* and *clear(c)* are considered. These are also true in the state description so the goal, *on(c,b)* can be achieved. The pre-conditions of *on(a,c)* are also considered and they are also true. Thus the ordering is {*on(c,b), on(a,c)*}.

STRIPS is significant because it provided a simple formalism for representing the planning problem without the need to explicitly define any frame axioms. Also the simple stack based implementation could be easily modelled on a computer. The planner described in the next section viewed the planning problem rather differently from STRIPS.

## 3.6 NOAH

This is said to be the first action ordering plan representation system (systems described previously were state space planners). Sarcedoti (1977) introduced the idea of the procedural net in his planner called NOAH (Nets Of Action Hierarchies)[35]. A procedural net is simply a connected network of nodes. These nodes represent actions (any operation that changes the state of the world) at varying levels of detail, organised into a hierarchy of partially ordered time sequences. It assumes that goals can be achieved in parallel until information to the contrary becomes available.

There are four different types of nodes: *goal, phantom goal, split and join*. In the net, nodes are annotated with '+' and '-', where '-' means that the associated action deletes some pre-conditions of the node labelled '+'. NOAH first builds a procedural net and each node is expanded in turn to produce a new and more detailed plan. The plan is then criticised by a set of critics to perform any necessary reordering of the nodes in the net. The following example is used to illustrate how NOAH works.

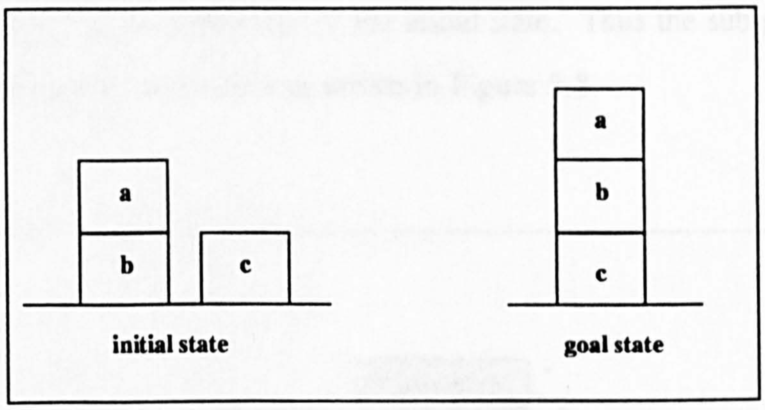

Figure 3.6 An Example to Illustrate NOAH

Description of initial state:

on(a,b), clear(a), clear(c)

Description of goal state:

on(a,b), on(b,c), clear(a).

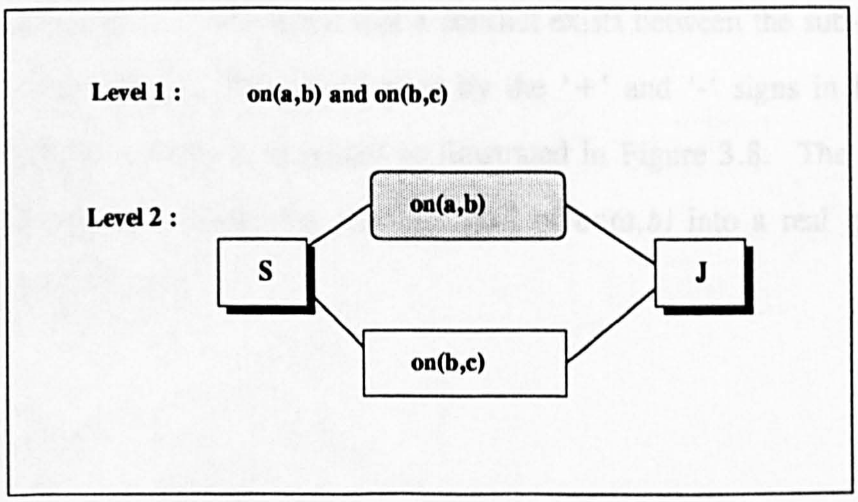

Figure 3.7 Levels 1 & 2 of Example

At Level 1, the partial plan is the goal, *on(a,b) & on(b,c)*. This level is then expanded and the goal is split into two sub-goals as seen in

Level 2 on Figure 3.7 above. The sub-goal, *on(a,b)* is shown as a phantom goal because it is already true in the initial state. Thus the sub-goal, *on(b,c)* is expanded next at Level 3 as shown in Figure 3.8.

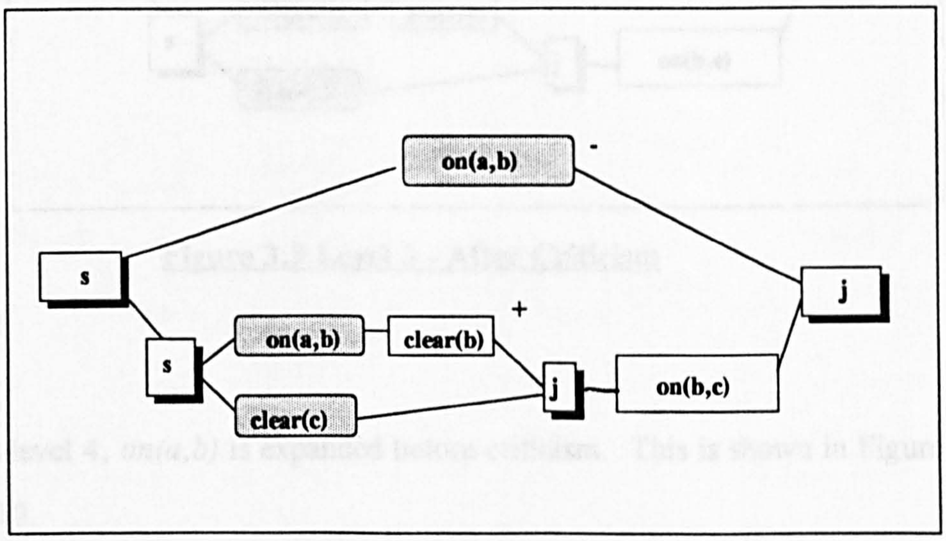

**Figure 3.8 Level 3 Before Criticism**

At Level 3, it was found that a conflict exists between the sub-goals, *on(a,b)* and *on(b,c)*. This is indicated by the '+' and '-' signs in Figure 3.8. Sub-goal on(b,c) is expanded as illustrated in Figure 3.8. The critics in the system will change the phantom goal of *on(a,b)* into a real goal as shown in Figure 3.9.

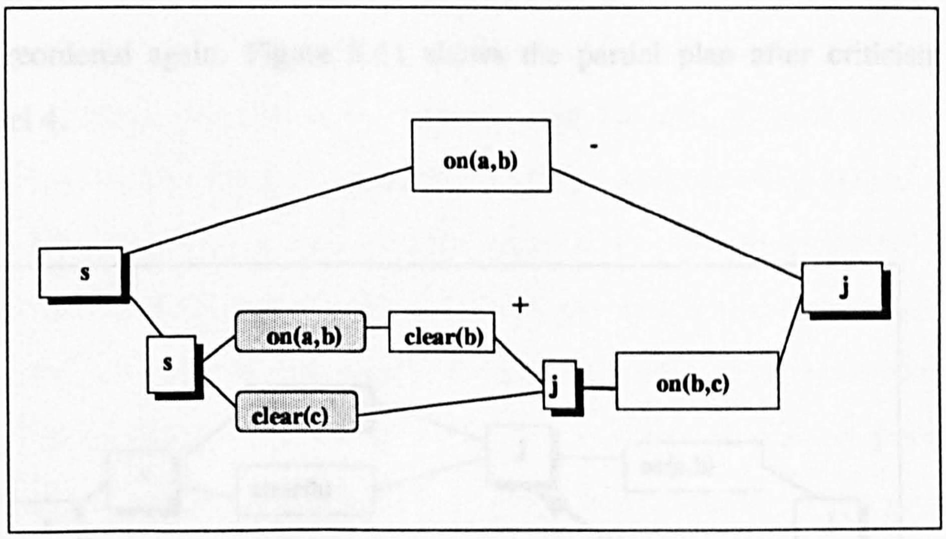

**Figure 3.9 Level 3 - After Criticism**

At level 4, *on(a,b)* is expanded before criticism. This is shown in Figure 3.10.

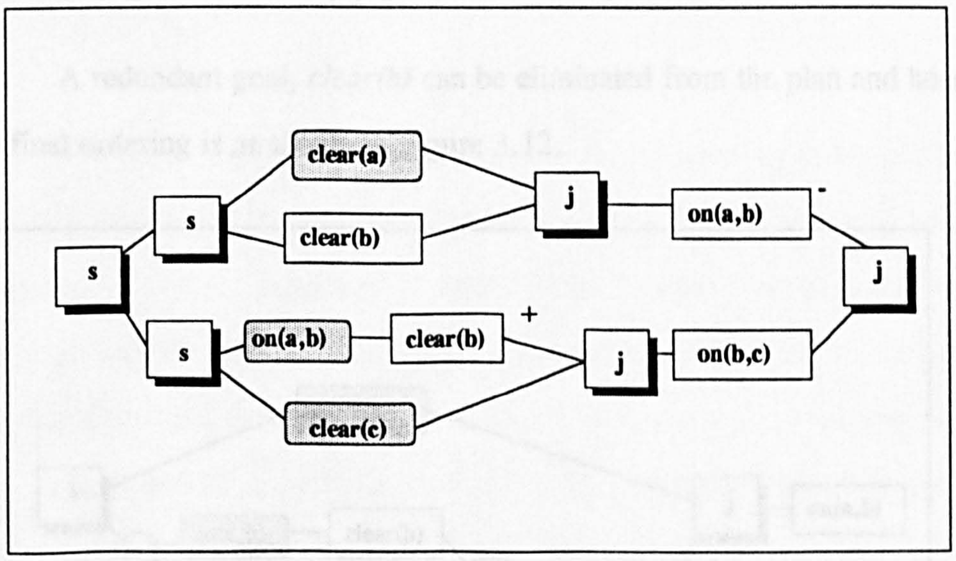

**Figure 3.10 Level 4 - Before Criticism**

A critic notices that *clear(b)* is asserted by one node (indicated by '+') and then deleted by another (indicated by '-'). Therefore, the partial plan has to be reordered again. Figure 3.11 shows the partial plan after criticism at Level 4.

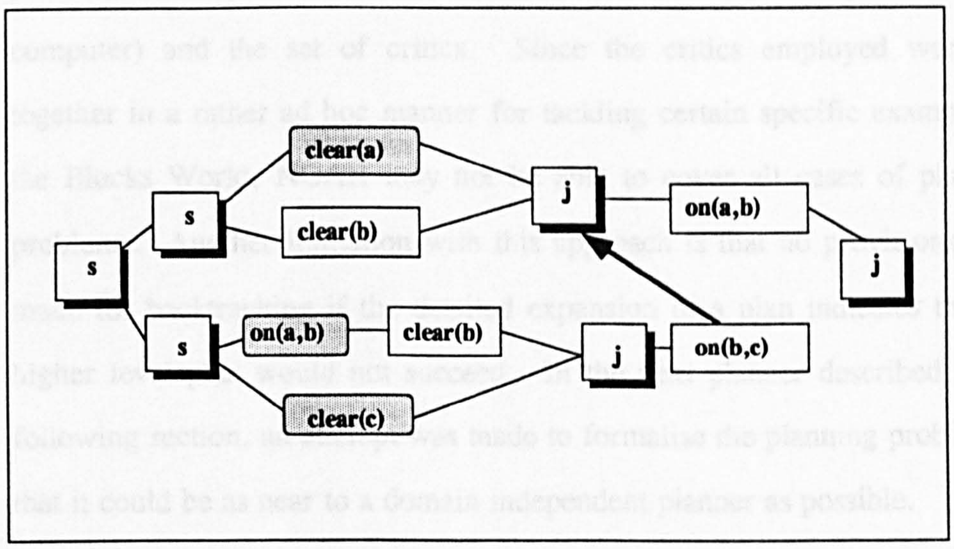

**Figure 3.11 Level 4 - After Criticism By Resolving Conflicts**

A redundant goal, *clear(b)* can be eliminated from the plan and hence the final ordering is as shown in Figure 3.12.

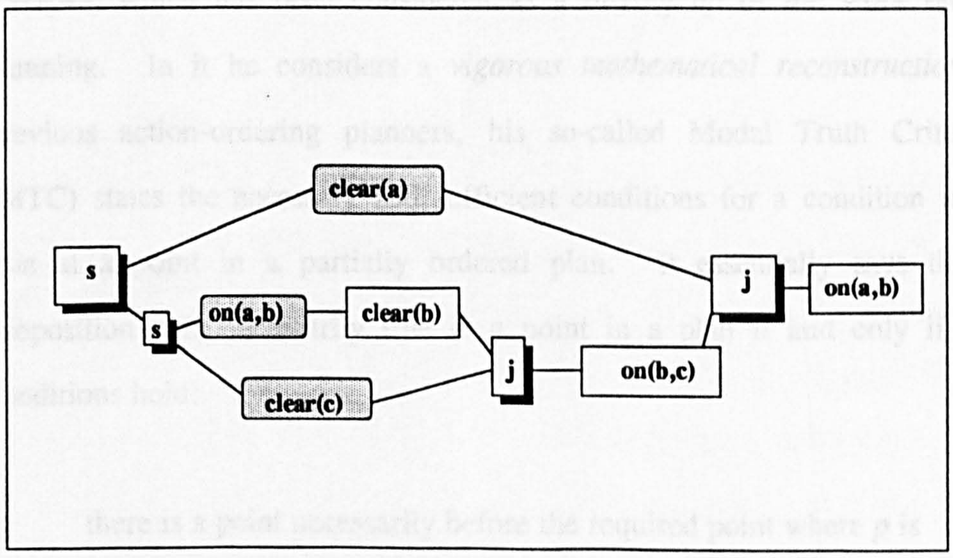

**Figure 3.12 Final Plan**

In summary, the planning algorithm of NOAH consists of inputting the procedural net and then expanding the (partial) plan to obtain an ordering. Critics are used during the expansion when conflicts occur. The power of NOAH lies in its formalism (which allows it to be represented in a computer) and the set of critics. Since the critics employed were put together in a rather ad hoc manner for tackling certain specific examples of the Blocks World, NOAH may not be able to cover all cases of planning problems. Another limitation with this approach is that no provisions were made for backtracking if the detailed expansion of a plan indicates that the higher level plan would not succeed. In the next planner described in the following section, an attempt was made to formalise the planning problem so that it could be as near to a domain independent planner as possible.

## 3.7 TWEAK

Chapman [36] provided a formalism for AI planning in his planner, TWEAK which has been considered as a *tidying up* of the work on AI planning. In it he considers a *vigorous mathematical reconstruction* of previous action-ordering planners, his so-called Modal Truth Criterion (MTC) states the necessary and sufficient conditions for a condition to be true at a point in a partially ordered plan. It essentially says that a proposition *p* is necessarily true in a point in a plan if and only if two conditions hold:

i) there is a point necessarily before the required point where *p* is necessarily asserted and;

ii) for every operator that could possibly come between the point of assertion and point of requirement, if the intervening operator possibly deletes a condition which might turn out to be *p*, then there must be another operator which restores the truth of *p* whenever the intervening operator deletes it.

In simplified terms, the MTC is a statement of the conditions under which an operator's pre-conditions will be true. Figure 3.13 is used to illustrate the meaning of the MTC.

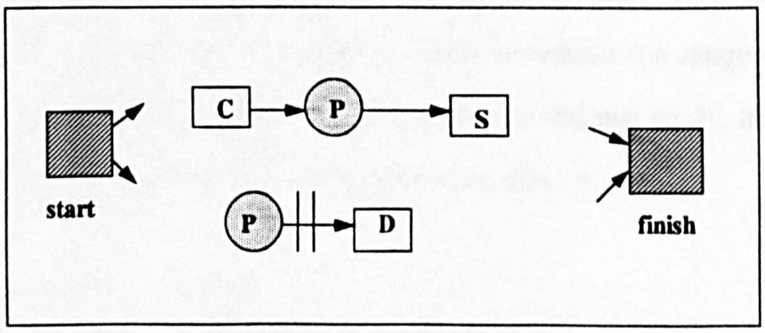

**Figure 3.13 The TWEAK Planner**

From Figure 3.13, operator *C* is said to contribute to condition *P* and *C* has *P* on its add-list. Operator *S* requires condition *P* to be true, i.e. *S* has *P* as one of its pre-conditions. Operator *D* has *P* as a pre-condition and deletes it. The deletion is indicated by the double line in the above figure. In the figure, objects to the left are ordered before objects to the right, so the *start* operator comes before everything else and *finish* is after everything else. Similarly, operator *C* comes before *S*, and *D* might possibly come after *C* and is before *S*. There may also be some operator after *start* and before *C* and *D* and after *S* and *D* but before *finish*. Thus in considering the truth of *S*, two points must be established:

i) *P* which is the pre-condition of *S* has to be achieved first, i.e. a contributor is needed for the condition that is necessary before *S* in the plan. If none exists, then one is created by adding an appropriate operator and install it into the net. If there is one, but not yet ordered before *S*, then the required operator is added into the net.

ii) any operators that possibly delete condition *P* have to be *prevented* from doing so; *P* is deleted if there is an operator, *D* in the plan that comes after *C* and before *S*. If no such deleters exist then the truth at *P* is guaranteed. Otherwise, order it outside the range over which *P* is expected to hold. Suppose *D* is a deleter of *P*, then it may be harmless if it is ordered before *C* or after *S*.

Basically, the mechanism of TWEAK states that in order to maintain the truth of a condition *S*, check and make sure that its pre-conditions (or contributors) are also true. In summary, TWEAK consists of the following actions which will make *P* necessarily true in *S* where *P* is a pre-condition and *S* is a goal:

i) establish a plan in which *P* occurs before *S*;

ii) adding a step before *S* in which *P* is asserted if necessary;

iii) re-order a step after *S*, if it is known to prevent *P* and hence *S*;

iv) if there is a step that will delete the pre-condition of *P*, add another step, *W* which will re-establish the truth of *P* and *S*.

These steps will be further elaborated in Chapter 5. Even though the heuristics in TWEAK are not new (e.g. similar to NOAH), it has provided a *global* representation to the planning problem. However this formalism cannot represent disjunction in a partially ordered plan, e.g. problems in the Blocks World where blocks can support more than one block.

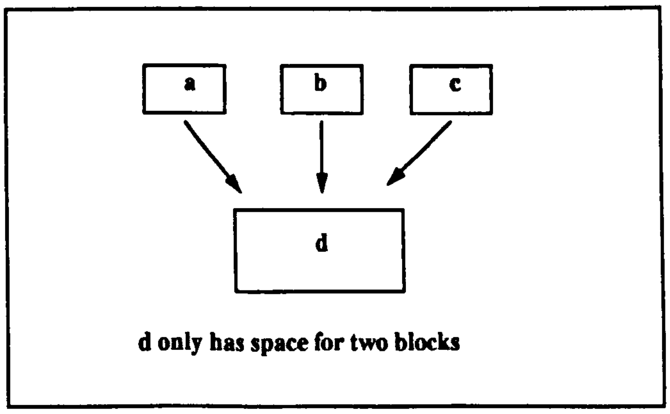

Figure 3.14 Stacking Three Blocks Onto One Block

The MTC does not detect the fact that all the three stacking actions cannot occur and will produce a plan as shown in Figure 3.15.

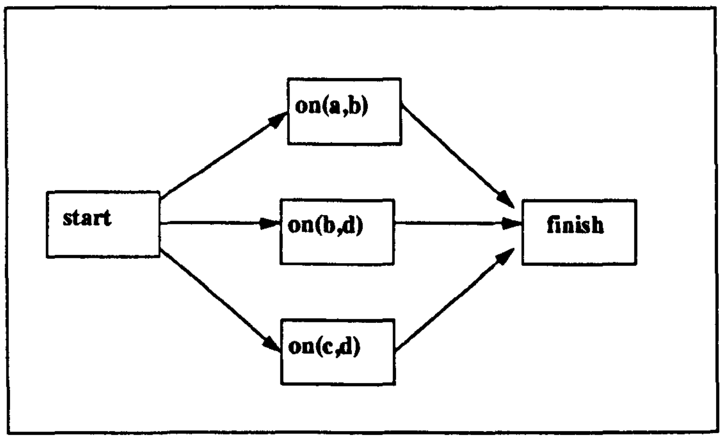

Figure 3.15 An Incorrect Ordering

Apart from this, the MTC cannot represent conditional actions, i.e. actions that perform a test and then do something that depends on the test's outcome (e.g. switching a light on if it is off and vice versa). Apart from these two limitations of TWEAK, the formalism provided a global view and a foundation to the planning problem. Hence in spite of its weaknesses, its rigorous representation is used with some modifications in the implementation of AAP.

Other planners which also deserve a mention are summarised in the Table 1.

| | |
|---|---|
| Hacker [37] | Search through a space of partial plans and plan representation is action-ordering. Has a gallery of critics to catch and correct known plans. Assumes that sub-goals are independent and thus can be achieved in an arbitrary order (so cannot solve Sussman's example). |
| Interplan [38] | Designed to detect sub-goal interactions and to correct them by analysis. Records links between actions and their effects on other sub-goals in a plan in a table called the ticklist. This ticklist could suggest the minimum set of reorderings required if an interaction was detected during the plan generation process. |
| Nonlin [39] | Initially designed to correct certain problems found in NOAH. Can search the space of alternative plans for a solution. |
| MOLGEN [40] | A hierarchical planner using the least commitment strategy. Used for planning gene-cloning experiments in molecular genetics. |

Table 1. Some AI Planners

## 3.8 Conclusion

The difficulties of the planning problem and the enormity of the problem of understanding human intelligence is evident in the fact that, after more than two decades of AI planning research a completely working domain-independent planner for real world planning problems is still as elusive as ever. Research in this area has concentrated on producing a planner that will be better than its predecessor. This has narrowed the focus of the research a great deal, the pattern of which is usually based on a previous planner which may or may not be coherent. On the other hand, this competitive spirit has led to a better understanding of the planning problem. It is now generally accepted that planning is difficult and is not easy to solve piecemeal or otherwise.

However it is believed that now the time is ripe to reflect on the work of AI planning and attempt to draw some benefits and lessons from it. This belief is reflected by the invasion into the engineering community of AI *products* in the form of expert systems. This could be due to two main reasons. Firstly, engineers are seeking alternative methods to solve their problems when conventional means fail. Adoption of the flexible manufacturing philosophy also requires more *flexible* approaches (such as AI techniques). Secondly, some in the AI community are impatient to test their findings in the real world. This is because if AI fails to deliver solutions (or even partial solutions) in the near future, interests in this area will surely dwindle as in the case of Machine Translation[6].

[6] Research in this area first started in 1949 in the UK, followed by massive investments in the US. However when it failed to produce any promisng results, in 1966 ALPAC (Automatic Language Processing Advisory Committee) in the US produce d a negative report for machine translation. Consequently, findings and researh in this area was substantially reduced. It was only recently that the subject of Machine Translation has attracted renewed interest .

# CHAPTER 4. AI TECHNIQUES FOR ASSEMBLY PLANNING

## 4.1 Introduction

A review of AI techniques which are available and a discussion on those that are relevant to assembly process planning is described in this chapter. A general discussion on assembly and the methods that are used to solve the assembly planning problems are also given. This chapter includes an examination of the methods in which knowledge can be represented and manipulated.

## 4.2 Assembly

Even though the assembly process is usually carried out manually and typically accounts for 40 - 60% of the total production time, development of computer aids is considerably quicker in the area of component manufacture rather than in assembly [41]. This is because apart from being mainly a manual task, there are no set procedures or rules that govern how a product should be assembled. Thus making it rather difficult to automate. Hence in this project, the assembly planner was developed with manual assembly in mind[1].

The method of assembly is normally determined at the design stage but it is not uncommon for the method to be changed further downstream,

[1] The process planning department visited during the course of this research dealt with manual assembly.

e.g. by assembly workers on the shopfloor. However the main function of assembly is to *join* all the individual parts or components together so that they perform the function that is specified by the designers. As any form of work that is performed on the parts that are going through the production chain costs money, ideally products should be designed in such a way as to avoid assembly. However this is only possible in very simple products because most assemblies must have a certain degree of mobility to achieve their desired functions. When different functions are necessary for individual parts, different material characteristics may be required. Hence assembly of these individual parts is inevitable. Also some parts are easier to produce by division into sub-parts. In addition there may be some particular functional conditions such as increased requirements of accessibility, demounting, cleansing and inspection that dictates the necessity for assembly.

The assembly process is usually divided into sub-assemblies. A sub-assembly is where one (or more) component(s) is (are) assembled with another component or base component (a component onto which others are assembled). The main reason for dividing the assembly process into sub-assemblies is because it is easier to construct the finished product from the individual sub-assemblies. For example, in the car industry, the assembly of the car is normally divided into various functional parts like the gearbox, engine, suspension, steering, etc. Each of these parts are also broken down into smaller sub-assemblies if possible, e.g. the sub-assembly of the piston-connecting rod is part of the assembly of the gearbox. This is also similar to the work done in AI planning where a goal is also broken down into sub-goals before actual planning occurs. Figure 4.1 illustrates the hierarchy of assembly process planning.

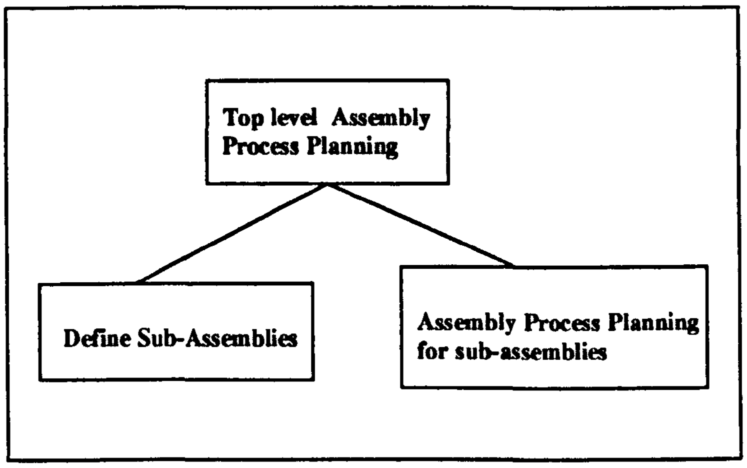
Top level Assembly Process Planning
Define Sub-Assemblies
Assembly Process Planning for sub-assemblies

**Figure 4.1 Hierarchy of Assembly Process Planning**

The assembly process planning stage usually consists of the following:

i) determination of the assembly parts and sub-assemblies;

ii) determination of the sequence of assembly operations;

iii) allocation of the work to respective assembly stations;

iv) calculation/estimation of the times and costs involved [42].

The estimation of times and costs would require substantial company related data. As for the determination of the major assembly parts and sub-assemblies, it is pre-determined by the function of design. Thus leaving the minor sub-assemblies for process planning by the process planner. This project had concentrated on the sequencing of assembly operations and illustrated how AI planning techniques can be used to solve this part of the assembly process planning problem with a planner.

## 4.2 1 Assembly Sequencing and AI Planning

In manual assembly, the operations usually involve a pair of components at any one time, i.e only two and no more parts are joined at a given time. This pair of components can be regarded as a mating pair, i.e. mating features on the respective components. Prior to the availability of AI methodologies, attempts at automating assembly sequence planning have been made using an algorithmic approach. If there are *n* mating pairs then the number of possible assembly sequences is *n!*. So, for say 5 mating pairs, the number of possible assembly sequences is 120. Thus, the number of valid assembly sequences can be very large even for a small number of components. Clearly it is not desirable to generate all the possible sequences and is also not the way in which a human works.

An example of such an algorithmic approach as suggested by Fazio and Whitney [43] is to obtain a set of rules for generating assembly sequences from a series of questions about the mating of part pairs and multiple parts which are directed to engineers. Due to the number of questions that are required in order to obtain a *complete* picture of the assembly, it can be very tedious to answer all the questions. Whenever minor changes occur, they can also drastically modify the available choices of assembly sequences. Hence requiring the user to repeat the question and answer session.

A semi-automatic approach along the lines of the above method is adopted by Rover for scheduling or allocating work on the assembly line taking into account changeable variables like manning levels and volume of cars to be produced[2]. Previously industrial engineers work out feasible work schedules for different work stations manually[3]. It was done by drawing a precedence network as shown in Figure 4.2. Nodes in the network represents the work or job to be done at one particular station on the assembly line. The numbers on the network refer to the work or job number (which may be a group of related operations) and smaller numbers have a higher priority than the jobs with bigger numbers. This process of producing the graph manually is found to be very tedious and problem is enhanced when frequent changes have to be made. The semi-automated system enables the industrial engineer to set up on a computer screen, the precedence network, the various nodes (i.e. operations) and where they are in relation to others. Thus instead of drawing it on paper the engineer is able to *draw* on the screen using a mouse.

[2] To adapt to customer demands, production fluctuates in the total number of a particular model and the different specifications offered within a model range. Hence the variations and changes is tremendous.

[3] The order in which a car is built consists of two levels of precedence, i.e. operations between members of the 'same' groups of operations (e.g. assembly of gearbox) and operations between groups (e.g. electrical system to engine). The process planning department generates the build method (i..e detail assembly sequences within groups) and then at the line balancing area, the schedule of work assignments is produced.

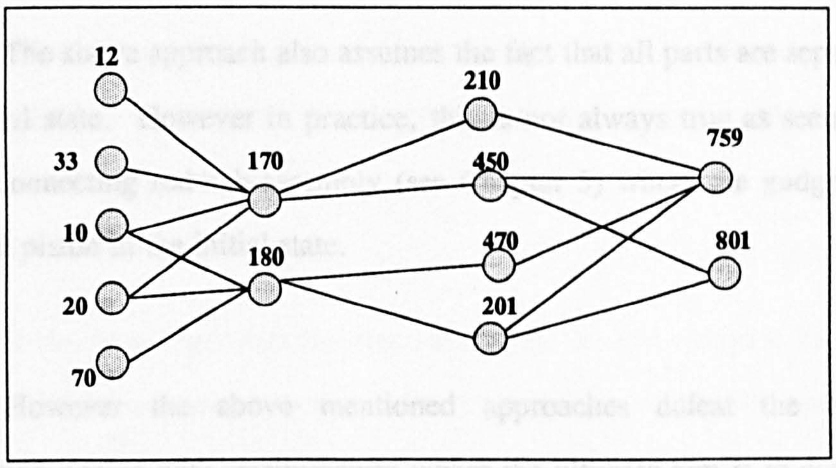

**Figure 4.2 An Example of a Precedence Network**

Once the engineer has completed *drawing* the network the system generates a job list in the required order. Whenever changes have to be made to the network, it can be easily retrieved and amended by the engineer and a new job list can be obtained. The manual drawing of the graph (once the relationships have been established) took e.g. three hours in two hundred and fifty operations application whereas the semi-automatic approach was able to reduce this time tremendously, i.e almost instantaneous response after amendments were made to the graph [44].

In an extension to the question and answer approach by Fazio et al, Sanderson et al [45] presented an approach to automate the assembly sequencing problem using relational AND/OR graphs. These graphs are used to represent all the assembly sequences from which a set of all possible sequences are derived. Ordering constraints are further employed to derive a set of possible or feasible sequences.

The above approach also assumes the fact that all parts are separate in the initial state. However in practice, this is not always true as seen in the piston-connecting rod sub-assembly (see Chapter 5) where the gudgeon pin is in the piston in the initial state.

However the above mentioned approaches defeat the aim of integrating design with manufacture where the ultimate aim is to achieve it with minimal or no human intervention. The alternative to an algorithmic approach is to apply AI techniques. In engineering terms, an AI approach is rather like the expert systems approach where relevant rules are acquired from experts (process planners in this case) using so called knowledge acquisition techniques.

Acquiring knowledge which is a vital ingredient of AI systems is recognised to be the hardest and often a painful part of the process of designing AI systems [46]. The main methods of knowledge acquisition are by reading relevant books and interviewing experts. Psychologists have identified various interviewing techniques. These range from actually talking to the experts to observing them at work. One of the methods that has been suggested is the repertory grid method where a set of objects in the domain is collected and the expert is asked for the similarities and differences between these objects [47]. From the answers, a set of rules are then derived. The main difficulty of using such a technique is the formulation of appropriate questions that will enable the system developer to gain enough knowledge to derive the relevant rules for the system. The clinical origins of these techniques is another obstacle to the success of these techniques as process planners may feel uneasy about the approach. Hence these methods which have been suggested by the psychologists are very difficult to use in practice.

It has been suggested that since acquiring the knowledge is seen as a bottle-neck to developing expert systems then the experts should develop the systems themselves. The argument against such cases is that the experts lack the experience in applying the best (knowledge representation and manipulation) techniques and approach for the development of expert systems. Therefore the efficiency and performance of the resultant system is far from desirable.

It is recommended that in order to produce working systems, involvement of experts is vital. They should be part of the team (consisting of programmers and *decision-makers*) that is involved in such projects. In this way, they would be willing and are available to provide the knowledge. Once the knowledge which is a vital ingredient of AI systems is obtained, it is important to be able to represent it in the best possible way.

## 4.3 Representing Knowledge

The philosophy of AI planning systems (or AI systems in general) is centred around the manipulation of knowledge. Therefore when attempting to build AI systems, it is essential to know how and in what form knowledge can be stored. In contrast to conventional computational methods where the knowledge of a problem is implicit in the algorithm (e.g. a program which manipulates data from a record will not work if the record structure changes), AI systems manipulate knowledge that is explicitly stated in a specially designated area normally known as a knowledge base. Examples of some common representation techniques are: production rules, semantic networks, frames, objects, scripts and predicate logic.

## 4.3.1 Production Rules

Production rules can be used to represent knowledge and they are a popular method of knowledge representation. They were first used by Newell and Simon for modelling human cognition [48]. The basic idea is that the database consists of rules called productions in the following format :

if A then B

where A is the premise (condition or antecedent) of the rule,

B is the consequence or conclusion.

An example of a production rule is:

if (table is clear) then

(put the block labelled A onto the table).

Sometimes the rule can be written as: A -> B ('->' is read as *implies*)[4].

Due to the structure of production rules, they are considered to be a natural way in which people express certain types of knowledge, e.g. statements about what to do in predetermined situations. The ability to add or delete individual production rules independently and thus allowing the system to be built in a modular way is also an advantage of this representation technique. However as the number of production rules increases, there is a large number of interactions between rules which may cause problems like inefficiencies (slowing down the response times) and difficulties in tracing the flow of the system. While production rules are naturally good for representing situation-action type of knowledge, they are however poor at expressing algorithmic knowledge. This is due to the isolation of production rules (they do not *call* one another) and the uniform size of the productions (i.e. no subroutines where one production may be composed of several sub-productions). In such cases, other programming languages offering the use of function calls and subroutines may be a better choice. Production rules are often used in AI programs to represent a body of knowledge about how people do a specific real world task like in medical diagnosis or mineral exploration[5]. Planning knowledge can be thought of as a type of situation-action knowledge e.g. *if the top block A is clear then put block B on top of block A*. Therefore the assembly planner in this project is based on production rules.

[4] It also resembles a Prolog rule as illustrated in Section 4.5.1.

[5] The MYCIN system which is used in aiding the diagnosis and selection of therapy for patients with bateremia or meningitis is a famous example of a production rule system.
The PROSPECTOR program has been used to help locate deposits of several minerals,like copper and uranium.

## 4.3.2 Semantic Networks and Frames

Semantic networks were originally developed to support the work of natural language processing. This formalism consists of nodes and arcs which link these nodes. The nodes represent objects or concepts about a problem and the arcs represent the relations between them. A simple example of a semantic net is illustrated in Figure 4.3.

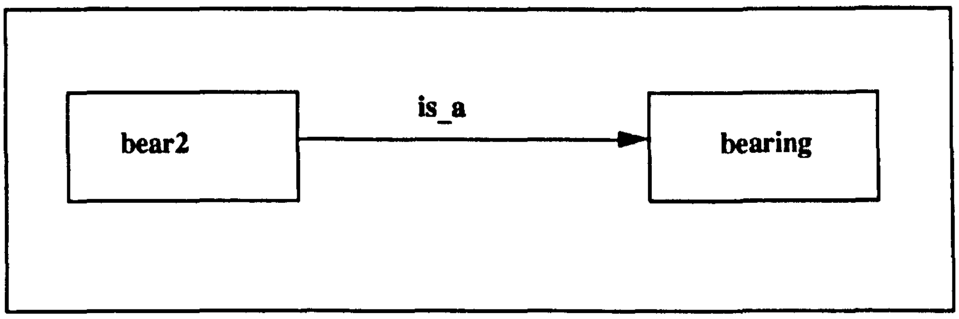

Figure 4.3 A Simple Semantic Net

The simple fact that *bear2* is a bearing is represented by the semantic net in the above figure. In semantic networks, knowledge is ordered in a tree-like structure usually with a top level node representing all objects.

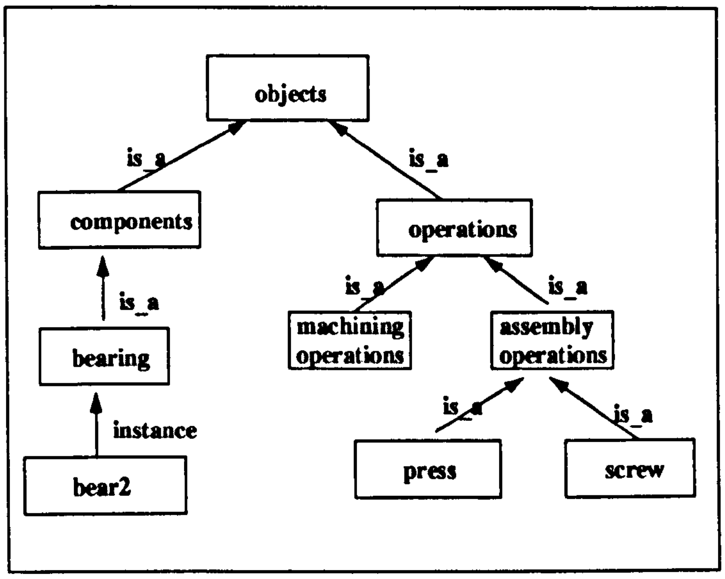

Figure 4.4 A Semantic Net to Illustrate Inheritance.

From Figure 4.4, the *is_a* and *instance* relations can be used to represent property inheritance which allows the building of specific models from generic ones. In the above figure, *bear2* is a specific instance of bearings and bearings *is_a* kind of component. Semantic nets were used by Jamie Carbonell in his tutoring program, SCHOLAR which answered questions about the geographical information stored in the net [49].

In contrast to logic (see section on Predicate Logic), there is no fixed notion of what a given representational structure in a semantic net means. The reasoning in semantic nets is governed by the procedures that manipulate the network. A reasoning mechanism that is used most widely is based on matching network structures. A query in the semantic network system is matched against the network database to see if such an object exists. For example, from Figure 4.5, if the question, *What are the* *assembly operations ?* is asked, a network fragment (or sub-net) resembling the figure below can be formed:

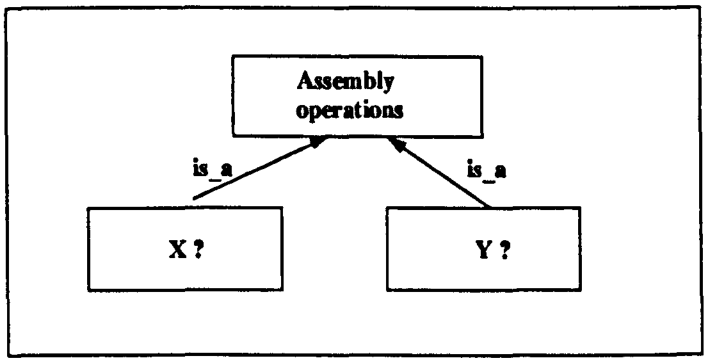

Figure 4.5 A Network Fragment

The matching mechanism is able to infer that *press* and *screw* are instances of assembly operations by matching the network fragment against the network database.

The concept of frames originally proposed by Minsky (1975) was developed from the early work on semantic nets. Frames are like an aggregate of network structures. They are based on the idea that new data can be inferred from previous experience. There are basically two main types of frames: generic and specific frames. Generic frames represent knowledge about a general concept and can be thought of as providing a prototype for information about actual objects. It consists of a type name and one or more default properties (or slots). Like semantic nets, a specific frame represents knowledge about a specific object, i.e. an instance of a generic frame. The specific frame may also inherit some properties (default values) from the generic frames. Figure 4.6 below shows an example of a specific frame, i.e the frame for *bearing2*. The generic frame for this could be the *bearing* frame which holds general information for bearings.

**Bearing2 Frame**

| | |
|---|---|
| **specialisation of :** | **Bearing** |
| **Inner diameter:** | **4.8cm** |
| **outer diameter:** | **6.8cm** |
| **weight:** | **250gms** |
| **action:** | **press** |
| **tools :** | **manual** |

**Figure 4.6 An Example Of A Specific Frame**

The generic frame is also able to suggest default values for a specific frame, if there is no contradictory evidence. It is also possible to attach procedures to the generic model (e.g. a tedious procedure for calculating the time of an operation) which allows these procedures to be used for specific frames. An extension of this procedural attachment is the idea of an object oriented programming system (OOPS). The OOPS is based on the concept that each item of data (or object) should contain within itself methods to specify how it can be processed. It also contains generic objects which dictates how specific objects should behave since it is inefficient for every object to contain all the procedures.

Scripts are frame-like structures which were developed for representing sequences of events [50]. They provide a formalism for representing people's everyday knowledge about stereotyped activities such as going to a restaurant or driving a car. Thus when confronted with a similar situation, the defaults in a generic script are used. This approach is used by Carbonell et al for translating stories [51]. The program called SAM (Script Applier Mechanism) attempts to understand short stories using a script to guide the interpretation of occurrences in the story. Once the appropriate script is found, the *slots* are filled with the appropriate information from the story. Based on this information, the system is then able to make inferences about similar events.

It is possible to stretch semantic nets or frame representation schemes too far, i.e. using nodes and links to represent objects and relations in the world. In such cases, the huge amount of computational effort required to process such large networks could become unmanageable. Furthermore, beliefs and ideas which are quite different from facts may also need to be represented. Semantic nets or frames may not be able to represent such *objects*. However the main problem with semantic nets or frames comes from the inheritance factor which is what made it so attractive a representation technique in the first place. It fails in cases where there are no strict inheritance of properties. This is best illustrated by the classic example below:

*All birds can fly .....*
*except penguins, ostriches, dead birds, ....*

The other problem with semantic nets is that of multiple inheritance. For example, a group of engineering components, such as turbine blades may inherit properties from the engine in which they are installed and another set of properties from the materials from which they are made. There are no problems if the properties are distinct but there is potential for inconsistencies of the knowledge if an object inherits the same property from more than one source. This leads onto the next section which describes how the above classic example may be solved.

### 4.3.3 Non-Monotonic Reasoning

In order to recognise special cases like the example in the previous section, it is sometimes necessary to change the truth value of a proposition from say *true* to *false* as more information becomes available. This kind of reasoning is known as non-monotonic reasoning as opposed to monotonic reasoning where propositions are added but not changed. As can be seen, human reasoning is clearly non-monotonic. It is often the case that plans may have to be revised whenever situations change, e.g. re-order the sequence in a process plan in the light of new constraints. The two most popular formalisms for this kind of reasoning are circumscription and default reasoning [52]. Circumscription assumes that no objects exist except those that are mentioned previously. Thus the statement that all birds can fly would represent the class of all birds that can fly excluding those that cannot. In default reasoning, guesses are made in the absence of contradictory evidences. Hence until some animals (such as penguins) can be found to contradict the statement that all birds can fly, it will be assumed that the statement is correct. The STRIPS representation also made use of default reasoning by assuming that the actions that are performed will not change any of the system's beliefs about the world states except those that are explicitly listed in the description, i.e. add and delete lists.

The Truth Maintenance System (TMS) by Doyle [53] is one of the first systems that support non-monotonic reasoning. In his system, those propositions that are currently true are marked as *in* and those that are not currently true are marked as *out*. Thus as new information or knowledge become available, a proposition that is *in* may be changed to *out* and then at a later date changed back to *in* should information becomes available again to support its truth. The system has to restore consistency whenever a contradiction occurs. A form of backtracking, dependency directed backtracking is used to maintain these markers. Instead of backtracking in a chronological manner, dependency directed backtracking records dependencies as the search progresses to allow a guided search using these dependencies.

Doyle's system is a justification based reason maintenance system. A belief is held to be valid if it is supported by a set of valid reasons for that belief. A datum is represented by the TMS as a node which is assigned a support status of being either *in* or *out*. Every conclusion derived along with the antecedents to that conclusion are stored as justifications. For example, given a rule: $A$->$B$ and the fact $B$, in order to conclude $A$, both the rule and fact are needed. Hence these are both recorded in the system. Given a new fact, if the antecedents of a rule are satisfied then the consequent is accepted. A node for that consequent would be created and its justification recorded in the *in* list. A node is said to be *in* if at least one of its justification is currently valid. Otherwise it would be *out* if it has no valid justifications. A variation of this is an assumption based TMS. In this system the underlying assumptions made for the facts are also stored in addition to the justifications. For example, given the rules, $B <\!\!-\!\!- A$ and $A <\!\!-\!\!- C$, the assumption of $C$ is sufficient to conclude $B$. Hence $C$ is recorded as an assumption to conclude the truth of $B$ [54].

The blackboard approach first developed in the HEARSAY-II speech recognition system is also used to model non-monotonic reasoning [55]. The blackboard system consists of a set of independent modules known as knowledge sources. Each of these knowledge sources contain specific knowledge about a sub-portion of the system and the blackboard is a shared data structure which all the knowledge sources have access to. In this way, the blackboard is rather like a manager overseeing the activities of a complete system and could maintain consistency of the knowledge by *marking* the data like Doyle's Truth Maintenance System. Since the knowledge sources are kept as separate modules, different inferencing mechanisms can be used for each of these modules. This is useful in cases where the characteristics of these modules are very different and may therefore require different inferencing mechanisms. Due to its inherent modular structure, the blackboard approach is said to be suitable for implementing large AI systems.

## 4.3.4 Predicate Logic And Prolog

As mentioned in Chapter 3, logic can be used to represent the meaning of sentences or knowledge. It is one of the first representation scheme used in AI work. The derivation of new facts from old facts can be mechanised using an automated version of theorem proving (see Appendix i). Hence theoretically, the deductions made here are guaranteed to be correct. This is what other representation schemes (such as semantic nets and frames) are not able to do.

Propositional logic is a form of formal logic. A typical proposition is *Yen is James's mother.* It is possible to assign the truth value *true* to this proposition[6] . Hence propositions are those *things* that can be either *true* or *false.* However individual propositions like these are not very interesting. Thus sentence connectives such as *and, or*, etc are used to join propositions together. If *A* and *B* are two propositions (or formulas) then the following definitions apply:

*A and B* is true, if *A, B* are both true.

*A or B* is true if at least one of *A* and *B* are true.

*A implies B* is true unless *A* is true and *B* is false.

*not A* is false if *A* is true and vice versa

[6] The notion of 'truth' is fundamental to logic systems. Terms such as *James's mother* and *James's food* are not propositions because it is not possible to assign truth values to them.

Thus given the formulas and their truth values, it is possible to make inferences and draw conclusions. For example, given:

*A or B*

and *not A*

then: *B* is true.

Further refinements of this system to allow more *expressions* to be represented is necessary, i.e. represent not only objects and relationships between objects but also generalise these relationships over classes of objects. This feature is found in Predicate Calculus which is an extension of Propositional Calculus. In addition to the use of connectives, it allows the representation of specific objects or individuals. An example of a predicate is *happy* and *happy(james)* is a predicate with one argument. Predicates can have more than one arguments. For example, *mother(yen,james)* is a predicate with two arguments. The notion of quantifiers are also used to refer to facts that are true of all or some of the members of the group. These are variables and quantifiers. A variable[7] is used to represent individuals that will vary with time. As mentioned in Chapter 3, there are two types of quantifiers namely, the universal quantifier ( $\forall$ ) to represent all the members and the existential quantifier ($\exists$) to represent some of the members of the group. Quantifiers are a means of talking about sets of individuals and what is true of them. As with propositional logic, it is possible to make inferences except that Predicate Calculus provides a more expressive way of saying things. In order to

[7] This is like a variable in computer programs.

allow all these to be represented and manipulated in a computer, the programming language, Prolog was developed.

The main drawback of such an automated system is that the manipulations (e.g. the procedures in resolution (Appendix i)) can become clumsy when there are large number of facts in the system. This leads to a combinatorial explosion in the possibilities of which rules to apply to which facts at each step of the proof. It seems more *knowledge* is needed to determine which facts are relevant in what situations in order to guide the proof. In Prolog, the selection of which rules to apply first is determined in a depth-first manner, i.e. rules that are ordered first will be *fired* first. In a way this prior knowledge is incorporated by the programmer when he/she orders the rules in the system. The sequence of the selection of rules can be altered by using the *cut* (written as *!*) facility which will be described later.

## 4.4 Processing Knowledge

There are two main strategies to processing knowledge, either deduce a solution from a set of initial data or prove that the goals are true. These approaches are called forward chaining and backward chaining respectively[8].

In backward chaining, the goal is usually broken down into sub-goals which makes it easier to prove (i.e. if all the sub-goals have been

[8] How these could be done in automated theorem proving systems are described in Appendix i.

proven then the goal is proved). This approach is best for solving problems which aim to pick the best choice from many enumerated possibilities, e.g. identification and diagnostic systems. As for forward chaining, it is usually used when it is not possible to enumerate all of the possible answers before hand. Some examples are configuration problems like circuit board design and office space layout. The following example is used to demonstrate how these two strategies work.

Example:
rule1: if john holds a studentship then he is a full-time student.
rule2: if john is a full-time student & he is a post-graduate student then his maximum demonstration load is 10 hours per week.
fact1: john holds a studentship.
fact2: john is a post graduate student.
Goal : john has a maximum of 10 hours of demonstration per week.

In backward chaining, the goal matches with the right hand side of rule2, so conditions of rule2 must be satisfied. John is a post graduate student, so it matches with fact2. Thus only the sub-goal, *john is a full-time student* remains. In the next cycle, this sub-goal matches with rule1 which is then reduced to - *john holds a studentship.* This matches with fact1 and so the original goal that john has a maximum of 10 hours of demonstration per week is indeed true.

In forward chaining, starting from fact1 and fact2, rule1 can be fired because fact1 matches its left hand side. Another fact, *john is a full time student* is then added to the set of facts. Rule2 can now fire because its left hand side matches the fact: john is a full time student and fact2. Thus *john has a maximum of 10 hours of demonstration per week* is proved.

The graph that is generated in a forward search can be very large as the number of possibilities increases. The backward chaining approach is a more directed approach and would be a more suitable reasoning mechanism for assembly planning problems.

## 4.5 Programming Techniques

Besides using a computer language, planning knowledge can also be modelled using an expert system shell. The former approach is used when more flexibility and control is needed, i.e. the designer of the system can make changes more easily and thus has more control over factors like the output display, input format, internal representation format and manipulation techniques. However using a computer language means that the development time is longer and sometimes more expensive than the latter approach (because the programmer must have knowledge of the programming language). There is a wide range of computer languages to choose from, some are better in certain aspects[9] than others. Prolog and Lisp are the two most popular languages for AI applications. Originating

[9] Because they have evolved from those aspects.

from France around 1970, Prolog is popular amongst the European AI community[10]. Written by McCarthy, LISP is the most popular AI language in the USA[11]. AI languages consist of functions for manipulating symbols which are central to modelling AI representation techniques outlined earlier. While it is recommended that one of these so-called AI languages be used to develop AI applications, it is also possible to develop AI based programs using more conventional languages like Pascal or C, except that in these cases the development time may be longer because the programmer may have to write more lines of code. An example of a process planner that is said to incorporate AI techniques which is written in PASCAL is TOM [56]. Sometimes the performance of systems developed in AI languages is not as desirable as systems written in conventional programming languages. This is however changing with the introduction of say, LISP machines which are dedicated machines for AI applications. Nevertheless these machines present problems of hardware compatibility.

Today shells have become a popular tool to produce prototype expert systems. The designer has a wide range of expert system shells available in the market to choose from, e.g. KES, CRYSTAL, GOLDWORKS[12]. These tools usually provide the programmer with a knowledge representation structure as well as a means of manipulating the knowledge. Hence the programmer only needs to add in the domain specific knowledge that is relevant to his/her particular application. As mentioned, they are very

[10] Prolog was invented by Alan Colmerauer and his associates as a result of research in logic programming.

[11] LISP is all about representing and manipulation of information in lists. Prolog is a resolution theorem prover.

[12] KES provides three main inference mechanisms, each of which are marketed as a separate entity. CRYSTAL is a production rule based system with menu type user interface. GOLDWORKS is a hybird environment providing a mixture of representation schemes: frames, production rules, OOP. All these shells are available on personal computers.

useful for developing impressive prototypes in a short period of time. However the programmer would be limited to the knowledge representation technique and manipulation technique built into the shell. Shells also have the reputation of being only good at solving problems that they are originally designed for. For example, the production rule system of the KES shell is ideally suited to diagnostic applications. Hence examples from the medical or biomedical domain have been used to demonstrate its power.

The next option is to provide a toolkit where there is a combination of representation and manipulation techniques for the programmer to choose from[13]. These toolkits usually cost a lot more (since they provide more features) and sometimes require sophisticated and dedicated machines to run on. Examples of some toolkits are ART and KEE.

It was decided to use Prolog in this project because:

i) other earlier AI planners used predicate logic which is the basis of Prolog;

ii) production rules are an integral part of the planning process and these are easily implemented in Prolog;

iii) backward chaining is an inherent feature of Prolog;

iv) Prolog is widely implemented on different hardware platforms;

v) prior experience of Prolog.

[13] This is partly addressed by the availablity of shells like GOLDWORKS. As machines become more sophisticated, the possibility of providing more features in shells have become a reality and sometimes a neccesity. Hence the division between toolkits and shells is not so clear any more.

Some of the basic features of Prolog can be found in Appendix i.

### 4.5.1 Assembly Planning In Prolog

A Prolog program can be regarded as a set of clauses (i.e. the knowledge base) and the Prolog interpreter as a theorem proving program which performs deductions on request from that set of clauses (see Appendix i). Since Prolog is restricted to accept only positive Horn Clauses, somehow one negative Horn clause has to be added for the proof to be possible. This is because resolving two positive clauses would produce a new positive clause and one consequent. However in refutation, it is needed to proof the empty clause and the consequent cannot be got rid of unless it is resolved with one consequent-free (negative) clause. Having two consequent-free clauses is of no use. Once one has been used, all the resolution products from then on will be negative clauses and positive clauses are needed to resolve these with. So the second negative clause is never used. Similarly for three, four, etc. This is the reason why Prolog restricts its database to positive Horn clauses. To allow resolution to work, one negative Horn clause is temporarily added. Hence from the user's point of view, this clause is the query, i.e. like a statement of what (s)he wants to proof [57].

In the assembly planner, this query is in the form of a description of the goal state of the problem. The planner then attempts to perform the resolution proof (i.e. that the negation of the goal state leads to an empty clause) using the Prolog interpreter. If the proof succeeds then the goal state (or the query) is true, and as a result of the proof certain variables become instantiated and the plan to achieve the proven goal state is produced as a result.

For example, in the description of the goal state, the assembly of a mating pair can be represented as follows:

*assemble(A1,A2,A3,A4).*

where *(A1,A2)* & *(A3,A4)* is a mating pair,

*A1, A3* refer to the part names,

*A2 & A4* refer to the features on the A1 & A3 respectively.

So in particular, the fact that the cap of a ball point pen is covering the bottom end of the pen can be represented as follows:

*assemble(cap,hole,pen,bottom_end).*

where *(cap,hole), (pen,bottom_end)* is a mating pair.

Similarly, the features on the parts may not be at the assembled state and the predicate, *clear* can be used to represent the opposite state to *assemble*. Hence,

*clear(pen,bottom_end).*

means that the bottom end of the pen is *clear*, i.e. not covered.

## 4.6 Conclusion

Assembly sequencing is a complicated activity that could be solved by decomposing the problem in order to reduce its complexity. This can be done by problem reduction where the problem is decomposed into smaller manageable sub-problems as in the sub-assemblies of the gear box of a car. The problem of finding a possible sequence of assembly actions can be an imposing one because as the number of parts and features increase, the number of possible mating combinations is increased enormously. An alternative to the algorithmic approach is to use AI planning techniques.

In order to apply AI planning techniques to assembly process planning, an appropriate knowledge representation scheme as well as knowledge processing strategy are essential. As most of AI planning research is based on a logical approach using state space representation, a good starting point is to follow this method initially. The main advantage of expressing programs in logic is that they can be defined in machine-independent human oriented terms. This means that they are easier to construct, i.e. converting from problem definition and thus easier to understand. The AAP described in the next chapter is implemented in this way, i.e. working from the definitions to obtain the code. Hence these programs are also easier to improve and are therefore more adaptable to other purposes.

Even though planning knowledge may be more akin to the formalism offered by production rules, semantic nets and the other representation techniques mentioned above are also attractive representation schemes in their own right. These representation schemes have evolved as a result of the need to tackle certain specific problems, e.g. scripts for natural language understanding. Semantic nets and frames are very useful in situations if the inheritance factor is an important feature of such applications. Today it is usual practice to make use of a hybrid representation scheme which includes a mixture of these representation schemes (which have their own individual merits for certain types of situations).

At the same time, increased capacities in terms of memory and processing speed, of AI toolkits enable say, the blackboard architecture to be adopted when developing a manufacturing system. In the context of manufacturing, with the trend towards integration the concept of the blackboard approach is attractive to develop a fully integrated system. With lessons learnt from the problem of the situation of the *islands of automation*, the various functions within a manufacturing system can neither be built nor exist in isolation any more. The complete system has to be conceived and implemented as a whole, rather like the planning problem. Finally, for the reasons mentioned above, predicate logic, backward chaining and hence Prolog was used for the development of AAP in this project.

# CHAPTER 5. AN AUTOMATIC ASSEMBLY PLANNER

## 5.1 Introduction

As mentioned in Chapter 2, since the process planning task is a subject that is not well written about, the starting point of the project therefore involved making field trips to the process planning department of ROVER in order to obtain some information on the process planning task. After the initial meeting, a number of similar meetings were set up subsequently with the main objective of acquiring the relevant knowledge in order to implement a process planning system.

## 5.2 Results of Field Trips

The process planning department in the company visited is divided into various sections with each one being responsible for a particular component (e.g. gearbox, suspension, etc) of the car. The average size of each section is around three to four members of staff. The initial stages of process planning consist of making rough process plans to obtain estimates on the costs of production. Rough costings are then made against each operation and part listed in the rough process plan. The costings have to be approved before actual production commences. The main activities of the process planning staff concerned are usually involved with producing drawings and process plans on the sub-assemblies concerned. Implementation of a complete process planning system is a long term exercise and requires full commitment from the company concerned in order to relieve an expert (or more experts) for the project.

As a starting point, a little planner was developed utilising the techniques of some of the AI planners described in Chapter 3. After gaining some exposure to the work of process planners, it was decided to investigate further a formalism for the basis of process planning. The planner had to be based on more general concepts in order to avoid the narrow path taken by previous researchers who had concentrated on planning for very specific parts in a set environment. Description of the Automatic Assembly Planner (AAP) can be found in the following sections. An initial planner was later extended and revised to cope with some assembly examples with an emphasis on improving the sequencing process.

## 5.3 Initial Tests

As mentioned in Chapter 3, the two main types of plan representation techniques are state space and action ordering plans. AAP was originally based on the state space representation and then modified to include aspects of action ordering plan techniques. The main reasons for choosing Prolog to model the planner have been discussed in the previous chapter. The objective of the planner is to state the order of operations that must be undertaken in order to achieve a particular assembly configuration. No attempt has been made at this stage to incorporate the time, costs and other constraints associated with process planning.

## 5.3.1 The Test Planner

Using the notion of state descriptions as described in STRIPS, a simple strategy for a planning program is to pick a single goal to work on. If it is already true in the initial state then proceed with the next goal otherwise make that goal true by choosing an action that will make it true. It may be necessary to recursively try and make it true (e.g. achieve its pre-conditions first if it is not already true in the intial state). When this goal has been satisfied then the next goal is tried until all the goals in the goal state description are true.

This simple strategy is summarised as follows:

i) check if the goal state already exists. If yes then the goal is achieved and nothing else needs to be done otherwise attempt to achieve that state.

ii) achieving the state:
check if there is anything obstructing or preventing the achievement of this state. If none then attempt to achieve that state otherwise unblock or remove obstacle(s) in order to achieve state.

iii) Repeat for all the goal states in the problem.

iv) Print the actions (produced as a result of achieving the above goal states)

An example of the sub-assembly of the piston-connecting rod which is a sub-assembly of the gearbox is chosen to demonstrate the test planner. This is illustrated in Figure 5.1.

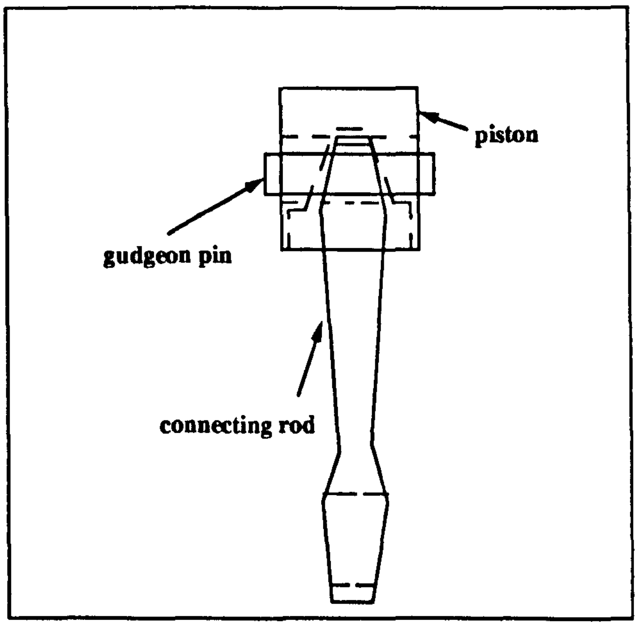

**Figure 5.1 The Piston-Connecting Rod Sub-Assembly**

Figure 5.1 also represents the configuration of the goal state of this sub-assembly, i.e. the connecting rod is inside the piston and the gudgeon pin is used to secure the connecting rod to the piston. An initial state of this configuration where the various part are separate is illustrated in Figure 5.2.

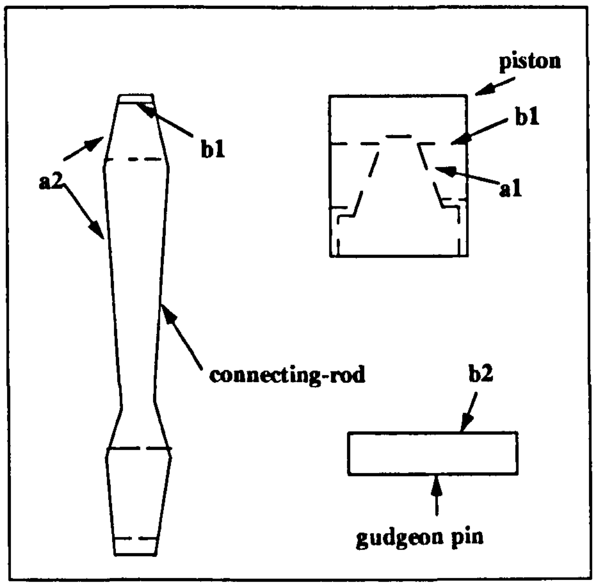

Figure 5.2 An Initial State of Piston-Connecting Rod Sub-Assembly

As mentioned in Chapter 4, the mating parts and their features have to be considered for assembly purposes. Working from the above example, the relevant features are named as indicated in Figure 5.2. Thus *a1* of *piston* mates with *a2* of *connecting rod* and *b1* of *connecting rod* mates with *b2* of the *gudgeon pin*. The next stage in the analysis of the problem involves identifying relevant predicates and actions for representing the configurations. The predicate, *inside* was used to denote the fact that the gudgeon pin is inside the piston and the action to achieve this is, *pushin*. The opposite of *pushin* is therefore *pushout*. The description of the initial state has to be stored as facts that are true at the start of the planning process. A database called *world* is used to contain the descriptions of the initial state. Hence the world database should contain the following facts at the initial state:

```
clear(a1,piston,in).
clear(a2,rod,out).
clear(b1,rod,in).
clear(b2,pin,out).
```

*in* and *out* in the above facts were used to indicate whether a feature is internal or external to the part respectively. The next stage of the exercise involved defining the relevant pre-conditions and post-conditions. An example of a pre-condition defined in Prolog is as follows:

```
result(pushout(V,W,X,Y)):-
        setof(inside(V,W,X,Y),world(inside(V,W,X,Y)),X1),
        X1 = [inside(V,W,X,Y)],
        assertz(action(pushout(W,Y))).
```

This can be interpreted as: the result of *pushout* part *W* from *Y* succeeds if feature *V* of *W* is *inside* feature *X* of *Y*. The *setof* predicate is used to find the information in *world* that matches this description. If it succeeds then the *pushout* action is asserted in the *action* database which keeps a record of the actions. The *assertz*[1] system predicate means to add to the end of the corresponding database. The opposite to this, i.e. removing a fact from the database is the *retract* system predicate. Other pre-conditions can also be defined in this manner.

[1] **asserta is a system predicate that will add facts to the beginning of the database and assertz will add facts to the end of the database. Similarly for retract.**

It should be noted that the *retract* predicate may only mark that particular predicate for removal rather than physically removing it. The actual removal would only occur when the top level goal is solved. For good programming practice, these predicates that will modify the initial state of the Prolog clauses and hence the possibility of modifying the intended meaning of the original program should be avoided, if possible. As seen later in the revised planner, system predicates of Edinburgh Prolog such as *record* and *recorded* were used instead to set up an internal database for manipulating data.

Similarly a post-condition of the planner can be defined in Prolog as follows:

```
state(inside(V,W,X,Y)):-
        result(pushin(V,W,X,Y)),
        retract(world(clear(V,W,out))),
        retract(world(clear(X,Y,in))),
        asserta(world(inside(V,W,X,Y))).
```

This can be read as: to achieve the state of *inside(V,W,X,Y)*, it is the result of *pushin(V,W,X,Y)*. The rest of the sub-goals are used to update the *world* database after execution of the action, *pushin*. On the other hand, the *result* predicate describes the pre-conditions of the action, *pushin* and has to succeed before the respective databases can be updated accordingly, i.e. the state *clear(X,Y,in)* will not be true after achieving the *pushin* action and the fact that feature *V* of *W* will be *inside* *X* of *Y* will be inserted into the database, *world*. The goal state description can be defined in Prolog as :

```
inside(a2,rod,a1,piston),
inside(b2,pin,b1,piston).
```

This means that the connecting rod is inside the piston and the mating pairs are {(a1,a2),(b1,b2)}, i.e. feature a1 of the piston mates with feature a2 of connecting rod and feature b1 of piston mates with feature b2 of the gudgeon pin and so on . The pin is inside the piston with mating features as (b1,b2). The action list associated with the configurations as illustrated in Figure 5.1 and 5.2 is given below:

```
action(pushin(rod,piston)).
action(pushin(pin,piston)).
```

The above action list means that the rod should be pushed into the piston, followed by the pin being pushed into the piston. However in the actual environment, the initial state is as shown in Figure 5.3. The assembly worker has to remove the pin from the piston before actually assembling the connecting rod.

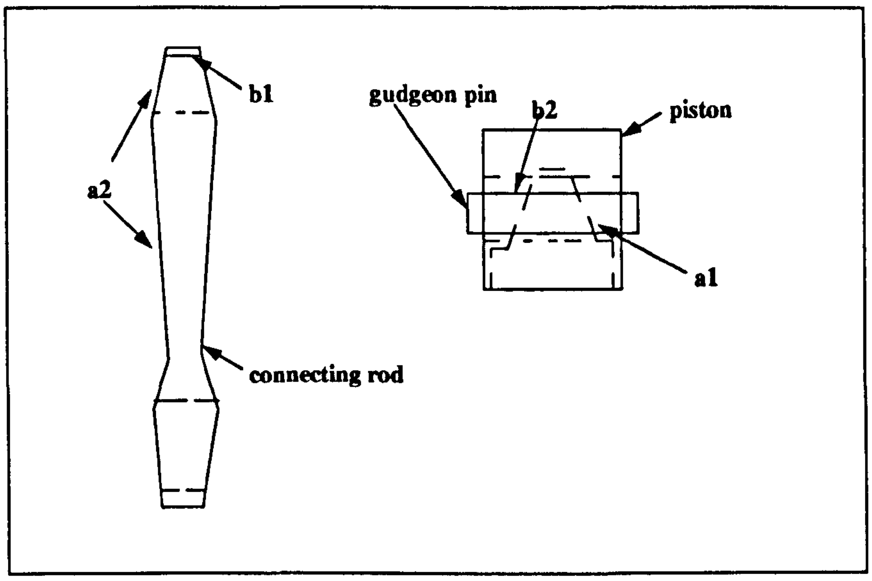

Figure 5.3 Actual Initial state of Piston Connecting Rod Sub-assembly

In this case, the actual action list is therefore:

*action(pushout(pin,piston)).*

*action(pushin(rod, piston)).*

*action(pushin(pin,piston)).*

A complete listing of this test planner can be found in Appendix iv. This simple planner only works correctly if the sub-goals are already ordered otherwise the action list could be wrong as shown in run-time example 2 of Appendix v. In most real cases, the sub-goals are not ordered and so planning involves considerable trial and error and sometimes undoing progress (as shown in Sussman's example) that has been achieved. Thus extra *planning* knowledge such as the ability of finding out what other sub-goals there are and also checking if they conflict with the present assumptions before attempting to achieve any given state is needed. This type of *planning* knowledge is often regarded in the literature as non-linear planning knowledge. Thus some kind of ordering procedures are required before the basic concept (i.e. that of achieving states) of the planner can be applied. In order to examine the planning process globally and not locally as in the simple strategy described earlier, action ordering planning knowledge has to be added to the planner.

In addition, the representation of the states (i.e. goal and initial) can also be a problem. The determination of what predicates and how many arguments to best represent the state descriptions are essential. Based on the part-feature pair description, a state description with four arguments would have been adequate. However an addition of two more arguments is made to indicate an extra pair, i.e. the part-reaction face pair as reaction faces are often used in manual assembly and which reaction face to use is indicated as early as the design stage. Hence extra information on which reaction face to use and where it is, i.e. the name of the part where the reaction face is has to be added as well. If the assembly does not require reaction faces then these two arguments will be written as *nil,nil*.

At the same time, the identification of individual mating surfaces was found to be too detailed and clumsy as the number of parts (in a sub-assembly) increases so more general state descriptions were developed. Hence instead of using predicates like, *inside,* a more general state descriptor, i.e. *assemble* was used. The opposite state of *assemble* is *clear*. The representations for the state descriptions are

*clear(Part,Feature)* *and*

*assemble(Part1,Feature1,Part2,Feature2,Reaction_face,Reaction_object).*

Historically much effort has concentrated on limiting the *undoing* of goals as illustrated by Sussman's example. A variety of planning heuristics are scattered in a number of AI planners which were used to alleviate this problem. The decision facing the author was which planning heuristics to use and therefore which AI planners to adopt. TWEAK was used because most of the other AI planners were complicated and ill-defined. Since it also claims to be a complete and correct planner[2], it therefore stands a better chance at solving general planning problems. Hence if it can solve general planning problems, it should be capable of solving assembly planning problems as well.

## 5.4 Approach Of The Automatic Assembly Planner

AAP consists of both the state space and action ordering representations. The first part is like an action ordering planner because it attempts to re-order the sub-goals (which are the input to the planner) into

[2] This means that if it produces a plan for a problem, the plan will solve the problem and if the planner fails to produce a plan then there is no plan that will solve the problem.

a list of sequential (or ordered) sub-goals. Once this has been achieved the basic idea of achieving states described above can then be applied to produce an action list. Incidentally this action list is produced as a result of the proof that the goal state is true (from the resolution principle). An overview of the Automatic Assembly Planner (AAP) is given in Figure 5.4.

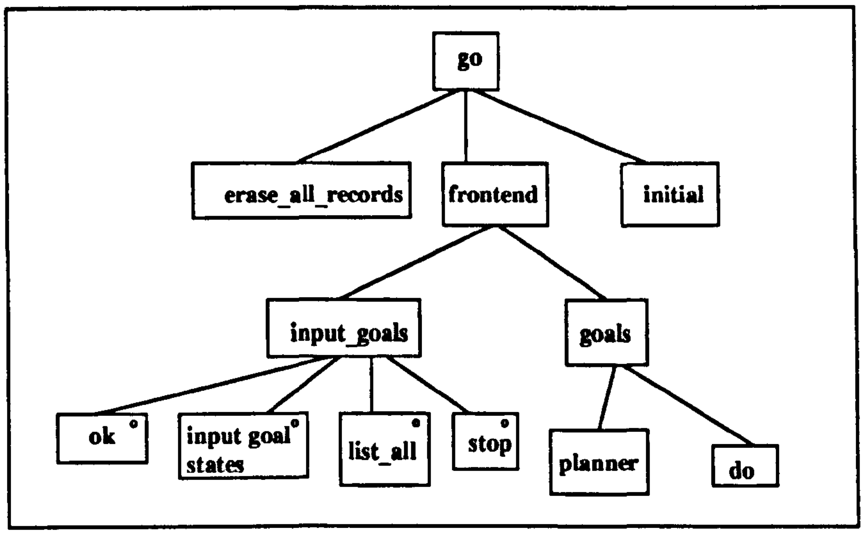

Figure 5.4 Overview of AAP

The root (or topmost goal) of the planner is *go* and consists of three sub-goals, i.e *erase_all_records*, *frontend* and *initial*. *erase_all_records* is for housekeeping purposes in order to reset the planner to the starting state. It erases all records that may have been set up previously by the planner. *Initial* is used to set up the initial states of the sub-assemblies, i.e. state descriptions based on Figure 5.3 will be recorded in the internal database, *world*. That means the key to this database is *world*. *Frontend* invokes the actual planner itself. It consists of two main sub-goals, *input_goals* and *goals*. The 'o' in sub-goals of *input_goals* is used to indicate that it is an option. *Input_goals* is used to read in the goal state description. There are four options here, i.e.

i) input a goal state description;

ii) *ok* to indicate end of goal state description;

iii) *list_all* to obtain a listing of the records in the world database.

iv) *stop* to terminate the planner.

Once this has been achieved, the next stage is to invoke the planner by calling *goals*. *Goals* is further divided into two main parts, i.e. the planner itself which is used to order sub-goals and *do* is used to produce the action list as described in the test planner previously.

The planner can be divided into three main parts which are:

i) appending to the *world* database the goal state descriptions and these are input by the user into the *newgoal* database;

ii) sequencing of the goal states to generate a *currentgoal* database;

iii) actions to be taken in order to generate an *action* list and updating of the *world* database at the same time.

## 5.4.1 Initialisation: Input of Goal States

The clause *initial* at the beginning of the planner is to record the initial states of the components to be assembled into a *world* list. The initialisation of the input goals is currently embedded in the program. For practical use, this would be replaced by either extraction of data from another system or interactive input. At the current state of development of the planner, any changes required in the initial configurations would have to be made by modifying the actual *initial* clause itself.

This part of the planner is mainly procedural and involves the instantiation of the initial states. The input of the goal states is defined in Prolog as follows:

```
input_goals(Goals,Option):-
            read(G),
            ((G == stop,
            Option = stop);
            ((G == list,
            list_all(world),
            Option = list, !);
            ((G == ok,
            Option = ok,
            findall(Y,recorded(newgoal,Y,_),Goals));
            ((recorded(newgoal,G,_);
            recordz(newgoal,G,_)),
            input_goals(Goals,Option))))).
```

The clause *input_goals(Goals,Option)* together with the built-in predicate *read(G)* allows each goal state to be typed in from the keyboard[3].

The format of a goal state is defined as follows:

*assemble(Object1,Face1,Object2,Face2,Reaction_object,Reaction_face).*

where *Object1,Face1* and *Object2,Face2* indicate the two mating components and their mating surfaces and *Reaction_object,Reaction_face* means between which component and surface the reaction force is to act. As mentioned earlier, if this specification is not required in a goal state, it is input as *nil,nil.*

For example, *assemble(bear2,whole,shaft,face2,shaft,face3).* means assemble bear2 to the feature face2 of a shaft with reaction face at face3 of the shaft, and *assemble(pin,whole,piston,t_hole,nil,nil).* means assemble the pin to t_hole of the piston (see Figures 5.8 and 5.9, Pages 121 & 122).

Each input goal state is checked immediately against the *newgoal* list to check if it is a double entry using the built-in predicate *recorded(newgoal,G,_).* If it is not, then it is recorded onto the list using

[3] Standard implementations of Prolog do not provide much in the way of assistance to the Prolog programmer to design sophisticated user interfaces. Some of the later implementations do provide tools for user interface design, e.g. the Prolog that runs in the Windows 3 environment marketed by Logic Programming Associates, UK.

the built-in predicate *recordz(newgoal,G,_)*. *recordz* will record a goal to the end of the list. By using a recursive approach, i.e. calling the clause *input_goals(Goals,Option)* again, the program will automatically prompt for input of the next goal state.

Finally, the three control options are provided by looking for three specific words from the input. If the input is *ok*, it will instantiate the variable *Option* as *ok* which indicates that the process of inputting goal states is complete and the planning can proceed onto the next stage, i.e. stage 2 of the planner by calling *goal(Goals)*. If the input is *stop* the program will be aborted. If the input is *list*, it will call the predicate *list_all(world)* to display all the initial states that have been recorded so far in the *world* database.

## 5.4.2 Sequencing

As established in Chapter 3, the main ingredient in planning systems is the maintenance of the truth of certain conditions at the relevant positions when ordering goals[4]. The task of sequencing is therefore to maintain the truth of all the sub-goals at each appropriate point in the planner. Hence

[4]This is known as the MTC in TWEAK

based on the truth criteria, the task of sequencing (i.e. reordering of the sub-goals) consists of four main parts which are summarised as follows:

i) Elimination of any redundant goal states and checking for conflicting goals.

ii) Generation of a *star* list which is defined to be a list of pre-conditions that are not present in the initial state but may be necessary for achieving the goal states.

iii) Ordering of the goal states.

iv) Inserting the necessary goal states from the *star* list to the existing ordered goal states.

The top level goal of the sequencer is defined in Prolog as follows:

```
goal(V):-
        .
        .
        redund_critic(V,Conf),
        ((var(Conf),
        findall(X,recorded(newgoal,X,_),U),
        addition(U),
        !,
        sorting,
        ((var(conflict),
        findall(Y,recorded(currentgoal,Y,_),[Z]),
        findall(S,recorded(star,S,_),W),
        insert_star(W,Z),
        findall(T,recorded(currentgoal,T,_),[C]),
        nl,
        write('The Ordered Goal is: '),
nl,
write(C),nl,nl,
.
.
.);
(nl,
write('Conflicting goals cannot be  achieved.... '),
nl,nl)).
```

The variable, *V* is instantiated[5] to a list containing all the goal states in the *newgoal* list before any sequencing occurs. After eliminating redundant goal states (if any) by the predicate *redund_critic*, the updated *newgoal* list is passed into the variable, *U* by using the *findall* predicate. Generation of a *star* list is done by the predicate, *addition*. Then the predicate, *sorting* sorts out the appropriate order of the goal states from the *newgoal* list. The variable, *Z* is instantiated to the currentgoal list whereas *W* is instantiated as the *star* list by the corresponding *findall* predicates. The final part of the sequencer is done by the predicate *insert_star* which inserts any necessary goal states from the *star* list to the *currentgoal* list. Finally the variable, *C* is instantiated as the ordered goal list which is written onto the screen before being passed onto the next stage of the planner which derives the appropriate action list.

[5]similar to initialisation in conventional programming languages. Recall the differences between these, i.e. destructive assignments in conventional programming languages is not relevant in logic programming.

#### 5.4.2.1 Elimination Of Redundant Goal States And Conflict Check

**Definition:**

A redundant goal state is defined as a goal state which happens to be one of the post-conditions of another goal state. A conflicting pair of sub-goals exists if one of the sub-goals is deleted by another sub-goal and vice versa, i.e. only either one of these sub-goals is achievable but not both.

Figures 5.5 and 5.6 are used to illustrate the redundant goal and conflicting goals situations respectively. In Figure 5.5, there are two goals, *A* and *B* to be ordered. *B* is a redundant goal if it is a member of the list containing the post-conditions of *A*. A conflicting pair of goals, *A* and *B* is said to exist if they clobber[6] one another as illustrated in Figure 5.6 where the *A* clobbers the pre-conditions of *B* and vice versa. Another conflicting goals situation also arises when a goal, *A* that is achieved in the final state is deleted by another goal, *B* which may be a redundant goal of another goal, say *C*. If this case is not detected, *B* may be eliminated from the input list by the redundant goals check (since it is a redundant goal) and in practice it means that no plans exist for such cases. Hence a redundancy as well as a conflict check must be carried out at the same time in order to avoid such goals being passed onto the next stage of the planner.

[6] This term is borrowed from TWEAK.

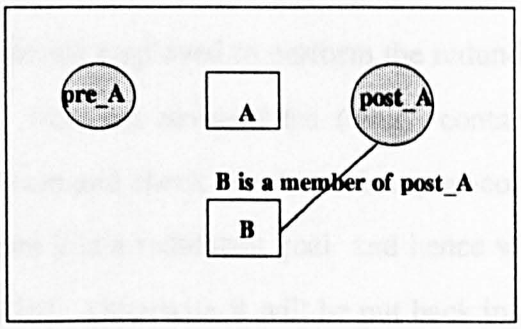

**Figure 5.5 Redundant Goal Case**

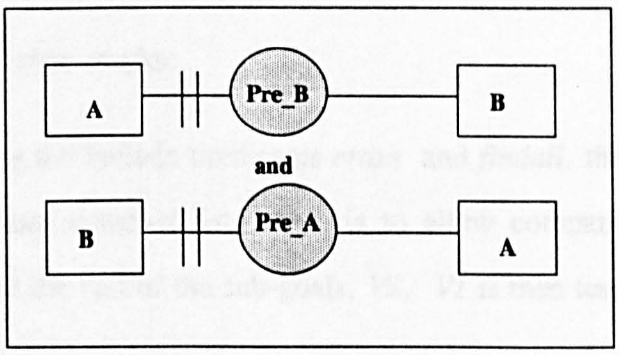

**Figure 5.6 Conflicting Goals Case**

The above mentioned checks are made by the *redund_critic* predicate which is written in Prolog as follows:

```
redund_critic([V1 | V2],Conf):-
        erase(newgoal,V1),
        findall(X,recorded(newgoal,X,_),V),
        test(V1,V,Conf,Redund),
        ((var(Conf),
          redund_critic(V2,Conf));
        !).

redund_critic([],Conf):-!.
```

The approach employed to perform the redundant check is to examine each sub goal from the *newgoal* list (which contains the list of goals for sequencing) in turn and check whether it is a post-condition of the other goal states. If so then it is a redundant goal and hence will not be recorded back to the *newgoal* list. Otherwise it will be put back into newgoal *list.* This is done by using the *test* predicate.

**How redund_critic works:**

By using the built-in predicates *erase* and *findall,* the head of the list is taken out from *newgoal* list. This is to allow comparison to be made between *V1* and the rest of the sub-goals, *V2*. *V1* is then tested with *V*

which contains the goal list (minus the head) for the above mentioned redundant and conflict goals situations. If *V1* and *V* is found to be a pair of conflicting goals then this check is terminated (with success) by the cut and then returns to the top level goal of the sequencing procedure where an appropriate message is displayed and the planner is halted. Otherwise, the *redund_critic* check is continued with the tail of the goal list, i.e. *V2*. Eventually the whole list is exhausted and the goal, *redund_critic* succeeds when the empty list is reached. This is satisfied by the call to the second *redund_critic* clause.

The *test* clauses used in redund_critic are defined in Prolog as follow:

```
test(v1,[],_,Redund):-
        var(Redund),
        recordz(newgoal,V1,_),
        !.

test(V,[],_,redundant):-!.

test(V1,[V2 | V3],Conflict,Redund):-
        cond(V1,V_pre,V_post),
        cond(V2,W_pre,W_post),
        conflict_check(V1,V_pre,V_post,W_pre,W_post,Conflict),
        ((var(Conflict),
          ((member(V1,W_post),
                test(V1,V3,Conflict,redundant)
           );
           ( test(V1,V3,Conflict,Redund)
           )));
        !).
```

The first *test* clause succeeds when the second term, i.e. the list to be compared with is empty. This indicates that the *test* check is complete and hence the goal, *V1* is recorded back onto the newgoal list if it is not a redundant goal (which is indicated by the uninstantiated variable, *Redund*). The second clause is also for when the test is completed but this time there is a redundant goal and hence the goal is not recorded back onto newgoal list. The last *test* clause means that if the *Conf* variable becomes instantiated by the *conflict_check* predicate then the cut is used to end this test. Otherwise a test is made for redundant goals using the *member* predicate. If the *member* clause succeeds then, the variable gets instantiated to *redund* and *test* continues with the remaining of the second list, i.e *V3*. This is to ensure that any underlying conflicts with *V3* can be detected. *Test* is also continued if the *member* clause fails, i.e. no redundant goal is found and hence the variable, *Redund* remains uninstantiated.

The *conflict_check* clause is written as:

```
conflict_check(V1,Pre1,Pre2,Post1, Post2,Conflict):-
        intersect(Pre1,Pre2,W),
          filter(W,W1),
          ((( W1 = =[]);
              subset(W1,Post1);
              subset(W1,Post2)
            ),
            ( not(member(V1,Pre2));
              member(V1,Post2)
            ),
           !);
          Conflict = conflict).
```

In the above clause, *filter* is used to remove any *clear(nil,nil)* states from the list which are generated if there are no reaction faces in that particular sub-assembly. An alternative to this could be to exclude these two arguments from the assembly description altogether. Despite increased programming effort, the *nil,nil* arguments were included to achieve a uniform format for the assembly state descriptions .

TWEAK, being a theoretical planner will attempt to achieve all goals. However in practice, it is necessary to detect conflicting goals before hand in order to prevent the planner from attempting to achieve impossible goals. Further, the theoretical implications of TWEAK would have sorted out the redundant sub-goals eventually but from a practical point of view, this check is needed so that when comparing the final action list against the ordered goal list, a direct translation would be possible. This can be illustrated by an example where there are say, five sub-goals, *a,b,c,d* and *e* that have been input to the planner and *e* happens to be a redundant goal, i.e. a post-condition of say, *d*. Figure 5.7 illustrates the difference between ordered list and action list that would be produced by AAP and TWEAK.

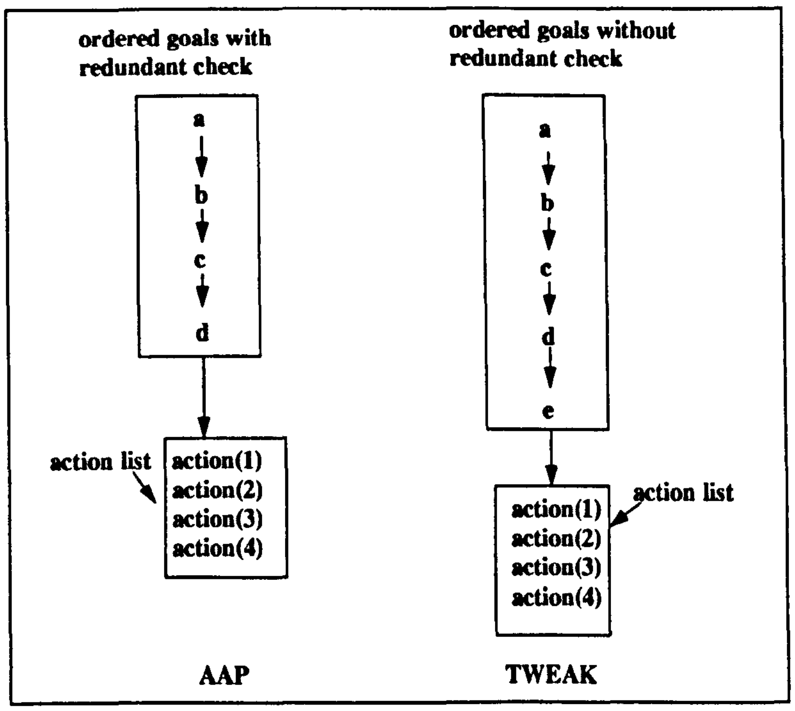

Figure 5.7 Difference Between AAP and TWEAK

If redundant sub-goals were not eliminated from the input list, they would also be ordered by the planner. In the above figure, sub-goal, *e* is a redundant goal so it is eliminated by AAP. This means that it would be easier to interpret the action list when comparing it against the ordered list in contrast with the ordered list on the right. From the output on the right hand side, it seems as though an action had been omitted. In addition, with the redundant check, it may be possible to reduce over specification of sub-goal states for an assembly.

Figure 5.8 shows an example of the primary shaft sub-assmebly, which is another sub-assembly of the gear-box.

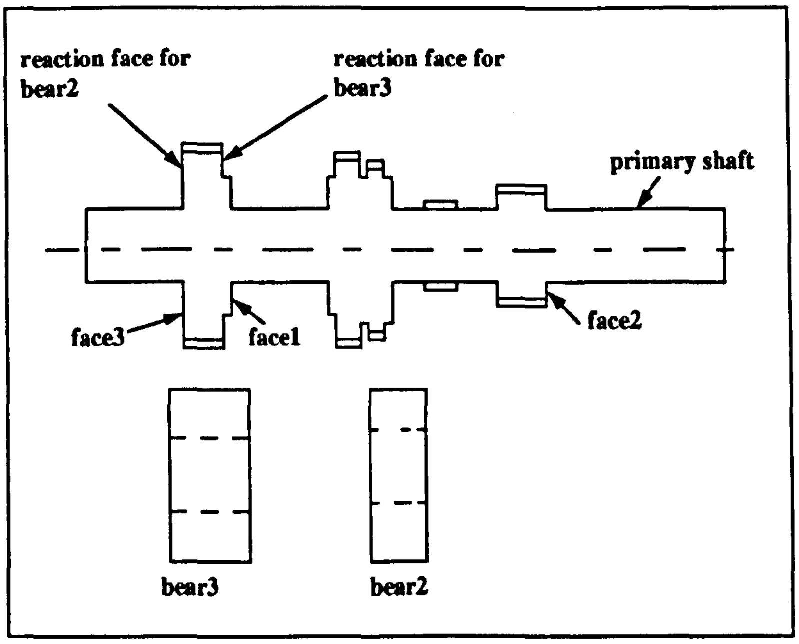

Figure 5.8 Primary Shaft Sub-Assembly

In this particular sub-assembly, when pressing the bearings onto the primary shaft, the respective reaction faces as indicated in Figure 5.8 are used. This imposes a constraint in that *bear2* has to be pressed onto the primary shaft first before *bear3* can be assembled. The goal state is illustrated in Figure 5.9 and it can be written as:

*assemble(bear2,whole,shaft,face2,shaft,face3),*

*assemble(bear3,whole,shaft,face3,shaft,face1).*

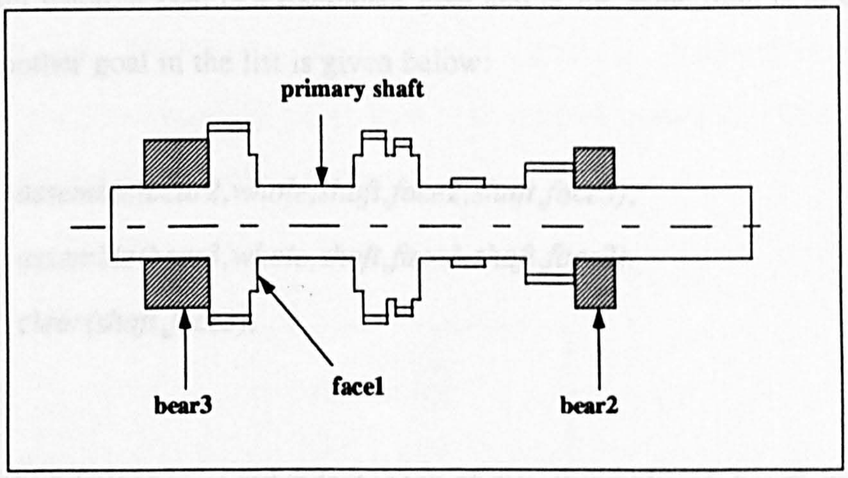

**Figure 5.9 Goal State of Primary Shaft Sub-Assembly**

As seen from the above figure, *face1* of the *shaft* is still clear after *bear3* has been assembled. Hence *clear(shaft,face1)* is a redundant goal in this example. An example of a conflict situation is if the goal state is input as follows:

*assemble(bear2,whole,shaft,face2,shaft,face3) and*

*assemble(bear3,whole,shaft,face3,shaft,face2).*

This is because if *bear2* were to be assembled onto *face2* first then *bear3* cannot be assembled because the reaction face it requires, i.e. *face2* is not clear. Similarly, if *bear3* were to be assembled first then the reaction face for *bear2*, i.e. *face3* will also not be clear for *bear2* to be assembled. Therefore in this case, the variable, *Conflict* would be instantiated, an error message will be displayed and AAP would terminate. An example of a situation where a goal is a redundant goal and at the same time is in conflict with another goal in the list is given below:

*assemble(bear2,whole,shaft,face2,shaft,face3),*
*assemble(bear3,whole,shaft,face3,shaft,face2).*
*clear(shaft,face3).*

*clear(shaft,face3)* is a redundant goal of the assembly of *bear2* but is in conflict with the assembly of *bear3*. Hence the above goal list is a conflicting goals type of situation.

After the successful completion of *redund_critic* check, all the goal states which have been accumulated in *newgoal* are then used in the next stage of the sequencer, i.e. the addition goal.

### 5.4.2.2 Generation Of Star List

**Definition:**

When a pre-condition, *P*, of a goal state, *S*, does not exist in the initial world state and there is no other goal states in the input list that will assert *P* before *S*, *P* will be defined as a star goal and has to be inserted before *S*, i.e. an action step may have to be taken to establish *P* for the goal, *S*, to be true. In addition, when there is a goal, *C* , before *S* which will *clobber* the pre-conditions of *P*, then *P* is inserted before *C*. This can be represented by the following figures.

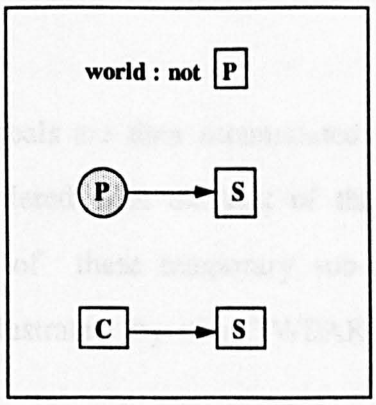

**Figure 5.10 Initial State**

In the above figure, both *C* and *P* contributes to the truth of *S* where *P* is not true in the world state. *P* is then added as a member of star list. Figure 5.11 indicates that *C* clobbers (represented by the '||') *Q* which happens to be the pre-condition of *P*. Thus the final order of these goals is as shown in Figure 5.12.

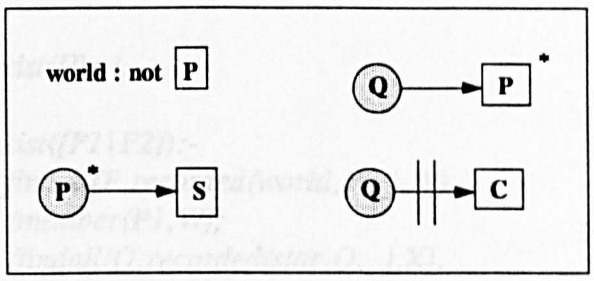

Figure 5.11 C Clobbers Q

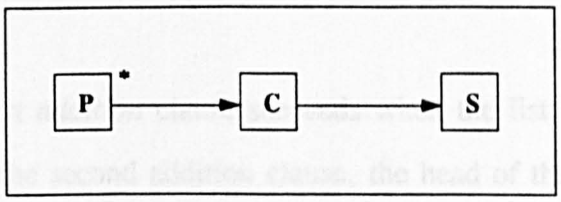

Figure 5.12 A Final Plan

All the *star* goals are then accumulated in a list called the *star* list which will be re-ordered with the rest of the sub-goals later on in the planner[7]. Addition of these temporary sub-goals may be necessary in certain cases as illustrated by the TWEAK planner using Sussman's example[58].

This step is achieved by the following Prolog code:

```
addition([]):- !.

addition([V1 | V2]):-
        cond(V1,Pre1,_),
        checkexist(Pre1),
        addition(V2).
```

[7] This is similar to the idea of promoting a sub-goal in TWEAK.

```
checkexist([]):-!.

checkexist([P1|P2]):-
        findall(P,recorded(world,P,_),W),
        (member(P1,W);
        (findall(Q,recorded(star,Q,_),X),
         (member(P1,X);
          recordz(star,P1,_)))),
        checkexist(P2).
```

The first *addition* clause succeeds when the list containing the goals is empty. In the second addition clause, the head of the list, *V1* is checked first followed by the remaining goals in the tail of the list, *V2*. The pre-conditions of *V1* is obtained by calling the *cond* goal (see section 5.5 for explanations on *cond*). Once this has been achieved, *checkexist* is called to find if the pre-conditions are true in the *world* database. If they are not true then they are recorded as a *star* goal into the *star* database provided that they have not been previously recorded. Again the individual goals in the pre-conditions list are examined by taking out the head, i.e. *P1* followed by the remaining elements in the tail, *P2*. This goal should also succeed when all the elements in *P2* have been examined and hence the first *checkexist* clause.

In the piston-connecting rod sub-assembly, the *pin* is inserted into *t_hole* of the *piston* in the initial state. This will be detected by the *addition* clause which will insert the following goals as temporary goals into the *star* list:

```
clear(pin,whole).
clear(piston,t_hole).
```

The feature, *t_hole* has to be cleared for the assembly of the connecting rod, so *clear(piston,t_hole)* is inserted as a *star* goal. In order to clear *t_hole* of the piston, the pin must not be there. Hence the whole of the pin must also be cleared. These temporary goals will be examined by the *insert_star* clauses for insertion into the main goal list (which are the permanent goals, i.e. must be true after achieving the assembly states). In the planner, certain conditions may arise where *star* goals are created but may not be necessarily inserted into the main goal list as seen in the piston connecting rod sub-assembly. This will be elaborated in 5.4.2.4.

### 5.4.2.3 Ordering Of The Goal States

The ordering of the goals is done using a simple *sorting* procedure. Figure 5.13 shows the possible sequencing outcomes for two goals, *A* and *B*.

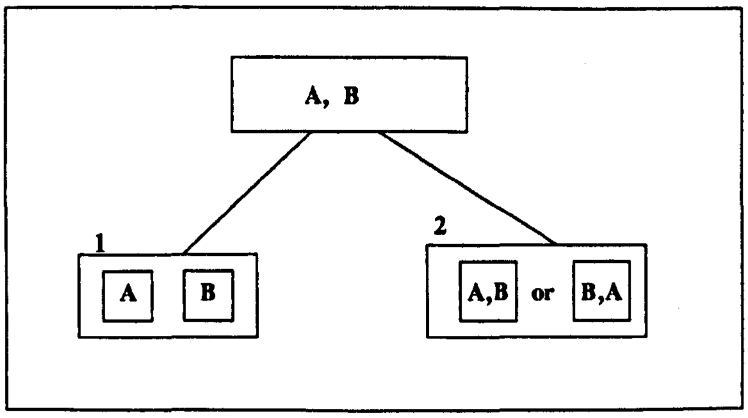

**Figure 5.13 Possible Outcomes For Goals, A & B**

From Figure 5.13 it can be seen that there are two possible achievable outcomes when ordering the goal states. The first one is when *A* and *B* do not interfere with each other and can be achieved in either order, i.e. 1 in Figure 5.13. In this case the sorted goal list is of the format, *par(A,B)* to represent the fact that goal *A* and goal *B* can be executed in parallel. The second outcome is when one goal state must be achieved before the other goal state. The format for this type of goals is *seq(A,B)* which means that the sequence of operation is to execute *A* followed by *B* [59].

For cases with more than two goals, nesting may occur. For example, for 3 goals *A, B, C,* the outcome may for example look like: *par(seq(A,B),C)*. In this case, goal *C* is parallel to the *compound* goal *seq(A,B)* in which goal *A* must be achieved before goal *B*.

The *sorting* procedure is to determine the ordering of the goal states to be achieved and link them accordingly to form a *currentgoal* list in the format as described above, e.g. *seq(A,B)*, etc. This procedure is written in Prolog as follows:

```
sorting:-
        findall(N,recorded(newgoal,N,_),P),
        P \== [],
        erase_all(toplist),
        order_critic((P),
        findall(L,recorded(toplist,L,_),HL),
        length(HL,Length),
        findall(C1,recorded(currentgoal,C1,_),C),
        ((  C \== [],
            linking(HL,C),
            erase_list(checklist,HL),
            findall(TL,recorded(temp_checklist,TL,_),TL1),
            record_list(checklist,TL1),
            erase_list(temp_checklist,TL1),
             record_list(temp_checklist,HL));
            ((Length = 1,
               [HL1] = H,
               HL2 =..[seq,HL1],
               recordz(currentgoal,HL2,_));
                (HL1 =..[par|HL],
                  recordz(currentgoal,HL1,_)))),
                record_list(temp_checklist,HL),
                erase_list(newgoal,HL),
                sorting.

sorting:-!.
```

In the sorting procedure, *toplist* is a buffer area to store the goal states which can be achieved first to allow the linking procedure to link these goals to the *currentgoal* list accordingly. In each pass *toplist* will be reset to an empty list before the procedure *order_critic* is called. The task of the *order_critic* clauses is to fill the *toplist* in each pass. Since the *currentgoal* list is empty to start with, there are only two possible cases to record the goals from *toplist* into *currentgoal.* The first one is if *toplist* contains only one goal, say goal *a.* In this case, *Currentgoal* will become *seq(a).* Obviously, any goals in the *toplist* in the next pass must be done after a. Hence the goal must be a sequential one. If *toplist* contains more than one goal, say *a* and *c*. *Currentgoal* will then become *par(a,c)* because goals *a* and *c* can be done in parallel. For subsequent passes the goal states in *toplist* will be linked to the goals of *currentgoal* list by calling the procedure *linking*. Eventually, the complete *currentgoal* list will be built up.

## a) Order_Critic

The Prolog code for the *order_critic* clauses are shown below:

```
order_critic([]):-!.

order_critic([N1|N2]):-
        erase(newgoal,N1),
        findall(N,recorded(newgoal,N,_),P),
        ordertest(N1,P,Flag_of_N1),
        (( var(Flag_of_N1),
           recordz(toplist,N1,_));
           !),
         recordz(newgoal,N1,_),
        order_critic(N2).


ordertest(_,[],_):-!.

ordertest(N,[N1|N2],Flag_of_N):-
        cond(N,Pre_N,Post_N),
        cond(N1,Pre_N1,Post_N1),
        intersect(Pre_N,Pre_N1,W),
        filter(W,W1),
        ((W1 == [],
           ordertest(N,N2,Flag_of_N));
          (subset(W1,Post_N1),
        Flag_of_N = tail);
        ordertest(N,N2,Flag_of_N)).
```

The definition for *order_critic* is as follows:

For two goals, *A* and *B* in which *A* is to be achieved before *B*. When *A* is compared with *B*, *A* can be achieved first and is hence recorded into the *toplist*. When *B* is compared with *A*, *B* cannot be achieved first and hence is omitted from the *toplist*.

The first *order_critic* clause succeeds when the *newgoal* list becomes empty, i.e indicating the end of this test. Using the same technique as the previous clauses (i.e. *redund_critic*, etc) if the list is not empty, the goal to be compared with the rest of the goals from the *newgoal* list is erased before a variable list, *P* is instantiated to the *newgoal* list (this is to avoid comparing the goal with itself). *Order_test* is then called which will indicate whether the goal can be achieved first. If so the variable, *Flag_of_N1* will remain uninstantiated and that goal is recorded onto the *toplist*. It is then put back to *newgoal* before calling the procedure *Order_critic* recursively to check the rest of the members in the list, i.e. *N2*.

As in previous clauses, the first *ordertest* clause is used to detect the end of the check, i.e. when the list to be compared with is empty. In this case the test succeeds with a cut. When the list is not empty, the pre and post conditions of the goal, *N* and the first member of the list, *N1* to be compared with are found using the *cond* predicate. If the intersection between the pre-conditions of *N* and the pre-conditions of *N1* is not empty then the members contained in this intersection set are investigated to determine whether they are a subset of the post-conditions of *N1*. If this is the case, then *N* is not a member of *toplist*. In the program *tail* is used to indicate if it is not a *toplist* member, in other words *Flag_of_N* gets instantiated to *tail*. The other case covers the situation where the intersection set is empty, suggesting that N and N1 do not clobber one another and *ordertest* is continued with tail, i.e. remaining elements of *N2*.

Once the *toplist* has been set up for subsequent passes, the *linking* procedure is called to synthesize the ordered goal list into the required format. Again before *linking* is called the members of the both *toplist* and *currentgoal* are obtained and *placed in HL* and *C* respectively.

## b). Linking

The *Linking* clauses are as follows:

```
linking([HL1|HL2],[C]):-
        C=..[F|_],
        link(HL1,C,Newargs,Linked),
        Linked == true,
        New_c=..[F|Newargs],
        recordz(checklist,HL1),
        linking(HL2,[New_c]).

linking([],[C]):-
        erase_all(currentgoal),
        recordz(currentgoal,C,_),
        !.
```

*Linking* is used to link goals in *toplist* to goals in the *currentgoal* list so that the correct ordered goal format can be synthesized. A variable, *Linked* is instantiated to true if the goals are linked. The variable, *Newargs* will contain the resulting goals which, when joined with the functor *F*, becomes the new *currentgoal*, if *link* is successful. Both variables, *Linked* and *Newargs* are determined by the *link* procedure. There are four cases of *currentgoal* formats that the *link* procedure has to consider, i.e.

case 1: *seq(Arg1)*

```
case 2: par(Arg1)
case3: seq(Arg1,....)
case4: par(Arg1,....)
```

where *Arg1* can either be a single or compound goal.

The Prolog code written for case 1 is as follows:

```
link(HL1,C,Newargs,Linked):-
        C=..[F,Arg1],
        F==seq,
        ((check_single_goal(Arg1),
          ((findall(Arg1,recorded(checklist,Arg1,_),[]),
            sequence_test1(HL1,Arg1),
           Newargs = [Arg1,HL1],
           Linked = true,
        !);
        Linked = false));
        (link(HL1,Arg1,Newarg1,Linked),
         Arg1=..[F1|_],
         Newarg2=..[F1|Newarg1],
         Newargs = [Newarg2])).
```

In the first instance, the currentgoal, *C* is decomposed into its functor (i.e. seq) followed by the rest of the goals. The built in predicate, =.. is used for examining structures, e.g. if *C* is *seq(a)* then *Arg1* will be *a*. The *check_single_goal* clause is used to detect compound goals. If it succeeds then *Arg1* is a single goal. *Checklist* is a buffer for storing goals that have already been linked in the first pass. This is to avoid comparing HL1 with goals from the same toplist as these goals must be parallel in the first place. If *checklist* does not contain *Arg1* then the *link* procedure proceeds with *Sequence_test1*. This is done to check if HL1 must be linked to Arg1. *Sequence_test1* succeeds if the intersection between the pre and post conditions of *HL1* and *Arg1* is not empty and that this set is a subset of the post conditions of *Arg1*. This suggests that *Arg1* is before *HL1* as illustrated in Figure 5.14 where the goal *a* has contributed to the success of *sequence_test1*. In this example, since it started with *seq(Arg1)*, then the outcome of linking *HL1* to *Arg1* must be *seq(Arg1,HL1)* and *Newargs* will be instantiated to *[Arg1,HL1]*.

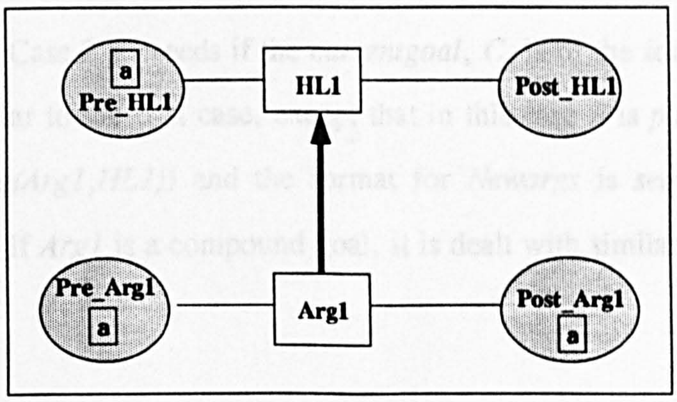

**Figure 5.14 A Sequence**

If *sequence_test1* succeeds then the *Linked* variable will be instantiated to *true* otherwise it will be marked as *false*. If *Arg1* is a compound goal then the *link* procedure is called again to attempt to link *HL1* to *Arg1* to form *Newarg1*.

The code for Case 2 is as follows:

```
Case 2:
link(HL1,C,Newargs,Linked):-
        C=..[F,Arg1],
        F==par,
        ((check_single_goal(Arg1),
((findall(Arg1,recorded(checklist,Arg1,_),[]),
  sequence_test1(HL1,Arg1),
  Newarg2=..[seq,Arg1,HL1],
  Newargs = [Newarg2],
  Linked = true);
 Linked = false));
(link(HL1,Arg1,Newarg1,Linked),
 Arg1=..[F1|_],
 Newarg2=..[F1|Newarg1],
 Newargs = [Newarg2])).
```

Case 2 succeeds if the *currentgoal, C* is of the format: *par(Arg1)*. It is similar to the first case, except that in this case *F* is *par*. The outcome is *par(seq(Arg1,HL1))* and the format for *Newargs* is *seq(Arg1,HL1)* in this case. If *Arg1* is a compound goal, it is dealt with similarly as in case 1.

For Case 3, *currentgoal* contains a sequence of goals with *F* being instantiated to *seq*, i.e. with C being of the form: *seq([Arg1| Args])* where *Args* is the tail of the goal list. Hence if *HL1* is successfully linked to *Arg1* where *Arg1* is a single goal, it implies that *Args* must be either a single or a compound goal. Hence there are two ways in which the *currentgoal* may be formatted :

i) where *Args* is a single goal, a. In this case the format of the goal is of the form: *seq(Arg1,par(a,HL1))*;

ii) where *Args* is a compound goal, *par(a,b)*. In this case the format of the goal is of the form : *seq(Arg1,par(a,b,HL1))*. This is because if *HL1* is after *Arg1* then it must be in parallel with what comes after *Arg1* and is hence inserted into the parallel list of goals.

If *Arg1* is a compound goal, it is dealt with similarly as in case 1 and case 2. Finally, if it fails to link *HL1* to *Arg1*, it will attempt to link *HL1* to the tail, *Args* by calling *link* again.

The Prolog code for case 3 is as follows:

```
Case 3:
link(HL1,C,Newargs,Linked):-
        C=..[F|[Arg1|Args]],
        F==seq,
        ((check_single_goal(Arg1),
         ((findall(Arg1,recorded(checklist,Arg1,_),[]),
           sequence_test1(HL1,Arg1),
           Args = [_|[]],
          ((check_single_goal(Args),
            Newarg2=..[par,Args,HL1]);
            (Args=..[par|Temparg2],
             append(Temparg2,HL1,Newtemparg2),
              Newarg2=..[par|Newtemparg2])),
            Newargs=[Arg1,Newarg2],
            Linked = true);
            (C1=..[F|Args],
            link(HL1,C1,Newarg1,Linked),
           Newargs=[Arg1|Newarg1])));
         (link(HL1,Arg1,Newarg1,Linked),
         (Linked = true,
           Newargs = [Newarg1|Args]);
           (C1=..[F|Args],
            link(HL1,C1,Newarg2,Linked),
           Newargs = [Arg1|Newarg2]))).
```

In the fourth case, *C* is of the format: *par([Arg1|Args])* where *Args* is the tail of the goal list. If *HL1* is successfully linked to *Arg1*, where *Arg1* is a single goal, the format for the *currentgoal* is of the form: *par(seq(Arg1,HL1),Args)*. For the situations when *Arg1* is a compound goal or when it fails to link *HL1* to *Arg1*, it is also dealt with in the same way as in case 3 described above.

The Prolog code for Case 4 is as follows:

```
Case 4:
link(HL1,C,Newargs,Linked):-
                C=..[F|[Arg1|Args]],
                F==par,
                ((check_single_goal(Arg1),
                ((findall(Arg1,recorded(checklist,Arg1,_),[]),
                sequence_test1(HL1,Arg1),
                Newarg1=..[seq,Arg1,HL1],
                Newargs=[Newarg1|Args],
                Linked = true);
                (C1=..[F|Args],
                link(HL1,C1,Newarg1,Linked),
                Newargs=[Arg1|Newarg1])));
                (link(HL1,Arg1,Newarg1,Linked),
                (Linked = true,
                Newargs = [Newarg1|Args]);
                (C1=..[F|Args],
                 link(HL1,C1,Newarg2,Linked),
                Newargs = [Arg1|Newarg2]))).
```

In the *linking* procedure, once the *link* procedure succeeds, *HL1* is placed into the *checklist* database and the *linking* procedure is then repeated with the tail of the *toplist,* i.e. *HL2* to the updated current goal, *New_c*. Again the end of this procedure occurs when *toplist* is empty, i.e. no more goals to be linked to the *currentgoal* list. The final result, *C* which contains the ordered goal list is then recorded back into the *currentgoal* database, as done by the second *linking* clause.

So far the *sorting* procedure is described for when the *currentgoal* list is not empty. If it is empty, (this means that there is nothing to link) and also when the *toplist* consists of only one goal then in this case it will be recorded into *currentgoal* list as *seq(goal)*. Otherwise if there is more than one goal in *toplist* then the currentgoal format is *par(goal,...)*. The *toplist* goal(s) that the procedure has worked with so far is recorded into *temp_checklist* to keep a record of the *toplist* goals that have already been examined. At the same time the *toplist* goals are also deleted from the *newgoal* list in preparation for the next pass where the *sorting* procedure is called again. Eventually, the *sorting* procedure is terminated (or succeeds) when the *newgoal* list (i.e. *P*) is empty. This is succeeded by the second *sorting* clause with the cut.

To illustrate the sorting procedure, suppose there are 4 goals, *C,A,B,D* in *newgoal* list with the final ordered goal being *par(seq(A,B),seq(C,D))*. The following steps will summarise how it works:

i) The findall predicate will instantiate P to [C,A,B,D].

ii) erase_all toplist will reset toplist to [].

iii) Call order_critic([C,A,B,D] and produce a toplist = [A,C].

iv) Currentgoal = par(A,C) because currentgoal is empty to start with and since more than one goal in toplist then they must be parallel goals.

v) Record A,C into temp_checklist.

vi) Erase A,C from newgoal list. So newgoal list is now [B,D].

vii) Call sorting again, with P=[B,D] and toplist =[].

viii) After order_critic, toplist becomes [B,D].

ix) Call linking which links [B,D] to par(A,C).

x) Finally, resulting currentgoal is par(seq(A,B),seq(C,D)).

After the successful completion of the *sorting* procedure the next task remaining of the sequencing stage is to insert the goals in *star* list (if any) to the main goal list, if necessary. This is done by the *insert_star* clause as described in the following section.

## 5.4.2.4 Insertion of Star Goals to Main Goal List

**Definition:**

Any sub-goal in the *star* list is inserted before a goal in the main goal list if in achieving that goal it will *cancel* the *star* sub-goal, unless the *star* sub-goal is one of the pre-conditions of that goal.

The clauses for performing this are given below:

```
insert_star([],C):-
        erase_all(currentgoal),
        recordz(currentgoal,C,_),
        !.
```

```
insert_star([S1|S2],C):-
        C=..[F|_],
        reorder(S1,C,Newargs,_),
        Current_goals=..[F|Newargs],
        insert_star(S2,Current_goals).
```

The first *insert_star* clause succeeds when the *star* list is empty, i.e no goals to insert. In this case, the original *currentgoal* list is removed and the updated *currentgoal* list (which could be the same as the old one, i.e. when there is no star list in the first instance) is recorded back. In the second clause, the star list is divided into head and tail, (i.e. *S1* and *S2* respectively) in order to insert one *star* goal at a time to the *currentgoal, C*. The functor, *F* of currentgoal is identified. By calling *reorder*, a star goal, *S1* is examined with currentgoal, *C* to form an argument called *Newargs*. By joining it back to the original functor, *F* the *currentgoal* is updated. *Insert_star* is then called recursively with the tail of the star goals, *S2* and the updated currentgoals until the *star* list is exhausted. Then the first *insert_star* clause will succeed by recording the updated currentgoal into the *currentgoal* database. The *reorder* clauses are used to insert the *star* goals into their appropriate places in the *currentgoal, C*. This is done by comparing *S1* with each goal in *C* in turn. Similar to the *sorting* procedure, there are four different formats of the *currentgoal* list as shown below:

| | |
|---|---|
| Case 1a. | *seq(a)or par(a)* |
| Case 1b. | *seq(par(a,b)) or par(seq(a,b)).* |
| Case 2a. | *seq(a,b,...) or par(a,b,...)* |
| Case 2b. | *seq(par(a,b),...) or par(seq(a,b),...),* |

Case 1a. is for single goals or the last or remaining goal in the *currentgoal* list. e.g. *seq(assemble(rod,whole,piston,axial_hole,nil,nil).* The *reorder* clause for this is given below:

<u>Case 1a.</u>

```
reorder(S,C,Newargs,Linked):-
        C=..[F|Arg1|[]],
        check_single_goal(Arg1),
        cond(Arg1,Pre_arg1,_),
        ((member(S,Pre_arg1),
          Newargs=[Arg1],
          Linked = true,
         !);
        ( sequence_test2(Pre_arg1,S),
          (( F == seq,
             Newargs = [S,Arg1]);
           ( F == par,
             Newargs = [seq(S,Arg1)])),
          Linked = true,
          !);
         ( Newargs = [Arg1],!)).
```

The first two sub-goals of reorder will succeed when C contains only one single goal, Arg1 (e.g. C = par(a) or seq(a)). A check is then made to see if the star goal, S is a pre-condition of Arg1. If it is, no insertion before Arg1 is necessary. This is because when an action is to be planned against the goal, *Arg1*, all its pre-conditions will be examined and subsequently promoted if any one of them is not yet true. Otherwise, the star goal, *S* will be compared with *Arg1* to see if *Arg1* will clobber *S*. If this is the case then the star goal, *S* has to be inserted before *Arg1*. The format will be either *seq(S,Arg1)* or *par(seq(S,Arg1))* depending on whether it starts with *seq(Arg1)* or *par(Arg1)*.

The clause for Case 1b is given below:

<u>Case 1b:</u>

```
reorder(S,C,Newargs,Linked):-
        C=..[_|[Arg1|[]]],
        Arg1=..[F1|_],
        reorder(S.Arg1,Newarg1,Linked),
        Newarg2=..[F1|Newarg1],
        Newargs = [Newarg2].
```

In case 1b, *Arg1* is a compound goal. Hence *star* goal, *S* cannot be checked directly against *Arg1* using *sequence_test2*. Instead the functor of *Arg1 (F1)* is identified, and *reorder* is called with *S* and *Arg1*. The argument, *Newarg1* that is returned will be joined back to *F1* before it is returned as *Newargs*.

The clause for the next case, case 2a is:

```
reorder(S,C,Newargs,Linked):-
        C=..[F|[Arg1|Args]],
        check_single_goal(Arg1),
        cond(Arg1,Pre_arg1,_),
        (( member(S,Pre_arg1),
           C=..[_|Newargs],
           Linked = true,
           !
           );
        (sequence_test2(Pre_arg1,S),
        ((F == seq,
          C=..[_|Newarg1],
          append([seq(S,Arg1)],Args,Newargs))),
        Linked = true,
!);
(C1=..[F|Args],
 reorder(S,C1,Newarg1,Linked),
 append([Arg1],Newarg1,Newargs),
 !)).
```

In case 2a, *currentgoal*, *C* contains more than one argument and the head of the list, *Arg1* is a single goal. In this case, *star* goal, *S* can be checked with *Arg1* directly as in case 1a. If *S* is one of the pre-conditions of *Arg1*, *Newargs* is kept the same as *[Arg1|Args]* and the variable, *Linked* is instantiated. If *S* passes *sequence_test2* procedure, it is linked to *Arg1* as *seq(S,Arg1)* to which the tail, *Args* is appended to form *Newargs*. If neither of these cases succeeds, *star* goal, *S* is to be checked against the tail, *Args*. This is done by joining the functor, *F* to *Args* to form *C1* so that *Arg1* is omitted before *reorder* is called again. The returned argument, *Newarg1* is appended to *Arg1* to form *Newargs* as in the previous case.

The clause for 2b is given below :

<u>Case 2b:</u>

```
reorder(S,C,Newargs,Linked):-
        C=..[F|[Arg1|Args]],
        Arg1=..[F1|_],
        ((reorder(S,Arg1,Newarg1,Linked),
          Newarg2=..[F1|Newarg1],
          append([Newarg2],Args,Newargs),
         not(var(Linked))),
        !);
        (C1=..[F|Args],
         reorder(S,C1,Newarg1,Linked),
         append([Arg1],Newarg1,Newargs),
```

*!)).*

Case 2b is similar to case 2a except that the head of the argument, *Arg1* is a compound goal. Like case 1b, the functor of *Arg1* is identified as *F1* and *reorder* is called with the *star* goal, *S* and *Arg1*. The returned argument, *Newarg1* is then joined back to the functor, *F1* and the tail, *Args* is appended to form *Newargs*. In order find out whether *S* is successfully linked to *Arg1*, the variable *Linked* is checked. If *Linked* is not instantiated (which indicates that *S* is not linked to *Arg1*), *S* has to be checked against the tail, *Args* as in case 2a.

When the *insert_star* goal succeeds, the final version of the *currentgoal* is retrieved from the *currentgoal* database and is written onto the screen. For example, if the input to the planner is:

*assemble(bear2,whole,shaft,face2,shaft,face3).*
*assemble(bear3,whole,shaft,face3,shaft,face1).*

The output of the sequencer will be:

*The Ordered Goal list is:*

*seq( assemble(bear2,whole,shaft,face2,shaft,face3),*
*assemble(bear3,whole,shaft,face3,shaft,face1).*

This information, i.e. the ordered goal list is then passed onto the next stage of the planner which produces an action list corresponding to the ordered goal list.

## 5.4.3 Producing the Action List

Once the goals in the input list have been ordered, the remaining task is quite straight forward. This is done by the *do* predicate. Based on the original idea of the test planner the clauses for producing the action list is as follow:

```
do([Y]):-
        Y =..[F,Arg],
        (F == seq;
         F == par),
        ((check_single_goal(Arg1),
          state(Arg1));
         do([Arg1])).

do([Y]):-
        Y=..[F|[Arg1|Args]],
        (F==seq;
         F==par),
        C1=..[F,Arg1],
        do([C1]),
        Cs=..[F|Args],
        do([Cs]).
```

An assumption made in *do* is that once the goals have been ordered then the action list produced will be according to the way in which the goals have been ordered irrespective of whether the goals are *seq* or *par* goals.

The argument for this is that if there is a *seq* goal then the ordering of the action list is as it is on the *currentgoal* list and so the action list is correct. However if it is a *par* goal (which indicates that the sub-goals can be done in parallel without affecting one another), except that the output format of the action is that the one that comes first in the ordered goal list will be done first. Since the plan is originally concerned with manual assembly and the manual assembly worker is only able to handle one task at a time so this interpretation is correct.

Hence in the first *do* clause, *seq* and *par* are extracted from the currentgoal, *Y* in order to obtain the original assembly goal states. These are then passed onto *state* which works in the same way as the *state* arguments of the test planner. The approach in the second *do* clause is to decompose the *currentgoal* list into head and tail until one goal state can be dealt with at a time. Hence in the second *do* clause when *Y* is passed in as a list, it is divided into a head, a tail and the functor, *F*. *F* is then attached back to the head, *Arg1* , i.e *seq(Arg1)* or *par(Arg1)*. When the *do* procedure is called recursively, this goal list with only one single argument is passed into the first *do* clause. In the first *do* clause, it will check if the argument is a single goal (i.e. consisting of one goal state only). If so, it will attempt to achieve that single goal state. If it is a compound goal, it will call the *do* procedure again which will further break down the compound goal until it can be dealt with. In the second *do* clause, after *C1* is achieved, *F* is attached back to the tail, *Args* as *Cs* and the *do* procedure is called again.

As mentioned earlier, two assembly states have been defined, i.e. *assemble* and *clear* where the associated actions are *press* and *remove* respectively. This definition is only a very rough one and in real situations further specific assembly states such as screw may have to be defined as well. However for the purpose of demonstrating the planner, the suggested assembly states and actions are found to be adequate. The clauses for the *assemble* state are given below:

```
state(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)):-
        findall(W,recorded(world,W,_),W1),
        member(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),W1).

state(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)):-
        result(press(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)).
```

Recall that the *state* clauses are also known as the post-conditions in the test planner. The first assemble *state* succeeds if there exists such a state already in the current state as indicated in the *world* database. If the first *state* clause fails then the second *state* clause is used in which the *press* action is suggested if it succeeds.

Similarly, there are two cases for the *clear* state (which is the opposite state of assemble) and they are as follow:

**Case 1:**

```
state(clear(Obj,Face)):-
        findall(W,recorded(world,W,_),W1),
        member(clear(Obj,Face),W1).
```

Case 2:

```
state(clear(Obj,Face)):-
        result(remove(Obj,Face,_,_)).
```

The first *state* clause for *clear* is true when it already exists in the current *world* state and hence no actions need to be done to achieve this state. In case 2, the *clear* state is achieved by simply taking the *remove* action.

There are two *result* clauses, one for the *press* action and the other one for the *remove* action. They are written in Prolog as follow:

```
result(press(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)):-
        cond(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),Pre,Post),
        Pre1=..[par|Pre],
        do([Pre1]),
        maintain_world_pre(Pre,Post),
        maintain_world_post(Post),
        recordz(action,press(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),_).

result(remove(Obj,Face,Obj1,Face1):-
        cond(clear(Obj,Face),Pre,Post),
        Pre1=..[par|Pre],
        do([Pre1]),
        (member(assemble(Obj,Face,Obj1,Face1,_,_),Pre);
         member(assemble(Obj1,Face1,Obj,Face,_,_),Pre)),
        maintain_world_pre(Pre,Post),
        maintain_world_post(Post),
        recordz(action,remove(Obj,Face,Obj1,Face1),_).
```

For the *assemble* goal state that requires a pressing action, as in the first *result* clause, the required pre-conditions and the resulting post-conditions are instantiated. The required pre-conditions are then checked if they already exist in the *world* state or achieved to be true by calling, *do([Pre1])*. Once the call to the pre-conditions succeeds, the *press* action can be taken which is recorded onto the *action* database. At the same time, the *world* state is maintained due to the pressing action by calling *maintain_world_pre* and *maintain_world_post*. Similarly, for a *clear* goal state with a corresponding *remove* action as in the second *result* clause above.

In order to update the relevant changes in the *world* database the following is used:

```
maintain_world_pre([],_):-!.

maintain_world_pre([Pre1|Pre2],Post):-
        (member(Pre1,Post);
                ( (Pre1=..[assemble|Args],
                Args = [Obj,Face,Obj1,Face1,_,_],
        ((recorded(world,assemble(Obj,Face,Obj1,Face1,_,_),Ref),
                erase(Ref));
        (recorded(world,assemble(Obj1,Face1,Obj,Face,_,_),Ref),
                erase(Ref))));
        (recorded(world,Pre1,Ref),
        erase)))),
        maintain_world_pre(Pre2,Post).

maintain_world_post([Post1|Post2]):-
        findall(W,recorded(world,W,_),W1),
        member(Post1,W1),
        maintain_world_post(Post2).
maintain_world_post([Post1 | Post2]):-
        recordz(world,Post1,_),
        maintain_world_post(Post2).

maintain_world_post([]):-!.
```

*maintain_world _pre* is used to erase any states that are deleted as a result of achieving a goal state. The criterion is that any pre-conditions in the *world* state that is not true in the post-conditions, i.e. when the call to *member(Pre1,Post)* fails, is erased. The first check is if the pre-condition is an *assemble* state and the second is for the same *assemble* state but with the second part-feature pair, i.e. *Obj1,Face1* as the first part-feature pair in the clause. This is because the *assemble* state may have been written with *Obj1,Face1* as the first part-feature pair in the *world* database. The third case is if the pre-condition is not an *assemble* state, e.g. a *clear* state. Similarly, *maintain_world_post* is used to add to the *world* states the post-conditions as a result of achieving a goal state. Before a post-condition is added, a check is made first to see if it already exists in the *world* database.

## 5.5 Finding The Pre and Post Conditions

As can be seen from the description of the planner so far, it relies heavily on the examination and manipulation of lists such as the input lists, pre-conditions and post-conditions. A major part of the planner is dependent on finding the preconditions and post-conditions of the relevant goal states in order to perform the tests such as *redund_critic*, *conflict_check*, etc in the planner. The derivation of the pre and post conditions of the goals states are performed by the *cond* clauses. There are three *cond* clauses for the *assemble* state and they are as follow:

```
cond(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),X,Y):-
findall(cross(Cr1,Cr2,Cr3,Cr4),recorded(world,cross(Cr1,Cr2,Cr3,Cr4),_),Cross),
findall(common(Co1,Co2,Co3,Co4,Co5),
        recorded(world,common(Co1,Co2,Co3,Co4,Co5),_),Common),
        ((member(cross(Obj1,Face1,Face,C_area),Cross),
                ((member(common(Obj1,Face,_,_,C_area),Common),
                        X1=clear(nil,nil),
                        Y1=clear(nil,nil));
                        (X1=clear(Obj1,Face),
                        Y1=clear(Obj1,Face))));
                (member(cross(Obj2,Face2,Face,C_area),Cross),
                 ((member(common(Obj2,Face,_,_,C_area),Common),
                        X1=clear(nil,nil),
                        Y1=clear(nil,nil));
                        (X1=clear(Obj2,Face),
                        Y1=clear(Obj2,Face))))),
                X2=[X1,clear(Obj1,Face1),clear(Obj2,Face2),clear(Rxnobj,Rxnface)],
                filter(X2,X),
                Y2=[Y1,assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),
                        clear(Rxnobj,Rxnface)],
                filter(Y2,Y),
                !.


cond(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),X,Y):-
        findall(at(A1,A2,A3),recorded(world,at(A1,A2,A3),_),AT),
        (((member(at(Obj1,Face1,Face11),AT),
                X1 = clear(Obj1,Face11),
                Y1 = clear(Obj1,Face11));
                (X1 = clear(nil,nil),
                Y1 = clear(nil,nil))),
        ((member(at(Obj2,Face2,Face12),AT),
                X2 = clear(Obj2,Face12),
                Y2 = clear(Obj2,Face12));
                (X2 = clear(nil,nil),
                 Y2 = clear(nil,nil))),
        ((member(at(Rxnobj,Rxnface,Face13),AT),
                X3 = clear(Rxnobj,Face13),
                Y3 = clear(Rxnobj,Face13));
                (X3 = clear(nil,nil),
                Y3 = clear(nil,nil))),
          X4 = [X1,X2,X3,clear(Obj1,Face1),clear(Obj2,Face2),clear(Rxnobj,Rxnface)],
        filter(X4,X),
        Y4 = [Y1,Y2,Y3,assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),
                clear(Rxnobj,Rxnface)],
        filter(Y4,Y)),!.
cond(assemble(Obj1,Loc1,Obj2,Loc2,Rxnobj,Rxnface),X,Y):-
   X1=[clear(Obj1,Loc1),clear(Obj2,Loc2),clear(Rxnobj,Rxnface)],
   filter(X1,X),
   Y1=[assemble(Obj1,Loc1,Obj2,Loc2,Rxnobj,Rxnface),clear(Rxnobj,Rxnface)],
   filter(Y1,Y),!.
```

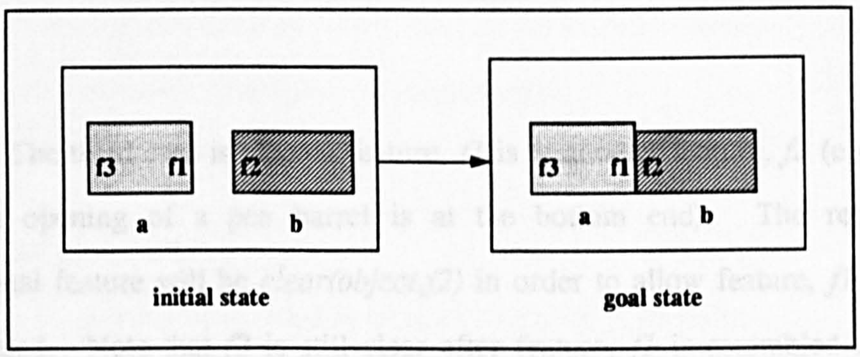

**Figure 5.15 A Diagram of a Simple Assembly**

In order to illustrate the principle of obtaining the pre and post conditions, Figure 5.15 is used. In the above Figure, the goal state is described as: *assemble(a,f1,b,f2,a,f3)* where the required pre-conditions are *clear(a,f1), clear(b,f2) and clear(a,f3)*. The post-conditions are *assemble(a,f1,b,f2,a,f3)* and *clear(a,f3)*. For such an *assembly* state, the relevant pre and post conditions are obtained from the third *cond* clause given above.

When a feature *Face1*, of an object, *Obj1* to be assembled *crosses* with another feature, *Face* of the same object, an additional pre-condition of clearing the second feature may be required in order to make the crossing area clear. It depends on whether another object which is assembled to the second feature of the first object provides a common area with the crossing area. If so, there is no need to clear the second feature because the crossing area will not be blocked. Otherwise, *clear(Obj1,Face)* is added onto the pre-condition list, and it will be added onto the post-condition list since the assemble action does not delete *clear(Obj1,Face)*. Such cases are defined in the first *cond* clause.

The third case is when a feature, *f1* is at another feature, *f2* (e.g. the bottom opening of a pen barrel is at the bottom end). The required additional feature will be *clear(object,f2)* in order to allow feature, *f1* to be assembled. Note that *f2* is still clear after feature, *f1* is assembled and is added to the list of post-conditions. This case is defined in the second *cond* clause.

Similarly the *cond* clauses for the *clear* state can be written as:

```
cond(clear(Obj,Face),X,Y):-
        findall(assemble(A1,A2,A3,A4,A5,A6),
                recorded(world,assemble(A1,A2,A3,A4,A5,A6),_),Assemble),
        member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble),
          findall(cross(Cr1,Cr2,Cr3,Cr4),
                   recorded(world,cross(Cr1,Cr2,Cr3,Cr4),_),Cross),
        findall(common(Co1,Co2,Co3,Co4,Co5),
                recorded(world,common(Co1,Co2,Co3,Co4,Co5),_),Common),
         ((member(cross(Obj,Face,Face3,C_area),Cross),
            ((member(common(Obj,Face3,_,_,C_area),Common),
             X1=clear(nil,nil),Y1=clear(nil,nil));
             (X1=clear(Obj,Face3),Y1=clear(Obj,Face3))));
           (member(cross(Obj1,Face1,Face3,C_area),Cross),
            ((member(common(Obj1,Face3,_,_,C_area),Common),
             X1=clear(nil,nil),Y1=clear(nil,nil));
             (X1=clear(Obj1,Face3),Y1=clear(Obj1,Face3))))),
        X2=[X1,assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface)],
          filter(X2,X),
        Y2=[Y1,clear(Obj,Face),clear(Obj1,Face1)],
        filter(Y2,Y),
        !.

cond(clear(Obj,Face),X,Y):-
        findall(assemble(A1,A2,A3,A4,A5,A6),
                recorded(world,assemble(A1,A2,A3,A4,A5,A6),_),Assemble),
        ((member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble),
          X1=assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),
          Y1=clear(Obj,Face),Y2=clear(Obj1,Face1));
         (X1=assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),
          Y1=clear(Obj,Face),Y2=clear(Obj1,Face1))),
        findall(at(At1,At2,At3),
                    recorded(world,at(At1,At2,At3),_),At),
        ((member(at(Obj,Face,Face2),At),
          X2=clear(Obj,Face2),Y3=clear(Obj,Face2));
         (member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble),
          member(at(Obj1,Face1,Face2),At),
          X2=clear(Obj1,Face2),Y3=clear(Obj1,Face2))),
        X=[X1,X2],
        Y2=[Y1,Y2,Y3],
        !.

cond(clear(Obj,Face),X,Y):-
        findall(assemble(A1,A2,A3,A4,A5,A6),
                recorded(world,assemble(A1,A2,A3,A4,A5,A6),_),Assemble),
        ((member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble),
          X=[assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface)],
          Y=[clear(Obj,Face),clear(Obj1,Face1)]);
         (X=[assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface)],
          Y=[clear(Obj,Face),clear(Obj1,Face1)])),
        !.
```

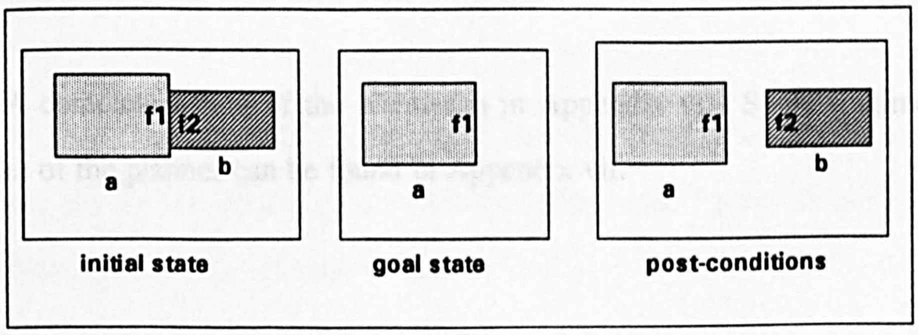

**Figure 5.16 Post-Conditions For a Simple Clear State**

As illustrated in Figure 5.16, for a simple clear state, *clear(a,f1)*, the pre-condition is *assemble(a,f1,b,f2,_,_)* and the post-conditions are *clear(a,f1)* and *clear(b,f2)*. This is defined in the third clear *cond* clause. In a case when the feature to be cleared crosses with another feature of the same object, it is necessary to check if any object which is assembled to the second feature has a common feature to the crossing area. If not, then the second feature must also be clear in the pre-condition, and will be clear in the post-condition. Such a case is defined in the first clear *cond* clause. In the last case the feature to be cleared is at or within another feature, then the second feature must also be clear in the pre-condition, and will be clear in the post-condition. This case is defined in the second clear *cond* clause.

When *do* succeeds, an appropriate *action* list will be produced and displayed on to the screen. Along with the *action* list, a view of the world database is also printed. Hence for the primary shaft example, the *action* list is:

```
press(bear2,whole,shaft,face2,shaft,face3).
press(bear3,whole,shaft,face3,shaft,face1).
```

A complete listing of the planner is in Appendix vi. Some runtime examples of the planner can be found in Appendix vii.

## 5.6 Miscellaneous Clauses

In addition to the clauses described so far, a number of other clauses are also used in the planner. Their roles are summarised below and their actual clauses can be found in Appendix vi in the listing of the planner.

i) The standard *member* clauses are used to check if an element is a member of a list. However for an assemble goal state the description, *assemble(Obj1,Face1,Obj2,Face2,_,_)* is in context equivalent to *assemble(Obj2,Face2,Obj1,Face1,_,_)*. Similarly for the *cross* and *common* predicates. Hence an additional *member* clause is used in the planner to check such cases.

ii) The *subset* clauses are are defined according to the standard definition of subset, e.g. [a,b] is a subset of [a,b,c,d] and [a,e] is not.

iii) The *intersection* clauses are defined according to the standard definition of intersection, e.g. [a,b] is an intersection set of [a,b,c,d] and [a,b,c].

iv) *Erase_list* is written to erase a list of elements from its database given the key to the database, e.g. world, action, etc. *Erase* is used to erase one member of a given list.

v) The *record_list* clauses are used to record a list of elements to the appropriate database.

vi) The *pp, ppx* clauses are used to print each element on a new line onto the screen.

vii) The *insert* clauses are used to insert a goal into the *currentgoal* database.

viii) The *append* clauses are used to append two lists, e,g, appending [a,b] to [c,d] will give [a,b,c,d].

## 5.7 Ball Point Pen Assembly

In addition to the examples previously mentioned, the assembly of the ball point pen was also used. The parts of the ball point pen are as shown in Figure 5.17 below and it can be assumed that it is also the initial state of this assembly. The final state is where refill is inside the pen barrel, the stopper is at the top end of the pen barrel and the cap is at the bottom end of the pen barrel as illustrated in Figure 5.18. The correct sequence of this assembly is that the refill must be inserted into the pen barrel before the cap. The assembly of the cap and stopper are independent actions and hence are parallel goals.

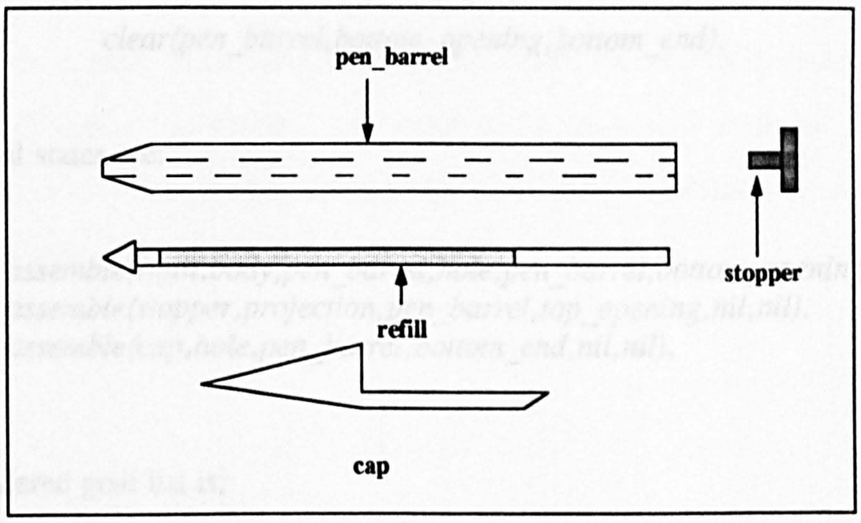

**Figure 5.17 Initial State of Ball Point Pen Assembly**

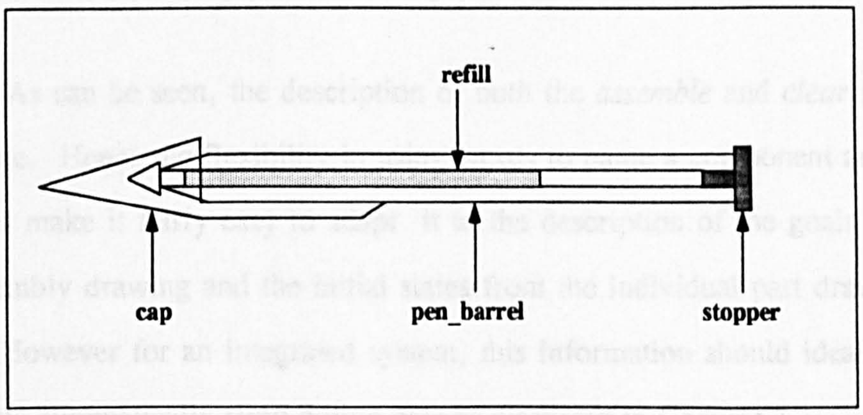

**Figure 5.18 A Possible Goal State For The Ball Point Pen Assembly**

The initial state descriptions are given below:

*clear(cap,hole),*
*clear(refill,body),*
*clear(stopper,projection),*
*clear(pen_barrel,hole),*
*clear(pen_barrel,top_opening),*
*clear(pen_barrel,bottom_opening),*
*clear(pen_barrel,bottom_end),*

*clear(pen_barrel,bottom_opening,bottom_end).*

The goal states are:

*assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).*
*assemble(stopper,projection,pen_barrel,top_opening,nil,nil).*
*assemble(cap,hole,pen_barrel,bottom_end,nil,nil).*

The ordered goal list is:

```
par(seq(assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening),
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil)),
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
```

As can be seen, the description of both the *assemble* and *clear* states is simple. Hence the flexibility in using words to name a component and its features make it fairly easy to adapt it to the description of the goals from an assembly drawing and the initial states from the individual part drawings [60]. However for an integrated system, this information should ideally be available automatically from design data as discussed in Chapter 2.

## 5.8 Summary

In considering geometric constraints, AAP had applied the fundamental principles of AI planning logic to derive an ordered sequence of operations. The strategy of the planner consists of defining the pre and post conditions, initial and goal states and then maintaining the truth of the assembly states by using planning heuristics. TWEAK's planning heuristics were used in addition to some practical considerations such as the redundant and conflicting goals check. The input goals are sorted prior to actual planning in order to avoid undoing of goals. Further discussion on AAP is included in the following chapter.

# CHAPTER 6. DISCUSSION

## 6.1 Introduction

As can be seen from AI planning literature, a vast amount of work has been done in this area. Therefore, it is not the intention of this research to produce yet another planner because most of the heuristics for the planning problem would seem to have already been suggested. The main objective of this project was to examine both the current AI planning and process planning situations and see what benefits process planning can obtain from AI planning. It also involves a study on the feasibility of applying AI techniques to process planning and assembly process planning in particular. As a result of applying some AI planning knowledge to some simple assembly examples, it is believed that AI planning techniques can be applied to assembly process planning and in particular when addressing planning problems, (whether for manufacturing or everyday life) they must be solved or viewed as a whole. This chapter provides a discussion on the findings and some recommendations to direct further and/or similar work in this area.

## 6.2 The AI Planning Situation

Much of planning research has been theoretical, designed for a particular specification (or application) and are not intended for use outside the research laboratory. This often means that an AI planner may not work in areas other than the application that it was originally designed for. In some cases when studied and tested by people other than the authors themselves, even some of the examples (used to model those particular planners) can even fail to work. The reasons for this could be due to lack of understanding of the methods used by the original author and also the ad hoc way in which heuristics were employed to solve specific examples [61].

When considering the application of AI planning techniques to real planning problems, the following general points should be considered:

i) size of the problem, i.e whether it is manageable using a computational approach;
ii) whether the specifications of the particular AI planner matches that of the application in mind[1] and the practical limitations of the planner;
iii) the quality or accuracy of the results;
iv) the ease of formalising the plans and this includes the format of the both the required input and output (e.g. the state descriptions);
v) implementation issues such as which programming language to use, the memory requirements of the problem and the hardware environment.
vi) linkage to other systems if necessary, e.g. linkage to design and manufacture of a manufacturing system.

The main reason for basing AAP on TWEAK was because it was fairly well described in the literature compared with other AI planners. Some practical problems that one may encounter when attempting to use AI

[1] If the AI planner is based on generic definitions and proven to be correct and complete mathematically then this problem can at least be reduced, which is why the need for a sound theoretical basis for planning.

planning techniques is that heuristics of planners used may not be sufficient for other examples. Apart from TWEAK, most of the AI planners to date were an improvement on their predecessors, i.e. ad hoc collections of heuristics and rules for applying these heuristics. Since there was no formal representation to the planning problem as such, an augmented AI planner usually consisted of an additional rule that was implemented to solve an example that its predecessor could not solve as seen in STRIPS and ABSTRIPS. Thus a good starting point is to adopt more generic techniques.

AAP is able to produce expected sequences with the given examples. Despite the fact that it is only based on geometric constraints, it has demonstrated a possible way of representing manual assembly problems in an AI planning domain using generic techniques, i.e deriving a possible plan from logic rather than using a look-up table such as used in the variant approach to computer aided process planning. The implication is that an AI planner with a set of general rules and facts is also adaptable to other areas of assembly planning, e.g. assembly of domestic appliances, toys, etc.

Certainly the next generation of AI planning work should concentrate on formalising the planning problem[2] and applying as well as testing them using more realistic cases instead of *Blocks World* examples. For assembly process planning applications, this could be done by using more realistic constraints such as those in the manual assembly environment, say for example reaction faces which are needed for assembly must be clear as

[2] as in TWEAK which is one of the first mathematical approach used in AI planning formalisms.

implemented in AAP. The current trend of AI planning research is in the area of plan execution and re-planning, if necessary [62]. The examples used are again based on Blocks World examples. The challenge facing the AI community is to convince people outside the AI community the validity of these techniques and their viability in real situations such as the manufacturing environment.

Like most AI projects, the development of AI systems for industrial or commercial use is a long term investment. This is perhaps the reason why many companies are reluctant to investigate and invest in long term AI projects. This has resulted in a number of expert systems that are developed for very specific purposes and in some cases behaving as an expensive substitute to a look-up table. What is lacking is some general guide-lines and techniques for developing AI systems that will benefit from the results of AI research so far in order to achieve the aim of producing truly integrated systems in the manufacturing sector. In the UK, the Department of Trade and Industry is sponsoring large projects which will be able to provide some answers to companies that cannot afford such long term investments [63].

Finally, an undeniable fact that has emerged from AI planning research is that the planning problem has to be considered as a whole. This includes planning problems in all aspects of life, albeit for manufacturing or for events of everyday life. The size of the problem can be reduced if it can be broken down into un-related parts as in the sub-assemblies of assembly.

## 6.3 The Logical Approach

As mentioned in Chapter 4, an advantage of using the logic approach is that the coding is more compact and is directly convertible from the specifications. This is not entirely true as seen in the Prolog language as some conversion, i.e. to the format of the Prolog syntax, is essential. Ideally, as little coding as possible should be necessary given the user's specifications but to date the logical approach is the nearest. The theory of logical systems (i.e. fifth generation programs) provides the basis for producing less errors when compared with programs written in lower level programming languages as less translations and hence less chances for errors. In safety critical systems such as computer systems that are used on the Airbus, it is suggested that a logic approach should be taken in order to reduce the chances of error.

The main drawback of the logical approach is that symbolic manipulations can become clumsy when there are large number of facts in the system. This leads to a combinatorial explosion in the possibilities of which rules to apply to which facts at each step of the proof. Even though in Prolog the selection of which rules to apply first can be set (by the ordering of rules) and changed (using the cut facility) by the programmer, it can still be very difficult to debug when the program does not behave in an expected manner[3]. Despite having the reputation of being a relatively easy language to program, it is also very easy to write Prolog code without actually knowing how or what it means. People who are used to

[3] The use of parenthesis can make a great difference to the program's behaviour.

programming in procedural languages often fall into the trap of treating it like a procedural language[4]. In such cases, programmers may be stuck with attempting to debug the programs. For example, in the implementation of AAP, when there are a number of alternatives to a rule, these alternatives have to be prioritised so that those with a higher priority are ordered first. When the cut facility is used in these alternatives, it may be difficult to predict where the search will terminate and the intended meaning of the program may change with the use of cuts. However like programming in general, some practice will help to alleviate misinterpretations of this type of program. Therefore some training in the logical programming approach may be necessary for people who are used to the conventional programming method.

Finally, even though the problem faced by AI planners is said to be difficult, it can be reduced when applied to real situations, like in assembly planning. Historically, the aim of AI planning is to produce a general purpose planner rather like the development of the general purpose computer. However specific planners for specific tasks as demonstrated by the actual working practice of planning for sub-assemblies may be a possibility for reducing the enormity of AI planning research.

[4] Even though Prolog is also considered as a procedural language, the meaning behind it (i.e. the resolution principle) is quite different from other conventional languages.

## 6.4 The CAPP Situation

Even though CAPP has been in existence since the first CAPP system presented by CAM-I in 1976, there have still not been many improvements in the area of integrating design with manufacture through process planning which is still considered to be the bottle-neck to integration. This could be due to the lack of knowledge on techniques for automating process planning on the part of the process planners themselves and lack of knowledge on the process planning task by system developers. Most of the process planning systems that are developed are in the area of machining because from the system designer's point of view there is more information regarding the process which are obtainable from handbooks and standard data. Thus leading to the development of variant type systems. Like AI planning, despite an improvement on the speed at which information can now be processed with the aid of sophisticated computers, the CAPP situation is also lacking in providing guide-lines and methodologies. Even though there is a large number of reported CAPP systems, most are very specific and are targeted for the manufacture of specific components. Future CAPP systems should take advantage of the results which could be obtained by looking for a common feature of all CAPP systems that is applicable to process planning in general. As suggested earlier the core logic of planning should be applicable to manual assembly in general. Furthermore automated assembly process planning should be able to reduce development lead times by enabling engineers to estimate costs for different plans which can be obtained by e.g. changing initial states configurations.

The assembly planner in this project is only concerned with the ordering of goal states/operations to achieve an overall assembly state from an initial state as described by a user. Other related work in assembly planning is concerned with robotic assembly path planning based on geometric modelling of mating features, where the shapes and sizes of mating features are analysed to compute feasible robotic paths in order to orientate the appropriate parts during assembly [64,65,66].

Since variant type process planning systems are more established commercially than generative type, companies that cannot afford the long term investments required for generative process planning might be better off with a variant type system. In some cases, variant type systems are more or less like text editors and this feature can also be tedious to users[5]. Therefore it is not uncommon for some companies to abandon these systems after a while. Lack of confidence in automated process planning systems could also be due to process planning systems being released to early (probably at the development stage) to the market. Thus when errors occur, users are discouraged from using them and often revert back to the manual systems. Purchasing a one off CAPP system may not solve the integration problem, therefore some companies are now making long term investments by working on the principle of a complete manufacturing system. This could be the only solution to integration, i.e. designing a system for all functions within manufacturing which is based on a common principle. A possible approach to the link between design and process planning is to have a feature-based modelling system instead of the conventional CAD systems. This is to enable feature information such as say, *this is a blind-hole* to be

[5] Tedius compared to manual system, because users may have to learn to use the system as well as operate a computer which may not be part of thier daily work.

incorporated in the design (or modelling) system. Hence the process planning approach could be a generative approach based on the feature information from the modelling systems, using both general and specific planning knowledge[6].

## 6.5 Philosophical Issues

From the view point of cognitive science, an interesting question that one may ask is whether human process planners do actually work like the variant approach, i.e. accumulate a set of standard plans and when presented with a new problem, produce a new solution by modifying plans for a similar problem. If indeed this is how they work, then the variant approach is perhaps more human-like and can be considered as an AI approach according to the definition of AI. This is contrary to the believe that the generative approach is similar to AI process planning research. In addition psychological evidence exists to indicate that people rely on a bank of knowledge accumulated from previous experience to interpret new situations [67]. This is rather similar to the variant type of approach. However from a philosophical view point, there are some who believe that the human mind is more *intelligent* than (and subsequently the mystique surrounding the whole issue of intelligence) simply relying on a huge look-up table for future references[7].

[6] Some companies in the UK are now taking this veiw with a plan stretching a number years, beyond 1992.
[7] A contradiction to this belief is demonstrated by Searle's Chinese Room where equally good translations (from English to Chinese and vice versa) were obtained from an operator with no knowledge of the languages concerned who was simply making references from a huge look-up table.

## 6.6 Findings From The Development Of AAP

### 6.6.1 Assembly Constraints

For the sake of simplicity, the type of assembly operation specified in the planner is *press* and *remove*. The first is used to suggest the assembling of two components rather than the literal meaning of the word press, while the *remove* operation is the reverse operation of *press*. These operations relate to the topology of the components and therefore can be termed geometric constraints. However realistically there are other constraints such as the type of operation to be used e.g. screwing, glueing and welding and also the type of tools to use, e.g. screw driver or hammer. Extra rules would need to be developed and a substantial amount of company specific data would be necessary in order to include these constraints. This is because different types of products require different types of assembly operations and hence different types of tooling.

Currently, AAP only knows about manual assembly that requires reaction faces. In order to accommodate other constraints, the format of the state descriptions using six arguments may have to be modified.

## 6.6.2 Input Description

In AAP, the input data is the goal state in the predicate format, i.e. assemble(.....) or clear(..). It is also necessary to remember that there are six arguments in the assemble predicate and two in the clear predicate. In addition the data in the initial state must be consistent with the goal state. This means that any spelling mistakes in the input would cause the AAP to fail. For real situations this is not acceptable. Therefore other varieties of examples are necessary in order to determine a better input description of the goal states. The input description should ideally be related to the data format from the design department, to allow a direct transfer of information between the process planning department and the design department. This fact has now been recognised and as suggested in Chapter 2, featured-based design packages are now beginning to emerge in the market. These feature-based CAD systems are at the moment expensive and sometimes the proprietors concerned may not be willing to provide source code information on these systems. Also these systems are at an early development stage and may not be entirely perfect even though they have been released into the market. Alternatively, another possible area of related research is to consider the design of feature based CAD systems using the Prolog language to allow it to be linked to the assembly planner easily if there were no feature-based CAD systems available.

### 6.6.3 Assembly Space Conflict

The kind of situations illustrated in Figure 3.14 of Chapter 3 where a planner is attempting to assemble a component onto another part or feature which does not have enough space due to a prior assembly, do not occur in the AAP because the mating pairs are specified in the first place. In AAP, the problem of more than one part being assembled to a designated feature which can only accommodate one particular part is not relevant. Hence if the definitions and constraints are specified in this way, such cases can be eliminated.

### 6.6.4 Level Of Detail

A problem with the logical approach is in the representation technique e.g. how to represent the state descriptions precisely to the level of required detail. It can get very complicated and tedious if too much detail is included but if it is under specified then this could lead to misinterpretations and insufficient knowledge about the constraints of the assembly. Another consideration is that AAP does not have an actual picture of the assembly as do humans and so specific pictorial descriptions may not be appropriate in this case. Therefore a more general description using the *assemble* and *clear* predicates were adopted in preference to the *inside* and *outside* predicate of the test planner. The *assemble* predicate first suggests there are two mating components with two corresponding mating surfaces, i.e.

*assemble(Component1,Face1,Component2,Face2).*

For example, cases like *on(a,b)* and *inside(a,b)* can be described as: *assemble(a,bottom_face,b,top_face)* and *assemble(a,out_wall,b,in_wall)* respectively which are still easily pictured by a user and at the same time is viewed in the same way by AAP. The *assemble* predicate also includes pre-specified constraints like a specified reaction face of one component to be used if any, e.g. *assemble(a,face1,b,face2,a,face3)* where *face3* is the reaction face of component *a*. For the *clear* predicate, it is sufficient to describe the feature of the component that has to be cleared. If there are any inter-relationships between any two features of the same component like two holes intersecting with one another then this can be described separately as a fact, e.g. like the *cross* predicate in AAP.

## 6.6.5 Additional Reaction Face

If two reaction faces are required for the assembly, i.e. one reaction face on each of the two components during assembly, then the *assemble* predicate as defined in AAP could be increased to accommodate eight arguments. In this case, most situations should have been covered. In the definition of the *cond* clauses, if both the *at* and *cross* situations exist at the same time in an assembly then the *cond* clauses will have to be modified accordingly in order to work in such cases.

### 6.6.6 Assembly Database

At the current stage of development, AAP is by no means ready for industrial use. In order to apply it to a practical situation, the next stage of development could involve developing a comprehensive database of sub-assemblies. This requires the involvement of a company that would be willing to provide such realistic data. In addition, it may be necessary to add more special relationships between features as seen in the *cross* predicate depending on the nature of the company's business.

### 6.6.7 Problem Size

In order to implement the above mentioned extentions to AAP, it would vastly increase its complexity and size. Due to historical reasons, the practice of analysing the complexity of Prolog programs is not as developed as for programs written in more conventional programming languages. As a rough measure, in analysing the time complexity of the sorting clause in the planner where *n* is the number of goal states, the number of comparisons required is said to be of the order of:

$$\sum_{1 < i < n}^{n > 2} (n-i+1)(n-i) = n^2-2ni+i^2+n-i$$

(See Appendix iii for the derivation of this formula) Hence for an assembly with a large number of sub-goals, the time will be significantly increased. However as mentioned in Chapter 4, it is usual practice to break down the assembly of a complete design into sub-assemblies according to the functional properties of the design. This means that the number of sub-goals in a sub-assembly should be manageable in practice. Hence if planning commences at the sub-assembly level, the time complexity of the problem should be reduced. For example, when the piston-connecting rod and primary shaft sub-assemblies are treated as a single problem containing a total of 4 sub-goal states, the time scale will be a factor of 20. When the two sub-assemblies are treated separately, the time scale is 4. This is a great reduction in the time.

## 6.6.8 Improving The Sorting Procedure

In particular, the *sorting* procedure in the planner had only considered the linking of one to one and one to many relationships between goals, e.g. *b* comes after *a* and *(b,c)* comes after *a* types of situations. The linking procedure will have to be modified accordingly to work for cases that have a many to one relationship, i.e. *d* comes after both *(b,c)* type of situations as illustrated in Figure 6.1.

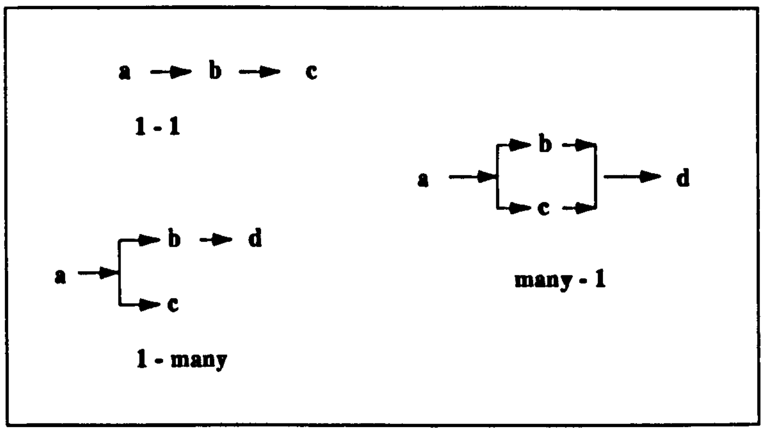

Figure 6.1 Types of Goal Relationships

## 6.6.9 Optimization Of The AAP Output

The AAP currently provides an output in the form of a list which may contain parallel or sequential goals or a mixture of both. The sequential goals must be done in the order that they are stated but the parallel goals can be done in any order. Since AAP finds the first solution, it would not necessarily be the only solution or even an optimal solution. The solution achieved is dependent on the ordering of the input data. In Appendix vii, Page a60, it can be seen that when the input is as follows:

```
assemble(bear2,whole,shaft,face2,shaft,face3).
assemble(bear3,whole,shaft,face3,shaft,face1).
assemble(pin,whole,piston,t_hole,nil,nil).
assemble(rod,whole,piston,axial_hole,nil,nil).
```

The output actions are to assemble the bearings followed by the assembly of the piston-connecting rod, i.e.

```
press(bear2,whole,shaft,face2,shaft,face3).
press(bear3,whole,shaft,face3,shaft,face1).
```

```
remove(piston,t_hole,pin,whole).
press(rod,whole,piston,axial_hole,nil,nil).
press(pin,whole,piston,t_hole,nil,nil).
```

In the second case (see Pagea61) the data remains the same but with a different ordering, the output is:

```
remove(piston,t_hole,pin,whole).
press(rod,whole,piston,axial_hole,nil,nil).
press(pin,whole,piston,t_hole,nil,nil).
press(bear2,whole,shaft,face2,shaft,face3).
press(bear3,whole,shaft,face3,shaft,face1).
```

The outputs are correct in both cases but no recommendations are made as to which is a better solution. In order to determine an optimal ordering of parallel operations, further information such as tooling and time of operations are needed.

## 6.6.10 Other Hardware Platforms

Even though AAP was implemented on a SUN workstation, there is now an availability of powerful microcomputers which are able to run Prolog applications. However until recently most versions of Prolog on microcomputers[8] do not have most of the facilities that are available on bigger machines. The speed of operations on these bigger machines are also much faster than microcomputers. Therefore most Prolog applications are

[8] e.g. Turbo Prolog on IBM microcomputers.

based on much faster machines as in the case of AAP. This situation is now changing with the availability of powerful microcomputers and vendors are also providing better and more complete implementations of Prolog for microcomputers. Hence it should be possible to run the AAP in a micro-computer environment as companies are more likely to be able to afford microcomputers than sophisticated work stations. This is to enable companies that cannot afford to invest in long term research work to benefit from the findings of research in this area.

## 6.7 Lessons And Future Systems

It is without doubt that the ideal for reducing production costs is to have a truly integrated computer system. This means that process planning must not only be linked to design but also to manufacture further downstream, i.e. actual scheduling of tasks for the shop floor in particular. The problem of integration with design has already been mentioned in the previous chapter. As scheduling assumes the process plans to be fixed, any changes in the sequencing of operations will affect scheduling as well. In a dynamic situation, it may not be possible to adhere to the recommendations of process planning, e.g. unavailability of specified tooling, and alternative measures have to be sought. These alternative solutions should not compromise the quality. Therefore, in an ideal integrated system, links are necessary both ways from process planning, the former so that process planning is able to accommodate changes and requests from design and the latter is necessary in order to pass on these changes as well as acting on the dynamic situation on the shop floor. It may be argued that since quality is not compromised due to alternative solutions being sought further downstream, then the process planning function need not be such a flexible one. However the view that is taken downstream may be a parochial one compared with that of process planning. Hence changes if any, should be fed back both ways as illustrated in Figure 6.2.

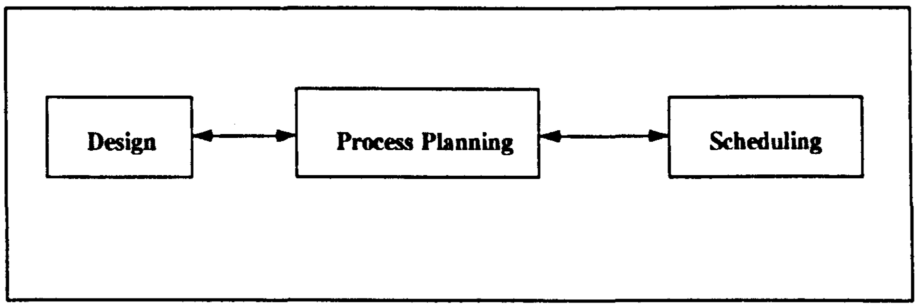

**Figure 6.2 Interactions of An Assembly Process Planning System**

Based on AI planning techniques, it is possible to link the process planning system to downstream activities as the new sequences can be derived from the logic of planning. Therefore, less programming effort is needed to cater for the changes when compared with the conventional programming approach. For example, AAP could be expanded by adding more constraints, such as the tooling constraints that are based on the dynamic situation on the shop floor and new sequences of operations can be generated. This kind of situation also enables information to be conveyed to process planners and managers for evaluation which may prompt further managerial action.

In considering the interface to design, it is not desirable to design interfaces to a particular CAD system as it would only be useful to that particular system. This problem was partly addressed by IGES which provided the means to convert data into a neutral format first in order to allow it to be transferred to other systems. Initially, it seems to be an attractive idea but the impracticalities of the concept, e.g. incompatibility of tape format between different systems, soon become apparent after a number of trials. Also the problem of extracting the necessary data for downstream applications such as process planning still remains unsolved. Various attempts were then made to interpret CAD data which is not only time consuming but also runs into the danger of being too specific for general use. Taking a closer look at the design, it has been suggested that since it was derived from a concept with functionalities attached to it, it would be beneficial if such properties were stated explicitly in the first place rather than having to derive them from the design at a later stage. Hence the current trends of CAD systems is towards functional based CAD systems where the manufacturing properties are stated together with the geometric features. It is not enough just to provide geometric features in CAD systems but rather the engineering features as well. This makes the validity of designing interfaces to geometric-based CAD systems questionable.

Hence if feature-based information is available from the design then the interface from design to process planning would be made slightly easier, e.g. extract the goal state and final state in manufacturing terms from the design. It is believed that there are no short cuts to the interface design because some degree of interpretation is still required, e.g. converting the manufacturing data into the format that is required by the (process planning) system further downstream. This task is made easier if the manufacturing system is built upon a common philosophy.

The key issues of future systems are interaction and integration. Interaction between different users of the system is necessary to allow a sharing of expertise amongst users within a department as well as crossing over into different functional boundaries (such as design and process planning department). Communication which is an essential element of such interactions has to be established and this could be achieved in the form of distributed knowledge based systems, possibly maintained by a blackboard architecture [68]. Integration provides the link up of these functionalities in a system in order to obtain answers efficiently. So far only the technical (i.e. software methodologies and hardware platforms) problems of integration has been mentioned but the management aspects of integration are equally important. This requires involvement of top management as well as involvement of people, i.e. users as well as manager.

Even though the motivation of this project is concerned with using AI techniques, one must also remember that AI need not be the only solution. Using AI techniques does not mean the exclusive use of *AI tools* as demonstrated by the ESCAPE system used by the FORD Motor company in America for collecting information and requests (e.g. requests for refunds) from dealers. An expert system module was embedded in a COBOL program (which was the original system) to alleviate the problems of frequent changes needed in the COBOL program [68]. Hence instead of rejecting AI techniques or embracing them exclusively, sometimes it may be worthwhile to consider a combination of both AI (or expert system) technology in conjunction with existing conventional methods.

Today the Fifth Generation Programme is technically over but in practice the computing industry is still dominated by procedural and conventional methodologies. However AI techniques in a reduced form can be seen in the shape of expert systems which have sprung up very rapidly over the last decade or so. This has sparked off much interest in AI from other non AI fields. Despite failing to produce a computer system (perhaps even in the next decade or so) that can behave (i.e. perceive, reason, understand, communicate, etc) like humans, AI techniques or its products will definitely play a part in people's everyday life[9] in one form or another. Perhaps future systems in general might have a planner as the heart of the system (rather like humans) which is able to solve problems in general. Specific problems requiring specific rules and facts could then be added to this system which is rather similar to a shell. Once the system is aware of new situations or applications, it is then able to learn and behave appropriately and this could be an incremental process. As the Fifth Generation Programme fades into history, there is now talk of the Sixth Generation Programme where the emphasis is on parallel processing and neural networks. The former would enable more powerful processing capabilities while the latter would provide an insight into advanced pattern recognition technology. A feasibility study is going on at the moment and

[9] For example, the Japanese (known to be keen on applying AI techniques) have produced an intelligent rice cooker.

if encouraging the program will commence in April 1992. This time the parties involved will consist of both Japanese and US/EC companies and universities. Thus enabling international cooperation [69]. The Sixth Generation Programme will be able to carry on the aims and objectives of the Fifth generation programme, i.e. the *search* for an artificially intelligent artifact.

# CHAPTER 7. CONCLUSIONS

AI planning techniques which have been developed in a Blocks World environment are relevant to the assembly process planning problem and AAP has demonstrated the feasibility of applying the fundamental planning heuristics of AI planning to assembly planning.

In real assembly situations the size of the problem is great. However it can be decomposed into sub-assemblies like the sub-goals in AI planning and can therefore be treated independently. Hence reducing the size of the problem.

The format of the input to AAP is important as it affects the ease of use of the planner. The alternative forms of input to AAP are either feature based descriptions or geometric CAD data but there are considerable difficulties in both areas.

The way in which the input is ordered closely affects the final output of the parallel goals. In order to optimise these parallel goals, further domain specific information is needed.

Further constraints such as those relating to operation and tooling are required for the operation of the AAP. In addition, access to a company database of assembly data would be essential.

Even though theoretically a generative planner is the ideal, there are nevertheless valid practical reasons why variant planners are used and will continue to be used.

# REFERENCES

[1] J. Kennedy et al, *Manufacturing Intelligence Market Surveys and Applications*, Department of Trade and Industry, MIT Division, 1989.

[2] K. Dell, *An introduction to computer-Aided Process Planning*, CIM Review, Fall 1987, pp 7-23.

[3] K. Srihari & T. J. Greene, *Computer-Aided Process Planning, A CAD/CAM Inteface*, CIM Review, Summer, 1988, pp 47 - 54

[4] Zozaya-Gorostiza, C Hendrickson, D R Rehak, *Knowledge-based process planning for construction and manufacturing*, Academic Press, 1989, pp 4 - 6.

[5] L. Alting, H.C. Zhang , *Computer Aided Process Planning : the state of art survey*, Int. J Prod. Res., 1989, Vol 27, No. 4 pp 553 - 585.

[6] C.H. Link, *CAPP, CAI automated process planning system*, Proceedings of the 1976 NC Conference, 1976, CAM-I, Ico. Arlington, Texas, USA.

[7] T.C. Chang, R.A. Wysk, *An introduction to automated process planning systems*, Prentice-Hall, 1985, pp 18 -22.

[8] J.L. Burbridge , *Production Flow Analysis*, The Production Engineer, April/May 1971.

[9] J.L. Burbridge, *The Introduction of Group Technology*, Wiley, New York, 1975.

[10] T.C. Chang, R.A. Wysk, *An Introduction to automated process planning systems*, Prentice Hall, 1985, pp 124 - 132.

[11] Y. Descotte , J.C. Latombe , *GARI: A problem solver that plans how to machine mechanical parts*, IJCAI 7, Vancouver, August 1981 pp 766 - 772.

[12] B.J. Davies 1986, *Expert systems in process planning*, 7the International conference on the Computer as a design tool, London, UK 2-5 September.

[13] K. Matsushima , et al, 1982, *The integration of CAD and CAM by application of artificial intelligence techniques*, Annals of the CIRP, 31(1).

[14] H.R. Berenji, B. Khoshnevis, *Use of AI in Automated Process Planning*, Computers in Mechanical Engineering, Sept 1986, pp 47-55.

[15] Li L., Bedworth D.D., *A semi-generative approach to CAPP using Group Technology*, Computers in Industrial Engineering, Vol. 14, No. 2, pp 127-137, 1988.

[16] A. Houtzeel, *A chain structured part classification system, (MICLASS) and group technology*, Proceedings of the 14th Annual meeting & Technical conference of the NC Society, March 13-16 , 1977, Pennsylvania, pp 383-401.

[17] A. Opitz, *A classification system to describe work pieces*, ed. MacConnell W.R., Pergamon Press, ELmsford, New York 1970.

[18] Wang Hsu-Pin, *Intelligent Reasoning for Process Planning*, Department of Industrial & Management Systems, Dec 1986, Ph. D. thesis.

[19] D.F. Theilen, J.F. Jones, *IGES - Data Exchange Between Dissimilar CAD/CAM systems*, Autofact (4) 1982, pp 20-33.

[20] A.C. Sanderson, L.S. Homem de Mello, Zhang H., *Assembly Sequence Planning*, AI Magazine, Spring 1990, pp 62-81.

[21] L.S. Haynes, G.H. Morris, *A formal approach to specifying assembly operations*, Int. Journal of Machine Tools Manufacture, Vol 28, No. 3, 1988, pp 281-298.

[22] Rocheleau D.N., Lee K.W., *System for Interactive Assembly Modelling*, Computer-Aided Design, Vol. 19, no. 2, March 1987, pp 65-72.

[23] Fazio T.L., Whitney D.E., *Simplified Generation of All Mechanical Assembly Sequences*, IEEE Journal of Robotics and Automation, Vol RA-3, No. 6, December 1987, pp 640-708.

[24] METCUT, Research Associates, Inc., *Machining Data Handbook*, Machinability Data Centre, Cincinnati, Ohio, 1980.

[25] E. Rich : *Artificial Intelligence*, McGraw-Hill, 1983, pp 1.

[26] A. Newell, H.A. Simon: *GPS, A program that simulates human thought, Computers and Thought*, (eds Feigenbaum, E A; Feldman J.) New York, McGraw-Hill 1963.

[27] E. Rich : *Artificial Intelligence*, McGraw-Hill, 1983, pp 99 - 102.

[28] L. Siklossy, *After 24 years in AI:Some Lessons*, Lecture notes in AI, edited by J. Siekmann, AI in Higher Education, CEPES-UNESCO International Symposium Prague, CSFR, October 1989 Proceedings, SPringer-Verlag, pp 159-167.

[29] N. Gupta, D. Nau, *Optimal's Blocks World Solutions are NP-hard*, Technical Report, Computer Science Dept., University of Maryland, USA.

[30] D. Chapman, *Planning for Conjunctive Goals*, Artificial Intelligence, 1987, 32, pp 333-377.

[31] C.C. Green, *Theorem Proving by resolution as a basis for question-answering systems*, Machine intelligence, Vol4 (eds B. Meltzer, D. Michie), New York, Elsevier Pub. Co 1969.

[32] R. Kowalski, *Logic for problem solving*, North-Holland, 1979 .

[33] R.E. Fikes, N.J. Nilsson:*STRIPS: A new approach to the application of theorem proving to problem solving*, AI, Vol 2, 1971, pp 189-208

[34] E.D. Sacerdoti, *Planning in a Hierarchy of Abstraction Spaces*, Artificial Intelligence, Vol 5, pp115-135, 1974.

[35] Sacerdoti E. D., *A structure for plans and behaviour*, Elsevier, New York, 1977.

[36] D. Chapman, *Planning for conjunctive Goals*, Artificial Intelligence, vol 32, 1987, pp 333 - 377.

[37] G.A. Sussman: *A computational model of skill acquisition*, MIT AI Lab Memo, AI-TR 297, Cambridge, Mass.

[38] A. Tate:*Interacting Plans and their use*, Proc of IJCAI, 1975, pp 215-218, Tbilisi, USSR.

[39] A. Tate, *Project Planning using a hierarchical non-linear planner*, Dept of AI, Report 25, 1976, University of Edinburgh.

[40] M.J. Stefik: *Planning with constraints*, Artificial Intelligence, Vol 16, 1981, pp 111-140.

[41] G. Boothroyd & P. Dewhurst, *Design For Assembly, A Designer's Handbook*, Department of Mechanical Engineering, Massachusetts, 1983.

[42] G. Spur, F.L. Krause, W. Grotte., *Planning of Assembly Sequences*, Lecture notes in Computer Science: Methods & Tools for Computer Integrated Manufacturing, Springer-Verlag, 1984, pp 106 - 111.

[43] T. L. Fazio & D. Whitney, *Simplified generation of all mechanical assembly sequences*, IEEE Journal of Robotics and Automation, Vol RA-3, no. 6, Dec 1987, pp 640 - 708.

[44] M. Allchurch, *Computer-Aided Line Balancing*, M. Sc. dissertation, University of Warwick, 1991, pp 42 - 76 .

[45] A.C. Sanderson, L.S. Homem de Mello, H. Zhang, *Assembly Sequence Planning*, AI Magazine, Spring 1990, pp 62-81.

[46] M. Welbank, *A review of knowledge acquisition techniques for expert systems*, Published by Martlesham Consultancy Services, British Telecom Research Laboratories, Ipswich, December 1983.

[47] A. M. Burton, N. R. Shadbolt, A. P. Hedgecock, G. Rugg, *A formal evaluation of knowledge elicitation techniques for expert systems*, Research and Development in Expert systems IV, edited by S. Moralee; Cambridge University Press, 1987.

[48] A. Newell, H.A. Simon, *Human Problem Solving*, Englewood Cliffs, N.J., Prentice-Hall, 1972.

[49] J.R. Carbonell , AI in CAI: *An AI approach to computer-assisted instruction*, IEEE Transaction on Man-Machnie systems, 1970, pp 190-202.

[50] R.C. Schank , R.P. Abelson *R P, Scripts, Plans, Goals and Understanding*, Hillsdale, N.J., Lawrence Erlbaum, 1977.

[51] J.G. Carbonell , R.E. Cullingford , A.V. Gershman , *Steps towards knowledge-based machine translation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 3, no. 4, 1981, pp 376-392.

[52] P.R. Cohen , E.A. Feigenbaum , *The Handbook of Artificial Intelligence, Volume III*, Pitman Books Ltd, 1983, pp 114-119.

[53] J. Doyle, *A Truth Maintenance System* , AI No. 12, 1979, pp 231 - 272.

[54] M.L. Ginsberg, *Readings in Non-monotonic reasoning*, Morgan Kaufman Pub, Los Altos, California, 1987.

[55] L. D. Erman , F. Hayes-Roth, V. R. Lesser & D. R. Reddy, *The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty*, Computing Surveys, Vol 12, No 2, June 1980.

[56] K. Matsushima, N. Okada, T. Sata, *The Integration of CAD and CAM by application of Artificial Intelligence*, Annals of the CIRP, Vol 31, No. 1, 1982.

[57] T. Richards , *Clausal Form Logic, An Introduction to the logic of computer reasoning*, Addison Wesley, pp 273 - 283, 1989.

[58] D. Chapman, *Planning for Conjunctive Goals*, Artificial Intelligence, 32 1987, pp 333-377.

[59] A.L. Dowd, Y. P. Cheung, *An Intelligent Planner For Assembly Process Planning*, INCOM 1989, 6th Symposium on Information Control Problems in Manufacturing Technology, Madrid, 26-29 sep 1989, Preprints of Volume 1, pp 19-23.

[60] Y. P. Cheung, A.L. Dowd, *An Artificial Intelligence Planner for Assembly Process Planning*, Presented at 6th International Conference on CAD/CAM Robotics & Factories of the Future, London 19-22 August 1991, International Society For Productivity Enhancement.

[61] N.A. Kartam D.E. Wilkins, *Towards a Foundation For Evaluating AI Planners*, AI EDAM, 1990, 4(1), pp 1 - 13.

[62] L. Morgenstern, *Replanning*, In proceedings of defense Advanced Research Projects Agency Knowledge-based Planning Workshop, 1987.

[63] J. Lamb, *Computer Weekly News*, October 17, 1991, pp 3.

[64] A.C. Sanderson, L.S. Homem de Mello, H. Zhang, *Assembly Sequence Planning*, AI Magazine,Spring 1990, pp 62 - 81.

[65] L. S. Haynes, G.H. morris, *A Formal Approach to specifying assembly operations*, Int. Journal of Machine Tools Manufacture, Vol. 28, No. 3, 1988, pp 281-298.

[66] R. J. Popplestone, Y. Liu, R. Weiss, *A Group Theoretic Approach to Assembly Planning*, AI Magazine, Spring 1990, pp 82 - 97.

[67] F.C. Bartlett, Remembering: *A Study in Experimental and social psychology*, Cambridge University Press, 1932, Reprinted in 1977.

[68] D. Bobrow, *Dimensions of Interaction*, AI Magazine, Fall 1991, pp 64-78.

[69] Newsletter of AISB (Society for the Study of Artificial Intelligence & Simulation of Behaviour), Summer 1991, No. 7, Page 3.

## **Bibliography**

Andreasen M., Lund T., Swift K.G., Kahler S., *Design for Assembly*, Bedford, IFS, 1988.

Barr A., Feigenbaum E.A., *The Handbook of Artificial Intelligence*, Volume 1, W Kaufmann, 1981.

Barr A., Feigenbaum E.A., *The Handbook of Artificial Intelligence*, Volume 3, W Kaufmann, 1983.

Bolter J.D., *Turing's man*, Penguin Books, 1984.

Bratko I., *Prolog programming for Artificial Intelligence*, Addison Wesley, 1986.

Chang T.C., Wysk R.A., *An introduction to automated process planning systems*, Prentice-Hall, 1985, pp 18 -22.

Cohn A.G., Thomas J.R., *Artificial Intelligence and its applications*, John Wiley, 1986.

Clocksin W.F., Mellish C.S., *Programming in Prolog*, Springer-Verlag, 1984.

Davis R., Lenat D.B., Knowledge-based systems in Artificial Intelligence, McGraw-Hill, 1982.

Fisher E., Moodie C.L., Martin-Vega L.A., *Proceedings of Manufacturing International 1990, Volume 1, Intelligent Manufacturing Structure, Control and Integration*, The American Society of Mechanical Engineers, 1990.

Forsyth R. *Expert Systems, Principles & Case Studies*, Chapman & Hall, 1984.

Dougherty E.R., Giardina C.R., *Mathematical Methods for Artificial Intelligence and autonomous systems*, Prentice-Hall, 1988.

Genesereth M.R., Nilsson N.J., *Logical Foundations of Artificial Intelligence*, Morgam Kaufmann, 1987.

Gill K.S., *Artificial Intelligence For Society*, John Wiley & sons, 1986.

Ginsberg M.L., Reinfrank M., de Kleer J., *Lecture Notes in AI, Non-monotonic Reasoning*, Proceedings of 2nd International Workshop Grassau, FRG, Springer-Verlag 1989.

Goos G., Hartmanis J., *Lecture Notes in Computer Science, Fundamentals of Artificial Intelligence*, An advanced course, Springer-Verlag, 1986.

Grady P.O., *Process Planning & Design For Manufacture*, International Journal of CIM, Volume 4(2), Taylor & Francis, 1991

Groover M.P., *Automation, Production Systems and Computer-Aided Manufacturing*, Prentice Hall, 1980.

Ham I, Hitomi K., Yoshida T., *Group Technology*, Kluwer Nijhoff Publishing, 1985.

Harel D., *The Science of Computing, Exploring the nature and power of algorithms*, Addison-Wesley, 1989.

Hawley R., *AI programming environments*, Ellis Horwood Ltd, 1987.

Hayes-Roth F., Waterman D.A., *Building Expert Systems, Volume 1*, Addison Wesley, 1983.

Jackson P., *Introduction to Expert Systems*, Addison Wesley, 1986.

Kennedy J., et al, *Manufacturing Intelligence Market Surveys and Applications*, Department of Trade and Industry, MIT Division, 1989.

Kidd A.L. *Knowledge Acquisition for expert systems, A practical handbook*, Plenum Press, New york, 1987.

R. Kowalski, *Logic for problem solving*, North-Holland, 1979.

Kurzweil R., *The Age of Intelligent Machines*, The MIT Press, 1990.

Matsushima K., et al, 1982, *The integration of CAD and CAM by application of artificial intelligence techniques*, Annals of the CIRP, 31(1).

Meritt D., *Building Expert Systems in Prolog*, Springer-Verlag, 1989.

Myer N.L., *Group Technology at work*, Society of Manufacturing Engineers, Dearborn, Michigan, 1984.

Nilsson N.J., *Principles of Artificial Intelligence*, Tioga Publishing, 1980.

Nilsson N.J., Problem solving methods in Artificial Intelligence, Computer Science Series, McGraw Hill, 1971.

O'shea T. Eisenstadt M., *Artificial Intelligence, Tools, Techniques and Applications*, Harper & Row Publishers, 1984.

Penrose R., The *Emperor's New Mind, concerning computers, minds and the laws of physics*, Vintage, 1990.

Pollock J.L., *How to build a person, A Prolegomenon*, MIT Press 1989.

Ramani S., Chandrasekar R., Anjaneyulu K.S.R., *Lecutre Notes in Artificial Intelligence, Knowledge Based Computer Systems*, Springer-Verlag 1990.

Rich E., *Artificial Intelligence*, McGraw Hill, 1983.

Richard T., *Clausal Form Logic, An Introduction to the logic of computer reasoning*, International Computer Science Series, Addison-Wesley, 1989.

Sacerdoti E. D., *A structure for plans and behaviour*, Elsevier, New York, 1977.

Schank R.C., Colby K.M., *Computer Models of thought and language*, W H Freeman & Co., 1973.

Seikmann J. (ed), *Lecture Notes in Artificial Intelligence, Artificial Intelligence in Higher* Education, Proceedings of CEPES-UNESCO International symposium, Prague, CSFR, October 1989,, Springer-Verlag, 1987.

Simons G.L., *Towards 5th Generation Computers*, NCC Publishing Co, 1983.

Sterling L., Shapiro E., *The Art of Prolog, Advanced Programming Techniques*, The MIT Press, 1987.

Sussman G.A., *A computer model of skill acquisition*, Elsevier 1975.

Tulkoff J., CAPP, *Computer Aided Process Planning*, Manufacturing update series, Pub. Computer and automated systems, Association of SME, Dearborn, Michigan, 1985.

Waterman D.A., *A guide to expert systems*, Addison-Wesley, 1986.

Wilensky R., *Planning, Understanding. A computational approach to human reasoning*, Addison Wesley, 1983.

Winstanley G., *Artificial Intelligence in Engineering*, John wiley & sons, 1991.

Zozaya-Gorostiza, C Hendrickson, D R Rehak, *Knowledge-based process planning for construction and manufacturing*, Academic Press, 1989.

# APPENDIX I. FUNDAMENTALS OF THEOREM PROVING AND PROLOG

In order to understand how the resolution principle works, some definition and fundamentals of logic are given below. It works on a simpler version of Predicate Calculus known as Clausal Form Logic (CFL). CFL is a simpler version of Predicate Calculus which allows logical manipulations to be performed more easily on the computer.

## 1.1 Introduction to Clausal Form Logic

Instead of logical formulas/sentences, CFL consists of literals and clauses. A literal is an atom (or fact) that is negated or unnegated, e.g. *happy(james)* is a literal (meaning James is happy).
A clause is a set of literals.

**Definition:**

If *p1,...,pm and q1,....,qn* are atoms ($m,n >= 0$) then *{p1,...,pm}* => *{q1,...,qn}* is a clause.

where the atoms, *p1,...pm* are known as the antecedents of the clause and *q1,...,qn* are the consequents of the clause. This is rather like a conditional represented by an *if-then* statement.

So, a clause is a conditional when it says:

if its antecedents are all true
then so is its consequent atom.

e.g. *clean(james)=>happy(james).*
(if james is clean then james is happy)

## 1.2 Truth Values

The truth value of any clause (i.e. whether it is true or false) as a whole can be computed simply as a function of the truth values given to its atomic components. Conventionally, '1' is used to represent *true* and '0' to represent *false*.

### 1.2.1 Definition of truth for clauses

From the definition of a clause, it is true unless:

i) all its antecedents are true and

ii) all its consequents are false.

What this means is a clause is true:

i) if antecedent (*p1*) is true then so is the consequent.

ii) if several antecedents (*{p1,..,pm}*) are all true then so is the consequent.

The fundamental concept underlying this definition is that :

i) no statement is both true and false and

ii) every statement is either true or false

## 1.3 The Empty Antecedent Set

{ } => { q } ----- (1)

Suppose, we have a clause with no antecedents and *q* is the only consequent then (1) is true if *q* is true and false if *q* is false. This is the unconditional statement that q is true, i.e.

```
{ } => q
     1  1
     0  0
```

It is customary to write the truth values of atoms beneath the atoms concerned and the truth value written beneath '=>' is the truth value of the whole clause. Similarly, a clause with no antecedents but several consequents is the unconditional statement that one or more of these consequents is true, i.e.

{ } => {q1,....,qn} ------- (2)

means that one or more of q1,....,qn is true.

Incidentally, this is equivalent to asserting the disjunction of the consequents.

**Example 1.3**

The statement, *vincent loves james* can be represented by the clause:

{} => loves(vincent,james) (from (2))

## 1.4 The Empty Consequent Set

{*p*} => {} ----- (3)

For general case:

{*p1,....,pm*} => {} ----- (4)

If one or more of *p1,...,pm* is false then clause (4) is true. If all of *p1,...,pm* are true then the clause is false, i.e. not all of *p1,...,pm* are true. This is equivalent to: *p1,...,pm* => *false*, i.e. denying the conjunction of antecedents.

**Example 1.4**

i) James is not naughty can be written as:

*naughty(james)* => *{}*

In general, "not p" where p is an atomic statement is represented as: " p => {} ".

ii) If James is clean but not wet then he is happy can be written as:

*clean(james),not_wet(james) => happy(james).*

but *not_wet(james)*is not a part of a clause in CFL. (Recall that CFL can contain only atomic formulae and the implication sign from definition.) So to convert to CFL we get:

*clean(james) => wet(james);happy(james)*

This is the same as saying that: if james is clean then he is either wet or happy. To add 'not-p' to one side of a clause, add 'p' to the other side instead. This is known as the Rule of Negation by Transfer.

## 1.4.1 Symmetry of Rule of Negation by Transfer

e.g. *If James is happy then he is not wet* is represented as:

happy(james),wet(james) => {}

from: happy(james) => not_wet(james).

## 1.5 The Empty Clause

The empty clause is represented by '=>'. The truth value of '=>' is false. This is because:

i) there are no false antecedents (from 1.2.1) and similarly,

ii) there are no true consequents (from 1.2.1 ).

Hence by definition "=>" is false, i.e. {T} => {F} is false.

## 1.6 Variables

In order to expand the clauses to represent different individuals at different times, variables can be used. e.g. => *loves(X,james).* is the unconditional statement that for any object X, X loves james, i.e. everyone loves james.

### 1.6.1 Substitution For Variables

An instance of a clause is obtained by the uniform substitution of one or more variables in the clause by a term. If each variable is uniformly substituted by a constant term, then the result is called a ground instance.

e.g. son(X,vincent),father(Y,X) => parent(vincent,X).

Substitute X=james, Y=vincent we get:

son(james,vincent),father(vincent,james)

=>parent(vincent,james).

## 1.7 Proof Theory for CFL

Resolution uses two types of rules (for deduction):

i) Substitution Rule;

ii) Cut Rule.

### 1.7.1 The Substitution Rule

Any uniform substitution instance of a clause may be deduced from that clause, e.g.

parent(x,y,z) =>mother(x,z);mother(y,z)

This means that if x and y are the parents of z, then x or y is the mother of z.

**Deduction:**

parent(x,y,z) =>mother(x,z);mother(y,z) --- (1)

x=yen, parent(yen,y,z) =>mother(yen,z);mother(y,z) y=vincent, parent(yen,vincent,z)

=> mother(yen,z);mother(vincent,z)

z=james, parent(yen,vincent,james)

=> mother(yen,james);mother(vincent,james)

This rule is not limited to the substitution of variables by constants; it is also possible to use the rule to substitute for other variables as well.

## 1.7.2 The Cut Rule

**Definition:**

If $P_1P => Q_1$ and

$P_2 => P\,Q_2$

then $P_1P_2 => Q_1Q_2$

where $P_1, P_2, Q_1, Q_2$ are sets of atomic formulae and $P$ is an atomic formula, i.e. cut out an atom P occuring on opposite sides of two clauses and merge the resultant clauses into a new one.

## 1.7.3 Using the Substitution Rule and Cut Rule Together

**Example:**

| | |
|---|---|
| Assertions: | Anybody who is somebody's father is male. --- (1) |
| | vincent is the father of James. --- (2) |
| Derive: | vincent is male --- (3) |

From (1) father(X,Y) => male(X) --- (4)

From (2) => father(vincent,james) --- (5)

Deduction:

From (4) & using substitution rule: {X = vincent}, {Y = james},

father(vincent,james) => male(vincent) --- (6)

From (5),

=> father(vincent,james) --- (7)

From (6) & (7) & using the cut rule,

=> male(vincent).

Therefore, vincent is male.

The proof for the above example can be displayed as a tree as shown in Figure a1.1.

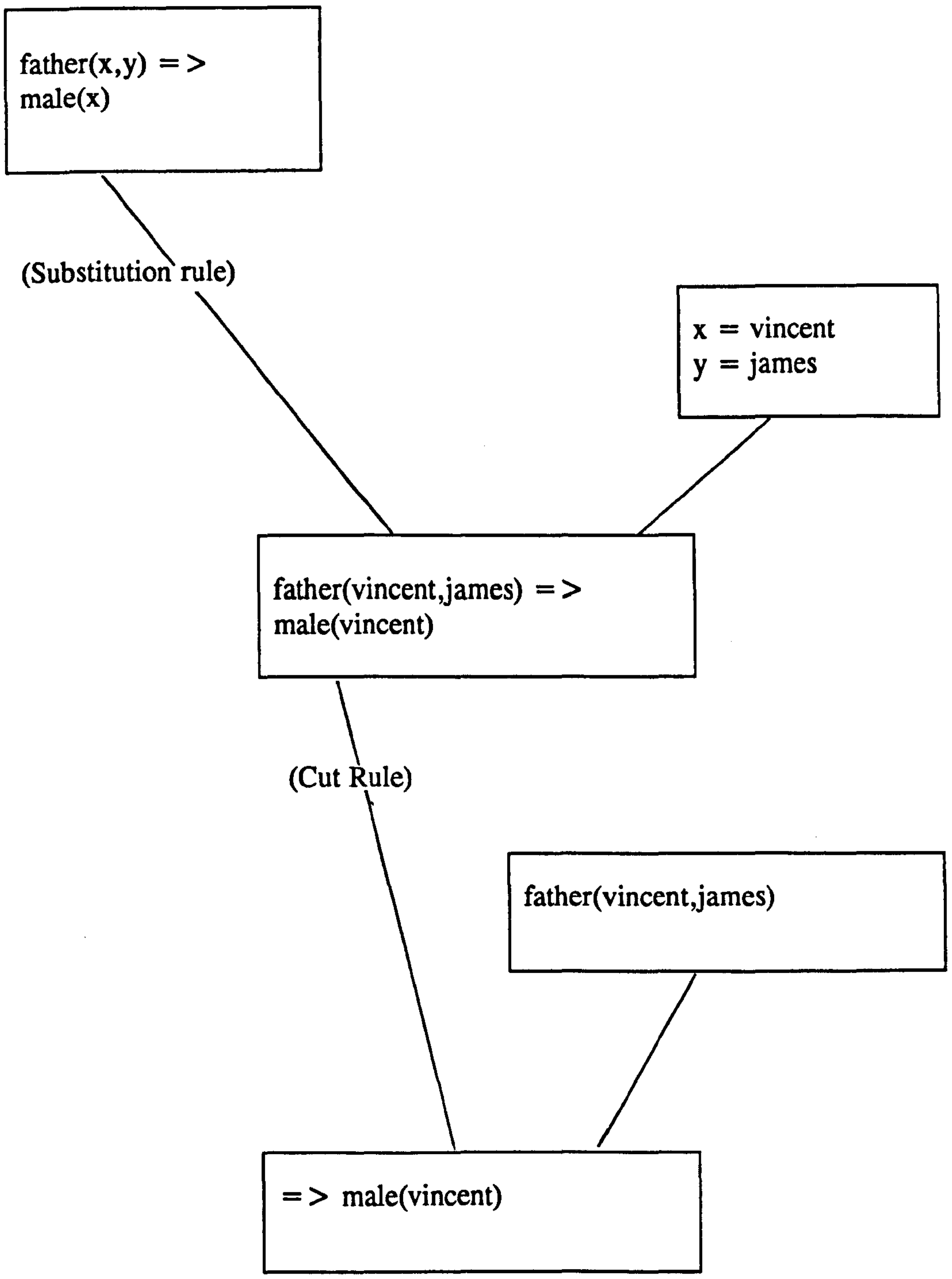

**Figure a1.1 Proof Tree for e.g. 1.7.3**

The method of the deduction in the above example is also known as forward chaining. The proof proceeds by working forwards from the premisses until the conclusion is reached. The refutation method also known as backward chaining proceeds by denying the conclusion (or goal) and then apply resolution backwards from that denial until some contradiction with the premisses is reached.

So using the same example, we negate the conclusion first and the proof tree as shown in Figure a1.2. We end up with the empty clause, which is logically false. Hence by applying the rules of resolution we have deduced something that is false. This means that, in any interpretation[1] that we choose, one or more of our original premisses must have been false. But we cannot reject the premisses that we begin with because we are trying to show that if they are true, then so is the conclusion. Hence our assumption, the denial of the conclusion must be wrong and so the conclusion must be true.

## 1.7.4 Advantages of Refutation

Even though the forward chaining approach seems more straight forward, the refutation method is the widely adopted approach to automated reasoning. The main reason for this is that backward chaining is much more focussed than forward chaining and involves less backtracking out of unprofitable or unpromising partial proofs.

[1] An interpretation is a particular specification (or instance) of the objects in a world (or problem).

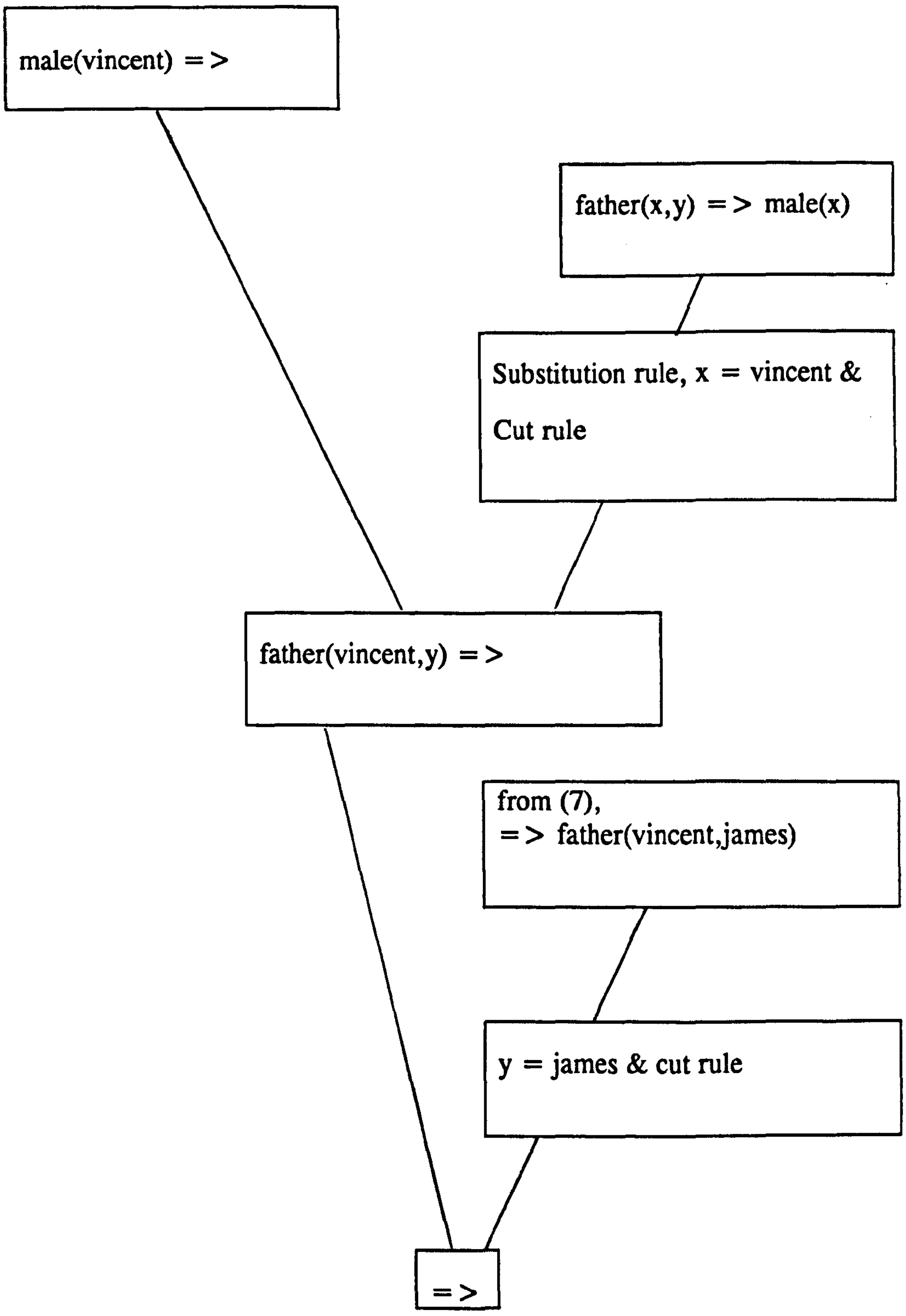

**Figure a1.2 Refutation Graph for e.g. 1.7.3**

## 1.8 Prolog

Prolog is said to be a cut-down (a reduced version of Predicate Calculus) logic machine designed to perform logical deductions using the resolution principle. Clauses that are entered into the database in Prolog have exactly one consequent, and zero or more antecedents. Clauses with one consequent are called positive Horn clauses and those with no consequents are called negative Horn clauses.

**Example:**

*b & d & c => a v b* can be written as two separate clauses:

*b & d & c => a*

*b & d & c => b*

where & means and, *v* means or.

These can be written in Prolog as:

*a :- b & d & e.*

*b:- b & d & e.*

(From the standard notation of Prolog: *conclusion :- conditions.*)

The standard notation for conjunctions in Prolog is the comma and for disjunction it is the semi-colon. For example,

```
goal:- sub_goal(a),
       sub_goal(b),
       sub_goal(c).
```

The above example means that to achieve goal, sub_goal(a) and sub_goal(b) and sun_goal(c) must be achieved. Similarly for the example below:

```
goal:-  (sub_goal(a);
         sub-goal(b);
         sub_goal(c)).
```

This means that to achieve goal, either one of sub|_goal(a), sub_goal(b) or sub_goal(c) has to be true. Any words that are written in upper case are treated as variables[2] by Prolog. Goals and constants begin with a lower case letter. Unlike conventional programming languages, once *instantiated* (initialise in conventional programming languages), variables in Prolog cannot be changed.

The most useful data structure is lists and the elements in a list are enclosed by *[* and *]* respectively. An example of a list is:

*[clear(pin,whole),clear(rod,whole].*

Notice that the elements are separated by commas. It can also be written as:

*[clear(pin,whole)|clear(rod,whole].*

The first element in the list is called the *head* while the remaining elements (recursively the rest of the list) in the list is called the *tail.* The symbol, |

2 The underscore, _, is used in Prolog as an annonymous variable , if it is not used elsewhere in the clause. This is to save having to think of different names.

is used to separate the *head* from the *tail*. In the above list, there is only one tail in the list, i.e. *clear(rod,whole)*.

Each prolog rule or fact is terminated by a period. The order of rules determines the order in which solutions are found. Changing the order of rules in a program would cause a different order of traversal of the search tree and a different order of finding solutions. This is because Prolog works in a depth-first manner. Consider the example below which is used to find whether an element is a member of a list:

*member(X,[X|Xs]).*
*member(X,[Y|Ys]):-member(X,Ys).*

As it is, the program will search the list until the desired element is found. If the order of these clauses are reversed then the program will always search to the end of the list. Incidentally, the order of solutions will also be affected. For example, if the query is: member(X,[a,b,c]. In the above program, the solutions will be:

X=a,X=b,X=c.

In the reversed version, the solutions will be: X=c,X=b,X=a.

If the search of a goal in a program contains an infinite branch then the program will not terminate. Non termination arises with recursive rules (which are often a feature of Prolog programs). An example of non termination is given below:

*Clauses:*

```
married(A,B):-married(B,A).
married(john,mary).
```

*Trace*[3]*:*

```
married(john,mary)
married(mary,john)
married(john,mary)
...........
```

The order of goals in a Prolog program determines the search tree. Unlike rule order, goal order affects the amount of searching that a program does in solving a query by determining which search tree is traversed. So, changing the goal order will change the search tree. For example,

```
Clause i.     son(X,Y):-
                     male(X),
                     father(Y,X).

Clause ii.    son(X,Y):-
                     father(Y,X),
                     male(X).
```

[3] Trace is a debugging facility of standard Prolog implementations which shows the programmer what it is doing at each call to the program.

| | | |
|---|---|---|
| *Facts:* | *father(ken,andrew).* | *male(andrew).* |
| | *father(ken,mary).* | *female(mary).* |
| | *father(ken,lucy).* | *female(lucy).* |
| | *father(vincent,james).* | *male(james).* |

Suppose the query is: son(P,ken), i.e. find who is the son of ken. With clause i., the search tree is given in Figure a1.3.

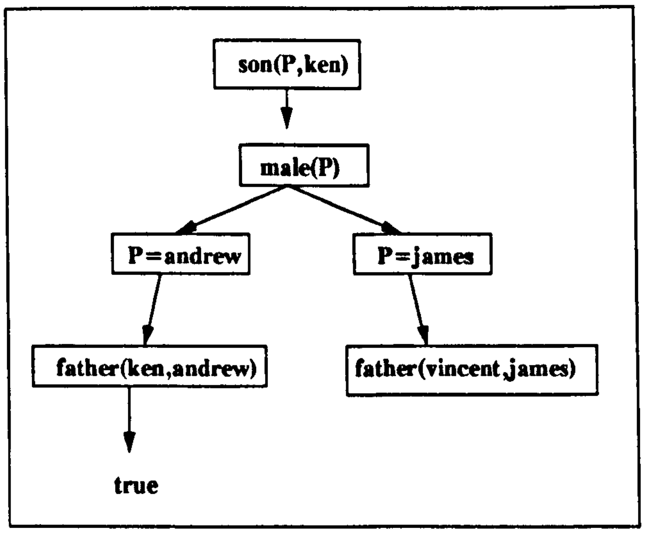

Figure a1.3 A Search Tree Of Clause i.

The search tree for the clause ii. is given in Figure a1.4.

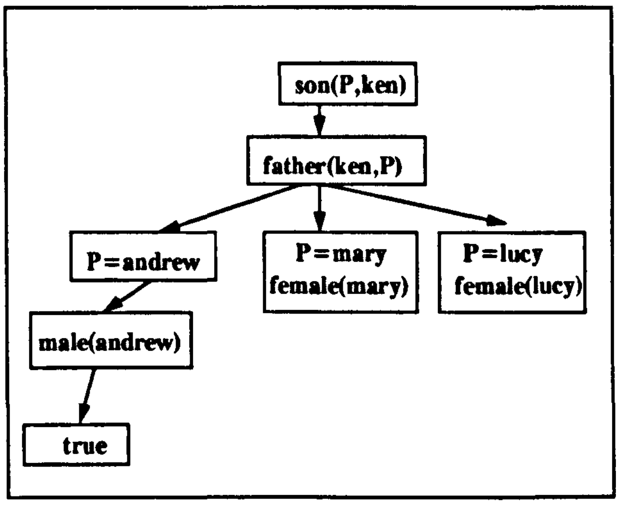

**Figure a1.4 A Search Tree Of Clause ii.**

Besides being a declarative language (defining rules and facts explicitly), Prolog is also considered to be procedural because of the its behaviour when ordering clauses and goals. This means that the execution of Prolog rules proceeds in a sequential manner according to the manner in which they have been ordered. Those that come first will be satisfied first. It works in a depth-first manner with backtracking when a sub-goal fails. A way in which the sequential order could be changed is by using the *cut* facility.

*Cut* is a system predicate for reducing the search space of Prolog programs by dynamically pruning the search tree. It can also be used purposely to prune paths that contain contain solutions in order to achieve a weak form of negation. The following clauses are used to represent a limited form of negation called negation as failure.

```
not(X):-X,!,fail.
not(X).
```

Having said earlier that changing the rule order only changes the order of solutions, in the above definition for negation, the rule order is essential for the intended meaning of negation. The cut-fail combination is used to allow early failure, i.e. to say that the search will not proceed. Cuts must therefore be used with caution otherwise the intended meaning of clauses with be affected. Further details of Prolog can be found in [2,3].

## References

[1] Richards T., *Clausal Form Logic, An Introduction to the logic of computer reasoning,* Addison Wesley, 1989 pp 273 - 283.

[2] L. Sterling, E. Shapiro, The Art of Prolog Advanced Programming Techniques, MIT Press, 1987.

[3] W.F. Clocksin, C.S. Mellish, Programming in Prolog, Springer-Verlag, 1986.

# APPENDIX II. A LIST OF SOME PROCESS PLANNING SYSTEMS

| SYSTEM | APPROACH | APPLICATION |
|---|---|---|
| APPAS [1] | Generative | Machining: milling,drilling |
| AUTAP [2] | Generative | Machining: rotational |
| AUTOCAP [3] | Variant | Machining: prismatic parts |
| CADCAM (extension of APPAS) [4] | Generative | Machining: hole making |
| CAP [5] | Variant | Machining: sheet |
| CAPP [6] | Variant | Machining: general |
| CAPP-I [7] | Variant | Machining: rotational |
| CIMS/PRO [5] | Generative | Machining: prismatic parts |
| COPICS [8] | Variant | Machining: general |
| CUTPLAN [9] | Variant | Machining: rotational, prismatic |
| EXCAP [10] | Generative | Machining: rotational |
| EXCAPP [11] | Generative | Machining: rotational |
| GARI [12] | Generative | Machining: general |
| GENPLAN [9] | Semi-generative | Machining: general |
| GT-CAPP [13] | Variant | Machining: general |
| INTELLI-CAPP [9] | Generative | Machining: general |
| KAPPS [14] | Generative | Machining: rotational, prismatic |
| MIPLAN [15] | Variant | Machining: rotational, prismatic |
| MITURN [16] | Variant | Rotational |
| Multi-CAPP II [17] | Variant | Machining: general |
| OMS [18] | Variant | Machining: general |
| OPEX [19] | Generative | Machining: rotational |
| TIPPS [5] | Generative | Machining: milling, drilling |
| TOM [20] | Generative | Machining: rotational |

**Note:**

This is not a complete list of reported process planning systems. Further references can be found in [21]

## References

[1] R.A. Wysk, *An automated process planning and selection program: APPAS*, Ph.D. thesis, Purdue University, West Lafayette, Indiana, USA, 1977.

[2] W. Eversheim, B. Holz, *Computer aided programming of NC-machine tools by using the system AUTAP-NC*, Annals of the CIRP, 31(1), 1982.

[3] A.J. Wright, et al., *Integrated knowledge based systems for process planning components*, 19th CIRP International Seminar on Manufacturing Systems, Penn. State, USA, 1-2 June 1987.

[4] T.C. Chang, R.A. Wysk, *Integrating CAD and CAM through automated process planning*, International Journal of Production Research, 1984, pp 877-894.

[5] T.C. Chang, R.A. Wysk, *An Introduction to Automated Process Planning Systems*, Prentice-Hall, 1985.

[6] C.H. Link , *CAPP, CAM-I automated process planning system*, Proceedings of the 1976 NC Conference, CAM-I. Ico. Arlington, Texas, USA.

[7] W. Jiang, H. Xu, *Capp systems and applications in China*, 19th CIRP International Seminar on Manufacturing Systems, Penn. State, USA, 1-2 June 1987.

[8] W. Eversheim, J. Schulz, CIRP technical reports, *Survey of computer-aided process planning systems*, Annals of the CIRP, 35(2), 1986.

[9] J. Tulkoff, *Process Planning: An historical review and future prospects*, 19th CIRP International Seminar on Manufacturing Systems, Penn. State, USA, 1-2 June 1987.

[10] B.J. Davies, et al. *The integration of process planning with CADCAM including the use of expert systems*, Proceedings of the international Conference on CAPE, Edinburgh, UK, April 1986.

[11] P. Du, J. Liu J., *The use of expert system in computer-aided process planning*, Proceedings of the 7th PROLAMAT Conference, Dresden, GDR, 14-17 June 1988.

[12] Y. Descotte, J.C. Latombe, *GARI: A problem solver that plans how to machine mechanical parts*, IJCAI 7, Vancouver, August 1981 pp 766 - 772.

[13] A.H. Strohmeier, *Implementing computer-aided process planning*, Rockwell International case study, CIM Review, Fall 1987.

[14] K. Iwata, Y. Fukuda, *Represenatation of know-how and its application of machining reference surface in computer-aided process planning*, Annals of the CIRP, 35(10), 1986.

[15] A. Houtzeel, *The MICLASS system*, Proceedings of CAM-I's Executive Seminar Coding, Classification and Group Technology for automated planning, CAM-I, Arlington, Texas, USA, 1976.

[16] J. Koloc, *Miturn, A computer-aided production planning system for numerically controlled lathes*, Proceedings of the second international conference on product development and manufacturing technology, University of Strathclyde, UK, April 1971.

[17] OIR Product News, Multi-II Group Technology System, 1987.

[18] M. Haas, T.C. Chang, *A survey on the usage of computer aided process planning systems in industry*, Purdue University, USA, January 1987.

[19] Gams M., et al. OPEX-an expert system for CAPP. The 6th International Workshop on Expert Systems and their applications, Avignon, France 28-30 April 1986.

[20] K. Matsushim., et al., *The integration of CAD and CAM by application of artificial intelligence techniques*, Annals of the CIRP, 31(1) 1982.

[21] Leo Alting, H.C. Zhang, *Computer Aided Process Planning: the state-of-the-art survey*, International Journal of Production Research, Vol 27, No. 4. 1989, pp 553-585.

# APPENDIX III. DERIVATION OF SORTING FORMULA

When n = 2, i.e. 2 subgoals to be compared (2 sub-goals in goal list):

the 1st sub-goal is compared with the 2nd sub-goal once,

and the 2nd sub-goal is also compared with the 1st sub-goal once.

therefore total comparisons = 2.

Similarly, for n = 3, i.e. 3 sub-goals to be compared :
the 1st sub-goal is compared with the 2nd, then 3rd sub-goal once each time, i.e. number of comparisons = 2
the 2nd sub-goal is compared with the 3rd, then the 1st once each time, i.e. number of comparisons = 2,
Similarly for the 3rd sub-goal where number of comparisons = 2.
In the worst case, only one sub-goal is put into the toplist in this pass.
In the next pass there will be 2 goals, i.e. n = 2 and the number of comparisons in the second pass = 2.
Hence the total maximum number of comparisons = (2+2+2) + 2

= 3*2 +2 = 8

with number of passes = 2.

For n =4,
In 1st pass , number of comparisons = (3+3+3+3) = 4*3
In 2nd pass, with n = 3 => (2+2+2)
In 3rd pass, with n = 2 => 2
with (4-1) passes.

Hence for n subgoals,
number of passes = n-1,

1st pass => n*(n-1)

2nd pass => (n-1)*(n-2)

..........

(n-1) pass => 2

Therefore the maximum number of comparisons= n*(n-1)+(n-1)(n-2) + ..

....+(n-i+1)(n-i)+...+2

$$= \sum_{i=1 \text{ to } n-1}^{n>=2} (n-i+1)(n-i)$$

where n >= 2 and 1<= i <n.

# APPENDIX IV. TEST PROGRAM FOR CONNECTING-ROD SUB-ASSEMBLY

```
now:-
        abolish(world,1),
        abolish(action,1),
        asserta(world(cross(a1,b1,piston))),
        asserta(world(cross(b1,a1,piston))),
        asserta(world(clear(a1,piston,in))),
        asserta(world(clear(a2,rod,out))),
        asserta(world(clear(b1,rod,in))),
        asserta(world(common(b1,piston,rod))),
        asserta(world(common(b1,rod,piston))),
        asserta(world(inside(b2,pin,b1,piston))).


goal(V):-
        now,
        do(V),
        listing(action).

do([X|Y]):-
        (setof(X,world(X),Y1);
         state(X)),
        do(Y).

do([]):-!.

/*---------------------- */
/*Post-conditions      */
/*---------------------- */
state(inside(V,W,X,Y)):-
        result(pushin(V,W,X,Y)),
        retract(world(clear(V,W,out))),
        retract(world(clear(X,Y,in))),
        asserta(world(inside(V,W,X,Y))).

state(clear(X,Y,in)):-
        result(pushout(V,W,X,Y)),
        retract(world(inside(V,W,X,Y))),
        retract(world(clear(X,Y,in))),
        asserta(world(clear(V,W,out))).

state(clear(X,Y,out)):-
        result(pushout(X,Y,V,W)),
        retract(world(inside(X,Y,V,W))),
        retract(world(clear(X,Y,out))),
        asserta(world(clear(V,W,in))).

/*---------------------- */
/*Pre-conditions        */
/*---------------------- */
result(pushin(V,W,X,Y)):-
        ((setof(clear(X,Y,in),world(clear(X,Y,in)),W1),
         W1=[clear(X,Y,in)];

result(pushout(V,W,X,Y)):-
        setof(inside(V,W,X,Y),world(inside(V,W,X,Y)),X1),
        X1 =[inside(V,W,X,Y)],
        setof(common(S,W,Y),world(common(S,W,Y)),S1),
        S1 = [common(S,W,Y)],
        state(clear(S,W,out)),
        assertz(action(pushout(W,Y))).

result(pushout(V,W,X,Y)):-
        setof(inside(V,W,X,Y),world(inside(V,W,X,Y)),X1),
        X1 = [inside(V,W,X,Y)],
        assertz(action(pushout(W,Y))).
```

## APPENDIX V. SAMPLE RUNS FOR TEST PROGRAM

```
phoenix% prolog

Edinburgh Prolog, version 1.5      (1st June 1987)
AI Applications Institute, University of Edinburgh

| ?- [pin2].
Warning: singleton variable Y1 in procedure do/1
Warning: singleton variable Z in procedure result/1
Warning: singleton variable P in procedure checkcross/3

pin2 consulted:  3784 bytes      3.15 seconds

yes

| ?- goal([inside(a2,rod,a1,piston),inside(b2,pin,b1,piston)]).

action(pushout(pin,piston)).

action(pushin(rod,piston)).

action(pushin(pin,piston)).


yes
| ?- goal([inside(b2,pin,b1,piston),inside(a2,rod,a1,piston)]).

action(pushout(pin,piston)).

action(pushin(rod,piston)).


yes
```

# APPENDIX VI. PROGRAM LISTING FOR AAP

```
/*---------------------------------------------------------------------*/
/* AN AUTOMATIC  ASSEMBLY   PLANNER                                    */
/*---------------------------------------------------------------------*/

/*---------------------------------------------------------------------*/
/*  Goal state of  primary shaft :                                     */
/*       assemble(bear2,whole,shaft,face2,shaft,face3),                */
/*       assemble(bear3,whole,shaft,face3,shaft,face1).                */
/*                                                                     */
/*  Goal state of con-rod:                                             */
/*       assemble(pin,whole,piston,t_hole,nil,nil)                     */
/*       assemble(rod,whole,piston,axial_hole,nil,nil)                 */
/*                                                                     */
/* Goal states of ball point pen :-                                    */
/*assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening)*/
/*assemble(stopper,projection,pen_barrel,top_opening,nil,nil).     */
/*assemble(cap,hole,pen_barrel,bottom_end,nil,nil).                 */
/*---------------------------------------------------------------------*/

/*initial:-
        recorda(world,clear(cap,hole),_),
        recorda(world,clear(refill,body),_),
        recorda(world,clear(stopper,projection),_),
        recorda(world,clear(pen_barrel,hole),_),
        recorda(world,clear(pen_barrel,top_opening),_),
        recorda(world,clear(pen_barrel,bottom_opening),_),
        recorda(world,clear(pen_barrel,bottom_end),_),
        recorda(world,at(pen_barrel,bottom_opening,bottom_end),_),
        recorda(world,clear(nil,nil),_).
*/


initial:-
        recorda(world,clear(bear2,whole),_),
        recorda(world,clear(bear3,whole),_),
        recorda(world,clear(shaft,face2),_),
        recorda(world,clear(shaft,face3),_),
        recorda(world,clear(shaft,face1),_),
        recorda(world,clear(rod,whole),_),
        recorda(world,clear(piston,axial_hole),_),
        recorda(world,assemble(pin,whole,piston,t_hole,nil,nil),_),
        recorda(world,cross(piston,t_hole,axial_hole,ta_area),_),
        recorda(world,common(piston,axial_hole,rod,se_hole,ta_area),_),
        recorda(world,clear(nil,nil),_),
        recorda(world,clear(cap,hole),_),
        recorda(world,clear(refill,body),_),
        recorda(world,clear(stopper,projection),_),
        recorda(world,clear(pen_barrel,hole),_),
        recorda(world,clear(pen_barrel,top_opening),_),
        recorda(world,clear(pen_barrel,bottom_opening),_),
        recorda(world,clear(pen_barrel,bottom_end),_),
        recorda(world,at(pen_barrel,bottom_opening,bottom_end),_).


erase_all_records:-
        erase_all(newgoal),
        erase_all(currentgoal),
        erase_all(world),
        erase_all(star),
          erase_all(action).


/*-----------------------------------------------*/
/*      Eliminate Redundant Subgoal             */
/*-----------------------------------------------*/
/*-----------------------------------------------------*/
/* redund_critic: check each subgoal in turn by 'test',*/
/*      eventually newgoal                            */
/*      contains no redundant subgoal.                */
/*-----------------------------------------------------*/

redund_critic([V1|V2],Conf):-
                        erase(newgoal,V1),
                findall(X,recorded(newgoal,X,_),V),
                        test(V1,V,Conf,_),
                        ((var(Conf),
                          redund_critic(V2,Conf));
                        !).

redund_critic([],_):-!.
```

```
/*-----------------------------------------------------*/
/*test:use to test if a subgoal is part of post-conditions*/
/*of other                                              */
/*subgoals. If so, there is no need to insert this     */
/*subgoal into goal list.                               */
/*Else insert to end of goallist                        */
/*-----------------------------------------------------*/
test(V1,[],_,Redund):-
                var(Redund),
                recordz(newgoal,V1,_),
              !.

test(_,[],_,redundant):-!.
test(V1,[V2|V3],Conflict,Redund):-
              cond(V1,V_pre,V_post),
              cond(V2,W_pre,W_post),
              conflict_check(V1,V_pre,W_pre,V_post,W_post,Conflict),
              ((var(Conflict),
               ((member(V1,W_post),
                 Redund = redundant,
                 test(V1,V3,Conflict,redundant)
               );
              (
                test(V1,V3,Conflict,Redund)
              )));
               !).

/*-----------------------------------------------------------------*/
/* addition:checks if precondition of subsequent subgoals already */
/*      true in world state                                        */
/*      put into star list if not true i.e. put a star in          */
/*      front of that subgoal as indicator                         */
/*-----------------------------------------------------------------*/
addition([]):-!.

addition([V1|V2]):-cond(V1,Pre1,_),
              checkexist(Pre1),
              addition(V2).

checkexist([]):-!.

checkexist([P1|P2]):-
               findall(P,recorded(world,P,_),W),
               ( member(P1,W);
                   (findall(Q,recorded(star,Q,_),X),
                    ( member(P1,X);
                      recordz(star,P1,_)))),
                 checkexist(P2).

/*----------------------------------------*/
/*     Resolve conflicts critic           */
/*----------------------------------------*/

/*-----------------------------------------------------------*/
/*sorting: if Conflict is uninstantiated, then no            */
/* invalid goals so can proceed with this                    */
/*test.  Boundary case when currentgoal                      */
/*is [], then can assert(z) HL (because                      */
/*length of HL is 1)                                         */
/*otherwise > 1 goal in toplist i.e. //                      */
/*goals so assert(z) par, goal then end.                     */
/*-----------------------------------------------------------*/

sorting:-
                 findall(N,recorded(newgoal,N,_),P),
                 P \== [],
                 erase_all(toplist),
                 order_critic(P),
                 findall(L,recorded(toplist,L,_),HL),
                 length(HL,Length),
                 findall(C1,recorded(currentgoal,C1,_),C),
                 ((  C \== [],
                 linking(HL,C),
                 erase_list(checklist,HL),
                 findall(TL,recorded(temp_checklist,TL,_),TL1),
                 record_list(checklist,TL1),
                 erase_list(temp_checklist,TL1));
                 ((Length = 1,
                 [HL1]=HL,
                 HL2=..[seq,HL1],
                 recordz(currentgoal,HL2,_));
                 (HL1=..[par|HL],
                 recordz(currentgoal,HL1,_)))),
                 record_list(temp_checklist,HL),
                 erase_list(newgoal,HL),
                 sorting.

sorting:-!.
```

```
erase(Key,Term):-
        recorded(Key,Term,Ref),
        erase(Ref).


/*--------------------------------------*/
/* findall:finds all the data stored   */
/*in the named database.               */
/*--------------------------------------*/
findall(Term,Generator,_):-
          asserta(found(mark)),
          call(Generator),
          asserta(found(Term)),
          fail.

findall(_,_,Answer):-
          collect_found([],Ansofar),
          !,
          Answer = Ansofar.

collect_found(Ans,Answer):-
          getnext(X),
          !,
          collect_found([X|Ans],Answer).

collect_found(Ans,Ans).

getnext(Term):-
          retract(found(Term)),
          !,
          Term \== mark.

record_list(_,[]):-!.

record_list(Key,[T1|T2]):-
         recordz(Key,T1,_),
         record_list(Key,T2).

erase_list(_,[]):-!.

erase_list(Key,[T1|T2]):-
         recorded(Key,T1,Ref),
         erase(Ref),
         erase_list(Key,T2).
```

```
list_all(Key):-
      findall(X,recorded(Key,X,_),Answer),
      nl,
      write('A view of database:  '),
      write(Key),
      nl,
      nl,
      pp([Answer],3).

/*----------------------------------------------*/
/* prints a list with each element per line    */
/*----------------------------------------------*/

pp([H|T],I):-
        !,
      J is I+3,
      pp(H,J),
      ppx(T,J),
      nl.

pp(X,I):-
      tab(I),
      write(X),
      write('.'),
      nl.

ppx([],_).

ppx([H|T],I):-
      pp(H,I),
      ppx(T,I).

/*--------------------------------------*/
/* link toplist to currentgoal list     */
/*--------------------------------------*/
linking([HL1|HL2],[C]):-
        C=..[F|_],
        link(HL1,C,Newargs,Linked),
        Linked == true,
        New_c =..[F|Newargs],
        recordz(checklist,HL1,_),
        linking(HL2,[New_c]).

linking([],[C]):-
        erase_all(currentgoal),
        recordz(currentgoal,C,_),
        !.

/*---------------------- */
/* case 1 : seq(a)      */
/*---------------------- */
link(HL1,C,Newargs,Linked):-
        C=..[F,Arg1],               /* f(arg1) */
        F == seq,                   /*seq(Arg1)*/
        ((check_single_goal(Arg1),  /* seq(a)  */
          ((findall(Arg1,recorded(checklist,Arg1,_),[]),
               sequence_test1(HL1,Arg1),
           Newargs = [Arg1,HL1], /* seq(a,b) */
           Linked = true,
           !);
          Linked = false));
        (link(HL1,Arg1,Newarg1,Linked),  /*Arg1 is compound goal*/
         Arg1 =..[F1|_],
         Newarg2=..[F1|Newarg1],
           Newargs = [Newarg2])).

/*---------------------- */
/*case 2 : par(a)       */
/*---------------------- */
link(HL1,C,Newargs,Linked):-
        C=..[F,Arg1],
        F==par,                        /* par(Arg1)   */
        ((check_single_goal(Arg1),
          ((findall(Arg1,recorded(checklist,Arg1,_),[]),
               sequence_test1(HL1,Arg1),
            Newarg2=..[seq,Arg1,HL1],
              Newargs = [Newarg2],
            Linked = true);
          Linked = false));
        (link(HL1,Arg1,Newarg1,Linked),
         Arg1=..[F1|_],
         Newarg2=..[F1|Newarg1],
           Newargs = [Newarg2])).

/*------------------------------ */
/*case 3 : seq(Arg1,........)  */
/*------------------------------ */
link(HL1,C,Newargs,Linked):-
        C=..[F|[Arg1|Args]],
        F==seq,
            ((check_single_goal(Arg1),
             ((findall(Arg1,recorded(checklist,Arg1,_),[]),
                  sequence_test1(HL1,Arg1),
               Args=[_|[]],
               ((check_single_goal(Args),
                 Newarg2=..[par,Args,HL1]);
                (Args=..[par|Temparg2],
          append(Temparg2,HL1,Newtemparg2),
          Newarg2=..[par|Newtemparg2])),
          Newargs=[Arg1,Newarg2],
            Linked = true);
            (C1=..[F|Args],
            link(HL1,C1,Newarg1,Linked),
            Newargs=[Arg1|Newarg1])));
            (link(HL1,Arg1,Newarg1,Linked),
            (Linked = true,
            Newargs = [Newarg1|Args]);
            (C1=..[F|Args],
            link(HL1,C1,Newarg2,Linked),
            Newargs = [Arg1|Newarg2]))).

/*---------------------------------------*/
/*case 4 : par(Arg1,............)      */
/*---------------------------------------*/
link(HL1,C,Newargs,Linked):-
         C=..[F|[Arg1|Args]],
         F==par,
         ((check_single_goal(Arg1),
           ((findall(Arg1,recorded(checklist,Arg1,_),[]),
                 sequence_test1(HL1,Arg1),
              Newarg1=..[seq,Arg1,HL1],
              Newargs=[Newarg1|Args],
              Linked = true);
             (C1=..[F|Args],
              link(HL1,C1,Newarg1,Linked),
              Newargs=[Arg1|Newarg1])));
           (link(HL1,Arg1,Newarg1,Linked),
            (Linked = true,
             Newargs=[Newarg1|Args]);
            (C1=..[F|Args],
             link(HL1,C1,Newarg2,Linked),
             Newargs = [Arg1|Newarg2]))).
```

```
/*--------------------------------------- */
/* check_single_goal(Arg):            */
/*  3 cases : Arg = inside(...), etc; */
/*      par(......);                  */
/*      seq(......).                  */
/*--------------------------------------- */
check_single_goal(A):- A=..[F|_],
                 F\==seq,
                 F\==par.

/*--------------------- */
/*not predicate       */
/*--------------------- */
not(P):- P,
       !,
       fail.

not(_).

sequence_test1(HL,Arg):-
        cond(HL,Pre_h1,_),
        cond(Arg,Pre_arg,Post_arg),
        intersect(Pre_h1,Pre_arg,I),
        filter(I,I1),
        ((I1\==[],
          subset(I1,Post_arg),!
          );
          (!,fail)).


sequence_test2(Pre_Arg1,S):-
        (cond(S,Pre_S,Post_S),
        intersect(Pre_Arg1,Pre_S,I),
        filter(I,I1),
        ((I1\==[],
          subset(I1,Post_S),
          !
         );
        (
         !,
         fail)
        )).


order_critic([]):-!.
order_critic([N1|N2]):-
                    erase(newgoal,N1),
                    findall(N,recorded(newgoal,N,_),P),
                    ordertest(N1,P,Flag_of_N1),
                      (( var(Flag_of_N1),
                         recordz(toplist,N1,_));
                         !),
                     recordz(newgoal,N1,_),
                     order_critic(N2).

ordertest(_,[],_):-!.

ordertest(N,[N1|N2],Flag_of_N):-
            cond(N,Pre_N,_),
            cond(N1,Pre_N1,Post_N1),
            intersect(Pre_N,Pre_N1,W),
             filter(W,W1),
             ((W1==[],
              ordertest(N,N2,Flag_of_N));
             (subset(W1,Post_N1),
              Flag_of_N = tail);
             ordertest(N,N2,Flag_of_N)).

/*------------------------------*/
/*to check conflicting goals   */
/*------------------------------*/
conflict_check(V1,Pre1,Pre2,Post1,Post2,Conflict):-
            intersect(Pre1,Pre2,W),
            filter(W,W1),
            ((( W1 == [];
                  subset(W1,Post1);
                  subset(W1,Post2)
                 ),
               (not(member(V1,Pre2));
                 member(V1,Post2)
               ),
              !);
              Conflict = conflict).

/*------------------------------------------------------------*/
/*to filter the state clear(nil,nil) before comparison.       */
/*------------------------------------------------------------*/
filter([],[]):-!.
filter([Wh|Wt],W):-
     (Wh==clear(nil,nil),
      filter(Wt,W));
     (filter(Wt,New_wt),
      ((New_wt\==[],
        W=[Wh|New_wt]);
       W=[Wh])).

/*--------------------------------------*/
/*to find the intersection of two sets */
/*--------------------------------------*/

intersect([X|R],Y,[X|Z]):-member(X,Y),
                          !,
                          intersect(R,Y,Z).

intersect([_|R],Y,Z):-intersect(R,Y,Z).

intersect([],_,[]).

/*------------------------------------------------*/
/*to check if X is a memeber of a list.          */
/*------------------------------------------------*/

member(X,[X|_]):-!.


member(X,[X1|_]):-
              ((X=assemble(Obj1,Face1,Obj2,Face2,_,_),
            X1=assemble(Obj2,Face2,Obj1,Face1,_,_));
                (X=cross(Obj1,Face1,Face2,C_area),
                 X1=cross(Obj1,Face2,Face1,C_area));
                 (X=common(Obj1,Face1,Obj2,Face2,C_area),
                  X1=common(Obj2,Face2,Obj1,Face1,C_area))),
            !.

member(X,[_|Y]):-member(X,Y).

/*------------------------------------------------------*/
/*to check if a list is a subset of the other list.     */
/*------------------------------------------------------*/
subset([],_):-!.

subset([A|X],Y):-member(A,Y),subset(X,Y).
```

```
append([],Ys,Ys):-!.

append([X|Xs],Ys,[X|Zs]):-
        append(Xs,Ys,Zs).

/*------------------------------------------------------*/
/*Order the sub-goals in goal list and star list       */
/*------------------------------------------------------*/

insert_star([],C):-
          erase_all(currentgoal),
          recordz(currentgoal,C,_),
          !.

insert_star([S1|S2],C):-
                C=..[F|_],
                reorder(S1,C,Newargs,_),
                Current_goals=..[F|Newargs],
                insert_star(S2,Current_goals).


/*------------ */
/*Case 1a      */
/*------------ */
reorder(S,C,Newargs,Linked):-
        C=..[F|[Arg1|[]]],
        check_single_goal(Arg1),
        cond(Arg1,Pre_arg1,_),
        ((member(S,Pre_arg1),
          Newargs=[Arg1],
          Linked = true,
          ! );
        ( sequence_test2(Pre_arg1,S),
          (( F == seq,
             Newargs = [S,Arg1]);
           ( F == par,
             Newargs = [seq(S,Arg1)])),
          Linked = true,
          !
        );
        ( Newargs = [Arg1],
          !
        )).
```

```
/*------------- */
/*Case 1b      */
/*------------- */
reorder(S,C,Newargs,Linked):-
        C=..[_|[Arg1|[]]],
        Arg1=..[F1|_],
        reorder(S,Arg1,Newarg1,Linked),
        Newarg2=..[F1|Newarg1],
        Newargs = [Newarg2].


/*---------------------- */
/* reorder - case 2a   */
/*---------------------- */

reorder(S,C,Newargs,Linked):-
        C=..[F|[Arg1|Args]],
        check_single_goal(Arg1),
        cond(Arg1,Pre_arg1,_),
        (( member(S,Pre_arg1),
          C=..[_|Newargs],
         Linked = true,
         !
        );
        ( sequence_test2(Pre_arg1,S),
          ( (F == seq,
            C=..[_|Newarg1],
            append([S],Newarg1,Newargs));
           (F == par,
            append([seq(S,Arg1)],Args,Newargs))),
        Linked = true,
         !
        );
        ( C1=..[F|Args],
          reorder(S,C1,Newarg1,Linked),
          append([Arg1],Newarg1,Newargs),
          !
        )).

/*---------------------- */
/* reorder - case 2b   */
/*---------------------- */

reorder(S,C,Newargs,Linked):-
        C=..[F|[Arg1|Args]],
        Arg1=..[F1|_],
        (( reorder(S,Arg1,Newarg1,Linked),
          Newarg2=..[F1|Newarg1],
          append([Newarg2],Args,Newargs),
          not(var(Linked)),
          !
        );
        ( C1=..[F|Args],
          reorder(S,C1,Newarg1,Linked),
          append([Arg1],Newarg1,Newargs),
          !
        )).


insert(W1,Z1,[Z2|Z3]):-
                    erase_all(currentgoal),
                      ((Z1\==Z2,
                     recordz(currentgoal,Z2,_),
                         insert(W1,Z1,Z3));
                     (recordz(currentgoal,W1,_),
                           addgoal([Z2|Z3]))).
insert(_,_,[]):-!.

addgoal([]):-!.

addgoal([Z2|Z3]):-
                recordz(currentgoal,Z2,_),
                   addgoal(Z3).

/*----------------------------------------------------*/
/*The pre- and post-conditions of the three states  */
/*----------------------------------------------------*/
cond(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),X,Y):-

findall(cross(Cr1,Cr2,Cr3,Cr4),recorded(world,cross(Cr1,Cr2,Cr3,Cr4),_),
 Cross),

findall(common(Co1,Co2,Co3,Co4,Co5),recorded(world,common(Co1,Co2,
 Co3,Co4,Co5),_),Common),


   ((member(cross(Obj1,Face1,Face,C_area),Cross),
   ((member(common(Obj1,Face,_,_,C_area),Common),
            X1=clear(nil,nil),
            Y1=clear(nil,nil));
            (X1=clear(Obj1,Face),
             Y1=clear(Obj1,Face))));
      (member(cross(Obj2,Face2,Face,C_area),Cross),
      ((member(common(Obj2,Face,_,_,C_area),Common),
            X1=clear(nil,nil),
            Y1=clear(nil,nil));
            (X1=clear(Obj2,Face),
             Y1=clear(Obj2,Face))))),
       X2=[X1,clear(Obj1,Face1),clear(Obj2,Face2),clear(Rxnobj,Rxnface)],
       filter(X2,X),

Y2=[Y1,assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),clear(Rxnobj,Rx
nface)],
            filter(Y2,Y),
            !.

cond(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),X,Y):-
            findall(at(A1,A2,A3),recorded(world,at(A1,A2,A3),_),AT),
            (((member(at(Obj1,Face1,Face11),AT),
                    X1 = clear(Obj1,Face11),
                    Y1 = clear(Obj1,Face11));
                    (X1 = clear(nil,nil),
                     Y1 = clear(nil,nil))),
            ((member(at(Obj2,Face2,Face12),AT),
                    X2 = clear(Obj2,Face12),
                    Y2 = clear(Obj2,Face12));
                    (X2 = clear(nil,nil),
                     Y2 = clear(nil,nil))),

            ((member(at(Rxnobj,Rxnface,Face13),AT),
                    X3 = clear(Rxnobj,Face13),
                    Y3 = clear(Rxnobj,Face13));
             (X3 = clear(nil,nil),
              Y3 = clear(nil,nil))),
            X4 =
[X1,X2,X3,clear(Obj1,Face1),clear(Obj2,Face2),clear(Rxnobj,Rxnface)],
            filter(X4,X),
            Y4 =
[Y1,Y2,Y3,assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),clear(Rxnobj,
Rxnface)],
            filter(Y4,Y)),!.
```

```
cond(assemble(Obj1,Loc1,Obj2,Loc2,Rxnobj,Rxnface),X,Y):-
   X1=[clear(Obj1,Loc1),clear(Obj2,Loc2),clear(Rxnobj,Rxnface)],
   filter(X1,X),

Y1=[assemble(Obj1,Loc1,Obj2,Loc2,Rxnobj,Rxnface),clear(Rxnobj,Rxnface
)],
   filter(Y1,Y),!.


cond(clear(Obj,Face),X,Y):-

findall(assemble(A1,A2,A3,A4,A5,A6),recorded(world,assemble(A1,A2,A3,A
4,A5,A6),_),Assemble),
        member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble),

findall(cross(Cr1,Cr2,Cr3,Cr4),recorded(world,cross(Cr1,Cr2,Cr3,Cr4),_),
Cross),

findall(common(Co1,Co2,Co3,Co4,Co5),recorded(world,common(Co1,Co2,
Co3,Co4,Co5),_),Common),


    ((member(cross(Obj,Face,Face3,C_area),Cross),
     ((member(common(Obj,Face3,_,_,C_area),Common),
        X1=clear(nil,nil),
        Y1=clear(nil,nil));
        (X1=clear(Obj,Face3),
         Y1=clear(Obj,Face3))));
    (member(cross(Obj1,Face1,Face3,C_area),Cross),
     ((member(common(Obj1,Face3,_,_,C_area),Common),
        X1=clear(nil,nil),
        Y1=clear(nil,nil));
        (X1=clear(Obj1,Face3),
         Y1=clear(Obj1,Face3))))),
        X2=[X1,assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface)],
        filter(X2,X),
        Y2=[Y1,clear(Obj,Face),clear(Obj1,Face1)],
        filter(Y2,Y),
        !.

cond(clear(Obj,Face),X,Y):-

findall(assemble(A1,A2,A3,A4,A5,A6),recorded(world,assemble(A1,A2,A3,A
4,A5,A6),_),Assemble),
       ((member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble)
,
    X1 = assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),
       Y1=clear(Obj,Face),
       Y2=clear(Obj1,Face1));
    (X1=assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),
       Y1=clear(Obj,Face),
       Y2=clear(Obj1,Face1))),
       findall(at(At1,At2,At3),recorded(world,at(At1,At2,At3),_),At),
       ((member(at(Obj,Face,Face2),At),
        X2=clear(Obj,Face2),
       Y3=clear(Obj,Face2));
       (member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble),
               member(at(Obj,Face,Face2),At),
               X2=clear(Obj1,Face2),
               Y3=clear(Obj1,Face2))),
               X=[X1,X2],
               Y=[Y1,Y2,Y3],
       !.

cond(clear(Obj,Face),X,Y):-
       findall(assemble(A1,A2,A3,A4,A5,A6),recorded(world,assemble(A1,A
2,A3,A4,A5,A6),_),Assemble),
       ((member(assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface),Assemble)
,
   X=[assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface)],
   Y=[clear(Obj,Face),clear(Obj1,Face1)]);
   (X=[assemble(Obj,Face,Obj1,Face1,Rxnobj,Rxnface)],
   Y=[clear(Obj,Face),clear(Obj1,Face1)])),
       !.


/*--------------------------------------------*/
/* planner: rearrange the goal list.          */
/* do : take actions to achieve each subgoal        */
/*listing : list the acitons                  */
/*--------------------------------------------*/
go:-
   erase_all_records,
   initial,
   frontend.

frontend:-
        write('Yes, Yen.....'),
        nl,
        write('Starting the planner.....'),
        nl,
        write('Type in the goal state(s).....'),
        nl,
        input_goals(Goals,Option),
        ((Option = stop, !);
         (Option = list,
         frontend);
         (Option = ok,
          goal(Goals),
            go)).

input_goals(Goals,Option):-
         read(G),
         ((G == stop,
           Option = stop);
          ((G == list,
            list_all(world),
            Option = list, !);
           ((G == ok,
             Option = ok,
             findall(Y,recorded(newgoal,Y,_),Goals));
            ((recorded(newgoal,G,_);
              recordz(newgoal,G,_)),
             input_goals(Goals,Option))))).


goal(V):-
        erase_all(checklist),
        erase_all(temp_checklist),
        erase_all(toplist),
        redund_critic(V,Conf),
        ((var(Conf),
         findall(X,recorded(newgoal,X,_),U),
         addition(U),
         !,
         sorting,
          findall(Y,recorded(currentgoal,Y,_),[Z]),
          findall(S,recorded(star,S,_),W),
          insert_star(W,Z),
          findall(T,recorded(currentgoal,T,_),[C]),
          nl,
          write('The Ordered Goal is: '),
          nl,
            write(C),
            nl,
            nl,
            do([C]),
            list_all(world),
            list_all(action));
           (nl,
            write('Conflicting goals cannot be achieved...'),
            nl,
            nl)).


/*-----------------------------------------------------*/
/* do : check if subgoal state already achieved,     */
/*else achieve the subgoal.                           */
/*state : achieve state of a subgoal                  */
/*-----------------------------------------------------*/

do([Y]):-
        Y =..[F,Arg1],
        (F == seq;
         F == par ),
        ((check_single_goal(Arg1),
        state(Arg1));
        do([Arg1])).


do([Y]):-
        Y=..[F|[Arg1|Args]],
        (F==seq;
         F==par),
        C1=..[F,Arg1],
        do([C1]),
        Cs=..[F|Args],
        do([Cs]).

/*-----------------------------------------------------*/
/* case1 : state already exists in current situation. */
/*-----------------------------------------------------*/
state(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)):-
    findall(W,recorded(world,W,_),W1),
    member(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),W1).
```

```
/*-------------------------------------------*/
/* case2 : take press action to assemble.    */
/*-------------------------------------------*/
state(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)):-
    result(press(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)).

/*-------------------------------------------------*/
/*case1: state already exists in current situation. */
/*-------------------------------------------------*/
state(clear(Obj,Face)):-
        findall(W,recorded(world,W,_),W1),
        member(clear(Obj,Face),W1).


/*-------------------------------------------*/
/*case2 : take remove action to clear.       */
/*-------------------------------------------*/
state(clear(Obj,Face)):-
    result(remove(Obj,Face,_,_)).

/*-------------------------------------------------*/
/*press : check if preconditions acheieved, else   */
/*      achieve preconditions.                     */
/*      assert action to action list.              */
/*      aintain world state interactively.         */
/*-------------------------------------------------*/
result(press(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface)):-
     cond(assemble(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),Pre,Post),
     Pre1=..[par|Pre],
        do([Pre1]),
     maintain_world_pre(Pre,Post),
     maintain_world_post(Post),
     recordz(action,press(Obj1,Face1,Obj2,Face2,Rxnobj,Rxnface),_).

/*-------------------------------------------*/
/* remove : check if either one case is true. */
/*      assert action to action list.        */
/*      if none of the case, then cut.       */
/*      maintain world state interactively.  */
/*-------------------------------------------*/
result(remove(Obj,Face,Obj1,Face1)):-
     cond(clear(Obj,Face),Pre,Post),
        Pre1=..[par|Pre],
        do([Pre1]),
     maintain_world_pre(Pre,Post),
      Pre = [assemble(Obj,Face,Obj1,Face1,_,_)],
    maintain_world_post(Post),
    recordz(action,remove(Obj,Face,Obj1,Face1),_).

/*------------------------------------------------------*/
/* maintain_world_pre & maintain_world_post :   */
/* to maintain the current world state after an    */
/* action has been taken.                            */
/*------------------------------------------------------*/
maintain_world_pre([],_):-!.

maintain_world_pre([Pre1|Pre2],Post):-
   (member(Pre1,Post);
        (
         (
          Pre1=..[assemble|Args],
          Args = [Obj,Face,Obj1,Face1,_,_],
          recorded(world,assemble(Obj,Face,Obj1,Face1,_,_),Ref),
                 erase(Ref)
        );
         (
          Pre1=..[assemble|Args],
          Args = [Obj,Face,Obj1,Face1,_,_],
          recorded(world,assemble(Obj1,Face1,Obj,Face,_,_),Ref),
                 erase(Ref)
        );

        (
               recorded(world,Pre1,Ref),
               erase(Ref)
        )
     )
   ),
   maintain_world_pre(Pre2,Post).

/*maintain_world_pre([],_).*/

maintain_world_pre([Pre1|Pre2],Post):-
   (member(Pre1,Post);
        (findall(W,recorded(world,W,_),W1),
         member(Pre1,W1),
        erase(world,Pre1))
   ),
    maintain_world_pre(Pre2,Post).
```

```
maintain_world_post([Post1 | Post2]):-
        findall(W,recorded(world,W,_),W1),
        member(Post1,W1),
        maintain_world_post(Post2).

maintain_world_post([Post1 | Post2]):-
      recordz(world,Post1,_),
      maintain_world_post(Post2).

maintain_world_post([]).
```

# APPENDIX VII. TEST RUNS FOR AAP

```
tern% eagle
eagle% cd planner
/home/eng4/es017/planner
eagle% prolog

Edinburgh Prolog, version 1.5      (1st June 1987)
AI Applications Institute, University of Edinburgh

| ?- [aap].


aap consulted:   25296 bytes      11.30 seconds

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(bear2,whole,shaft,face2,shaft,face3).
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  ok.

The Ordered Goal is:
seq
        (assemble(bear2,whole,shaft,face2,shaft,face3),
        assemble(bear3,whole,shaft,face3,shaft,face1))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        assemble(pin,whole,piston,t_hole,nil,nil).
        clear(piston,axial_hole).
        clear(rod,whole).
        clear(shaft,face1).
        assemble(bear2,whole,shaft,face2,shaft,face3).
        assemble(bear3,whole,shaft,face3,shaft,face1).


A view of database:  action

        press(bear2,whole,shaft,face2,shaft,face3).
        press(bear3,whole,shaft,face3,shaft,face1).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  ok.

The Ordered Goal is:
seq(assemble(bear3,whole,shaft,face3,shaft,face1))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        assemble(pin,whole,piston,t_hole,nil,nil).
        clear(piston,axial_hole).
        clear(rod,whole).
        clear(shaft,face1).
        clear(shaft,face2).
        clear(bear2,whole).
        assemble(bear3,whole,shaft,face3,shaft,face1).
```

```
A view of database:  action

        press(bear3,whole,shaft,face3,shaft,face1).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  assemble(bear2,whole,shaft,face2,shaft,face3).
|:  ok.

The Ordered Goal is:
seq
        (assemble(bear2,whole,shaft,face2,shaft,face3),
        assemble(bear3,whole,shaft,face3,shaft,face1))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        assemble(pin,whole,piston,t_hole,nil,nil).
        clear(piston,axial_hole).
        clear(rod,whole).
        clear(shaft,face1).
        assemble(bear2,whole,shaft,face2,shaft,face3).
        assemble(bear3,whole,shaft,face3,shaft,face1).


A view of database:  action

        press(bear2,whole,shaft,face2,shaft,face3).
        press(bear3,whole,shaft,face3,shaft,face1).
```

```
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(pin,whole,piston,t_hole,nil,nil).
|:  assemble(rod,whole,piston,axial_hole,nil,nil).
|:  ok.


The Ordered Goal is:
seq
        (assemble(rod,whole,piston,axial_hole,nil,nil),
        assemble(pin,whole,piston,t_hole,nil,nil))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        clear(shaft,face1).
        clear(shaft,face3).
        clear(shaft,face2).
        clear(bear3,whole).
        clear(bear2,whole).
        assemble(rod,whole,piston,axial_hole,nil,nil).
        assemble(pin,whole,piston,t_hole,nil,nil).
```

```
A view of database: action

        remove(piston,t_hole,pin,whole).
        press(rod,whole,piston,axial_hole,nil,nil).
        press(pin,whole,piston,t_hole,nil,nil).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(rod,whole,piston,axial_hole,nil,nil).
|:  assemble(pin,whole,piston,t_hole,nil,nil).
|:  ok.


The Ordered Goal is:

seq
        (assemble(rod,whole,piston,axial_hole,nil,nil),
        assemble(pin,whole,piston,t_hole,nil,nil))


A view of database: world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        clear(shaft,face1).
        clear(shaft,face3).
        clear(shaft,face2).
        clear(bear3,whole).
        clear(bear2,whole).
        assemble(rod,whole,piston,axial_hole,nil,nil).
        assemble(pin,whole,piston,t_hole,nil,nil).


A view of database:  action

        remove(piston,t_hole,pin,whole).
        press(rod,whole,piston,axial_hole,nil,nil).
        press(pin,whole,piston,t_hole,nil,nil).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
|:  assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
|:  assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
|:  ok.

The Ordered Goal is:
par(seq
        (assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening),
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil)),
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        assemble(pin,whole,piston,t_hole,nil,nil).
        clear(piston,axial_hole).
        clear(rod,whole).
        clear(shaft,face1).
        clear(shaft,face3).
        clear(shaft,face2).
        clear(bear3,whole).
        clear(bear2,whole).
        assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil).


A view of database:  action

        press(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        press(cap,hole,pen_barrel,bottom_end,nil,nil).
        press(stopper,projection,pen_barrel,top_opening,nil,nil).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
|:  assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
|:  assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
|:  ok.

The Ordered Goal is:
par
        (assemble(stopper,projection,pen_barrel,top_opening,nil,nil),
        seq(assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening)
,
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil)))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        assemble(pin,whole,piston,t_hole,nil,nil).
        clear(piston,axial_hole).
        clear(rod,whole).
        clear(shaft,face1).
        clear(shaft,face3).
        clear(shaft,face2).
        clear(bear3,whole).
        clear(bear2,whole).
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
        assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil).


A view of database:  action

        press(stopper,projection,pen_barrel,top_opening,nil,nil).
        press(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        press(cap,hole,pen_barrel,bottom_end,nil,nil).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
|:  assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
|:  assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
|:  ok.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
|:  assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
|:  assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
|:  ok.
```

```
The Ordered Goal is:
par
        (assemble(stopper,projection,pen_barrel,top_opening,nil,nil),
        seq(assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening)
,
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil)))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        assemble(pin,whole,piston,t_hole,nil,nil).
        clear(piston,axial_hole).
        clear(rod,whole).
        clear(shaft,face1).
        clear(shaft,face3).
        clear(shaft,face2).
        clear(bear3,whole).
        clear(bear2,whole).
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
        assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil).


A view of database:  action

        press(stopper,projection,pen_barrel,top_opening,nil,nil).
        press(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        press(cap,hole,pen_barrel,bottom_end,nil,nil).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
```

```
Starting the planner.....
Type in the goal state(s).....
|:  assemble(bear2,whole,shaft,face2,shaft,face3).
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  assemble(pin,whole,piston,t_hole,nil,nil).
|:  assemble(rod,whole,piston,axial_hole,nil,nil).
|:  ok.



The Ordered Goal is:
par(seq
        (assemble(bear2,whole,shaft,face2,shaft,face3),
        assemble(bear3,whole,shaft,face3,shaft,face1)),
        seq(assemble(rod,whole,piston,axial_hole,nil,nil),
        assemble(pin,whole,piston,t_hole,nil,nil)))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        clear(shaft,face1).
        assemble(bear2,whole,shaft,face2,shaft,face3).
        assemble(bear3,whole,shaft,face3,shaft,face1).
        assemble(rod,whole,piston,axial_hole,nil,nil).
        assemble(pin,whole,piston,t_hole,nil,nil).


A view of database:  action

        press(bear2,whole,shaft,face2,shaft,face3).
        press(bear3,whole,shaft,face3,shaft,face1).
        remove(piston,t_hole,pin,whole).
        press(rod,whole,piston,axial_hole,nil,nil).
        press(pin,whole,piston,t_hole,nil,nil).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(rod,whole,piston,axial_hole,nil,nil).
|:  assemble(pin,whole,piston,t_hole,nil,nil).
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  assemble(bear2,whole,shaft,face2,shaft,face3).
|:  ok.



The Ordered Goal is:
par(seq
        (assemble(rod,whole,piston,axial_hole,nil,nil),
        assemble(pin,whole,piston,t_hole,nil,nil)),
seq
        (assemble(bear2,whole,shaft,face2,shaft,face3),
        assemble(bear3,whole,shaft,face3,shaft,face1)))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        clear(shaft,face1).
        assemble(rod,whole,piston,axial_hole,nil,nil).
        assemble(pin,whole,piston,t_hole,nil,nil).
        assemble(bear2,whole,shaft,face2,shaft,face3).
        assemble(bear3,whole,shaft,face3,shaft,face1).


A view of database:  action

        remove(piston,t_hole,pin,whole).
        press(rod,whole,piston,axial_hole,nil,nil).
        press(pin,whole,piston,t_hole,nil,nil).
        press(bear2,whole,shaft,face2,shaft,face3).
        press(bear3,whole,shaft,face3,shaft,face1).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(bear2,whole,shaft,face2,shaft,face3).
|:  assemble(pin,whole,piston,t_hole,nil,nil)
|:  .
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  assemble(rod,whole,piston,axial_hole,nil,nil).
|:  ok.


The Ordered Goal is:
par(seq
        (assemble(bear2,whole,shaft,face2,shaft,face3),
        assemble(bear3,whole,shaft,face3,shaft,face1)),
seq(assemble(rod,whole,piston,axial_hole,nil,nil),
        assemble(pin,whole,piston,t_hole,nil,nil)))
```

```
A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(pen_barrel,top_opening).
        clear(pen_barrel,hole).
        clear(stopper,projection).
        clear(refill,body).
        clear(cap,hole).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        clear(shaft,face1).
        assemble(bear2,whole,shaft,face2,shaft,face3).
        assemble(bear3,whole,shaft,face3,shaft,face1).
        assemble(rod,whole,piston,axial_hole,nil,nil).
        assemble(pin,whole,piston,t_hole,nil,nil).


A view of database:  action

        press(bear2,whole,shaft,face2,shaft,face3).
        press(bear3,whole,shaft,face3,shaft,face1).
        remove(piston,t_hole,pin,whole).
        press(rod,whole,piston,axial_hole,nil,nil).
        press(pin,whole,piston,t_hole,nil,nil).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(rod,whole,piston,axial_hole,nil,nil).
|:  assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
|:  assemble(pin,whole,piston,t_hole,nil,nil).
|:  assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
|:  assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
|: ok.



The Ordered Goal is:
par(seq
        (assemble(rod,whole,piston,axial_hole,nil,nil),
        assemble(pin,whole,piston,t_hole,nil,nil)),
        seq(assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening)
,
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil)),
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil))


A view of database: world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        clear(shaft,face1).
        clear(shaft,face3).
        clear(shaft,face2).
        clear(bear3,whole).
        clear(bear2,whole).
        assemble(rod,whole,piston,axial_hole,nil,nil).
        assemble(pin,whole,piston,t_hole,nil,nil).
        assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil).


A view of database: action

        remove(piston,t_hole,pin,whole).
        press(rod,whole,piston,axial_hole,nil,nil).
        press(pin,whole,piston,t_hole,nil,nil).
        press(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        press(cap,hole,pen_barrel,bottom_end,nil,nil).
        press(stopper,projection,pen_barrel,top_opening,nil,nil).
```

```
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
| ?- go.
Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
|:  assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
|:  assemble(bear3,whole,shaft,face3,shaft,face1).
|:  assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
|:  assemble(bear2,whole,shaft,face2,shaft,face3).
|:  ok.

The Ordered Goal is:
par(
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil),
seq(assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening),
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil)),
seq(assemble(bear2,whole,shaft,face2,shaft,face3),
        assemble(bear3,whole,shaft,face3,shaft,face1)))


A view of database:  world

        at(pen_barrel,bottom_opening,bottom_end).
        clear(pen_barrel,bottom_opening).
        clear(nil,nil).
        common(piston,t_hole,rod,se_hole,ta_area).
        cross(piston,t_hole,axial_hole,ta_area).
        assemble(pin,whole,piston,t_hole,nil,nil).
        clear(piston,axial_hole).
        clear(rod,whole).
        clear(shaft,face1).
        assemble(stopper,projection,pen_barrel,top_opening,nil,nil).
        assemble(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        assemble(cap,hole,pen_barrel,bottom_end,nil,nil).
        assemble(bear2,whole,shaft,face2,shaft,face3).
        assemble(bear3,whole,shaft,face3,shaft,face1).
```

```
A view of database:  action

        press(stopper,projection,pen_barrel,top_opening,nil,nil).
        press(refill,body,pen_barrel,hole,pen_barrel,bottom_opening).
        press(cap,hole,pen_barrel,bottom_end,nil,nil).
        press(bear2,whole,shaft,face2,shaft,face3).
        press(bear3,whole,shaft,face3,shaft,face1).


Yes, Yen.....
Starting the planner.....
Type in the goal state(s).....
|:  stop.

yes
.
```