A User Modelling Approach to Computer Based Advice Generation

John Michael Lewis

PhD

University of Edinburgh

1990

Abstract

The need for intelligent advice, help and tutoring, generated automatically by computers is widely accepted. Recently, efforts to achieve these requirements have focused upon research into user models and improved user-machine interaction. This thesis develops a mechanism for detecting when an individual has a problem with the computer system, and generates hypotheses which describe possible problems that he[1] may possess. The method uses Plan Recognition to suggest plans that a user may be following. A Chart Parser is used to achieve this plan recognition task. This technique is capable of detecting multiple, interleaved and incomplete plans, without prior knowledge of the user's intended goals.

A model of users' beliefs is developed for part of the UNIX[2] operating system. This model is capable of representing misconceptions that users possess about the domain.

The combination of these techniques and heuristics to suggest when the user is experiencing difficulty, enables the computer to initiate the advice process and determine the basis for the advice.

---

1. Throughout this thesis, the words "his" and "him" are used. This usage does not imply that the author assumes that users of computer systems are male, but is merely to aid readability of the thesis.

2. UNIX is a trademark of Bell Laboratories.

Declaration

I declare that this thesis has been composed by myself, and that the work that it describes is my own

John M. Lewis

To Shen-Chi

Acknowledgements

I would like to thank Peter Ross who has given constant support over the years. John Jones was of invaluable assistance in the initial stages of the work, and latterly, Helen Pain has supplied useful comments and given encouragement. The views of other post-graduate students have also been invaluable: most notably, David Hintze and Colin Williams.

Life would have been much more frustrating were it not for James Kwa, Gary Roberts, and Richard Caley, who helped solve tiresome problems concerned with generating a thesis on a computer.

Thanks must also go to my wife Christina, my parents, my brother Robert and his family, Mukesh Patel, Andy Lowe, Nang Chan, and Alain Senteni, all of whom gave much needed moral support.

I would like to thank everyone in the Department of Artificial Intelligence for providing the environment in which the work could be performed. Thanks also to Westland Helicopters Ltd for sponsoring the work, and Alan Cort, John Wheeler, Les Botham and Cliff Howell for their involvement. Also recognition should be given to staff at Napier Polytechnic of Edinburgh, who have encouraged me to complete my thesis.

This work was funded by a grant from the Science and Engineering Research Council, with sponsorship from Westland Helicopters Ltd.

Contents

Chapter 1:

An Introduction to Automated Advice 1

1.0 Introduction 1
1.1 Human-Computer Interaction 2
1.2 Intelligent Advice 5
1.3 User Modelling 8
1.4 Objectives 10
1.5 UNIX - An Application Domain for Automated Advice 12
1.5.1 The UNIX Operating System 13
1.6 Thesis Structure 18
1.7 Summary 19

Chapter 2:

User Modelling and Plan Recognition 20

2.0 Introduction 20
2.1 User Modelling 21
2.1.1 Modelling the Extent of Users' Knowledge 22
2.1.2 Adapting the Model 24
2.1.3 Modelling Misconceptions 26
2.1.4 Summary 30
2.2 Plan Recognition 32
2.2.1 The MACSYMA ADVISOR 34
2.2.2 POISE 36
2.2.3 BELIEVER 37
2.2.4 Circumscription-based Plan Recognition 39
2.2.5 Plan Recognition and Discourse 45
2.2.6 GUIDON II 46
2.2.7 PROUST 47
2.2.8 Plan Recognition in Story Understanding 48
2.2.9 Summary of Plan Recognition 49

2.3 Summary **51**

Chapter 3:

An Analysis of the Actions of UNIX-Users **52**

3.0 Introduction **52**
3.1 Data Logging **54**
3.1.1 Analysis of "logged data" 54
3.2 Problem Solving **56**
3.2.1 Problem Design 57
3.3 Observations from the Experiment **59**
3.3.1 Misconceptions 59
3.3.1.1 "**" misconception 59
3.3.1.2 ".. is Home" 61
3.3.1.3 "Name Specifies Absolute Location" 61
3.3.1.4 "./directory/" 62
3.3.1.5 "Manual Syntax" 63
3.3.1.6 "cp directory to directory" 64
3.3.1.7 "Location" 65
3.3.2 Lack-of-Knowledge 66
3.3.2.1 "Commands" 66
3.3.2.2 "Parts of Commands" 66
3.3.2.3 "Filestore Trees" 68
3.3.2.4 "Wild Cards" 70
3.3.3 Errors 70
3.3.3.1 "Typing" 70
3.3.3.2 "Pattern Matching" 71
3.3.4 Habits 71
3.3.4.1 "Move to See" 71
3.3.4.2 "Locate at Departure / Destination" 72
3.3.4.3 "cp,rm" 73
3.3.4.4 "ls" 73
3.3.4.5 "Non-Default Options" 73
3.3.4.6 "Pattern Matching" 74
3.3.5 Goals and Plans 74
3.3.6 Validity of Results 76
3.4 Comparison between "logged" and "experiment" data **78**

3.4.1 Analysis of User Profiles ... 78
3.4.1.1 Long-Term Users ... 81
3.4.1.2 New Users ... 81
3.4.2 General Observations ... 82
3.4.3 Interpretation of Results ... 83
3.5 Discussion ... 84
3.6 Summary ... 88

Chapter 4:
Modelling Users' Beliefs about UNIX Commands ... 89

4.0 Introduction ... 89
4.1 Modelling Users' Problems ... 90
4.2 Modelling UNIX Commands ... 93
4.2.1 A Model of the UNIX Filestore ... 93
4.2.2 A Model of UNIX Commands ... 95
4.2.3 Using the Model of UNIX Commands ... 98
4.3 A Model of the User's beliefs about UNIX Commands ... 101
4.3.1 The User Model of Commands ... 101
4.3.2 Learning a User Model of Commands ... 103
4.4 Modelling a User's Problems ... 106
4.4.1 Detecting Mistakes ... 106
4.4.2 Modelling the User's Lack-of-Knowledge ... 107
4.4.3 Modelling the User's Misconceptions ... 108
4.5 Discussion ... 113
4.5.1 Representation ... 113
4.5.2 The Learning Algorithm ... 115
4.5.3 Models of Misconceptions ... 117
4.5.4 Application to Other Domains ... 118
4.5.5 Comparisons with Alternative Methods of Modelling Misconceptions ... 119
4.6 Summary ... 121

Chapter 5:
Active Chart Parsing and Plan Recognition ... 122

5.0 Introduction **122**
5.1 Parsing Using an Active Chart 123
5.1.1 The Basics of Chart Parsing 123
5.1.1.1 The Fundamental Rule 125
5.1.1.2 The Parsing Policy 127
5.1.2 An Example Parse 129
5.1.3 Pausing the Parsing Process 136
5.2 Chart Parsing and Plan Recognition **138**
5.2.1 Incremental Parsing 138
5.2.2 Context 143
5.2.3 Fragmented Plans 146
5.2.4 Interleaved and Enclosed Plans 151
5.3 A UNIX Grammar **160**
5.3.1 The Form of the Grammar 163
5.3.2 Incomplete Grammars 165
5.3.2.1 Recognising Cliches 166
5.3.2.2 Generating Plans 169
5.4 Discussion **177**
5.4.1 Bidirectional Chart Parsing 177
5.4.2 Comparison with other Plan Recognition Techniques 179
5.4.3 Complexity and Efficiency Considerations 180
5.4.4 Extending the Grammar for User Modelling 187
5.5 Summary **189**

Chapter 6:

A Chart-Based Approach to Advice Generation **190**

6.0 Introduction **190**
6.1 Generating Multiple Hypotheses **191**
6.1.1 Backtracking Solution 195
6.1.2 Multiple Contexts Solution 198
6.2 Maintaining Multiple Hypotheses **201**
6.2.1 Maintaining Partially Instantiated Edges 205
6.3 Analysing the Chart **208**
6.3.1 Detecting When the User Requires Advice 208
6.3.2 Using the Chart to Detect When a User Requires Advice 210

6.3.3 Determining the User's Problems ... 212
6.4 Example Advice Generation ... 214
6.4.1 Example Advice derived from a Potential Typing Error ... 214
6.4.2 Example Advice derived from a Potential Path Error ... 218
6.4.3 Example Advice derived from a Potential Misconception ... 220
6.5 Alternative Advice ... 226
6.6 Discussion ... 231
6.7 Summary ... 235

Chapter 7:
Discussion and Conclusions ... 236

7.0 Introduction ... 236
7.1 System Evaluation ... 237
7.2 Discussion of the System Components ... 239
7.2.1 User Modelling ... 239
7.2.1.1 Model of Commands ... 239
7.2.1.2 Plan Grammar ... 241
7.2.2 Plan Recognition ... 242
7.2.3 Misconception Detection ... 248
7.2.4 Integrating Plan Recognition, User Modelling and Advice Generation ... 250
7.3 Further Work ... 252
7.3.1 Short Term ... 252
7.3.2 Long Term ... 253
7.4 Conclusions ... 255

References ... 256

Appendices *263*

I	Example Logging Script	264
II	Text of Questions for Data Gathering Experiment	265
III	Part of a Typical Data Script	268
IV	An Example showing the Development of a Misconception	271
V	An Example of Planning and Re-planning in Error Recovery	273
VI	Model of UNIX Commands	276
VII	User Model of UNIX Commands	289
VIII	A Sample Plan Grammar	309
IX	Example UNIX Cliche Detection	315
X	Chart Analysis Data	316
XI	Chart Listing for a UNIX Command Sequence Containing a Typing Error	320

Papers in Support of Candidature *324*

Detecting and Modelling User's Beliefs about UNIX 325

Plan Recognition for Intelligent Tutoring Systems 329

Chapter 1

An Introduction to Automated Advice

1.0 Introduction.

This chapter introduces the problems which people experience when using computer systems, and explains how advances in the field of "Human-Computer Interaction" (HCI) can benefit these users. We propose a method of improving the use that people make of computers through computer-generated "intelligent" advice for the user. This advice is described as being "intelligent" because of the ability to adapt it automatically to individual users' problems, their beliefs and their tasks-in-hand. This includes the computer being able to determine when the user requires advice and what advice is needed, without explicitly being asked. The constituents of such a system are identified. In particular a User Model which represents the user's beliefs about the computer system and which can adapt to evidence of these beliefs changing over time is needed. Also the user's intentions must be recognised through observation of his actions.

As a vehicle for developing the ideas on automated advice, the UNIX operating system is used as an application domain to investigate different parts of the advice-giving system. The UNIX operating system is briefly described in section 1.5 so that the reader can appreciate the concepts and notation used throughout the thesis. Finally, the structure of the thesis is described and the chapter summarised.

1.1 Human-Computer Interaction.

The topic of Human-Computer Interaction (HCI) has been recognised as being vitally important as a means to make computers easier to use (Jerrams-Smith, 1986). The form that improvements to HCI should take is still wide open to debate, although it is generally agreed that the behaviour of the computer should be improved to meet the needs of the user (rather than simply relying on the user adapting to the system). In the past, designers of computer systems have relied upon the user adapting to the design of the computer's hardware and software. This has been made possible because "People are so adaptable that they are capable of shouldering the entire burden of accommodation to an artifact" (Draper and Norman, 1986 (P.1)). Proposed solutions to the problem of making computers easier to use take many forms, for example:

- Designing programs that are based upon the tasks that users wish to perform with the system (for example; Davenport and Weir, 1986).
- Providing more information to the user that is relevant to his task (for example; Mason, 1986; Jerrams-Smith, 1986).
- Using natural language interfaces (for example: Wilensky, 1984; Quilici et al, 1986).
- Using structured environments to guide the user (such as menus).
- Using pictorial information such as icons or graphics (for example; Boggild, 1986).
- Using different hardware devices (such as "Mice", or Touch screens).
- Providing Self-Adaptive interfaces (for example; Cooper, 1988).

Each of these approaches to HCI solves particular interaction problems, but introduces new problems, and leaves many others unsolved. For example, the use of menus to structure users' input to the computer

overcomes the problem of the user typing invalid commands. However, the user is then restricted to following particular sequences of actions which constrains users at all levels of expertise to those sequences. It is also unclear as to the best form of the menu structure, there being a tradeoff in the depth / breadth aspect of menus (Raymond, 1986). Another potential problem is that users may not be aware of the purpose or consequences of some of the options available. Buxton examines different hardware interface devices and discovers that no currently available device solves the interface problem for all tasks (Buxton, 1986). For example, he discovers that different kinds of "joysticks" are more appropriate to some tasks than others because of their particular idiosyncrasies.

Davenport's approach (Davenport and Weir, 1986), of structuring the design of programs according to the tasks that users wish to perform, does not overcome the problem that often users try to achieve their tasks in very different ways when they have the freedom to choose. It may well be the program structure that constrains the way that the users think, in which case it is likely that Davenport's approach will constrain users to one particular method of achieving tasks.

Hollnagel argues that the HCI should take into account the abilities and capacities of the user (Hollnagel, 1983), and Draper and Norman suggest that one view of the computer is that "The computer can be thought of ..... as a personal assistant, where the goals and intentions of the user become of primary concern" (Draper and Norman, 1986 (P.1)). This is the view of the computer that we wish to adopt, where the computer is helping the user to achieve his task, as well as being used to actually solve the constituent problems.

This thesis relies on the assumption that all forms of HCI pose problems for the user, since the behaviour of the machine will not correspond to every user's mental model of the system's behaviour. This assumption is based upon the observation that humans cannot always convey their thoughts to other humans, even though the communication between them is

much richer than between human and machine. Rather than attempt to develop the "ideal" physical human-computer interface, we are concerned with detecting the problems that users have with existing computer systems. If these problems can be identified, then they can be used in the generation of automated advice. The justification for this is based upon the assumption that humans will always need to learn how to use a computer system - no matter how simple the interface (just as we need to learn to walk, eat, drive cars, etc), and that humans must adapt in some way to the interface. Two such examples are the use of icons and the mouse for computers such as the Macintosh[1]. Informal observation of novice users indicates that neither of these devices are entirely intuitive to use, and that some advice has to be given about their use. That is, users have incomplete knowledge and possess misconceptions about computer systems, and this problem pervades all computer hardware and software.

The aim of computer-generated advice, then, is to recognise the user's misconceptions, lack of knowledge and errors, and help him to correct these. These aims were expressed by Hartley (Hartley and Pilkington, 1987 (P.41)) as the need ".... to help users build a conceptually consistent model of the application domain". However, for this model to be of value to the user it must also be accurate.

---

1. Macintosh is a trademark of Apple Laboratory Inc.

1.2 Intelligent Advice.

A strong case for an intelligent user-system interface was made by Jerrams-Smith (Jerrams-Smith, 1986). She describes features which should form part of an interface, these falling into two broad categories: *General Communication Considerations*, and the *Particular Needs of an Individual.* She maintains that there should be two components to the user-model, a general user model and a specific user model for each individual. The main features of interest that she identified are:

- The interface should train the user to make the best of the system.
- Feedback and a sense of presence should be provided.
- There should be a personalised response to each user.
- The interface should offer help only when necessary.
- Users should be protected from their mistakes.
- Answers to questions should take into account the user's intentions.
- Constructive messages and positive reinforcement causes faster learning for novices.
- Experienced users must feel in control.

These requirements were used by Jerrams-Smith to develop an "Intelligent Assistant", called SUSI. An intelligent assistant is a computer program that analyses the user's actions when he is attempting to solve a problem using a computer-based tool, and offers advice based upon this analysis. The program behaves like a friend who is looking over the user's shoulder, usually remaining silent, but offering timely advice either voluntarily, or when asked by the user. SUSI did not completely fulfill these criteria since the adaptation to individual users was minimal, and the ability to offer advice without being asked, simply relied upon the recognition of certain pre-specified "non-optimal" sequences of commands. A better sequence was then suggested to the user, but the user's beliefs about the system were not taken into account when formulating this new

sequence. Such beliefs about commands and how these are combined to form plans should affect the information that is offered to the user and the way in which this advice is presented.

Shrager and Finin support the concept of computers generating advice *automatically* and *voluntarily* (Shrager and Finin, 1982; Finin 1982). They view the process of advice-giving as a "complex, interactive process" between the user and the computer. They identify steps in this process, including:

- Deciding if the user requires help.
- Deciding what information the user needs.
- Determining what information the computer can provide which will best satisfy the user's needs.
- Determining whether the user understands the advice.

Their program, WIZARD, provides advice to inexperienced VAX VMS users. The advice is based upon the automatic recognition of correct but inefficient command sequences. Advice is then generated to improve the user's performance. However, WIZARD suffers from the same problems as SUSI, due to its superficial view of the user's tasks and beliefs.

The features identified by Jerrams-Smith and Shrager et al are consistent with the aims of this thesis, except that we will not be concerned with protecting the user from his mistakes, with the format that the advice will take, nor with developing a methodology to determine if the user has understood the advice. The features that they identify imply that the individual must be taken into account both in terms of detecting problems and formulating advice. This is also suggested by Jackson and Lefrere, who state that:

> In addition to having access to knowledge about the domain, an advice-giving system needs to be able to reason about the current state of the interaction. This in turn requires the ability to infer a

user's goals from his inputs and determine what plan of action he is following. (Jackson and Lefrere, 1984).

That is, the computer needs a *model* of the user if it is to have the ability to generate intelligent advice, and this model must be capable of:

- Determining the user's tasks (goals) and his methods (plans) for achieving these - ie *Plan Recognition*.
- Inferring the user's knowledge and beliefs.
- Representing the user's plans, goals, knowledge and beliefs.

1.3 User Modelling.

User Modelling has been an active area of research since the early 1970's. It is concerned with furnishing a computer with knowledge about individuals through explicit models of people's capabilities. These models can take the form of a simple classification of the user's expertise into "expert" or "novice" categories; through to complicated models which dynamically alter to take account of the user (for example, Goldstein, 1982). The subject has arisen from the difficulty that people experience whilst using computers and the desire to make computers interact with people in a way that takes account of their physical and mental capabilities.

One particular area where it was thought that user models were needed was in Computer Aided Instruction (CAI) (Self, 1974). However, the last fourteen years have not fulfilled the promise of greatly improved CAI through such modelling techniques. This situation has forced researchers to reappraise the use of user modelling, and the tide has now turned on the respectability of using these techniques. People now claim that the problems are too complex (Self, 1988b) and that user modelling should be abandoned in favour of more traditional techniques (for example, through the careful analysis and tailoring of solutions before the system is used).

In contrast to these underlying feelings, this thesis supports the view that user modelling is *possible, useful* and *necessary* for future computer systems, provided that the boundaries of the user model are clearly defined. Self still supports such a view (Self, 1988a), but has now modified his thoughts on the subject to produce some guidelines on how user models can be realised (Self, 1988b). In these guidelines, Self maintains that user modelling is an important component of Intelligent Tutoring Systems (ITS) and argues that by clearly defining the need for the model, and not expecting the ITS to be omniscient, that the model can be simplified to a level that is attainable.

Sparck-Jones attempts to identify possible roles of user models in expert systems, the kind of information that such models require and how this information can be obtained from the user (Sparck-Jones, 1984). She supports the view that user modelling is necessary for many advanced programming tasks, such as tutoring. However, she recommends that the modelling should be considered as a means for improving existing performance, rather than attempting to model as much of the user as possible and then applying this model to the problem. She also takes particular note of the relationship that programs must have with human beings, and classifies different types of program according to what part humans play in their application. She indicates that the task of obtaining information for the user model is extremely difficult and will be limited to certain easily-measured characteristics.

1.4 Objectives.

This work is driven by the requirement for improved Human-Computer Interaction. However, the objectives of the research are not to produce a perfect Man-Machine Interface (MMI). Instead, the work is focussed on the automatic generation of advice, and more specifically the detection of users' problems that will trigger such advice. This problem is two-fold: to detect when a user requires advice, and to determine the problems that he is experiencing. This goes further than the work by Finin and Jerrams-Smith, but stops short of making any serious attempt to correct the user's mental model of the system. Thus the main objectives of this work are to:

- Develop a technique which can be used to recognise user's intentions through the observation of their actions (Plan Recognition).
- Develop a method of user modelling which models a set of beliefs that might account for the user's behaviour in the application domain. This models changing beliefs, misconceptions, and the extent of knowledge within the domain.
- Integrate user-modelling and plan recognition so that instances when the user requires help can be identified, and the user's underlying problem can be determined.

The user model does not model the user's actual beliefs, which is an impossible task. Instead, it models a set of hypothesised beliefs that might account for the observed behaviour. The user's beliefs might not (and probably will not) be the same as the hypothesised beliefs. However this is still a useful model since it offers possible accounts for the observed behaviour to which the user can relate. When the advice-giving program identifies that the user has made an error and offers advice based upon hypothesised beliefs which are consistent with his behaviour there are two scenarios: If the user possesses these beliefs, then he will be able to

understand the advice and modify his actions accordingly. Alternatively, the user might not in fact possess the hypothesised beliefs, but he will be better able to understand the problem that has occurred through the causal chain which is offered as advice. This explanation is based upon the hypothesised beliefs which acts as a prompt for the user to seek an explanation to a problem, and it also provides a frame of reference for the user. Thus the explanation based upon the hypothesised beliefs either exposes the problems with the user's beliefs, or prompts the user to seek a problem and an explanation.

For the above objectives to be realised, the concepts must be tested and therefore an application domain chosen. The chosen domain is the UNIX operating system.

1.5 UNIX - An Application Domain for Automated Advice.

The UNIX operating system was chosen as an appropriate application domain for automated advice because:

- It is a widely used computer-based tool.
- It is well understood and well documented.
- It poses a wide range of problems to users.
- There is a wide range of possible activities that users can follow.
- Data on command usage had already been gathered and some analysis performed.

A decision was made to restrict the domain to UNIX commands concerned with the manipulation of files and directories rather than with commands which process the contents of files (for example, filters such as "grep"). This subset was chosen because of its wide usage by users with all levels of experience, and the ease with which the commands and their effects can be modelled.

Throughout this thesis a basic understanding of UNIX concepts and commands is needed. For this reason, a brief introduction to UNIX will now be given, but see (Kernighan and Pike, 1984) for a general description and introduction to UNIX, and the UNIX programmer's manual (University of California, 1983) for a complete description of the facilities available. The shell used throughout this work is the C shell running under Berkely UNIX BSD.4.2.

1.5.1 The UNIX Operating System.

UNIX is a widely used command-driven operating system, which was designed to have a terse nature of interaction to enable sequences of commands to be combined together easily. Commands typically consist of a command name, flags (which modify the effects of the command), and a series of arguments upon which the command operates. These command elements are entered on a single line and are separated by spaces. UNIX assumes that the first word appearing on a line is the command name and that it is followed by any flags and then the arguments to the command. The command names take the form of mnemonics, usually being composed of two or three letters, but these mnemonics are not always apparently appropriate for their actions. The commands that are used in this thesis are shown in fig 1.1.

cp	copy a file or directory.
mv	move / rename a file or directory.
ls	list the contents of a directory.
rm	remove / delete a file or directory.
mkdir	make / create a new directory.
rmdir	remove / delete an empty directory.
cd	change working directory.
cat	concatenate files / list the contents of a file.
pwd	display the current working directory.
man	online manual / help.
top	editor (this is not a standard UNIX command).

fig 1.1.
A summary of UNIX commands used.

A file is a store of data which contains, for example, text or a program. A directory is a special kind of a file which contains information about where files are located, and is used by the user to group together files

which have particular purposes. The UNIX nomenclature for files and directories can be confusing. In manuals, a file often takes the more general meaning of a node in the filestore hierarchy (a file or directory). In this thesis such a meaning is not used, and a file specifically does *not* include directories as a sub-class. A node in the filestore, which can include both files and directories, will be called a *filestore node*.

The commands used in this thesis are concerned with manipulating the location and existence of files and directories within the filestore (although some of the commands also affect the contents of files - for example; top, cat). Each command has, typically, many different applications which vary both through the context in which they are used and by modifying the action of the command through the use of flags. Flags are usually specified immediately after the command name as a list of letters prefixed by a "-" character. For example, "ls -l" is the "ls" command (modified using the "l" flag) to list the contents of the current working directory with detailed information (for example; time of last change, size, etc) about the contents of the directory.

The UNIX filestore is based upon the concept of a hierarchical structure of directories each of which may contain files. This enables the user to structure the way that his files are stored to reflect their purpose, by grouping certain files into a directory (An example filestore hierarchy is shown in fig.1.2). Due to this hierarchical structure, each file must have a *path* associated with it (in addition to the name), so that ambiguity between files of the same name is avoided. This path describes the directories between the top of the filestore tree ("/") and the node being accessed. For example, /usr/students/fred unambiguously specifies file "fred" in the filestore hierarchy. Each user can move around the filestore to aid him in his task. This is achieved by each user possessing a *current working directory* which specifies his current position within the filestore hierarchy. Files and directories can be addressed as offsets from this point.

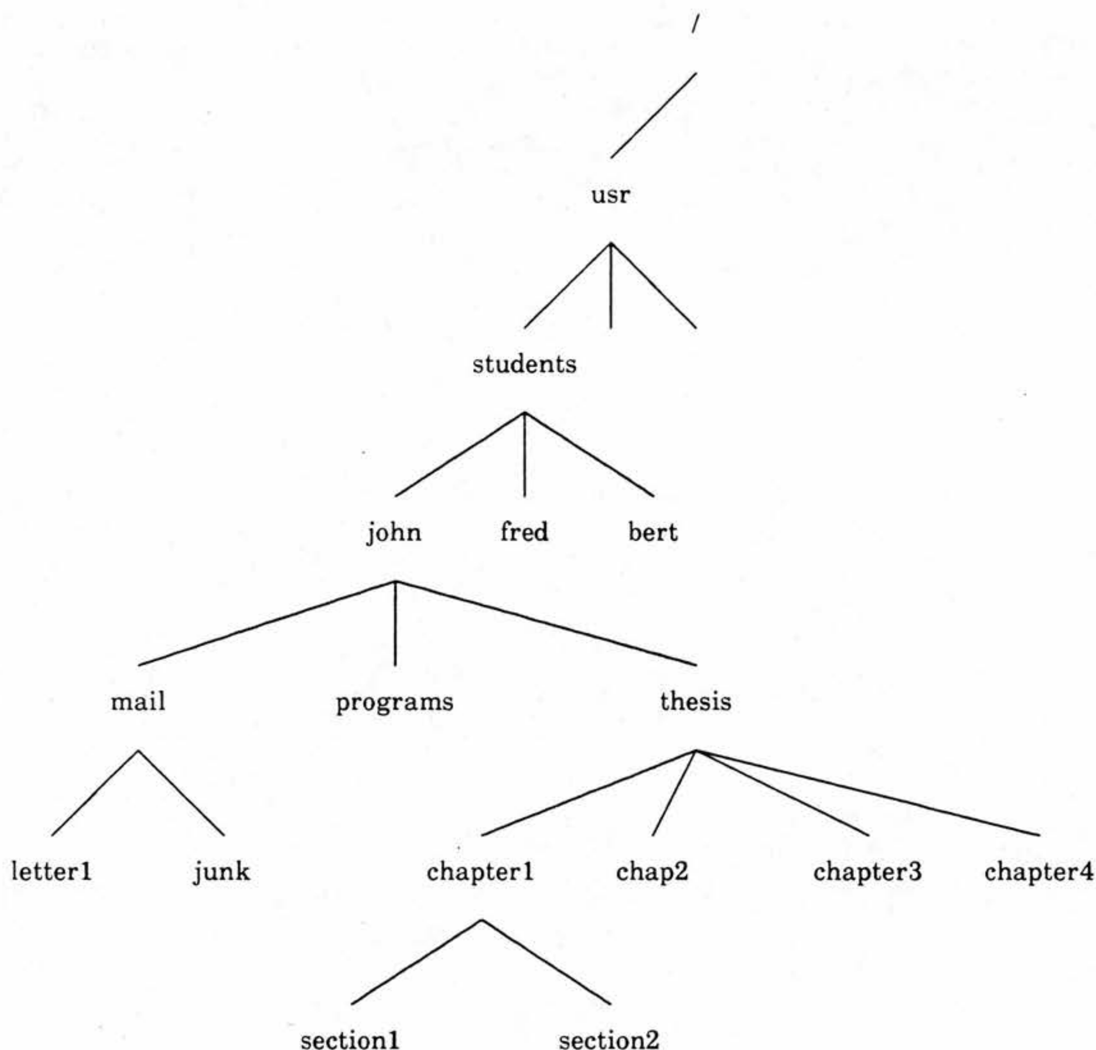


fig 1.2
Part of a UNIX filestore.

For example, in fig 1.2, if the current working directory is /usr/students/john/mail, then the file "letter1" can be accessed by typing the name of the file without the path (ie. letter1). This spares the user from having to specify long path descriptors for each file that he wants to access.

There is a notation for specifying the current directory ".", the next directory up the tree (towards the root) as "..", the user's home directory as "~" (this is the directory where the user is situated when he first logs in - usually at the root of his personal filestore tree), and other users' home directories as "~<Name>" (where <Name> is the user's login name. For example ~fred). An example of the use of these shorthand facilities is given in fig 1.3. This example is based upon the filestore hierarchy of fig 1.2.

Home directory is /usr/students/john
Current working directory is /usr/students/john

```
% ls
mail thesis programs
% ls mail
letter1 junk
% ls ..
john fred bert
% ls ~
mail thesis programs
```

fig 1.3
Example UNIX Session (Based upon the filestore given in fig 1.2).

There are other shorthand facilities available on this particular implementation of UNIX. Two frequently used facilities are:

- Automatic filename completion (performed by pressing the "escape" key, which causes UNIX to complete the current word as far as possible without ambiguity. For example "cp le<ESC>" is completed to "cp letter1").
- Pressing the "control z" key which gives a list of the current options available.

UNIX allows sequences of commands to be entered on a single command line by separating the commands with a semi-colon. For example, "cp letter1 letters; ls" which executes the copy command followed by the list command. This facility is not widely used by novice UNIX users and is not specifically included in the subset of UNIX dealt with by this thesis, since it adds no complexity to the subset. The only possible problem with such a composite command is that it might not be clear to which part of the command an error message refers. However, nearly all UNIX error messages name the command which caused them.

UNIX uses the concepts of *pipes* and *re-direction*. Pipes enable the output from one command to be fed directly into another command. For example, passing the output from "ls" into "wc" to count the number of files in a directory. This is achieved by typing "ls | wc -w". Here, the "|" represents the pipe. This allows complicated functions to be performed by passing data through several different commands. Re-direction allows the input or output of a command to be directed to or from a file. For example, sending the output from "ls" to a file, using "ls > temp". Normally the default input to a command is the keyboard, and the default output is the screen. Thus commands such as "cat" have the side-effect of outputting the contents of the file to the screen - thereby displaying the file's contents. Pipes and file re-direction will not be modelled in this thesis work.

Associated with each file and directory are protection attributes, which specify who can access that object, read it, write to it or execute it. Users are divided into three groups: The user himself; members of a particular group of users; and other users. Each filestore node has protection attributes which specify which of these groups can access it for reading, writing, or execution.

1.6 Thesis Structure.

Chapter 2 describes other work in User Modelling, Plan Recognition and Misconception Detection.

Chapter 3 examines the problems that users experience when using the UNIX operating system, and highlights the main problem areas and scope of the problems that should be addressed by an automated advice-giving system.

Chapter 4 develops a representation and model of users' beliefs about UNIX commands, based upon the observations made in Chapter 3.

Chapter 5 introduces the concept of Chart Parsing which is developed as a technique for analysing users' actions and inferring possible plans that they are following.

Chapter 6 develops a method of integrating advice generation, based upon using the Chart Parser to maintain multiple possible interpretations of plans

Chapter 7 discusses the resulting system and its components, makes suggestions for further work, and draws conclusions from this work.

Throughout the thesis, extracts from the program used in the research will be used. The programming language used is C-PROLOG. An understanding of this language is assumed.

1.7 Summary.

Users of computer systems *need* intelligent advice in order to make full use of the computer tool that they are using. Intelligent advice requires an understanding of the domain, what the user is attempting to achieve, and the user's beliefs. To be able to develop such an understanding requires that the program contains a user model that is consistent with the user's actions. This model is not an accurate model of the user's beliefs, but it accounts for his actions in a plausible manner. Advice based upon this user model will provide the user with a better understanding of problems that he encounters, enabling him to correct them. Without such a *specific* model, any advice is general and may mislead or confuse the user.

Chapter 2

User Modelling and Plan Recognition

2.0 Introduction.

Chapter 1 introduced the need for Intelligent Advice to be generated by computer systems and identified that such systems must be based upon a model of the user. This model needs to represent those attributes of the person that are important in order that the program can adapt to that individual. The problem of generating intelligent advice requires that this model should be able to represent both the *extent* of the user's knowledge of the application and the *misconceptions* that he possesses which are relevant to the task. In addition, the modelling task involves making hypotheses about the user's intentions through the observation of his actions. This is achieved through the recognition of plans and goals that are consistent with the observed actions.

This chapter considers aspects of existing work in user modelling which are important for an advice giving system. Particular attention is taken to work on Plan Recognition, since the information which will form part of the user model will be derived from the analysis of the user's actions and the recognition of his plans.

2.1 User Modelling.

A general definition of User Modelling is; *The Detection and Representation of an individual's physical and mental capabilities, to a degree adequate for providing him with responses suited to his specific needs.*

Since different computer programs place different requirements upon users, the form and content of the user model should vary considerably from application to application to accommodate this. Also, users do not have the same knowledge of the machine or application as each other. In addition therefore, the model should vary from user to user. However, user modelling does not always fulfill these desires and the models used in different programs vary considerably in their ability to model individuals. At one extreme, the user model may exist as a static pre-defined set of boolean attributes which is considered to represent the user's beliefs. A more complicated version of these models might rate the user's ability (on a range from 1 to 10, say) to achieve particular tasks; or record the number of occasions that these tasks have been observed and missed (for example, WEST (Burton and Brown, 1982)). At the other extreme the model may be a complicated network of inter-related aspects of the user which changes as his behaviour is observed over time, for example the Genetic Graph (Goldstein, 1982).

Important aspects of the user model for the UNIX advisor are that it should:

- Be able to model the extent of the user's knowledge in the domain. It is important that this can be modelled so that *useful* advice can be generated. This advice is useful because it will neither tell the user things that he already knows, nor confuse him with concepts that are too complicated for him to grasp.
- Adapt dynamically as evidence of the capabilities of the user are observed. This is necessary because the model must *learn* about the user's possible beliefs from the observation of his

actions as the session progresses (since it is intended that the advice giving system should intervene as infrequently as possible).

- Be able to model misconceptions that users' possess about the system. This capability is needed in order that the advice giving system is capable of offering meaningful remedial advice to correct these misconceptions.

These three aspects of the user model will now be discussed in more detail, taking examples from existing programs which employ user modelling. For a description of techniques used in user modelling programs, see (Jones, 1985). Wenger gives an excellent review of user modelling for intelligent tutoring systems in (Wenger, 1987).

2.1.1 Modelling the Extent of Users' Knowledge.

An important aspect of an advice giving system, is that it requires a model of what the user believes about the system. In this description the *extent* of the user's knowledge about the domain is defined as being *those aspects of the system about which the user believes and which are correct.* This definition excludes misconceptions (beliefs about aspects of the system which are not correct) which will be discussed in section 2.1.3.

In the past, user modelling has tended to focus upon the extent of an individual's knowledge in a domain. This is mainly due to the relative ease with which we can establish what a user should know about a system to be able to use it effectively (as compared with the problem of determining all possible misconceptions that a user might have about the system). These models are generally called *overlay* models (Carr and Goldstein, 1977), since a user's beliefs can be considered as a subset (or overlay) of those beliefs that are considered necessary to make effective use of the system. Such a model is in effect a sub-set of an idealised expert's knowledge of the system. Examples

of such systems are WEST (Burton and Brown, 1982), and GUIDON II (London and Clancey, 1982).

WEST (Burton and Brown, 1982) is a tutor for the simple board game "How the West was Won". The WEST tutor records the student's use of arithmetic combinations, and the special moves that he makes during the game. These moves are compared against the possible moves that could have been made, and a record is kept of the opportunities taken and missed. This results in the model representing the student's play as a subset of an expert's moves (both in terms of the arithmetic constructs used and the special moves made). *Issues* and *examples* are used as a method of tutoring the student. The student is considered to require certain skills, or *issues* to play the game. The student's moves are assessed against an ordered list of possible moves generated by an expert. *Issue recognisers* are used to determine which issues the student failed to use. General advice (in the form of pre-stored text) is offered to the user and specific advice using an *example* of this issue (based on his position in the game) is used to reinforce the tutorial information.

In contrast to the simple expert in WEST, GUIDON II (London and Clancey, 1982) considers the domain of medical diagnosis. In this domain the expert system NEOMYCIN (Clancey and Letsinger, 1981) is used, which contains correct rules for medical diagnosis. NEOMYCIN is an expert system which has explicit knowledge about how to perform a diagnosis, as well as knowledge about infectious diseases. The knowledge consists of production rules, and the student's knowledge is considered to be a sub-set of these rules. The student is set a diagnosis problem, he then requests information about symptoms, and tests. GUIDON II uses NEOMYCIN to solve the diagnosis, and the student's requests are matched against the expert's diagnosis path. If the student's diagnosis path deviates from NEOMYCIN's, this triggers advice. The student's knowledge is modelled as a strict subset of NEOMYCIN's rules, there being no ability to accommodate other, possibly correct, diagnosis paths.

LMS (Sleeman, 1984) represents students' beliefs about arithmetic and algebra as an ordered list of production rules. These rules consist of conditions and actions, which collectively form an executable model. The rules are pre-determined and are applied to the algebra problem in an attempt to account for the student's solution to the problem. Only one rule can be applied at any one time, this being determined through the ordering of the rules and their conditions. The rules are executed in a cyclic manner, until there are no more rules with satisfied conditions. The resulting trace of rules given by LMS is used as a record of the student's beliefs about algebra, and can be used as a basis for tutoring him.

2.1.2 Adapting the Model.

A model of a user operating an interactive system must adapt to the user's changing beliefs. This adaption is needed because different users behave in different ways, and an individual's beliefs change as he discovers new features of the system. Also, the model must hypothesise about the user's beliefs from observation of his actions, continually refining the model to obtain an accurate "picture" of these beliefs.

Most user models are not adaptive, but static. Static models do not modify their contents as the session progresses, but are usually set at the start of the session. For example, the user might be asked to rate his skill in a range from 1 to 10, from novice to expert. The system will then load a model *stereotype* for that ability, and the model will not be altered during the session.

Stereotypes are used in GRUNDY (Rich, 1979), which is a program to recommend books to library users. GRUNDY obtains a user profile by asking the user to describe himself, and then invoking *stereotypes* associated with particular attributes. For example, these attributes might be that the person is female, and does not have a television. A book is recommended to the user based upon the attributes associated with the different stereotypes.

The relative importance of these attributes being assigned through either conflicting or supportive evidence from the different stereotypes that have been invoked. Generally, such stereotype models do not adapt to the user as the session progresses. The method also suffers from the user having to grade his performance without prior knowledge about how the program will behave. However, notable exceptions to these techniques are those of Goldstein (Goldstein, 1982), and Self (Self, 1988b).

The Genetic Graph (Goldstein, 1982) is an attempt to model how people learn procedural skills. It was proposed as a general method for modelling procedural skills and consists of a network of rules connected by links which represent relationships between the rules. For example, a rule can be considered as a generalisation or specialisation of other rules, or as an analogy. The Genetic Graph can also contain incorrect rules (deviant rules) that the student may learn. As the student learns, his knowledge is represented as the state of the network, this being an overlay on the complete network. The genetic graph approach was tested in the WUSOR coach for the computer adventure game WUMPUS. The student's behaviour when using WUMPUS is compared to that of an expert, which enables the links and rules known by the user to be deduced.

Recently, Self (Self, 1988b) suggested that one method of achieving an intelligent tutoring system would be to make the computer *learn* and *collaborate* with the user. The student-computer interactions should be designed so that the information needed by the ITS is supplied "naturally" by the user. This would appear to be the ideal solution to modelling the user, however there are great difficulties involved in following such an approach. One such problem is that the program must model how the student learns. As yet there is no adequate implementation of this approach.

2.1.3 Modelling Misconceptions.

The computer modelling of people's misconceptions has not received much attention. There are, however, three notable attempts to do this; DEBUGGY (Burton, 1982), WHY (Stevens et al, 1982), and the MACSYMA ADVISOR (Genesereth, 1979 and 1982).

DEBUGGY (Burton, 1982) is a program that analyses the problems that children have with arithmetic subtraction. The program is based upon a detailed study of the problems that children experience with subtraction. This study resulted in a set of rules being identified which describe the possible behaviours that children exhibit when attempting subtraction problems. The rule set (in network form) contains both correct and incorrect *(buggy)* rules about how to perform subtraction. The application of one or more of these rules can transform a subtraction problem into a set of possible answers. The rules used to obtain the answers given by the child, represent the model of the child's knowledge of subtraction. DEBUGGY is used to analyse a series of problems that have been given to children. Thus, the program has access to the subtraction problem, the answer given by the child, and the correct answer. The program implicitly assumes that the child is attempting to get the correct answer to the problems. Using the set of subtraction rules, DEBUGGY determines the rules that the child possesses for subtraction. Heuristics have to be used by the program to determine which rules are most likely to be possessed by the student, since a particular answer could be obtained by applying different combinations of rules. For example, DEBUGGY has the heuristics to use as few rules as possible, and to use correct rules preferentially to buggy rules. DEBUGGY needs the problem and the child's solution to be given. In the UNIX domain this would correspond to the user's intentions being known, which is clearly difficult to determine.

LMS (Sleeman, 1984) uses a fixed set of mal-rules in a similar way to DEBUGGY. These mal-rules are variations of correct rules and they represent misconceptions that students possess about algebraic

manipulation. The application of these mal-rules to achieve the student's solution of a problem, helps with identifying the student's underlying misconceptions.

Another approach to modelling misconceptions is that used in WHY (Stevens et al, 1982), which is an attempt to build an intelligent computer aided instruction program. The chosen domain of expertise is "the causes of rainfall", for which the program has a causal model. This causal model is used to represent the underlying reasons for rainfall occurring (for example; geographic location, or air masses meeting), so that this knowledge can be imparted to a student using the program. The example of the causal model, shown in fig 2.1, is given by Stevens et al for the causes of heavy rainfall.

WHY can detect missing, or additional sub-steps in the causal model (which Stevens refers to as misconceptions). However, they are aware of the limitations that a particular representation places upon the ability to detect problems. They state that:

> The types of misconceptions in a student's knowledge that a system can diagnose are heavily dependent on the knowledge represented in the system. The script structures in the WHY system are able to represent misconceptions that result because of missing substeps or extra substeps in the various scripts. However, these are only two of several misconceptions that occur (Stevens et al, 1982).

They observed the interaction of a small number of bugs which manifest themselves in a large number of misconceptions. Indeed, they state that they were unable to classify many errors due to "non-obvious interactions" between bugs. They suggest that one method of overcoming such difficulties is to use multiple ways of describing the system, each emphasising a different aspect of the system.

As in the DEBUGGY model, WHY uses explicit descriptions of misconceptions, which were found by careful analysis of the domain and students beliefs about the domain.

Heavy Rainfall

1: A warm air mass over a warm body of water absorbs a lot of moisture from the body of water

precedes

2: Winds carry the warm moist air mass from over the body of water to over the land mass

precedes

3: The moist air mass from over the body of water cools over the land area

causes

4: The moisture in the air mass from over the body of water precipitates over the land area

fig 2.1

Part of the Causal Model for Rainfall (from Stevens et al, 1982).

A different approach to modelling and detecting misconceptions is used in the MACSYMA ADVISOR (Genesereth, 1979, 1982). MACSYMA is a large computer program which is a tool to help people to manipulate complicated algebraic expressions. The ADVISOR is a program which helps a user of MACSYMA to identify their problems with their use of that tool. The form of this help, or advice, is in the recognition and description of the user's misconceptions about MACSYMA.

The ADVISOR is unusual because of its ability to detect the user's misconception with the MACSYMA program by analysing the user's actions and reconstructing the *plan* that he is following. The analysis depends upon the user invoking the ADVISOR program when he encounters a problem, and then articulating his goal to the ADVISOR. The analysis of the user's problem is performed by a heuristic problem solver, MUSER, which generates a graph of goals and sub-goals to achieve the specified task. MUSER is a problem solver which uses databases of knowledge about particular domains in order to generate correct solutions to the problem. In the ADVISOR, the knowledge-base is about the use of MACSYMA. It is assumed that the user's misconceptions and errors are due to incorrect and incomplete knowledge of this knowledge-base. Unlike DEBUGGY there is not a fixed set of explicit misconceptions in the knowledge-base, rather the misconceptions are derived from a divergence of MUSER's solution from the user's actions.

The ADVISOR requires a description of the user's goal, which is used by MUSER to generate a graph of possible solutions to achieve the goal. A misconception or error is detected by fitting the user's actions to the solution graph. When the user's actions no longer fit into this graph, then the point of departure can be assumed to be the error or misconception. The type of misconceptions that are detected are the result of values that are returned by the *fetch* database access function. Fetch is used to search the MUSER database to find a command to achieve the required goal and returns the command and its prerequisites. The divergence of the user's commands from MUSER's command sequence can be explained by the user possessing a misconception about the prereqisites for a command. However, the ADVISOR must ask the user whether he believes in the misconception before it is accepted that this is the user's actual problem (since there may be multiple possible interpretations for a deviation from possible solution paths).

The approach that the ADVISOR takes in misconception and error detection is very different to the DEBUGGY or WHY approaches. Instead of having built-in misconceptions and errors which were determined by a lengthy analysis of users' behaviour, a correct model of problem solving is used. This model is based upon the problem-solving strategies used by novices. The misconceptions and errors are then determined as a divergence from a correct solution, and are therefore described at the level of elements in the solution rather than a high-level description of a bug given by WHY. This means that the misconception does not have an explicit high-level description, but is a description of what has gone wrong.

2.1.4 Summary.

A user model for an advice giving system needs to model the extent of the user's beliefs, be able to adapt to his changing beliefs, and be able to model misconceptions that he possesses.

The extent of a user's beliefs are usually modelled with overlays. However, these models suffer from the limitation that they cannot incorporate incorrect beliefs that the user might possess, except as pre-determined errors or misconceptions. Therefore such models are severely limited in their use for offering advice, since this advice can be based upon part of the user's beliefs only (those which can be modelled using the overlay). The model also excludes any alternative views of the system, which may be equally valid but not part of the expert model.

Although there are user models which adapt in some way to the user's beliefs as he solves his task, these tend to be based upon overlay models, with the overlay on the underlying knowledge varying to accommodate the user's beliefs. These overlay models cannot easily model misconceptions or alternative methods to achieve a solution.

Misconceptions tend to be modelled through the use of buggy models. These models have the limitation of only being able to handle a specific number of pre-defined misconceptions. The ADVISOR evades this problem by having a correct model of the domain and detecting a user's deviation from this. However, this approach depends upon having a description of the user's goal and using a top-down as well as a bottom-up matching process.

2.2 Plan Recognition.

Plan Recognition is the task of inferring an actor's intentions from the observation of a sequence of actions performed by that actor. For example, the observation that "John switches a kettle on" might lead an observer to make the inferences that John is going to make a cup of tea. However, the observer could make other possible inferences - That John is going to make a cup of coffee, make rice, etc. The observation might also change the observer's model of the actor's beliefs, adding for example the belief that "switching a kettle on causes water to boil", or that "you do not need to fill the kettle with water before boiling it" - this last belief exposing a possible misconception on the part of the actor. Such inferences about the actor's goals and beliefs are important for giving intelligent advice because the recognition of these plans and misconceptions can be used to trigger advice automatically ("You need to fill the kettle before boiling it").

Attempts have been made at performing plan recognition in many areas of Artificial Intelligence research. For example; Intelligent Advice Generation (Genesereth, 1982; Carver et al, 1984), Intelligent Tutoring Systems (London and Clancey, 1982), Understanding Discourse (Allen and Perrault, 1980; Litman and Allen, 1984; Sidner, 1985; Cohen, 1978), Story Understanding (Schank, 1975; Wilensky, 1983), Psychological Modelling (Schmidt et al, 1978), and Program Debugging (Johnson and Soloway, 1985).

The term *Plan Recognition* is not clearly defined (Charniak and McDermott (Charniak and McDermott, 1985) use the term *Motivation Analysis* instead of plan recognition) and it also embraces different interpretations. For example, plan recognition can be taken to mean the task of inferring an actor's plan to achieve a stated goal with a sequence of actions; or it might include determining the goal itself. The definition that we will use for plan recognition is:

Plan Recognition is the task of determining the plans that the user is following and the goals that he is attempting to achieve, through the

observation of his actions and based upon the assumption that people follow planned behaviour in the environment under consideration.

Work by Suchman (Suchman, 1987) suggests that people do not always have complete plans to achieve their goals. Suchman is concerned with the problem of human-machine interaction and how users formulate their actions. She analysed people using photocopier machines and found that users often do not have pre-determined plans which they follow to achieve some goal. Instead, users have ".....preconceptions about the nature of the machine and the operations required to use it, combined with moment by moment interpretations of evidence found in and through the actual course of its use" (Suchman, 1987. P.119). Suchman named the actions performed in this manner *Situated Actions.* Users who perform situated actions do not have a plan, but perform actions dependent upon the prevailing conditions.

The method by which the Trukese navigate at sea is used by Suchman as an example of situated actions (Suchman, 1987. P.viii). The Trukese set sail with the objective of arriving at their destination, but with no detailed plan of how to achieve this. They have no course mapped out at the start of the voyage, instead they steer according to the prevailing conditions of wind, tide, waves, stars, etc. At any time they know in which direction to steer, but they cannot describe the course that they have taken. This is contrary to planned activity, in which the navigator plots a course at the start of the voyage and follows this course from point to point.

It is not a claim of this thesis that users always follow planned behaviour. Indeed, data gathering exercises concerning the actions of UNIX users identified numerous sequences where this is not the case. However, the fact that the user is not necessarily following a plan does not negate the value of plan recognition as a tool for identifying problems that the user might have. In using a photocopier, operating system or any other machine the user may perform a sequence of actions for which he has no initial plan. These actions take into account the contextual information at the time that

they are performed. Yet these same actions may, in retrospect, fit into a sequence that matches a pre-defined plan. The recognition of this plan does not then mean that the user had been following that plan from the outset. However, the plan does form evidence that the user has achieved something more abstract than a single action.

Using a plan as the basis for automated advice enables the advice to be expressed in a form which makes sense to the user. The advice can be couched in the framework of the actions that he has performed or could have performed, and combinations of such actions. It is not important to this thesis whether the user was indeed following a plan or he was making situated actions. It is the observed actions which are important, since they form a source of information which can be used for generating appropriate advice. The recognition of plans will enable this information to be used for advice generation.

The following sections describe other plan recognition work that has been performed. This is followed by a summary which describes the relative importance of the different techniques.

2.2.1 The MACSYMA ADVISOR.

The ADVISOR (Genesereth, 1979 and 1982) acts as a debugging tool for users of the MACSYMA algebra manipulation tool (see section 2.1.3.3). When someone discovers that they have a problem with using MACSYMA (for example, they might have a result which does not appear to correspond with what they expect), they invoke the ADVISOR to help them trace the problem. This requires the user to state what his intended goal is, then the ADVISOR attempts to fit the user's actions into a plan to achieve this goal.

The ADVISOR requires an explicit description of the user's goal (obtained by asking the user), and the sequence of commands that the user

entered. The ADVISOR also needs to be invoked by the MACSYMA user when he is experiencing a problem. From this information the ADVISOR works top-down from the goal and bottom-up from the actions. Possible, partially instantiated plans are generated "top-down" from the goal, by a planner (MUSER), which develops valid plans according to a model of novice user's behaviour. The observed actions are then combined with the plan to instantiate variables. If there is any ambiguity about which plan the user is following, this is solved by interrogating him.

The ADVISOR makes several simplifying assumptions which enable it to perform plan recognition. First, the user is assumed to be attempting to achieve a single goal, and following a single plan to achieve this goal. Also, every action that is performed by the user must form part of this plan. In many domains where automated advice is required it is unlikely that it will be possible to fulfill these criteria. It is assumed that users do not have any problems in planning to achieve their goal, but their problem is related to the detail of MACSYMA commands. It is also assumed that MUSER is capable of modelling all possible plans that novice users generate, which would be an impossible task to achieve in many domains. However, MUSER's approach assumes that it is capable of achieving this if the user's problem is to be identified accurately. Finally, the user needs to be aware that he has a problem before invoking the ADVISOR. This may not always be possible; for example, the user may think that he has completed his task correctly without recognising that the results are incorrect. A related problem to this is that the ADVISOR requires the user to articulate his problem. It is not clear how this could be achieved in many domains.

However, the ADVISOR is one of very few attempts at determining user's problems whilest they are solving complicated tasks. The ability of the ADVISOR to analyse the user's actions and isolate errors and misconceptions is required by advice giving systems. For the ADVISOR to work in the UNIX domain it requires the ability to deal with multiple, interleaved plans, where the user is attempting to achieve unspecified goals. It is not possible to modify the ADVISOR to work with an unspecified goal,

since it requires this goal in order to recognise the user's plans. In advice giving systems it is desirable that advice is generated automatically, in addition to being initiated by the user. This extends the advice capability to the area where the user does not know that he has a problem, or he is unable to articulate his problem. The ability to perform plan recognition incrementally is desirable, incorporating actions as they are observed, rather than in the "single-shot" capability of the ADVISOR. Such incremental plan recognition enables problems to be located by the advice giving program without being initiated by the user.

2.2.2 POISE.

POISE (Carver et al, 1984) is an intelligent interface to an office automation system. The aim of POISE is to assist a user of the office automation system by detecting and correcting errors, and completing the user's plans. Unlike the ADVISOR, POISE allows the user to be working on more than one goal concurrently, and does not require an explicit description of these goals to be supplied. However, POISE does assume that all of the actions form part of one of these plans. POISE attempts to determine the user's current goals and plans and to use these as a basis for giving advice about errors (actions that do not fit into any current plan and that cannot start a new plan) that have been made, or how to complete a plan. The importance of POISE is in its use of a Blackboard Expert System Architecture (Nii, 1986a), upon which it develops plan and goal hypotheses. Heuristics are used to reduce the number of active plan interpretations on the blackboard at any time. The implementation also makes use of a Truth Maintenance System (Doyle, 1979) to ensure that the assumptions made when forming an hypothesis are consistent with any new information.

The representation of plans uses a hierarchy, in which each node specifies the sub-goals necessary to achieve that goal. There are also conditions which constrain the parameters of these sub-goals.

POISE gives an incremental approach to plan recognition, incorporating new information into the interpretations as the session progresses. However, the nature of the application domain simplifies the problem significantly. For example, only certain actions can start a valid plan (this being analogous to the use of discourse markers (Sidner, 1985), which indicate when a speaker makes a shift in intention); this allows simple errors to be located in the form of actions which do not fit any plan and cannot start a new plan. Such assumptions are acceptable in the domain of office automation, where the adherence to standard procedures is important. However, in other domains such as UNIX, this assumption would be unacceptable because the syntax rules are not so limiting. POISE has no notion that users can possess misconceptions, any command which does not fit the plan is explained as being a mistake which must be corrected. Another limitation is that only plans which conform to the given plan hierarchy can be considered to be correct, even though there may be other correct ways of achieving a goal in other domains. This leads to the problem of having to develop a complete and correct plan hierarchy for any domain under consideration.

The advantages of the POISE system are the explicit representation of interpretations on the blackboard, the use of heuristics to reduce the number of alternative interpretations under consideration and the ability to recognise concurrent plans. However, the approach is difficult to extend to more complicated domains in which the user is allowed to possess misconceptions and make errors.

2.2.3 BELIEVER.

BELIEVER (Schmidt et al, 1976 & 1978; Sridharan and Schmidt, 1977) is a model of a psychological theory of how human observers understand the actions of others. Understanding these actions involves inferring the actor's goal and generating a plan. The intended ability of the program is that it should accept a sequence of actions (eg. Steve walked to

the 'fridge. He opened the 'fridge. He took out an ice cream.....), be able to summarise the salient features of the sequence, and be able to predict the next action.

The knowledge in BELIEVER is divided into domains which supply the data for the *World model* (what is true in the world), *Person model* (what the actor believes is true in the world) and *Plan model* (which contains hypotheses about what the user might be following). Each of these models contains specific instances of concepts and relations. When an action is observed, BELIEVER looks in the knowledge bases for schemas associated with particular actions and objects. These schemas have pre-conditions associated with them which are checked against the world and person models, to ensure that the pre-conditions are believed by the actor and are true in the world. Each schema has a goal associated with it, which is asserted in the plan model when that schema is activated.

An *Expectation Structure* (expected plan) is generated by a top-down refinement of the goal which was suggested by the action, to form a plan tree. This occurs through the instantiation of sub-goals due to the actions being observed. The suggested expectation structure is guided by the person model and the goals suggested by objects (eg. ice cream - for eating). If the system finds an action which does not fit into the expectation structure, then an attempt is made to refine the expectation structure to accommodate the new action.

As with the ADVISOR, it is assumed that the actor is following a single plan to achieve one goal, and that all of the actions performed by the actor form part of this plan. There is a limited ability to deal with erroneous actions, since incorporation of such actions into the plan would cause the expectation structure to be altered and the original (possibly correct) expectation to be discarded. However, if the action cannot be incorporated into the expectation structure, then the action is explained as being an error. The system cannot account for errors through the actor possessing a misconception, since it is assumed that the actor has perfect planning

knowledge and possesses a subset of correct knowledge about the domain. This implies that misconceptions are not allowed. The plan recognition process is incremental, allowing new actions to be added to the observations and incorporated into the expectation structure.

2.2.4 Circumscription-Based Plan Recognition.

An attempt has been made to formalise plan recognition by using a circumscription-based approach (Kautz 1985, 1986 and 1987). Kautz uses this formalised theory to perform plan recognition for a wider set of problems than considered by other researchers. He states five problems that plan recognition should be able to manage. It should:

- handle uncertainty. For example, the problem of a sequence of actions not uniquely identifying a particular plan. He does not, however, address the problem of the observation being incorrect.
- be able to draw conclusions and make predictions from a set of observed actions which do not uniquely define a plan.
- not jump to premature conclusions.
- handle temporal reasoning.
- allow actions to form part of several parallel plans.

Kautz does not assume that the user's goal is available to the program, but that this should be determined through the plan recognition process. As with other plan recognisers, the plans are represented as a hierarchical network of actions connected by *specialisations* and *decompositions*. Kautz originally used the domain of cookery to demonstrate his ideas, but he extended the ideas over a number of other domains later, including advice giving for UNIX. Kautz uses the example of plan hierarchies in cookery (fig 2.2). In this hierarchy: *"make-spaghetti" is a specialisation of "make-noodles"; and "make-pasta-dish" can be decomposed into the constituents "make-noodles", "boil" and "make-sauce".*

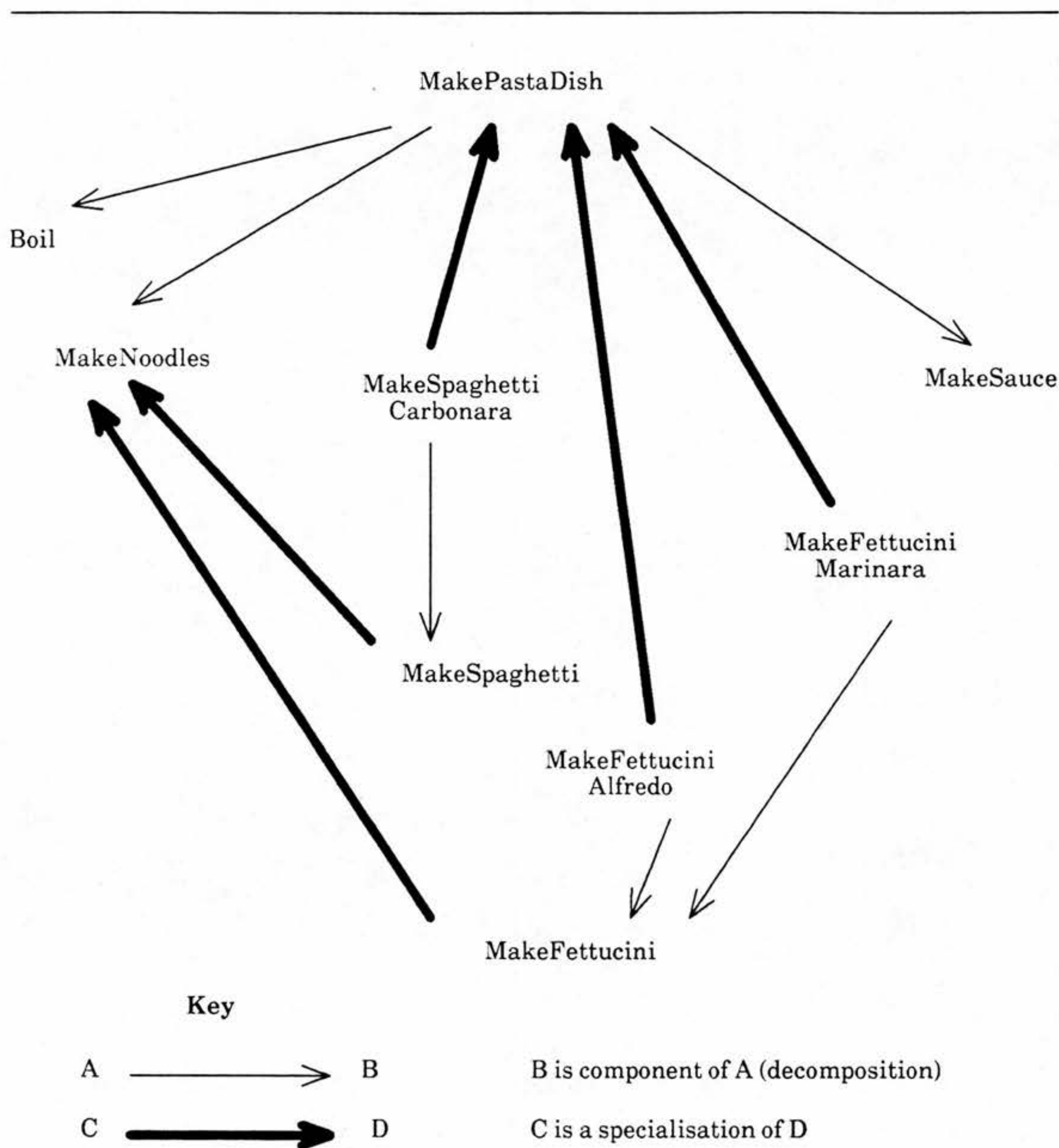


fig 2.2

Part of cookery plan hierarchy (from Kautz, 1986).

Each action has temporal limits associated with it, which allows the actions to be observed in any order without the plan recognition process

being affected. Goals are recognised, through the specialisation and decomposition links, which are consistent with all of the observations. This results in a number of different possible explanations being formed. Kautz then uses heuristics to collapse together as many constituent components into as few plans as possible to minimise the number of end points (goals). This approach makes the recognition of multiple, interleaved plans particularly easy, as the different plans just fall out of the recognition process. Also, because the actions contain temporal information the observations can be made in any order.

A brief description of the representation of actions the plan hierarchy, and the plan recognition process is given for completeness (see (Kautz, 1986) for a more complete description).

Kautz uses a representation of actions of the form:

#(E9, MakePastaDish).. 2.1

where E9 is an instance of MakePastaDish, and # is the "occurs" predicate.

The specialisations and decompositions are then specified as axioms:
An example of a specialisation axiom is:

$\forall e.\#(e, \text{MakeSpaghetti}) \supset \#(e,\text{MakeNoodles})$ 2.2

This axiom (axiom 2.2) describes that instances of MakeSpaghetti are also specialisations of MakeNoodles.

An example of a decomposition axiom is:

$\forall e.\#(e, \text{MakePastaDish}) \supset$
$\quad \exists tn.\#(S(1,e), \text{MakeNoodles})\&$

#(S(2,e), Boil)&
#(S(3,e), MakeSauce)&
Object(S(2,e)) = Result(S(1,e))&
hold(noodle(Result(S(1,e)), tn)&
overlap(T(S(1,e)), tn)&
during(T(S(2,e)), tn) 2.3

This axiom (axiom 2.3) describes that MakePastaDish is decomposed into MakeNoodles, Boil and MakeSauce. The axiom also describes that the object that is boiled is the result of subaction 1; the object created (Noodles) exists for some transitional time (tn); and that the action of boiling the Noodles is performed during the time that the noodles are in existence. There would be other such conditions for the action MakeSauce.

Disjointedness can be represented by axioms. For example, the fact that "an action cannot be *both* an instance of MakeFettuciniAlfredo and an instance of MakeFettuciniMarinara" is represented by:

∀e.#(e, MakeFettuciniAlfredo) $\bigtriangledown$
#(e,MakeFettuciniMarinara) 2.4

where $\bigtriangledown$ means "not and".

To enable plan recognition to be performed, Kautz makes two assumptions; the Assumption of Specialisation Completeness and the Assumption of Decomposition Completeness.

Specialisation completeness is represented by adding axioms of the form:

∀e.#(e, MakeNoodles) ⊃
#(e,MakeFettucini) ⊕
#(e,MakeSpaghetti) 2.5

Axiom 2.5 states that the only ways of specialising MakeNoodles are MakeFettucini or MakeSpaghetti ($\oplus$ is the "exclusive or" operator).

Decomposition completeness is represented by adding axioms of the form:

$\forall e.\#(e, \text{MakeNoodles}) \supset$
$\quad \exists a.\#(a,\text{MakePastaDish})\ \&\ e = S(1,a)$ 2.6

Axiom 2.6 states that MakeNoodles is the first sub-action of MakePastaDish. When an instance of MakeNoodles occurs, then the inference is made that a MakePastaDish will be performed.

The axioms 2.2 to 2.6 are generated before actions are observed, these axioms are then used to perform plan recognition. For example; a MakeSpaghetti action is observed, giving an instance:

$\#(E1, \text{MakeSpaghetti})$ 2.7

Abstracting up the hierarchy with axiom 2.1 gives:

$\#(E1, \text{MakeNoodles})$ 2.8

which recognises that the action is consistent with the goal to MakeNoodles.
Decomposition using axiom 2.6 gives:

$\#(K01, \text{MakePastaDish})$ 2.9

which recognises that the action forms part of the goal MakePastaDish. Axioms 2.6 and 2.3 combine to suggest that a "boil" is expected.

#(S(2,K01), boil) 2.10

The combination of the observed instances of actions with the generalised axioms enables goals to be inferred which are consistent with the observed actions. If observations are made which are conflicting, then predictions are made that are consistent with these observations. For example, (from fig 2.2) if the observation is made that the action is either MakeFettucini or MakeSpaghetti, then the goal MakePastaDish is recognised, as is the prediction that the next action will be a boil.

The representation used by Kautz depends upon the decomposition and specialisation hierarchies which appear to be difficult relationships to determine and write in any non-trivial domain (although Kautz has shown hierarchies for simple examples in several domains (Kautz, 1987)). These representations are fixed and cannot be adapted to model an individual's beliefs, though there is no apparent reason why relations should not be added, deleted and changed to reflect this. The theory assumes that the observed actions are intended by the actor, forming a meaningful part of one of his plans, and that they have been observed correctly. There is no mechanism in the theory for coping with errors or misconceptions on the part of the actor, but the implementation should be able to ignore actions which do not form a part of any plan.

The plan recognition process supports the incremental development of plans by building upon plans determined previously. The mechanism allows this incremental development to occur in a non-monotonic way, with new observations refuting previous inferences.

The appeal of this plan recognition scheme is its generality and its theoretical grounding on circumscription. However, intelligent advice has additional complications since the actor (user) can make errors and possess misconceptions, and there is no apparent way of incorporating these extensions into the system. Also, Kautz's work assumes that there is an

action taxonomy which contains the plan knowledge for the system. This action taxonomy must be complete and correct for circumscription to work, which is a severe problem when the system is confronted by a real-world problem.

2.2.5 Plan Recognition and Discourse.

Allen and Perault were interested in modelling cooperative behaviour exhibited in the discourse between two actors (Allen and Perault, 1980). They wished to infer an actors "wants" and "beliefs" from an observation of his actions. Such actions existed as utterances between a patron and clerk at a train station. They were attempting to model the helpful behaviour of the clerk, who would often give help that was not explicitly asked for. The work depended upon viewing the utterances as speech acts (Cohen, 1978), which are utterances that have the effect of modifying the actor's beliefs. They made the assumptions that actors (people) are:

- Rational agents who are capable of forming and executing plans to achieve their goals.
- Often capable of inferring the plans of other agents from observing those agents perform actions.
- Capable of detecting obstacles in another agent's plans. Obstacles being goals in the plan that the agent cannot achieve without help.

Allen's task is to write a program that infers the actor's goal from a single utterance (the goal being implicit in the utterance), and determines the actor's plan for achieving this goal. If there are any obstacles in the plan, then the program determines what the actor needs to know so that he can achieve the goal. An utterance should then be formulated to inform the actor of this information. For example, the actor (patron) might ask "When does the Montreal train leave", the program infers that the actor's goal is to catch

the train and that to do this he must know which platform to go to. So the program adds the extra information about the departure platform to the reply, answering "3.15 at gate 7" (instead of merely answering "3.15").

The recognition is performed on a single utterance in a very limited domain, which simplifies the plan recognition task considerably, there being only a few possible plans to recognise. The work by Litman (1984), builds upon Allen's work, enabling plan suspension and resumption to be modelled by using discourse markers (words which indicate a change of the speaker's intentions). These are used to recognise when the speaker is switching between plans.

TRACK (Carberry, 1988) uses plan recognition to determine a speaker's intentions from his utterances in an information-seeking dialogue. The dialogue is between an *information seeker* (user) and an *information provider* (system). She observed that actors performing such co-operative dialogue follow an organised pattern of behaviour. This enabled her to isolate heuristics to suggest the likely goal of the information seeker. She uses a *context model* to store information about the goals that the information seeker is following. This model relates the current goal to previous goals that the information provider thought were being followed. Utterances made by the information seeker suggest certain (possibly many) goals, which are analysed using the heuristics which determine the single goal that the information seeker is attempting to achieve. This goal is added into the context model, which can then be used by the information provider to generate useful responses.

2.2.6 GUIDON II.

GUIDON II (London and Clancey, 1982) is an Intelligent Tutoring System for teaching students how to perform medical diagnosis (see section 2.1.1). GUIDON II traces the questions that the student is asking and determines the diagnosis that he is attempting to "prove" (assuming that the

student is following a plan). Based upon the recognised diagnosis, GUIDON II offers the student help if necessary. The importance of GUIDON II for plan recognition is in the use of knowledge bases from an expert system as a model of the student's understanding and using this to infer his goal from an observation of his actions. There is a student modeller in the program, IMAGE, which interprets the student's actions and performs the plan recognition in order that the diagnosis path is recognised.

Plan recognition is used in the ODYSSEUS program (Wilkins et al, 1986). ODYSSEUS is an apprentice program that learns expertise from the observed actions of a specialist. ODYSSEUS uses the domain of medical diagnosis, and attempts to *justify* the actions of a specialist when he performs a diagnosis. Evidence for justifications are posted on a blackboard by knowledge sources which analyse the specialist's actions. If there is insufficient evidence that there is a justification for an action, then this is treated by ODYSSEUS as being a *learning opportunity*. Such a learning opportunity indicates that the expert system's knowledge base is incorrect or incomplete, or that the heuristics for justifying the actions need modifying. By triggering a dialogue with the specialist, the knowledge base or justification heuristics can be altered to improve the expert system's performance at diagnosis.

2.2.7 PROUST.

PROUST (Johnson and Soloway, 1985) applies plan recognition to the problem of understanding Pascal programs, with a view to finding semantic bugs in these programs. PROUST also suggests how these bugs could be fixed and how they arose. Johnson claims that "debugging requires knowledge of the programmer's intentions", however the assumption is made that the code is intended to solve the specified problem. That is, it is assumed that the student has an accurate and complete model of the intended task. It is not clear that this assumption would hold under any but the simplest of programming problems. The student's goal is explicit in the

description of the programming task, and the program must be syntactically correct. PROUST uses a library of plans to achieve the goal and attempts to match the student's program to such a plan. Once the best fit is found, a library of bugs is consulted to determine the problems that the user has.

The problems with this approach are that the plan recognition must be performed on the complete student's program, and not built-up incrementally as the program develops. Also, PROUST is limited by a fixed number of possible plans which might not correspond with the student's solution to the problem, and a fixed set of bugs that it can apply.

2.2.8 Plan Recognition in Story Understanding.

The task of understanding stories can be considered to be that of recognising goals and plans. The SAM program (Schank, 1975) uses scripts associated with specific situations to describe the information implicit in those situations, and to answer questions about the content of the story. SAM has a static goal structure which identifies the goals and sub-goals associated with a script, and enables SAM to answer "why" questions. For example, a restaurant script has the goals of eating, sitting down, ordering, etc... In answering a question, SAM can look at the next level in this hierarchy (for example, "Why did John go to the Restaurant?" SAM knows that going to the restaurant is a sub-goal of eating, and can respond with "To eat").

A particular problem is SAM's inability to cope with conflicting goals (where there are several goals which cannot all be satisfied) or changing goals. SAM has no situation-independent method of resolving these conflicts, so any information about how to treat a problem has to be encoded for that instance. A more general way of treating these problems was required, which led to the development of PAM (Wilensky, 1983). Wilensky recognised the need for a more general description of the goals, and proposed using meta-plans as a way of achieving this. However, both

SAM and PAM are unable to cope with the problem of unexpected actions which do not fit the script.

2.2.9 Summary of Plan Recognition.

Plan recognition has been used in many different areas of Artificial Intelligence, its importance being fundamental to many areas of current research. The plan recognition employed usually makes simplifying assumptions: The goal being known; the actor following a single plan; the actor making no errors and possessing no misconceptions; or the actor having perfect planning knowledge. The most general approach to plan recognition is that of Kautz, who has theoretical backing for part of his work, though resorting to heuristics to select between alternative plans that the actor might be following. However, Kautz's work requires a complete and correct action taxonomy. Also his work does not appear to be easily extensible to deal with errors and misconceptions, which must be modelled in an intelligent advice system.

Plan recognition which allowed for misconceptions is performed by the ADVISOR, but the user's goal must be available and the assumption is made that the user is following a single goal. However, this work is important because of its ability to recognise misconceptions. POISE addresses the problem of recognising the user's goal, however the only problems that POISE can recognise are when the user enters syntactically incorrect sequences of commands. This can be achieved only where the command grammar is relatively simple, unlike in the UNIX domain where the ordering of commands is not tightly constrained.

An alternative technique is to associate particular goals with certain actions used in Believer and SAM. The problem with such an approach is that an action might form part of a misconception or an error,

and that in realistic domains the number of possible goals that an action suggests may be extremely large.

In conclusion, there are no generally accepted methods for performing plan recognition, but the techniques used vary from application to application as the requirements placed on the plan recogniser vary (for example, whether the system needs to be able to recognise misconceptions, the nature of the domain, etc).

2.3 Summary.

Two important aspects of user modelling are the model itself and the ability to infer an actor's beliefs through the observation of his actions.

Researchers have been working in these areas for several years, but there are few general techniques which can be applied directly to the problem of modelling a user of the UNIX operating system. Instead, the approaches tend to have been tailored to a particular problem.

However, important aspects of the modelling problem have been identified for the UNIX Adviser, these being:

- The model should be capable of modelling the extent of the user's correct beliefs about the domain.
- The model should be capable of modelling misconceptions that the user has about the domain.
- The model should be capable of adapting to reflect the user's beliefs, both when new evidence about the user's beliefs is found, and when his beliefs change.
- The model should be capable of obtaining information through the observation of the user's actions, which requires the use of plan recognition.
- The goals that the user is attempting to achieve and his plans for doing this should form part of the user model, since they will expose the user's beliefs.

In the following chapters, a model of the user's beliefs about UNIX, a method for achieving Plan Recognition, and a mechanism for combining these techniques will be developed. It is intended that this system will fulfill the above criteria. The first step in achieving this is to analyse how people interact with UNIX, so that a realistic model can be developed.

EX BIBL. UNIV. EDINBURGEN.

Chapter 3

An Analysis of the Actions of UNIX-Users

3.0 Introduction.

To make the ideas suggested in this thesis more concrete, an application domain was chosen in which users have problems. Such a system is the UNIX operating system. UNIX offers a set of tools for manipulating, organising and analysing files. These tools are invoked through a set of mnemonic commands, and offer a terse environment which is poorly suited to novices (Norman, 1981). Problems that users possess are not adequately dealt with by help currently available in UNIX. This chapter determines some of these problems. This will enable a model of users' possible beliefs about the use of UNIX commands and a method for recognising users' possible problems to be developed.

Two methods of gathering data about the way that different users use the UNIX operating system are described. These methods are: the passive collection of commands issued by users, and recording the way that users solve set problems using UNIX. A qualitative analysis of the data obtained is made, which identifies possible errors, misconceptions and lack-of-knowledge that users have. In addition, a comparison is made between the data obtained by the two different methods.

Both sets of data were analysed for a subset of UNIX concerned with the manipulation of files and directories, and the maintenance of a filestore area. This was chosen because of its applicability to all UNIX users, and the relatively large proportion of commands issued that are involved with this task (Ross et al, 1985). However, the filestore domain could not be entirely isolated from all other domains, thus the analysis includes the "pattern-matching" and "location" (method of specifying files and moving about the filestore) domains.

Two methods of gathering data about UNIX Users were used:

- **Continuous Data Logging.** This technique involved a group of twelve volunteers who allowed their sessions on the Artificial Intelligence Department Vax computer to be monitored.

- **Problem Solving.** This technique involved the group of volunteers being asked to solve three problems concerned with the manipulation of a UNIX filestore area.

These two techniques will now be discussed in more detail.

3.1. Data Logging.

A group of twelve users, consisting mainly of students studying for an MSc in Knowledge-Based Systems in the Department, volunteered to supply data on their use of UNIX commands. Most of these students had little experience of UNIX or programming at the start of the course, which is when the data was gathered. They were asked to place a command line in their ".login" files on the departmental Vax computer (which runs 4.2 Berkeley UNIX). This command switched on logging of the UNIX commands that they issued to a file, resulting in all of the commands that the user issued being recorded in a "logging file". This method of data collection was passive and "unseen" by the user. Indeed, when the volunteers were interviewed they stated that they did not feel that their actions were being watched. Each time that the user "logged on", a date and time header was generated and appended to the file. Then as commands were typed, they were added to the file, each UNIX command corresponding to three lines of data stored, viz;

- The command line as issued (prefixed by a "#").
- The command line as expanded by the shell.
- The Status code (prefixed by a "##"): 0 indicates that the command has been performed without an error, other codes are given if an error is encountered (the code is dependent upon the command).

This logging mechanism was purposely built into the publicly used command interpreter, specifically for monitoring volunteers, and had also been used to gather data over a period of about a year. A typical script obtained by this method is given in Appendix I.

3.1.1. Analysis of "logged data".

The analysis of the logged data presented a difficult task for several reasons:

- Large amounts of data had been collected, but this contained relatively few examples of the user attempting to achieve significant tasks in the filestore domain.
- Detecting the goal that the user was trying to achieve, could at best be hypothesised from the actions. Also, the state of the filestore that the user achieved might not be the goal state that the user wished to attain, or the goal might have altered throughout the session. It was rarely feasible to ask the user directly because of the delay between the data being gathered and analysed.
- Detecting the underlying misconceptions and extent of knowledge about the domain was hindered by the above.

Despite these problems, past analysis has yielded some useful information (Ross et al, 1985). They identified a number of interesting command sequences, and indicated that such sequences were infrequent (and therefore embedded in large amounts of uninteresting and seemingly unplanned sequences of commands).

To overcome the problems given above, a series of tasks were presented to the same volunteers who were supplying the "logging" data. These tasks were designed to present the user with precisely defined goals in the filestore domain, in an attempt to make the user's lack-of-knowledge and misconceptions more visible.

3.2. Problem Solving.

The aims of presenting the problems were to:

- Detect misconceptions that the user has about commands and concepts within the UNIX filestore domain, and other closely related domains.
- To detect the extent of knowledge of the user within the domains.
- To observe sub-goals and plans which could have been used to achieve the stated goals.
- To detect habits that the user has when using the system (for example, using "ls -al" frequently).

The data gathered concerning users' misconceptions, extent of knowledge, goals, plans and habits is used to develop a model for the UNIX filestore domain. The purpose of this user model is to account for observed action sequences and generate hypotheses about possible user beliefs. These beliefs might not correspond with the user's actual beliefs, but at the very least they provide a story that may help the user understand that he has a problem and the nature of this problem.

The same twelve users who volunteered to supply data, were asked to solve three file manipulation problems on the computer. This gave a record of each user's ability, goals and methods of achieving these goals, which helped in the analysis of the problem scripts. The problems were designed to:

- Be easily understood by the subjects.
- Offer no hints about a method for achieving the specified goals.
- Be tasks that might typically be performed, or might be met in the future by the users.

- Address usage of the concepts of: moving and copying files, blocks of files and sub-trees; and the relative positions of files in a file tree with respect to the present working directory.

3.2.1. Problem Design.

A set of three problems was presented to the subjects, in which the subject was asked to manipulate files in a filestore area. The environment was configured so that it appeared to the subject that he was user "john", with the usual default profile (for example, History Substitution, Home directory, "~", etc.) working in the usual way (though because the subject did not have a login name, but was logged in under my account (jml), "~john" did not specify his home directory! This caused a problem with two of the users who were not aware of the use of "~", usually using "~name" to specify their home directory. These users had to be given tutorial help during the experiment to help them reach their home directory).

Each time the experiment was started the filestore area was created in the "/tmp" directory and the first question was presented to the subject via the visual display unit. A UNIX script was started which automatically wrote the subject's key strokes and the system's responses, to a file. The subject was requested to type the " ↑ z" (control-z) character at the end of the first problem, which finished the script and started the next problem. At the beginning and end of each problem, a copy of the filestore tree was written to the script file to record the state of the tree. At the end of problem three, the subject was automatically logged out. A paper copy of the questions was supplied to each of the subjects so that they could refer to the instructions throughout the experiment (rather than being restricted to reading the instructions at the start of the experiment).

The problems were presented as tasks to transform the filetree from one filestore "picture" to another. This method was used in an attempt to exactly specify the desired end states, yet not suggest a method of achieving the task. The design of the problems was derived through the observation of

logged data, and it was hoped that this would enable the problems that users have with file manipulation to be isolated. The form of the wording and presentation of the problems was achieved through discussion with members of the department (who did not take part in the experiment). These problems were tested using another set of volunteers, to verify the neutrality of the questions and to ensure that the problems could be easily understood by the subjects. Changes were made to the problems before they were presented to the group of volunteers. The problem texts are given in Appendix II, and a sample script is given in Appendix III.

The problem data was analysed in two different ways:

- Command sequences were analysed by hand to determine possible sub-goals and plans being followed. These observations are presented in Section 3.3.

- Commands, observed in both the experimental and logging scripts, were categorised and used to complete a "User-Profile". These "User-Profiles" could then be used to compare the commands used in the experiment with those used under normal circumstances. This comparison could then give an indication of the validity of the tasks, and the extent of the user's knowledge about UNIX. This analysis is presented in Section 3.4.

The experiment was well constrained, with clearly identified goals for the users to achieve. This made the analysis of the data easier to achieve than observing logged data through the users' normal use. After the experiment, users were consulted if there was any doubt about what they were trying to achieve.

3.3. Observations from the Experiment.

Qualitative analysis of the problem "solution scripts" identified:

- Misconceptions about commands.
- Lack-of-knowledge about commands and concepts.
- Typical errors.
- Habitual behaviour.
- Different methods for achieving the goals.

3.3.1. Misconceptions

Several misconceptions were observed in the solution scripts. These misconceptions are used to identify the class of misconceptions that are incorporated in the user model in Chapter 4. The following descriptions of misconceptions gives a complete list of the observed misconceptions.

3.3.1.1 "**" misconception.

This misconception concerns the use of wildcard character matching in commands. The misconception was exhibited by the user attempting the command;

```
mv appendix* appendices/appendix*
```

Here the user appeared to want the "*" to match across the "mv" arguments, being instantiated to successive possible values. Thus he expected the command to be equivalent to the command sequence:

```
mv appendix1 appendices/appendix1
mv appendix2 appendices/appendix2
mv appendix3 appendices/appendix3
```

In the experiment, the misconception was reinforced (in the case of one user) by the command "mv chapter* chapters/chapter*". This works, but not in the way expected by the user (in fact the shell expands the command before passing it to "mv". Thus "chapter*" expands to "chapter1 chapter2 chapter3 chapters", and "chapters/chapter*" expands to nothing. This gives "mv" the desired arguments - "mv c*" would have worked just as well under these circumstances). The fact that the "buggy" syntax worked in this case compounded the "**" misconception for the user.

The "**" misconception was found to be widespread among the subjects performing the experiment. A possible explanation for this misconception is through the incorrect extrapolation of the simple move and copy commands, which take single files as both arguments. ie. "mv file1 file2" (Such explanations have been used (Matz, 1982) to account for errors that High School students make when solving Algebra problems). The case where the last argument is a directory (eg. "mv file directory", or "mv file1 file2 directory") had typically not been mastered by the users. Hence, a natural extension of the command and pattern matching is; "mv files files", leading to the "**" misconception.

A variation of the "**" misconception, using the "?" wildcard instead of "*", was observed for one subject. The "?" wildcard is infrequently used, its effect is to match a single character (for example "fred?" would match "fred1" or "fredx", but not "fred32").

The "**" misconception is not associated with any one command, but with the UNIX command interpreter. Therefore, to produce a model of this misconception would involve a model of the UNIX command interpreter. This model is not developed in this thesis due to the complexity of the problem, also such a model is independent of a command model and it would be possible to develop this at a later date.

3.3.1.2 ".. is Home".

This misconception was exhibited by a user who expected "cd .." to always relocate the current working directory at the "Home Directory" (ie. "/tmp/john" in the experiment). An explanation for this misconception is that it arose from the subject's personal file area being configured without any sub-directories. Just before performing the experiment, the user had created a sub-directory for the first time, and was in the process of moving files into it. Thus the user's experience with file trees was extremely limited, and at that point in time "cd directory", "cd .." was always sufficient to move around his filetree. In the experiment, this misconception did not show until the user moved two levels deep into the filestore. Then a "cd .." command was issued followed by an "ls", and the user saw that most of the tree had apparently been deleted (at which point he gave up).

Another example of similar behaviour (observed in the logged data) was of a user who had a shallow filestore (which had one level of sub-directories), and could therefore move around it with "cd" and then "cd directory". Thus a path description effectively never had to be given, and the user never learnt about the fuller meaning of "..".

The ".. is Home" misconception tends to manifest itself in the "cd" command, although the user's problem is with the concept of the filestore rather than the "cd" command. Therefore a model of the concept of the hierarchical UNIX filestore is required to model this problem. Such a model is not developed in this thesis, since it can be considered independently from a command model.

3.3.1.3 "Name specifies Absolute Location".

This misconception was exhibited by users specifying a simple file name or directory name, when a path name was required. For example, in problem 1 of the tasks, the following command sequence was typed:

```
1 mkdir chapters
2 cd book
3 ls
4 mv chapter4 chapters
```

Command 4 causes a file rename; perhaps "mv chapter4 ../chapters" was intended.

This misconception occurred when changing directory and for cp, mv, etc., often leading to files being renamed instead of "changing their locations". Such behaviour was also exhibited as errors, and forgetting the present working directory. However, there was also evidence that users with shallow filestores have this misconception (since the users had difficulty in correcting the errors, once they became aware of them).

As with the ".. is home" misconception, the "name specifies absolute location" misconception is related to beliefs about the structure of the UNIX filestore, rather than any one command.

3.3.1.4 "./directory/".

This misconception was specific to one user, and it occurred when he specified a directory location. The user always added the preceding "./" and trailing "/" to the directory name. For example:

```
mv chapter? ./chapters/
```

In this example the user believes that the last argument can be a directory, provided that the "./ /" syntax is included (which is redundant). When interviewed, the subject gave the reason "UNIX needs to know the difference between a file and a directory". The trailing "/" arose because the user always uses "ls -F" to list directories; this has the effect of marking

directories with a trailing "/". The initial "./" arose through expecting to have to specify a full path description, and using "." as being short for the present working directory. The subject explained his beliefs as being a generalisation about the "/" notation for directories.

This example shows how seemingly surface syntactic problems that users have, may be derived from deeper misconceptions (for example, that a directory argument must always be a full path description).

The "./directory/" misconception is rooted in a misconception about command syntax. A model of command syntax is required to model this misconception. Such a model is not developed in this thesis, since it is unclear how to associate the intended meaning with commands using incorrect syntax.

3.3.1.5 "Manual Syntax".

The "mv -" misconception is an example where the user has misunderstood the manual. The user with this misconception gives additional syntax to "mv files directory", to give "mv - files directory" (the "-" flag is required when the first argument starts with a "-"). This misconception arose when the user was attempting to find the syntax for moving files to a directory, from the manual. The user started by copying the exact syntax from the manual, including "[" and "]", and continued through a sequence of incorrect commands attempting to get the syntax correct. The first command that worked was the "mv -" command, and since then (two months before the experiment was performed) this command variation had been used constantly (see Appendix IV for a record of the formation of this misconception).

This example illustrates the point that the format of entries in the manual can influence the beliefs that the user has, especially when the feedback from the computer supports his misconceptions. It appears to be a

common problem that users do not understand the syntax used in the manual.

The "manual syntax" is not associated with any one command, but is associated with a model of command syntax.

3.3.1.6 "cp directory to directory".

This misconception is concerned with the belief that "cp" will copy a directory containing files from one filestore location to another, with the same syntax as "mv". This is more than just a lack-of-knowledge about "cp -r", since the command is positively expected to work (possibly as an extrapolation from the "mv" command). An example of this misconception is shown in fig 3.1:

The example illustrates the strong belief that the user had in the command. The user tried to use the command twice without checking whether it worked, then attempted to "cd" to the new "directory", which failed. Also "ls -F" gave the information that two executable files had been made (shown by a trailing "*" on the listing), rather than directories (shown by a trailing "/"). So the user did not seem to be making full use of the information available by the "-F" option of "ls".

1 cp book/appendices book.bak/appendices	# copy directory to directory
2 cp book/chapters book.bak/chapters	#command repeated
3 ls -F	
4 cd book.bak	
5 ls -F	
6 cd appendices	# Try to cd to a new directory.
FAIL	

fig 3.1

An example of the "cp directory to directory" misconception.

The "cp directory to directory" misconception is associated with the "cp" command. A model of individual commands is developed in this thesis which is capable of modelling this misconception.

3.3.1.7 "Location".

The user might have had misconceptions about his location in the filestore, or the location of files (which is more than just an error, because the user believes that the locations are different from the real locations). Generally, these misconceptions were corrected quickly through command failure, or the use of "ls", or "pwd", etc.

The "location" misconception tends to result in command failure, which tends to break the user's misconception. Therefore, this problem is modelled as an error instead of a misconception. The error model is achieved by expanding the command issued by the user to include the filestore context. Incorrect contexts give rise to location errors, which are used in this thesis to identify file path errors.

3.3.2. Lack-of-Knowledge.

The user was observed performing command sequences that indicated a lack-of-knowledge. The behaviour that was observed is given in the following sections, and this will be used to develop the user model in Chapter 4.

3.3.2.1 "Commands".

The lack-of-knowledge that a command exists was difficult to detect. The mere fact that a command was not used (for example, using "cat a > b", instead of "cp a b"; or "cp a b, rm a" instead of "mv a b") was insufficient evidence for the assumption to be made that the user does not know about that command. There might be a complicating factor like the user thinking that "cp cannot be used for copying groups of files"; in which case an explanation facility needs to be able to model the user's beliefs about the command. However, the ability to model the user's belief that the command can be used is important. Advice will be based upon the assumption that the user does not know about the command, and this might then help him to understand his problem with the command. The ability to model the knowledge of commands is developed in the user model in Chapter 4.

3.3.2.2 "Parts of Commands".

Normally, users know only a sub-set of UNIX commands. Therefore, commands and parts of commands will be unknown by users. Even in the small and frequently used "filestore" command sub-set, there appear to be parts of commands that are not known. Examples found in the experiment were:

- **"mv (cp) file(s) to a directory".**

The ability to specify a directory without filenames as a location for sending files in "mv" and "cp", was often not known. For example:

```
mv chapter1 chapters/chapter1
```

instead of:

```
mv chapter1 chapters
```

This could be part of one cause for the "**" misconception, since a logical extension of this lack-of-knowledge, is the use of pattern matching to specify groups of files.

- **"cp -r".**

The use of the "cp -r" option made the solution to Question 3 of the experiment (see Appendix II) very simple. This question showed the different strategies that users adopted in an attempt to solve the problem. There were two basic strategies:

i. The users who "stuck" to their knowledge, and ignored the possible existence of a new command (these tended to be "long-term" users of the system, with one subject believing that there was not a "copy tree" command (there was not on the previous version of the operating system). They had their own particular methods of achieving this goal; for example, by creating and running an executable file, which uses the editor to take the effort out of specifying each file separately).

ii. The users who found new commands (from the manual) and tried to use them. These users had difficulty in using the command because the manual entry for "cp -r" is ambiguous. Also one user attempted to find an entry in the manual but could not find a relevant one (the manual does not mention

explicitly the ability to copy directories in the summary), and resorted to another plan.

- **"Options".**

There are numerous options (for example, seventeen for "ls", plus combinations), and users tend to be reluctant to use options, or stick to a few well known ones.

The user model developed in this thesis is capable of modelling partial knowledge of commands. This information is maintained for each command, rather than making the assumption that the user generalises concepts across commands.

3.3.2.3 "Filestore Trees".

There was commonly an apparent lack-of-knowledge about concepts in the filestore tree. For example, that ".." is the directory "one level up", or "~" is the user's Home directory.

These concepts were applicable both to "cd'ing" about the tree, and specifying locations of files and directories. Thus, the lack of these concepts caused several types of behaviour to be observed, for example:

- **"Top then Down"**

The user could not "cd" up the tree in small steps, but used "cd" to the top and then "cd" down instead.

```
cd
cd book/chapters
```

instead of

```
cd ../chapters
```

This is not necessarily a lack-of-knowledge, since the user might not have remembered his location. In which case changing directory to the top of his filestore located his current directory in a known place. However, with particular users this behaviour was observed whenever they moved across the file tree, suggesting that it was a lack-of-knowledge.

- **"Full path"**

The user specified the full path for any position up or across the tree. For example:

```
cd /tmp/john/book/chapters
```

instead of

```
cd ../chapters
```

- **"Sit at Top"**

The user sat at the top node so that both arguments of a "mv", say, could be specified down the tree. The user could not specify positions up the tree. For example:

```
cd chapters
.
.
.
cd ..
mv backup/chapter1 chapters
```

instead of:

```
cd chapters
mv ../backup/chapters .
```

The lack-of-knowledge about filestore trees could, in principle, be modelled with a model of the filestore tree and a model of the command interpretter. However, such models are not developed in this thesis, which concentrates on the problems associated with individual commands.

3.3.2.4 "Wild Cards".

The Wild card concept was generally known by the subjects, yet only a sub-set was used. For example "?" and "[...]" were infrequently used. Even when question 1 virtually forced the use of "mv chapter[123] chapters" a "*" was used instead - giving the error message "mv: rename: File Exists" (meaning that it cannot rename "chapters" as "chapters"). Although this error message does not affect the result (the command behaves as desired), it is not clear whether the error message was expected by the subjects. Typically, there was a tendency to use "rm *" and "ls *" showing specific instances of usage (that "rm *" means "remove all files" as one concept) which have not been abstracted to other commands.

A model of the command interpreter is required to model problems associated with wild cards. Such a model is not developed in this thesis.

3.3.3. Errors.

The errors that were detected with the use of UNIX are described below.

3.3.3.1 "Typing".

Typing mistakes accounted for many errors, with few users making use of the "ESC" name completion facility available on the local UNIX system. The ability to identify typing errors is included in the user model developed in Chapter 4. This is achieved by interpreting the command

typed by the user to give different possible interpretations based upon the modification of the spelling of that command.

3.3.3.2 "Pattern Matching".

Patterns matching more files and directories than the user had envisaged, caused several errors in the experiment. An example was the "mv chapter* chapters" case (which matched directory "chapters" as well as the files "chapter1", "chapter2" and "chapter3"), where a user might know that "*" matches all character sequences, yet overlook particular occurrences of the pattern. An alternative explanation to this might be a failure in planning the pattern matching, or having a misconception that the pattern will only match valid occurrences (and matching "chapters" does not make sense in this case).

Pattern matching errors require a model of the command interpreter. Such a model would be complicated and has not been developed in this thesis, although it would be possible.

3.3.4. Habits.

Numerous habits that users have were observed, examples of which are given below. Identification of these habits is important since user's of command-driven systems experience the problem of reaching a knowledge "plateau". Users retain habits to achieve tasks that can be achieved in a more efficient way.

3.3.4.1 "Move to See".

Using the command "cd" to change directory to the directory that they wish to list with the command "ls". This was observed in the

experiment, when the users have completed a problem, and do not have to complete another task, they give behaviour such as:

```
ls
cd chapters
ls
cd ..
cd appendices
ls
```

Looking at the "logging" scripts indicated that usually the users knew about "ls" taking arguments, suggesting that the behaviour was just a habit.

This habit replaces a single command by an equivalent sequence of commands called a *cliche*. The recognition of cliches is developed in this thesis through the user model and plan recognition techniques.

3.3.4.2 "Locate at departure/destination".

Some users always positioned themselves so that a "mv" or "cp" could be made from either the point of departure, or destination, for example:

```
mv chapter1 /tmp/john/book/chapter1
```

It is difficult to tell whether this was a misconception; that the destination node should always be specified in full for the "mv" command, or if it was just a habit. Again, this habit can be recognised as a cliche.

3.3.4.3 "cp, rm".

Instead of using "mv", the combination of "cp" and "rm" were often used, when it was known that the user had used "mv" before. This could be merely a habit, or a misconception that "cp, rm is safer than mv". Again the reason for the behaviour is difficult to detect, possibly being due to the user having experience of different operating systems.

Again, this command sequence can be considered to be a cliche. Such cliches are difficult to handle with conventional plan recognition techniques, which usually require an explicit statement of each cliche. This thesis develops an alternative technique which does not require cliches to be specified in a grammar.

3.3.4.4 "ls".

"ls" is often used at intervals for no particular reason, except perhaps reassurance, boredom, thinking time, etc. It is not important that this habit is modelled since it has no detrimental effects. However, it is important that commands interspersing "interesting" actions do not prevent the recognition of the user's plans. Therefore, a technique for dealing with such fragmented plans is developed in this thesis.

3.3.4.5 "Non-Default Options".

Some users always gave certain commands with particular options (for example, "ls -al"), which were not apparently needed every time it was used. It is difficult to detect when the user requires the additional information and when the command is typed as a habit. Since this problem requires a very detailed model of the user's beliefs, it will not be developed further in this thesis.

3.3.4.6 "Pattern Matching".

Full use of pattern matching was not generally made. Patterns which matched only the differing parts of the pattern tended to be used (for example, "chapter*" instead of "c*" - which would have the same effect). The general rule "use matching for as few characters as possible", seemed to be applied. However, there did appear to be other uses for pattern matching observed: To act as an abbreviation for only one possibility, or to mean "all" (eg. "rm *"). Only two subjects appeared to have a good understanding of pattern matching.

A model of the command interpreter would be required to model the pattern matching habits. For simplicity, this model has not been developed.

3.3.5. Goals and Plans.

The conceptual simplicity of the goals for the experimental problems, meant that there were few solution strategies. However, these strategies were then altered by the user's knowledge, misconceptions, habits and error recovery (including re-planning); to produce unique scripts. This analysis is used to develop a plan grammar for UNIX commands that is used as the basis for plan recognition in this thesis.

A selection of the general initial plans for the problems are given in fig 3.3. In problem 1 the user was asked to copy groups of files; problem 2 involved moving a filestore tree; and problem 3 involved copying a filestore tree.

Problem 1.

i.

Make directory chapters.
Block shift chapter1-3
to chapters.
Shift chapter4 to chapters.
Make directory appendices.
Block shift appendix1-3
to appendices.
Tidy.
Check.

ii.

Rename directory book as chapters.
Block shift chapter1-3 to
chapters.
Make directory appendices.
Block shift appendix1-3 to
appendices.
Check.

Problem 2.

i.

Make new directory book
Relocate directory
appendices in book.
Relocate directory
chapters in book.
Check.

ii.

Make new file tree.
Copy chapter1-4 and appendix1-3
to new tree.
Check.

Problem 3.

i.

Block copy tree (cp -r)
Check.

ii.

Make new file tree.
Copy chapter1-4 and appendix1-3
to new tree.
Check.

fig 3.3
Alternative methods for achieving goals.

The actual implementation of these sub-tasks varied from user to user; for example, a block shift was implemented in two different ways shown in fig 3.4.

```
cp appendix1 appendices      or      mv appendix* appendices
cp appendix2 appendices
cp appendix3 appendices
ls
cd appendices
ls
cd
rm appendix1
rm appendix2
rm appendix3
```

fig 3.4

Two Implementations of the Block Shift goal.

Error recovery was also observed, and the modification of sub-goals and plans associated with errors. An example of a user attempting to solve question 3 of the experiment, and recovering from errors is given in Appendix V.

3.3.6. Validity of Results.

The validity of the data obtained from the experiments was questionable for a number of reasons:

- An unfamiliar and alien filestore was imposed upon the users. Users may wish to structure their filestore in a particular way

so that they minimise their work-load, and this structure might not correspond to the structure of the filestore in the problems.

- Tasks were imposed upon the user. A user may never wish to copy a tree structure.
- The feeling of being watched, and not wishing to look foolish (someone is going to analyse my actions).

In order to check the validity of the results, the experiment was performed only with subjects for whom logging scripts were available. Therefore a record of typical behaviour for each of the subjects was available, which recorded their use of the machine under more normal circumstances.

3.4. Comparison between "logged" and "experiment" data.

A comparison was made between the data gathered by the experiment and the continuously logged data. The following comparisons are the results of observations and the comparison of command usage from fig 3.5.

3.4.1. Analysis of User Profiles.

The users could be divided into two main groups; those that were new to the system, and the long-term users (those that had been using the system for over one year - marked with a "#" in fig 3.5).

	Subject Number											
	A	B	C#	D#	E	F	G	H	I	J	K	L#
Command / Concept												
mkdir												
directory	*l*	*l*	*lp*	*p*	*lp*	*p*	*p*	*lp*	*p*	*lp*	*lp*	*lp*
directories								*lp*				
ls												
no arguments	*lp*	*p*	*lp*	*lp*	*lp*	*lp*	*lp*	*lp*	*p*	*lp*	*lp*	*lp*
directory			*lp*	*lp*	*lp*	*lp*		*l*		*p*	*lp*	*l*
directories				*l*	*p*							
- R (tree)												
-a				*l*	*p*	*l*				*l*	*l*	*l*
-l			*l*	*l*	*p*	*l*				*l*	*l*	*p*
Other Flags			*l*	*lp*							*l*	
cat												
file	*p*		*l*	*lp*	*l*	*l*			*l*	*l*		*lp*
files				*l*		*l*				*l*		
options												
cd												
no arguments	*l*		*l*	*l*		*lp*		*l*		*lp*	*l*	
~x		*l*	*l*	*lp*		*l*		*lp*			*l*	*l*
~/x												*l*
".."	*p*	*p*	*l*	*lp*	*lp*	*lp*	*lp*				*lp*	*lp*
single level	*lp*	*p*	*l*	*l*	*lp*	*lp*	*lp*	*lp*	*p*	*lp*	*lp*	*lp*
multi-level	*p*	*l*	*l*	*l*	*lp*	*lp*		*lp*	*p*	*l*	*l*	*l*
pattern matching												
*		*p*	*p*	*p*	*p*	*lp*	*p*	*p*	*p*	*p*	*lp*	*p*
?				*p*								
[]												

Key:

l	Indicates commands observed in the user's "logging script".
p	Indicates commands observed in the user's solutions to the experiment.
#	Indicates a user who has used the UNIX system for several years ("long term user").

Summary of Users' Profiles.

fig 3.5.

Command / Concept	A	B	C#	D#	E	F	G	H	I	J	K	L#
cp												
file to file	*l*	*p*	*l*		*lp*	*p*	*l*	*l*	*p*	*lp*	*p*	*lp*
file to directory			*l*									*l*
files to directory			*l*					*lp*				
-i												
-r (tree)	*p*	*p*	*lp*		*p*		*p*					*p*
"."												*p*
".."						*lp*						*p*
"~"			*l*				*l*	*lp*				
pattern matching		*			*	*	*		*			
mv												
file to file	*p*	*p*	*l*	*lp*	*l*	*lp*				*p*	*lp*	*l*
file to directory	*p*		*lp*	*l*			*lp*					*l*
files to directory	*p*		*p*				*lp*					*p*
directory to directory						*p*					*p*	*p*
-f												
-i												
"."												*p*
".."												*lp*
"~"			*p*				*p*					*l*
pattern matching		*				*						*
rm / rmdir												
file		*p*	*l*	*l*	*lp*	*l*	*l*	*l*	*p*	*lp*	*l*	*l*
files				*l*	*lp*	*l*	*p*	*lp*	*p*	*p*	*lp*	
directory	*p*	*p*	*l*	*p*	*lp*	*p*	*p*	*p*			*p*	*p*
directories								*p*				
-r (tree)			*lp*			*p*						
-f			*l*									
-i			*l*									

Key:

l	Indicates commands observed in the user's "logging script".
p	Indicates commands observed in the user's solutions to the experiment.
#	Indicates a user who has used the UNIX system for several years ("long term user").

Summary of Users' Profiles.

fig 3.5 (Cont'd).

3.4.1.1. Long-Term Users (Subjects C, D and L).

These users completed the tasks fairly confidently, yet with different methods.

Subject C did not use "mv directory directory" for problem 2, but used "cp -r" and "rm -r" to achieve the same effect. The "logging" script confirmed this to be typical behaviour, and when interviewed the user claimed that this method was "safer".

Subject D did not use "cp" at all, but used "cat a>b" instead. Again the logging script confirmed this as being typical behaviour, and when interviewed the user said that he did not know about the "cp" command. The use of "cat" also caused problems with pattern matching in problem 3, since a directory destination could not be specified. This promoted the user's belief in the '**' bug, though the user knew the correct syntax for "mv files directory" and had not exhibited this bug before.

Subject L was sure about the commands that he issued and had strong beliefs that there was no easy way of copying a tree. The user explained that there used to be a "cptree" command, but this was not a feature of the new implementation (Berkeley 4.2) of UNIX. To achieve the "copy tree" goal the user created and ran an executable file which performed the copy commands needed in a long-hand version of the plan, using the editor to take the "donkey work" out of typing the commands.

3.4.1.2. New Users (Subjects A,B,E,F,G,H,I,J and K).

These subjects were new to this particular version of UNIX, some being used to other operating systems, others new to computers. Of particular note was subject E, who had used an operating system that accepted the "**" method of copying and moving files. This user consistently tried to use the feature on this system, and did not appear to be daunted by the commands' failure and apparent inconsistency (possibly due to the "mv

chapter* chapters/chapter*" working by chance). The logging script also showed attempted use of the "**" syntax.

3.4.2. General Observations.

The "cp/mv file(s) to directory" commands were used by very few users, although both problems 1 and 2 were easily solved by their use. There also appears to be a correlation between users not knowing the "mv/cp to directory" concept and attempted pattern matching to do a block copy with the "**" bug (subjects B,E,F,G and I). However, subject D who used "cat" instead of "cp" exhibited the "**" bug and yet the "mv file directory" command was known to the user. Also subjects H, J and K used only simple pattern matching (eg. "ls *"), and subject A used no pattern matching at all.

"cp -r" was used more in the experiment than generally observed in the logging script (only subject C made general use of the command). The reason for the increased usage was because it made problem 3 a simple task and the subjects expected that there was an easy way to achieve the task. Also, the task of copying a whole tree is likely to be one that is used infrequently in general, and therefore its use would probably not be observed in the "logging" scripts.

Little use was made of flags by the users in either general use, or during the experiment. Notably, no use was made of "ls -R" to recursively list the tree during the experiments.

The use of "~/" with "cd" is surprisingly low, only being observed for subject L. Two subjects used "cd ~name" for this purpose, otherwise appropriate use of "cd ..", or "cd" did the trick.

For pattern matching, "*" was generally used in the experiment (by eleven subjects), though only two users had been observed using "*" in the

"logging" scripts. Only subject D had been observed using "?" and no usage of "[...]" had been observed.

3.4.3. Interpretation of Results.

When comparing the "logged" versus "experimental" data from fig 3.5, account needs to be taken of the way that the data was collected. The logging data was mainly collected over a period of two months, near the beginning of term (data had been gathered for a period of over a year for two volunteers). Many of the users were MSc students, and over this period their use of the machine was light. Thus, the absence of "*l*"'s in fig 3.5 does not necessarily mean that the user does not know about a feature, just that it had not been used during the sampling interval. For example, subjects B and I did not appear to know about "ls with no arguments" according to the "logging" script, yet they made use of the command in the experiment. Also, the analysis does not record whether the commands had succeeded or failed (This would not be a trivial task since we would need to know whether the command fulfilled the user's goal, not if the computer could perform the instruction without generating an error).

3.5. Discussion.

The use of experiments to gather data about UNIX users has identified that users do have misconceptions about the system, make errors and have incomplete knowledge about the system.

The observed misconceptions appear to fit into two categories; stable and transient misconceptions. A stable misconception is one that has been retained over a large number of commands (for example, the "./directory/" misconception, which had remained for over a year). A transient misconception is one that lasts for only a few commands (for example, the incorrect belief about the location of a file). The transient misconceptions tend to be broken quickly by an error, or the user noticing unexpected files in a directory listing. The user model would need to be able to model both of these characteristics if it were to shed light upon the user's activities.

Some numerical analysis of commands issued by UNIX users has been performed (Hanson et al, 1984). This work shows that only a small number of commands are used frequently (10% of commands account for nearly 90% of command usage). They claim that the frequently used "core" commands are used in groups. Some commands (eg. "ls" and "cat") are used before and after a large number of other commands, and others (eg. "cp") are usually preceded by a particular command (eg. "chmod" before the "cp" command).

Other work concerning the analysis of UNIX command sequences (Bannon et al, 1983) claims that complex, interleaved tasks are performed by users and that the command sequences are structured and coherent. They state that:

> In our empirical data, the annotations that users added to their history lists showed that they view their interactions with the computer in terms of goals rather than system commands (Bannon et al, 1983).

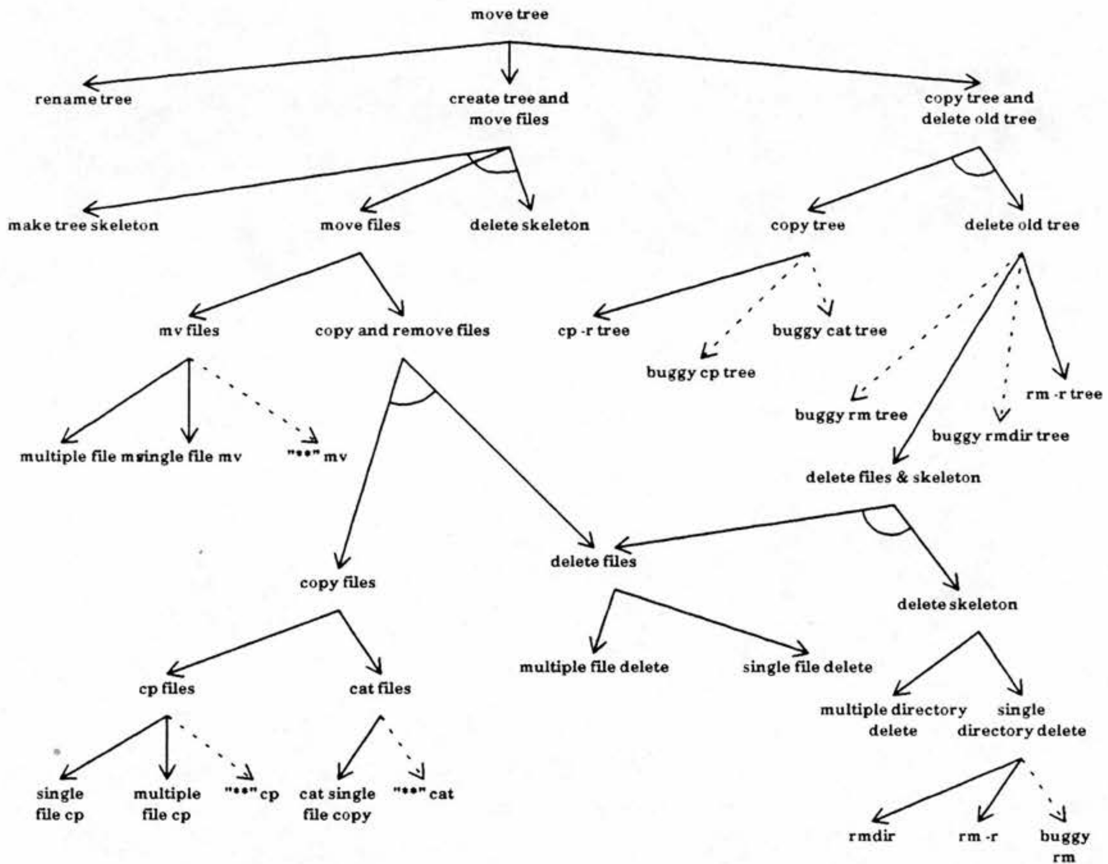


fig.3.6.
A Decomposition of the "move tree" Goal.

The results from the experiments show structured command sequences, but it could be that this structuring has been imposed upon the user by the experiment. The logging scripts show little visible structuring in general, with simple tasks being performed. Only occasionally are complex, planned sequences encountered. Little interleaving of tasks has been detected in the scripts. However, the ordering of sub-plans to achieve a goal varies from user to user, which could be accounted for by the interleaving of

plans or through a different decomposition of the goals. A plan hierarchy was developed to account for users' actions when attempting to move files (fig.3.6). This hierarchy consists of an "and / or" tree of actions to achieve the goal "move tree". The hierarchy was developed by analysing the commands issued by users as solutions to the data gathering experiments. The goals were known for each of the scripts from the experiments, and the assumption was made that all actions are issue in order to achieve these goals. The dashed lines in the hierarchy represent incorrect actions which were a result of the user possessing misconceptions. The hierarchy is not intended to be complete, but it will be used as the basis for developing a plan grammar to test the ideas developed in this thesis.

The development of some misconceptions seems to be driven "bottom up". That is, the user knows what he wants to do, and it is the error messages, manual entries and incorrect extrapolation of commands to cover more cases, that produce the misconceptions. In particular error messages often give harmful feedback by confusing the user when he needs support. These messages undermine the user's confidence in commands and concepts. The compounding of misconceptions can occur when a command apparently achieves the desired effects that the user has in mind, when in fact it is pure chance that the command works (for example, "cp chapter* chapters/chapter*" described in section 3.3.1.1).

The results show that users possess misconceptions about commands and concepts even in such a small command subset that the filestore domain offers, giving supporting evidence that the type of user-model needed to describe this domain would also be applicable to UNIX as a whole. In other parts of UNIX it is extremely likely that users do not know about the existence of commands, for example "grep", or "uniq". Also, when these commands are used, it is probable that users will develop misconceptions about their usage.

The results also showed that the filestore could not be totally isolated from other domains, the users' actions also depending upon

pattern-matching and the locations of files and the present working directory. This indicates that a user-model would have to model at least several domains and the dependencies between them (for example, the "**" misconception can be viewed as the interaction of the lack of the "mv file directory" command, and the use of pattern matching to specify more than one file).

The experiments in the filestore domain exhibited the planning and re-planning strategies of users, and illustrated the necessity for plan-recognition in order that the user's actions can be "understood".

The experimental situation did appear to colour the actions observed (as observed by the use of the "cp -r" command), but this can be justified by the expectation that at some point the users will want to achieve the tasks specified by the experiment. If a more complete and accurate picture of UNIX users were required, then a much larger sample of users would need to be taken and the methods of collecting data more carefully studied so as not to colour the results (Draper, 1983). The gathering of this data was not intended to give a complete and accurate description of UNIX users, but to supply data which could be used to support ideas on plan recognition in the UNIX domain and the development of a user model.

A model of UNIX commands enables a wide variety of misconceptions, lack-of-knowledge, and errors to be modelled. These models are used with plan recognition techniques developed in Chapter 5 to enable advice to be generated which is specific to the user's problems. This advice generation is developed in Chapter 6, and is based upon the user having incomplete knowledge of commands and parts of commands, misconceptions about commands, command cliches, typing errors and filestore errors. The structure of the UNIX filestore is an integral part of the problem of offering advice, therefore a filestore model is required by the UNIX ADVISOR. This filestore model is used to enable typing errors and file path errors to be accomodated in the advice.

3.6 Summary.

The data logging and "problems" have supplied large amounts of data upon which a representation of users can be built. Clearly, these experiments will not cover all the possible plans, errors and misconceptions that can occur, but they give the basis for developing a user model and a mechanism for recognising the problems experienced by UNIX users.

The data gathering indicates that UNIX -users have problems with specifying individual commands. These problems can be divided into lack-of-knowledge of commands or parts of commands, misconceptions about the preconditions and effects of commands, and errors in typing or path specification.

Chapter 4 develops a user model of possible beliefs about UNIX commands. This model enables misconceptions and the extent of the user's knowledge to be modelled for each command. A model of the filestore structure is also developed which gives the UNIX ADVISOR the capability of identifying typing errors and path errors.

Chapter 4

Modelling Users' Beliefs about UNIX Commands

4.0 Introduction.

A prerequisite for being able to detect a user's lack-of-knowledge and misconceptions, is the ability to model these problems. The analysis of the UNIX domain described in Chapter 3 suggested a range of such problems that UNIX users experience. This chapter develops a model of the user's beliefs about UNIX commands. The model is able to represent a class of misconceptions that users have, and their lack-of-knowledge of these commands. The model adapts to an individual's beliefs through the observation of actions issued by the user. The information contained in the user model can then be used as the basis for offering advice. However, no claims are made in this thesis that the model corresponds to the user's mental model of UNIX. The model is purely a method for representing possible misconceptions that users may possess, and the extent of their knowledge about UNIX. Such a model can then be used as the basis for automatic advice generation.

4.1 Modelling Users' Problems.

The analysis of the commands that users issued, to achieve tasks in the UNIX filestore, provided evidence that many of the users' problems can be described through:

- Misconceptions. The user has incorrect beliefs about the preconditions or effects of a command. For example, the user might believe that "rm book" will remove directory "book" from the filestore (an incorrect precondition for the "rm" command, which must be files in simple cases), or the user might believe that "cp book book.bak" has the effect of copying the directory "book" to create a new directory "book.bak" (an incorrect belief about the effects of "cp").
- Lack of Knowledge. The user is unaware of the existence of commands or parts of commands. For example: not knowing about the "cp" command, or not knowing about the "move directory" part of the "mv" command.
- Errors. These occurred in two forms; Typing errors (for example "cp mbx mbox.bak" instead of "cp mbox mbox.bak"), and Path errors where the user gives an incorrect path description for a file or directory (for example "cp /tmp/john/book/chapter1 chapters" instead of "cp /tmp/jml/john/book/chapter1 chapters").

From the analysis of the use of UNIX commands (Chapter 3), it was decided to model the misconceptions and lack-of-knowledge associated with an incomplete or incorrect functional model of commands. This range of problems was selected because they are clearly defined and extensible to other UNIX commands. A simple STRIPS-like model (Fikes and Nilson, 1971) can be used to represent the commands and the associated problems. Problems associated with the command interpre ter would require a much more complicated user model.

An incomplete model of commands will enable the user's lack-of-knowledge to be modelled. For example, the user might not know about the "-r" flag for copying file trees (Section 3.3.2.2), that it is possible to move files to a directory (Section 3.3.2.2), or not knowing of the existence of the "cp" command (Section 3.4.2). An incorrect functional model of commands will enable the user's misconceptions about the commands to be modelled. For example, attempting to copy a directory with a simple form of the "cp" command (see Section 3.3.1.6), attempting to delete a directory using a simple form of the "rm" command, or attempting to copy a file tree with the "cp -r" command variant and specifying the destination incorrectly (Section 3.3.2.2).

Misconceptions about pattern matching, for example, the "**" misconception (Section 3.3.1.1) and path specification, for example, ".. is home" (Section 3.3.1.2) will not be modelled since they would need to use a much more complicated model of the command interpreter.

Errors were observed in the mis-typing of names (Section 3.3.3.1), and errors could also be made in the incorrect specification of a file path. Errors will not form part of the user model. Instead, they will be detected through the incorrect instantiation of the detailed command representation used by the command modeller.

This thesis does not attempt to model the user's learning processes since this would be a very difficult task. Misconceptions can be caused by numerous sources, and it is not clear how users develop these misconceptions. It is also not clear when and how generalisations are made by the user for commands, or between commands. Without this information it is not possible to develop a model of learning. Instead, this thesis develops the idea of user modelling as a search for evidence concerning the extent of knowledge, errors and misconceptions which are consistent with the user's correct and incorrect behaviour. Even this much simpler task is very

difficult, since errors and misconceptions can combine in many ways to produce other (perhaps correct) behaviour.

4.2 Modelling UNIX Commands.

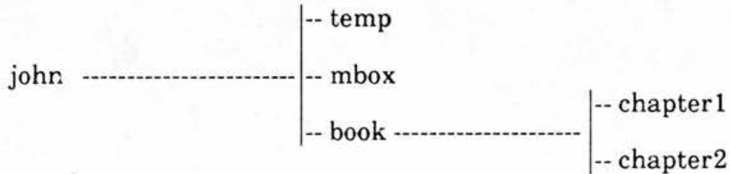
The task of the user model of commands is to be able to represent the lack-of-knowledge and misconceptions that a user possesses, and not how these problems arose. Therefore, the model will represent the user's beliefs without providing any details about the roots of those beliefs.

A STRIPS-like (Fikes and Nilson, 1971) model of the preconditions and effects of UNIX commands is described. This is used in conjunction with a similar model describing a range of possible beliefs that a user could have about UNIX commands. A comparison of these two models enables the user's misconceptions and lack-of-knowledge to be represented. These models of UNIX commands are executable models, which require a corresponding model of the UNIX filestore for their operation. In effect, the user models perform a simulation of the filestore. This is necessary to determine the effects of different parameters contained in the model.

4.2.1. A Model of the UNIX Filestore.

The UNIX filestore model is maintained as a list of "*Filestore Nodes*". Each filestore node represents a file or directory in the UNIX filestore, and has a list of attributes associated with it. These attributes describe the essential features (for the file manipulation commands) of the node, and can contain any information about the node (for example, the last time the file was read or written, the contents of the file, etc). However, the current model uses the following attributes; The *Name* of the node (ie. the name of the file or directory), the *Location* of the node in the filestore (eg. "john/book", where "john" is the root of the filestore), the *Links* to sub nodes from that node (eg. a directory may contain files and directories), the *File type* of the node (ie. whether the directory is a file or directory), and whether the file is *Executable* (directories and some files are executable). Other file node attributes such as file size, creation dates, access permissions, etc, could

be modelled using this scheme. However, for simplicity these additional attributes were omitted. An example filestore is shown in fig. 4.1;



An example filestore.

fig.4.1.

In the filestore shown in fig 4.1, the file "chapter1" has attributes:

Name	chapter1
Location	john/book
Links	[]
FileType	file
Executable	not_executable

and the directory "book" has attributes:

Name	book
Location	john
Links	[chapter1, chapter2]
FileType	directory
Executable	executable

Two operators "*add_node*" and "*delete_node*", add and delete a specified node and sub nodes from the filestore model respectively. By applying combinations of these operators to the filestore, the behaviour of the UNIX filestore commands can be simulated and a model of the current filestore maintained. This model of the filestore is needed to interpret the

context of commands as they are typed by the user, and therefore enable the semantics of that particular command to be deduced.

4.2.2. A Model of UNIX Commands.

The UNIX filestore commands are modelled as STRIPS-like (Fikes and Nilson, 1971) representations of the *preconditions* and *effects* of the commands.

The preconditions consist of two parts: A list of conditions that must hold for the command's arguments (for example, the file type = "file"), and constraints detailing the necessary relationships which must hold between arguments (for example, in fig 4.2, that the created file must be different from the existing file; [Name1|Location1] <> [Name2|Location2]). The filestore location for each file is maintained as a list, thus the requirement that the source and destination files are different is achieved through ensuring that the corresponding lists do not match.

The effects take the form of a list of additions to, and deletions from the current UNIX filestore that arise due to the command.

Each UNIX command has, typically, a number of distinct tasks which it can perform. For example, the "cp" command can be divided into the following variants (ignoring "cp" with more than two arguments, and taking examples from the filestore shown in fig.4.1);

i. copy a file to an already existing file ("cp mbox temp").
ii. copy a file or directory to form a new file or directory ("cp mbox mbox.bak").
iii. copy a file to an existing directory ("cp mbox book").

These functional components of the command are modelled as distinct command variants in the UNIX command model, with preconditions and effects. For example, the command variant to copy a file or directory to an existing directory is given in fig.4.2.

```
unix_command(                                                           % 1
        copy_dorf_to_directory,                                         % 2
        cp,                                                             % 3
            args(   [                                                   % 4
                    [
                    Flags1          range  [[]]                         % 5
                    ],
                    [
                    Name1           range [_] ,                         % 6
                    Location1       range [_],                          % 7
                    Links1          range [_],                          % 8
                    FileType1       range [file, directory],            % 9
                    Executable1     range [_]                           % 10
                    ],
                    [
                    Name2           range [_],                          % 11
                    Location2       range [_],                          % 12
                    Links2          range [_],                          % 13
                    FileType2       range [directory],                  % 14
                    Executable2     range [_]                           % 15
                    ]
            ]),                                                         % 16

      constraints(  [                                                   % 17
                    [Name1|Location1] <> [Name2|Location2]              % 18
                    ]),                                                 % 19

      actions([                                                         % 20
                    [add_node,      [                                   % 21
                                    Name1,                              % 22
                                    [Name2|Location2],                  % 23
                                    [],                                 % 24
                                    file,                               % 25
                                    Executable1                         % 26
                                    ]                                   % 27
                    ]
            ])                                                          % 28
      ).                                                                % 29
```

Command model for UNIX command to "copy a file or directory to a directory".

fig.4.2

The UNIX command variants are maintained as PROLOG unit clauses. Each clause has the predicate name "unix_command" (line 1) followed by five arguments;

The first argument (line 2) gives a name to the command variant ("copy_dorf_to_directory").

The second argument (line 3) gives the command type ("cp").

The third argument (lines 4 to 16) gives a list of the arguments that the command variant will accept. In fig.4.2. there are three arguments: the flags (line 5), the source file (lines 6 to 10), and the destination file (lines 11 to 15). Usually the argument list consists of a flag list, and lists corresponding to the arguments that the command takes. Each argument in the list takes the form of a list of parameters, for example, "FileType1 range [file, directory]" (line 9). In this instance the variable "FileType1" is allowed to match a description which has a value of "file" or "directory". The full description of the argument, therefore, specifies the possible range of filestore nodes that could match the argument. A parameter with the range "_" can match any value.

The fourth argument (lines 17 to 19) describes the constraints which must hold between parameters, for the command to be valid. In fig.4.2, this describes that the source and destination filestore nodes must be different.

The fifth argument (lines 20 to 28) describes a list of actions that the command variant will perform. In fig.4.2, this is adding a node to the filestore.

4.2.3. Using the Model of UNIX Commands.

When the user types a command this is translated into a complete, fully instantiated description of the command by consulting the current filestore context model. This translation is achieved by searching the filestore model for filestore nodes which match the command's arguments. The form of the filestore description is:

[Name, Location, Links, FileType, Executable]

where the elements of this list have the same meaning as that given in section 4.2.1. For example, with the filestore given in fig 4.1, the arguments for the command "cp mbox book" expand to:

"mbox" expands to [mbox, [john], [], file, not_executable]
"book" expands to [book, [john], [chapter1, chapter2], directory, executable].

The command is matched against the possible filestore commands (defined in fig.1.1) and the flags used are made into a list of flags, giving a fully expanded command for "cp mbox book" of:

```
[
   cp,                                                        % 1
   [],                                                        % 2
   [mbox, [john], [], file, not_executable],                  % 3
   [book, [john], [chapter1, chapter2], directory, executable] % 4
]
```

that is; the command "cp" (line 1), with an empty list of flags (line 2), and the two expanded arguments each describing a filestore node (lines 3 and 4).

The next stage is to consult the UNIX command dictionary (model) in an attempt to match this command against one of the possible valid command descriptions. The expanded command matches the command variant "copy_dorf_to_directory" (shown in fig 4.2), giving the instantiated command model shown in fig 4.3.

```
unix command(                                                          % 1
        copy_dorf_to_directory,                                        % 2
        cp,                                                            % 3
            args(   [                                                  % 4
                        [
                        []                      range   [[]]           % 5
                        ],
                        [
                        mbox                    range [_] ,            % 6
                        [john]                  range [_],             % 7
                        []                      range [_],             % 8
                        file                    range [file, directory],  % 9
                        not_executable          range [_]              % 10
                        ],
                        [
                        book                    range [_],             % 11
                        [john]                  range [_],             % 12
                        [chapter1,chapter2]     range [_],             % 13
                        directory               range [directory],     % 14
                        executable              range [_]              % 15
                        ]
                    ]),                                                % 16

            constraints(    [                                          % 17
                        [mbox,john] <> [book,john]                     % 18
                        ]),                                            % 19

            actions([                                                  % 20
                        [add_node,      [                              % 21
                                        mbox,                          % 22
                                        [book,john],                   % 23
                                        [],                            % 24
                                        file,                          % 25
                                        not_executable                 % 26
                                        ]                              % 27
                        ]
                    ])                                                 % 28
        ).                                                             % 29
```

An instantiated UNIX command description for command variant "copy file or directory to directory".

fig 4.3

The uninstantiated arguments in the UNIX model become instantiated to the values of the arguments of the command, if the parameters lie within the specified range of values (for example, in fig.4.2

line 9 gives a possible range of values of [file,directory]; thus the parameter FileType1 can be instantiated to one of these values only). The instantiation of the model results in the instantiation of the constraints and effects (in fig.4.3); to give constraints "[mbox, john] <> [book, john]" (line 18), and effects of "add node([mbox, [john,book], [], file, not_executable])" (lines 21 to 27). The constraints are evaluated to ensure that they hold true, and the effects of the command are executed in the filestore model, to generate the next filestore context. It is not possible for the execution of the instantiated command to fail, because the command is syntactically and semantically validated before the execution is attempted. In the case where the user types a command that would fail, this may be due to the command not being fully instantiated, not matching a valid UNIX variant, or the constraints not holding. In any of these cases, the command interpretation will not reach the stage where it is executed in the filestore simulation.

4.3. A Model of the User's beliefs about UNIX Commands.

The model of the user's beliefs about UNIX commands (user model of commands) is based upon the model of UNIX commands and the filestore model. However, this model must be able to exhibit behaviour which is different to the "standard" command model. That is, the model must be able to represent the misconceptions and lack-of-knowledge that the user might possess.

4.3.1. The User Model of Commands.

The user model of commands is similar to the UNIX model of commands, except the preconditions and effects of the command variants can be altered to suit the observed behaviour of the user. Each attribute of the command has a range of values that the parameter can take (in the same way as the UNIX model represented possible parameter values). In addition there is also a list of possible ranges which could be substituted into the range slot. These possible facets are used to define different possible user models by modifying the range of values that a parameter can take. This in turn affects the behaviour of the model. For example, the file type might take the values of "file", "directory", or it could take either value ("file" or "directory"). The selection of these parameter values affects the behaviour of the command variant, different parameters representing different beliefs that the user has about the command. For example the "cp" command "copy-dorf-to-directory" variant has an initial user model (the user model before any changes are made - ie no account has been taken of the user's behaviour to date) shown in fig.4.4.

```
user command(inactive,                                                        % 1
        copy_dorf_to_directory,                                               % 2
        cp,                                                                   % 3
            args(   [
                    [
                    Flags1 range  ?                                           % 4
                                  possible [[[]]]                             % 5
                    ],
                    [
                    Name1         range ?                                     % 6
                                  possible [[ _]],                            % 7
                    Location1     range ?                                     % 8
                                  possible [[ _]],                            % 9
                    Links1        range ?                                     % 10
                                  possible [[ _]],                            % 11
                    FileType1     range ?                                     % 12
                                  possible [[file], [directory],              % 13
                                           [file,directory]],                 % 14
                    Executable1   range ?                                     % 15
                                  possible [[not_executable],                 % 16
                                           [executable], [ _]]                % 17
                    ],
                    [
                    Name2         range ?                                     % 18
                                  possible [[Name1],[ _]],                    % 19
                    Location2     range ?                                     % 20
                                  possible [[Location1],[ _]],                % 21
                    Links2        range ?                                     % 22
                                  possible [[[]],[Links1],[ _]],              % 23
                    FileType2     range ?                                     % 24
                                  possible [[directory]],                     % 25
                    Executable2   range ?                                     % 26
                                  possible [[not_executable],                 % 27
                                           [executable],[ _]]                 % 28
                    ]
            ]),
        constraints(  [
                    [Name1|Location1] <> [Name2|Location2]                    % 29
                    ]),
        actions([
                    [add_node,    [                                           % 30
                                  ? possible [Name1, Name2],                  % 31
                                  ? possible [Location1, Location2,           % 32
                                          [Name1|Location1],                  % 33
                                          [Name2|Location2]],                 % 34
                                  ? possible [[], Links1, Links2],            % 35
                                  ? possible [file, directory, FileType1,FileType2],% 36
                                  ? possible [not_executable,executable,      % 37
                                          Executable1, Executable2]           % 38
                                  ]
                    ]
            ])
        ).
```

Initial user model for command variant "copy file or directory to directory".

fig.4.4

In this description, each parameter is given an initial range of "?" (for example, line 12 "FileType1 range ?") which represents that no range has yet been selected. There is also a set of possible ranges for the parameters (for example, "FileType1" has possible ranges of [file], [directory], [file,directory], (lines 12 to 14)). At any instant in time the user model of commands represents the user's knowledge as the "range" parameters, so the initial representation of the user shows that he has no knowledge. The "possible" values represent different values for the parameter that the user may believe. The selection of such a belief depends upon gathering sufficient evidence to support such a choice.

The model has an additional argument "inactive" (line 1), which specifies whether the command variant has been used yet. Thus the flag "inactive" shows that the command variant has not been used, and the user might not be aware of the facility. If some part of the command variant has been used, the command flag becomes "active", though this does not necessarily mean that all features of the command are known by the user.

This model is not just an overlay model since incorrect arguments are allowed in the model. This gives rise to command variants which have incorrect preconditions and effects. The set of possible variants is very large since it arises from the combination of all possible command parameters. Thus the model is very flexible and does not have just a limited number of buggy commands.

4.3.2. Learning a User Model of Commands.

The user model of commands is adapted to represent the user's beliefs about the preconditions and effects of commands, by observing the user's behaviour. Examples of the user's actions are incorporated into the user model to develop a model of the user which is able to adapt to the user's knowledge. Consider the situation where a user always types correct and valid commands. Then, whenever a command is observed, the user model of

commands can be updated to include the command. If examples of all possible command variations are given to the model, then the user model will be equivalent to the UNIX model. However, up to this point the model will reflect the extent of the user's knowledge. For example, after observing the user's command "cp mbox book", it might be considered that there is sufficient evidence to incorporate this instance into the user model. This would transform the initial model to the form shown in fig.4.5.

Comparing this model with the UNIX model for the same command variant (fig.4.2), shows differences in the ranges of parameters. For example; The user model has; "FileType1 range [file]" (line 12, fig.4.5), whereas the corresponding range for the UNIX model is "[file, directory]" (line 9, fig.4.2). In this case, the model of the user's beliefs about the command variant is a subset of the UNIX model (that is; the model describes the user as not having complete knowledge of that command variant).

So far this model has made the simplifying assumption that the user always types correct commands. The user model needs to be extended to model the user's problems.

```
user command(active,                                                        % 1
        copy _dorf _to _directory,                                          % 2
        cp,                                                                 % 3
            args(   [
                    [
                    Flags1          range  [[]]                             % 4
                                    possible []                             % 5
                    ],
                    [
                    Name1           range [ _]                              % 6
                                    possible [],                            % 7
                    Location1       range [ _]                              % 8
                                    possible [],                            % 9
                    Links1          range [ _]                              % 10
                                    possible [],                            % 11
                    FileType1       range [file]                            % 12
                                    possible [[directory],[file,directory]], % 13
                    Executable1     range [not _executable]                 % 14
                                    possible [ [executable],[ _]]           % 15
                    ],
                    [
                    Name2           range [ _]                              % 16
                                    possible [],                            % 17
                    Location2       range [Location1]                       % 18
                                    possible [[ _]],                        % 19
                    Links2          range [[]]                              % 20
                                    possible [[Links1],[ _]],               % 21
                    FileType2       range [directory]                       % 22
                                    possible [],                            % 23
                    Executable2     range [executable]                      % 24
                                    possible [[ _]]                         % 25
                    ]
            ]),
        constraints(    [
                    [Name1|Location1] <> [Name2|Location2]                  % 26
                    ]),
        actions([
                    [add node,      [                                       % 27
                                    Name1 possible [Name2],                 % 28
                                    Location1 possible [Location2,          % 29
                                            [Name1|Location1],              % 30
                                            [Name2|Location2]],             % 31
                                    [] possible [Links1, Links2],           % 32
                                    file possible [directory, FileType1,FileType2], % 33
                                    not _executable possible [executable,   % 34
                                            Executable1, Executable2]       % 35
                                    ]
                    ]
            ])
        ).
```

User model for command variant "copy file or directory to directory", after observation of the command "cp mbox book".
fig.4.5

4.4. Modelling a User's Problems.

Now that a model of the correct behaviour of UNIX commands, and a variable model of different possible beliefs that users have about UNIX commands have been developed, it is possible to use this model to represent a class of problems that users have with UNIX commands. These problems are; Lack-of-knowledge of entire commands or part of a command's functionality, and misconceptions about the preconditions or effects of commands. In addition to these problems, users make mistakes which must be detected by the system.

4.4.1. Detecting Errors.

Users often make mistakes when using UNIX. Two particular problems are: typing errors, and path errors. Both of these types of mistake are detected by allowing commands to be interpreted with such mistakes. When a command is instantiated by consulting the simulated filestore, allowances can be made for the mistakes. These allowances are:

- Typing errors are allowed for the command and its arguments. Allowable mistypings are for the word to: contain an extra letter; be missing a letter; the letters to be a permutation of the correct letters; or for one of the letters to be incorrect.

- Path errors are allowed, where the name of the filestore node is correct, but the directory path for that node is incorrect.

Potential typing errors and path errors are generated when the command is translated into the full command description (section 4.2.3). In the case of typing errors, a single incorrect letter, one letter too many or too few, or a transposed letter is allowed. This is then matched against possible commands or filestore nodes. In the case of path errors, a name can match a filestore node at any location. These path and typing errors will be used for

the detection of errors (section 6.3.3) by instantiating command models to commands containing errors or misconceptions.

Both the path errors and typing errors are one-off mistakes that should not be modelled as misconceptions (although this is a fairly weak assumption to make, especially for the path descriptions). As such, they do not form part of the user model, but are used for generating advice about the presence of the error.

4.4.2. Modelling the User's Lack-of-Knowledge.

The user's lack-of-knowledge is modelled in two ways:

- As command variants in the user model which are as yet unused ("inactive"). For example, the command variant "cp_dorf_to_directory" might not be used by an individual, therefore there is no evidence to suggest that the user knows about that particular command variant.

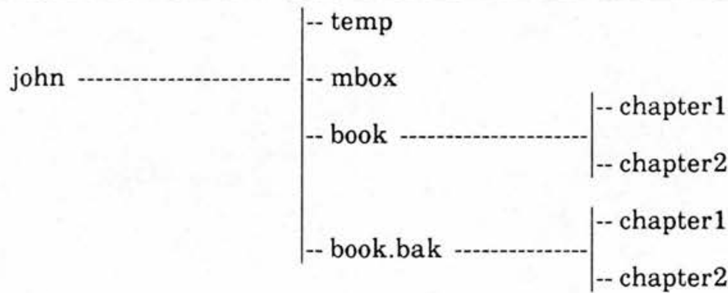
- As command variants in the user model which have not been developed to a degree where they describe all the possible features of that command variant. For example, a user might not have shown that he can move a directory into a directory whereas he can move a file into a directory (Both of these features being modelled in the "mv_dorf_to_directory" command variant shown in Appendix VI).

A particular user's lack-of-knowledge of a command can be determined by comparing the user model for a command or command variant, with the UNIX model for that command or command variant. For example, comparing the "user" and "UNIX" models for the "cp_dorf_to_directory" variant (figs. 4.5 and 4.2), shows that the user might be unaware of the feature which allows an executable file to be copied

(line 14, fig.4.5, and line 10, fig.4.2), and that the destination and initial locations need not be the same (line 18, fig.4.5, and line 12, fig.4.2).

4.4.3. Modelling the User's Misconceptions.

The user's misconceptions about commands are embodied in incorrect preconditions or effects of the command in the user model. The misconceptions for a particular command can be found by comparing the user model against the UNIX model of the command. The difference in the parameters in the preconditions and effects of the command represents one possible model of the beliefs of the user. For example, there might be evidence to support the belief that the user expects copying a directory to create a new directory. If the command "cp book book.bak" was executed on the filestore in fig.4.1, such a belief would result in the filestore given in fig.4.6.



A filestore tree created by a misconception about the command variant "copy file or directory to directory".

fig.4.6

The misconception is then captured as the difference between the two models, where parameters in the user model are not a subset of the corresponding UNIX model. The two models are shown in fig.4.9. (UNIX model) and fig.4.10. (User model). Comparing these two models, the effects

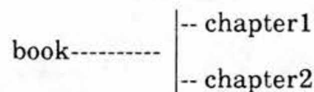
that the models cause are different (lines 21 to 27 in fig.4.9., and lines 27 to 32 in fig.4.10.). The effects of the UNIX model are to:

add_node,[Name1,[Name2|Location2],[],file,Executable1]

but the user model is:

add_node,[Name1,[Name2|Location2],Links1,Filetype1,Executable1]

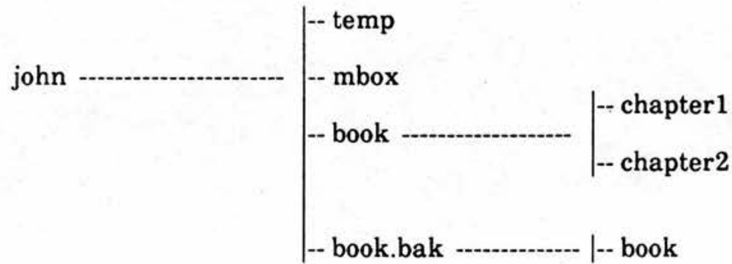
that is; the file type created by the command in the user model is determined by the file type of the first argument. For example, copying a "directory" creates a "directory". Whereas, in the UNIX model the file type created is always a "file". Also, there are no links to sub-nodes in the UNIX model, but the user model creates a node with links to copies of the sub-nodes of the file being copied. This enables the tree structure shown in fig 4.7 to be copied using the command, transforming fig.4.1. into fig.4.6.



Filestore Tree Structure.

Fig.4.7.

The UNIX command would have resulted in the tree structure shown in fig.4.8. if directory "book.bak" had been created first. This is the "copy directory" misconception (section 3.3.1.6).



Filestore Tree Structure resulting from application of the UNIX command "cp book book.bak".

Fig.4.8.

The model of UNIX commands is given in appendix VI, and the user model of commands is given in appendix VII.

```
unix_command(                                                        % 1

        copy_dorf_to_directory,                                      % 2
        cp,                                                          % 3
                args(   [                                            % 4
                        [
                        Flags1              range  [[]]              % 5
                        ],
                        [
                        Name1               range [_] ,              % 6
                        Location1           range [_],               % 7
                        Links1              range [_],               % 8
                        FileType1           range [file, directory], % 9
                        Executable1         range [_]                % 10
                        ],
                        [
                        Name2               range [_],               % 11
                        Location2           range [_],               % 12
                        Links2              range [_],               % 13
                        FileType2           range [directory],       % 14
                        Executable2         range [_]                % 15
                        ]
                ]),                                                  % 16

        constraints(    [                                            % 17
                        [Name1|Location1] <> [Name2|Location2]       % 18
                        ]),                                          % 19

        actions([                                                    % 20
                        [add_node,      [                            % 21
                                        Name1,                       % 22
                                        [Name2|Location2],           % 23
                                        [],                          % 24
                                        file,                        % 25
                                        Executable1                  % 26
                                        ]                            % 27
                        ]
                ])                                                   % 28
        ).                                                           % 29
```

Command model for UNIX command variant "copy a file or directory to a directory".

fig.4.9.

```
user command(active,                                                   % 1
        copy_dorf_to_directory,                                        % 2
        cp,                                                            % 3
            args(  [
                   [
                   Flags1        range  [[]]                           % 4
                                 possible []                           % 5
                   ],
                   [
                   Name1         range [_]                             % 6
                                 possible [],                          % 7
                   Location1     range [_]                             % 8
                                 possible [],                          % 9
                   Links1        range [_]                             % 10
                                 possible [],                          % 11
                   FileType1     range [file,directory]                % 12
                                 possible [],                          % 13
                   Executable1   range [_]                             % 14
                                 possible  []                          % 15
                   ],
                   [
                   Name2         range [_]                             % 16
                                 possible  [],                         % 17
                   Location2     range [_]                             % 18
                                 possible  [],                         % 19
                   Links2        range [Links1]                        % 20
                                 possible  [],                         % 21
                   FileType2     range [directory]                     % 22
                                 possible  [],                         % 23
                   Executable2   range [executable]                    % 24
                                 possible  [[_]]                       % 25
                   ]
            ]),

      constraints( [
                   [Name1|Location1] <> [Name2|Location2]              % 26
                   ]),

      actions([
                   [add node,     [                                    % 27
                                  Name1 possible [Name2],              % 28
                                  [Name2|Location2] possible [],       % 29
                                  Links1 possible [Links2],            % 30
                                  FileType1 possible [FileType2] ,     % 31
                                  Executable1 possible [Executable2]   % 32
                                  ]
                   ]
            ])
      ).
```

User model for command variant "copy file or directory to directory", after observation of the command "cp book book.bak".

fig.4.10.

4.5. Discussion.

The methods used to model users' beliefs about the UNIX operating system affect the ability of the advice-giving system to detect the user's problems. The main considerations are:

- The representation chosen to model users' beliefs about the UNIX domain.
- The learning algorithm used to develop a particular instance of a user model.
- Whether the representation reflects the actual misconceptions that users possess about UNIX.
- Whether the representation techniques could be mapped onto different application domains.

Each of these aspects will now be discussed further, then comparisons will be made with other misconception representation schemes.

4.5.1. Representation.

The representation used for the user model does not *explicitly* represent misconceptions and lack-of-knowledge. Instead, these can be identified by comparing the user model with the UNIX-model. It would be possible to make misconceptions explicit by, for example, asserting the misconceptions as unit clauses (see fig.4.11), or by introducing a misconception "flag" to the user model command descriptions. Representing misconceptions explicitly would allow the user model to be used to warn the user of potentially harmful effects of a command that has been issued, but before it has been executed (for example; if the user has a misconception about a command that he has issued, advice could be given to the user, or confirmation could be obtained before the command is executed). It would also be possible to retract the misconception once evidence has been gathered that suggests that the user no longer believes in it (For example,

the user might have been observed to use the command correctly where he would have used the misconception before.

```
misconception(                                                    % 1
        copy_dorf_to_directory,                                   % 2
        cp,                                                       % 3
            args(  [
                   [
                   Flags1        range  [[]]                      % 4
                   ],
                   [
                   Name1         range [_],                       % 5
                   Location1     range [_],                       % 6
                   Links1        range [_],                       % 7
                   FileType1     range [file,directory],          % 8
                   Executable1   range [_]                        % 9
                   ],
                   [
                   Name2         range [_],                       % 10
                   Location2     range [_],                       % 11
                   Links2        range [Links1],                  % 12
                   FileType2     range [directory],               % 13
                   Executable2   range [executable]               % 14
                   ]
            ]),

      constraints( [
                   [Name1|Location1] <> [Name2|Location2]         % 15
                   ]),

      actions([
                   [add node,    [                                % 16
                                 Name1,                           % 17
                                 [Name2|Location2] ,              % 18
                                 Links1,                          % 19
                                 FileType1,                       % 20
                                 Executable1                      % 21
                                 ]
                   ]
            ])
      ).
```

Explicit misconception for copying a filestore tree derived from an over-generalisation of the command variant "copy file or directory to directory".

fig.4.11.

The representation used for the user model enables a spectrum of possible misconceptions and lack-of-knowledge to be modelled. The model does not appear to be overly restrictive about the type of misconception that it can generate, although these are limited to problems that arise due to an incorrect functional model of commands. The user model does not model people's misconceptions about pattern matching or file path specification.

4.5.2. The Learning Algorithm.

The learning algorithm used in developing the user model is simplistic in several ways. The algorithm assumes that the user's knowledge about UNIX commands is *monotonically* increasing with use of the operating system. This is implicit in the method for selecting the command parameters in the user model. When a command variant is modified to incorporate more instances of usage, the parameters are selected so that the command variant describes the present use of the command and *all past behaviours* of the command that have been observed. This is achieved through the careful ordering of the lists of possible command parameters which form part of the user model. The mechanism for selecting the parameters ensures that when a parameter value is changed the new parameter is a generalisation of the previous value (for example, a file-type of [file,directory] is a generalisation of [file]). This mechanism prevents the model from specialising a command parameter, and the model cannot backtrack to a previous value. One solution to this problem would be to use a truth maintenance system (Doyle, 1979), which would allow the information in the model to change non-monotonically.

The justification for the use of this monotonic learning algorithm is that the user's knowledge will, in general, be increasing. Also, it would be difficult to detect a user *not* knowing a particular instance of a command variant, since evidence has been obtained to suggest that the user knows about that instance of the use of the command. Therefore, strong evidence

needs to be obtained to refute this information. Such information would need to be obtained from the consideration of tasks that the user is attempting to achieve, and which could have been achieved using the command instance under question. This suggests that information should be stored about the probability that particular command instances are known by the user, and it is not clear how to implement this in the present representation scheme. A possibility would be to associate a probability with each possible parameter value, and the model would "flip" between different states as the parameter values varied. For example; observation of a particular command instance would increase the probability of certain parameter possibilities, and decrease the values of others. The actual parameter value being used at any one time would depend upon the probabilities of the relevant "possible parameter values". Maintenance of such a scheme might involve heuristic analyses about the intent of commands, their relative frequencies, and alternative methods of achieving the task. This would then be capable of modelling the user forgetting instances of the use of UNIX, or the situation where other people might tell the user to type a particular command to achieve some task (but the user does not understand this usage of the command). Such situations cannot be modelled with the current representation scheme.

The learning algorithm does not make generalisations across commands. For example, discovering that the user knows how to copy a file to a directory may imply that he also knows how to move a file to a directory. However, there is no such uniformity about UNIX commands and other command-driven systems are even worse (for example, MSDOS). It would be possible to make these links in the models of commands, and this may lead to certain misconceptions being hypothesised which arise through the incorrect generalisations (for example, generalising moving a directory using the "mv" command to the "cp" command). Such a scheme has not been implemented in the UNIX Advisor, but such information could be useful in selecting between competing hypotheses about misconceptions. The problem with such an approach is that these links would need to be pre-determined for all users.

Learning is used in SIERRA (Van Lehn, 1987), which is a program that learns arithmetic skills incrementally through a series of *lessons*. Each lesson is a set of positive examples which introduce one new subprocedure, which will be used to generalise the previously learnt procedure. Unlike SIERRA, the problem of learning the user model for UNIX requires that the examples can contain errors, and that the user may introduce more than one new concept to the UNIX Advisor.

4.5.3. Models of Misconceptions.

The misconceptions described so far have been concerned with the functionality of the commands. The question arises as to whether such a representation captures the user's misconceptions, or merely the *symptoms* of these misconceptions. For example; the problem of copying a sub-tree using a simple copy command and expecting this to create a new sub-tree, might be just a symptom. The misconception might be rooted in the user having had previous experience with other operating systems with which he is making analogies.

These "deeper" reasons for the behaviour are much more difficult to determine and model than the functional view of commands. Such reasons would require knowledge of the user's past experiences, and general knowledge of the world. It would be difficult, if not impossible, to place bounds on the knowledge needed. Thus the problem becomes either computationally expensive, or intractable. Further, the benefits of such explanations for behaviour are questionable. The purpose of the user model is to aid the user in developing a complete and consistent model of the UNIX operating system, and it should be possible to achieve this with the existing model (although analogies could improve the learning process).

4.5.4. Application to Other Domains.

The UNIX domain enables the representation to be partitioned into discrete, independent functional descriptions of the commands. At present, any domain which was being considered would have to be partitioned in a similar way. This would not cause a problem with most computer-based systems, since they tend to be written using functionally independent commands to achieve different tasks. However, the representation also requires that the parameters used in the command descriptions should be capable of representing the possible misconceptions. Selecting these parameters depends upon a careful analysis of the domain and users' problems with its use, but it also depends upon the task being achieved by the command. For example; in an editor the command "↑b" might mean "move the cursor back one character". Such a command is very simple, and it is difficult to envisage how users could have misconceptions about such commands (perhaps they cannot, but merely make mistakes), and if they do have misconceptions, how should they be modelled? Thus the commands in the domain to be modelled need to have a certain amount of structure which enables users to possess misconceptions about the commands. In domains such as editors, it seems more likely that users have misconceptions about the use of *command sequences*, which the present representation is unable to model.

The representation of commands is able to model the user's knowledge of the functionality of UNIX commands. The model does not address the misconceptions related to the file path specification (for example, ".. is home misconception") or pattern matching (for example, "** misconception"). However, the model captures the user's lack-of-knowledge about commands (for example, being able to move a directory). Errors are not represented in the user model since they do not reflect the user's beliefs. Instead, they are suggested through the incorrect instantiation of the command translation.

4.5.5. Comparisons with Alternative Methods of Modelling Misconceptions.

DEBUGGY (Burton, 1982) needs the problem and the child's solution to be provided. In the UNIX domain this would correspond to the user's intentions being known, which is clearly difficult to determine. There are, however, strong similarities between the representation used in DEBUGGY and the representation used in the UNIX user model. Both models represent the function of actions, and how these functions might be altered to produce different behaviour. The altered models represent misconceptions, which in the case of DEBUGGY are fixed and explicit, and in the UNIX user model are variable. However, the misconceptions are not totally explicit in the UNIX user model (though the model could result in the generation of explicit buggy-like misconceptions).

As in the DEBUGGY model, WHY (Stevens et al, 1982) uses explicit descriptions of misconceptions, which have been found by careful analysis of the domain (for example, the "cooling-by-contact" misconception). However, the UNIX Advisor does not rely upon such pre-defined high-level misconceptions, instead the misconceptions remain implicit and are derived as a consequence of the chosen *representation* of the system.

The approach that the ADVISOR (Genesereth, 1979) takes in misconception and error detection is very different to the DEBUGGY or WHY approaches. Instead of having built-in misconceptions and errors which were determined by a lengthy analysis of users' behaviour, a correct model of problem solving is used. This model is based upon the problem-solving strategies used by novices. The misconceptions and errors are then determined as a divergence from a correct solution, and are therefore described at the level of elements in the solution rather than a high-level description of a bug given by WHY. This means that the misconception does not have an explicit high-level description, but is more a description of what has gone wrong. The UNIX Advisor follows such an

approach, where the advisor determines how commands can be put together to create higher-level tasks. Differences occur because Muser cannot generate incorrect solutions, whereas the UNIX Advisor creates multiple possible interpretations based upon different bugs (this is necessary because the UNIX Advisor does not have a description of the user's desired goal state).

4.6. Summary.

A model of the user's beliefs about UNIX commands has been developed. This model requires examples of the use of the commands so that a model specific to a particular individual can be learned. With this model it is possible to model a range of misconceptions and lack-of-knowledge that has been observed in users of the UNIX operating system.

Chapter 5

Active Chart Parsing and Plan Recognition

5.0 Introduction.

The ability to generate plausible beliefs that a user might have about the use of UNIX commands is insufficient for giving advice. The advice generation problem requires that advice should be supplied to the user when he has a problem. This advice should take into account the user's present tasks as well as his knowledge of the domain. Therefore, the user's plans and goals need to be inferred from the observation of his actions. This task is known as plan recognition. A typical approach to plan recognition is to compare observed actions against a library of plans. The circumscription approach (Kautz, 1986) was not used because of the complexity of the plan recognition problem in the UNIX domain, its requirement for a complete grammar and there being no clear way to incorporate errors and misconceptions. The blackboard approach (Carver et al, 1984) was not used because of the overhead involved with running the expert system (Hayes-Roth, 1985). A flexible approach with relatively low computational overheads is to parse the action sequence according to a plan grammar. This chapter develops the use of Chart Parsing techniques to achieve plan recognition.

5.1 Parsing using an Active Chart.

In Natural Language Understanding, sentences are parsed to build syntactic structures which capture the structure of the utterance. There is, typically, some ambiguity in deciding which grammar rules to apply as the parse progresses. In a conventional depth-first and backtracking approach, such as the use of a PROLOG definite clause grammar, this can lead to inefficient parsing due to the repeated recomputation of parts of the parse.

Active Chart Parsing was developed to overcome these problems. It is based on the concept of *well-formed substrings* (Earley, 1970). In this approach, a record is kept of the constituents of the parses which have been found, and the partial parses which have still to find constituents. These partial parses can then be used later in the parse without having to recompute them. The chart parser used was based upon an existing parser (Ross, 1989). There are many descriptions of chart parsing, see for example (Winograd, 1983) or (Thompson, 1981). However, for completeness, I will outline the basic concepts of Chart Parsing.

5.1.1 The Basics of Chart Parsing.

Given a sentence to be parsed, the words of the sentence are first separated by vertices. For example, the sentence "The man saw the ball" is shown in fig 5.1.

V_0 the V_1 man V_2 saw V_3 the V_4 ball V_5

where;

$V_0 ... V_n$ are the vertices.

Insertion of Vertices in a Sentence.

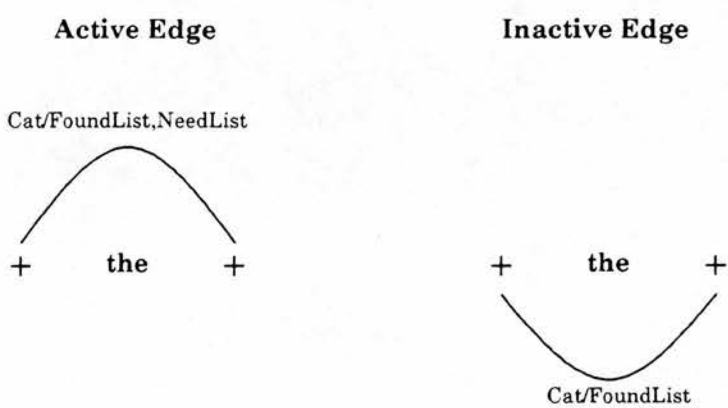
fig.5.1.

The chart records the elements of the parse found as *Active and Inactive Edges*. Inactive Edges correspond with parts of the parse which have found all of their constituent parts. Active Edges correspond with parts of the parse which require more constituents.

For each active and inactive edge a record is kept of:

- Its Category (what the edge will form when it is complete - for example, a sentence).
- Its Start Vertex (or "left hand end" of the edge).
- Its End Vertex (or "right hand end" of the edge).
- The Constituents found so far.
- The Constituents which are needed to complete the edge (none in the case of inactive edges).

The chart can be expressed in a diagrammatical form, where each edge in the chart is shown as an arc in the diagram. Active edges are shown as arcs "above" the words and Inactive edges are shown as arcs "below" the words. These are depicted in fig 5.2.



where:

Cat is the category of the edge (for example, "NP", or "Det").
FoundList is a list of the constituents of the edge that have been found so far.
NeedList is a list of the constituents of the edge that have yet to be found.

Diagramatical Form of Active and Inactive Edges.
fig.5.2.

5.1.1.1 The Fundamental Rule.

The *Fundamental Rule* determines how new edges should be added to the chart. When an active edge *meets* an inactive edge the fundamental rule adds an edge to the chart if the extension conditions are satisfied. These extension conditions are given in fig 5.3.

An active edge with End Vertex "V_{Ae}" meets an inactive edge with Start Vertex "V_{Is}" if:

- $V_{Ae} = V_{Is}$, *and*
- *the category of the inactive edge is the next constituent that the active edge needs.*
- *the edges have not met before.*

Extension Conditions for Active and Inactive Edges.

fig 5.3.

The form of the edge that is added to the chart is also determined by the fundamental rule, and is given by the addition conditions in fig 5.4.

When an active edge meets an inactive edge and the extension conditions are fullfilled, a new edge is added with the form:

- *Start Vertex = Start Vertex of the Active Edge.*
- *End Vertex = End Vertex of the Inactive Edge.*
- *Constituents found = Constituents found in the Active Edge + Category of the Inactive Edge.*
- *Constituents to find = Constituents to find in the Active Edge - Category of the Inactive Edge.*
- *If there are no more Constituents to find then the new edge is Inactive, otherwise the new edge is Active.*

Conditions for Determining the Form of an Edge to Add.

fig 5.4.

No edges in the chart are ever removed or altered. This is necessary so that *all* possible parses are found, not just the first parse.

5.1.1.2 The Parsing Policy.

The application of just the fundamental rule will not cause the parsing process to operate. In addition, empty active edges need to be added to the chart, and the way that this is achieved determines the parsing *policy*.

A "Bottom-up" parsing policy is achieved by adding empty active edges triggered by the completion of inactive edges. Informally, this results in behaviour described by; "I have found category 'Cat', now find all possible grammar rules which use category 'Cat' as their first constituent". That is, the parse is driven "bottom-up" from what has been observed. If:

- An inactive edge has just been created with start vertex "V",and
- There is a grammar rule that requires the category of the inactive edge as its first constituent, and
- There is no empty active edge corresponding to this grammar rule, at this vertex.

then an empty active edge is added to the chart at vertex "V" with;

- Start Vertex = "V".
- End Vertex = "V".
- FoundList is empty.
- NeedList is a list of the constituents that the new category needs.

One edge is added to the chart for each grammar rule which meets the above conditions.

A "Top-down" parsing is achieved by adding empty active edges triggered by the addition of active edges. This follows the behaviour "I am looking for category 'Cat', which can be found by finding all the constituents

of 'Cat'". That is, the parse is driven "top down" from what is required to obtain a complete parse (described in the grammar). If:

- An active edge has just been created with end vertex "V" requiring next constituent "Cat", and
- There is not an empty active edge at vertex "V" which corresponds to the grammar rule describing the constituents of "Cat".

then an empty active edge is added to the chart at vertex "V" with;

- Start Vertex = "V".
- End Vertex = "V".
- FoundList is empty.
- NeedList is a list of the constituents that the new category needs.

With both parsing policies the chart needs to be initialised by adding inactive edges which correspond to the dictionary entries for each word. In addition, the "top-down" policy needs an empty active edge to be added, which corresponds to the "top level" that needs to be found in the grammar (the Sentence in the grammar in fig 5.5).

The parsing process continues by applying the fundamental rule and the policy until no more edges can be created, then the parse halts.

5.1.2 An Example Parse.

Consider the grammar shown in fig 5.5 and the sentence "The man saw the ball".

S	::=	NP, VP
NP	::=	Det, N
VP	::=	V, NP
V	::=	saw
N	::=	man | ball
Det	::=	the

Where:

S	=	Sentence
NP	=	Noun Phrase
VP	=	Verb Phrase
V	=	Verb
N	=	Noun
Det	=	Determiner

A Simple Grammar.

fig.5.5.

The parse is performed according to the "bottom-up" policy and proceeds as follows:

Inactive edges are added for the dictionary entries of each word (fig 5.6).

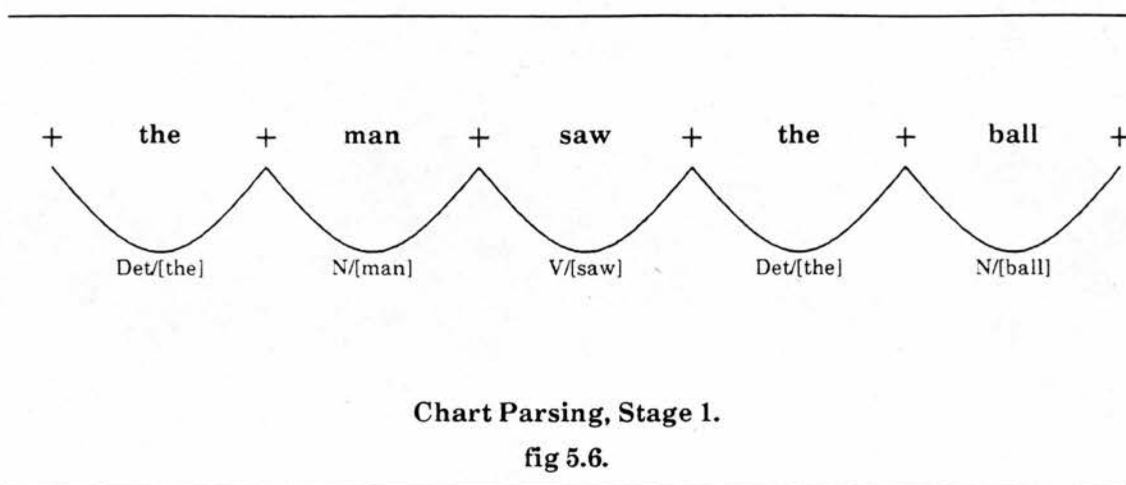


Chart Parsing, Stage 1.

fig 5.6.

The addition of these inactive edges causes the "bottom-up" policy to add empty active edges wherever possible, according to the grammar rules. For example, a determiner can be used to start a noun phrase, leading to empty active edges with the form NP/[],[Det,N] being added at the start vertex of any inactive edge spanning a determiner. This gives rise to the chart in fig 5.7.

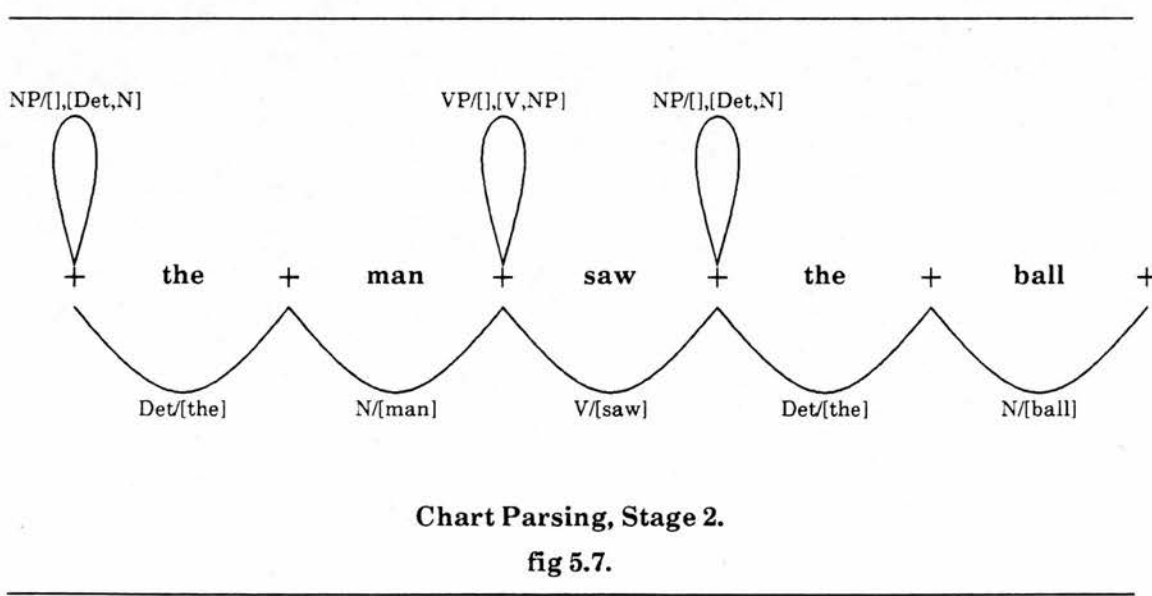


Chart Parsing, Stage 2.

fig 5.7.

Next, the fundamental rule is applied to each pair of active and inactive edges which meet (according to the rules given in fig 5.3), causing the addition of new edges. For example, the active edge "NP/[][Det,N]" will combine with an inactive edge with the category "Det". The resulting chart is shown in fig 5.8.

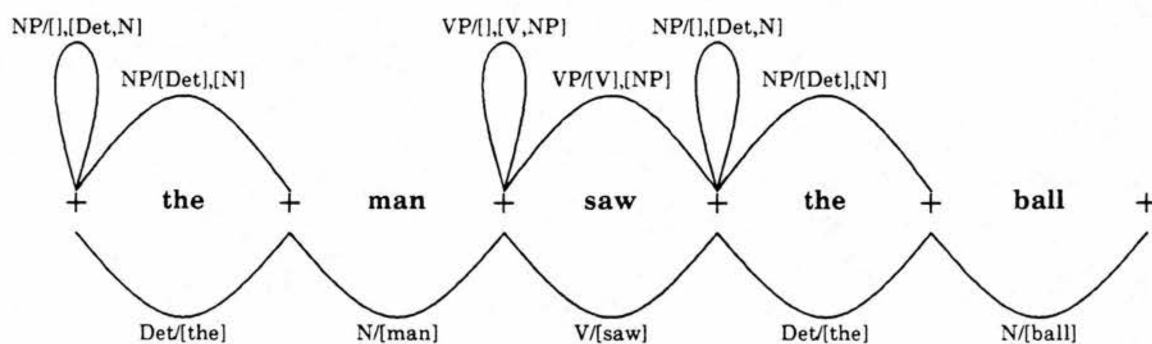


Chart Parsing, Stage 3.

fig 5.8.

The fundamental rule is applied again to any new active and inactive edges that meet, giving the chart in fig 5.9.

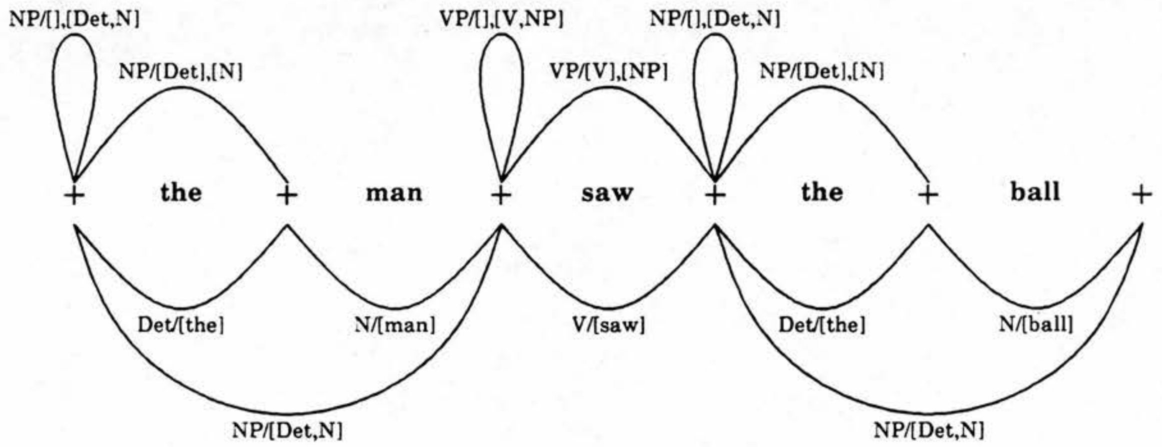


Chart Parsing, Stage 4.
fig 5.9.

The addition of new inactive edges causes the "bottom-up" policy to be applied to these edge. New empty active edges are added to the chart according to the grammar rules, giving the chart shown in fig 5.10.

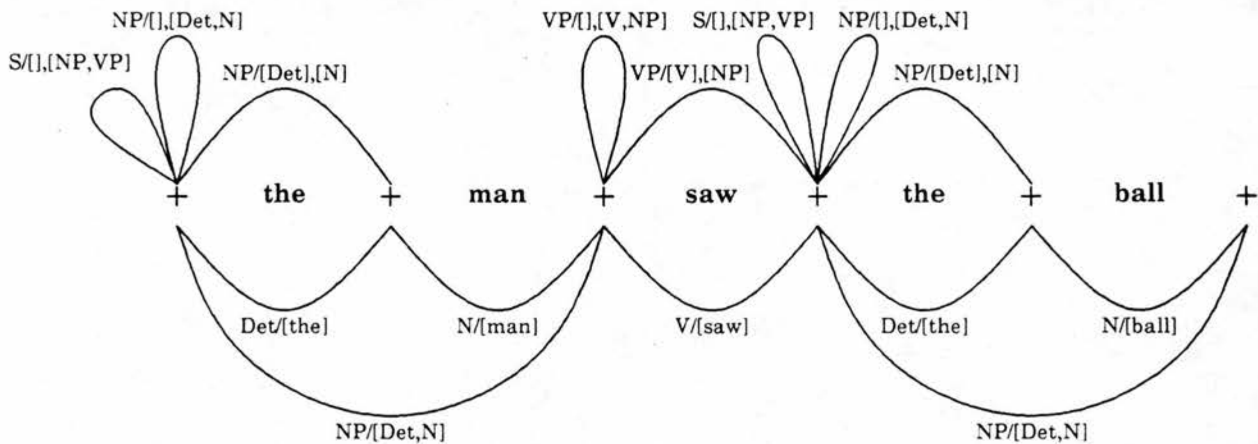


Chart Parsing, Stage 5.
fig 5.10.

The repeated application of the bottom-up policy and fundamental rule results in the last two remaining stages in the development of the chart; shown in fig 5.11 and fig 5.12.

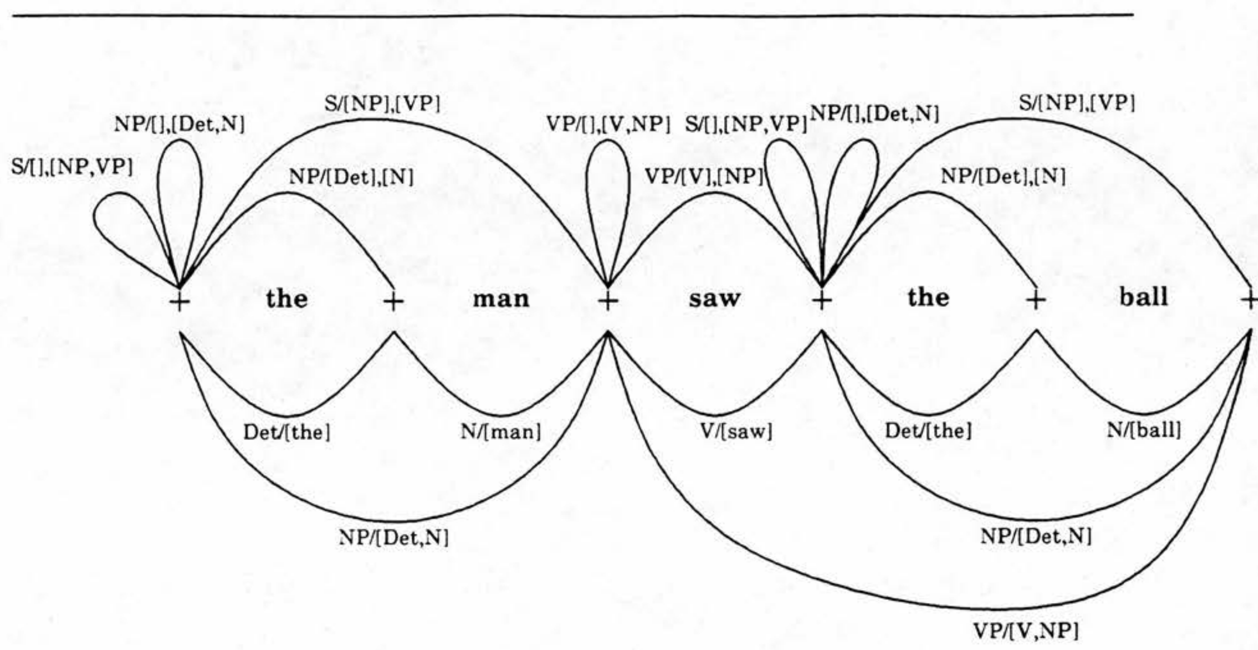


Chart Parsing, Stage 6.

fig 5.11.

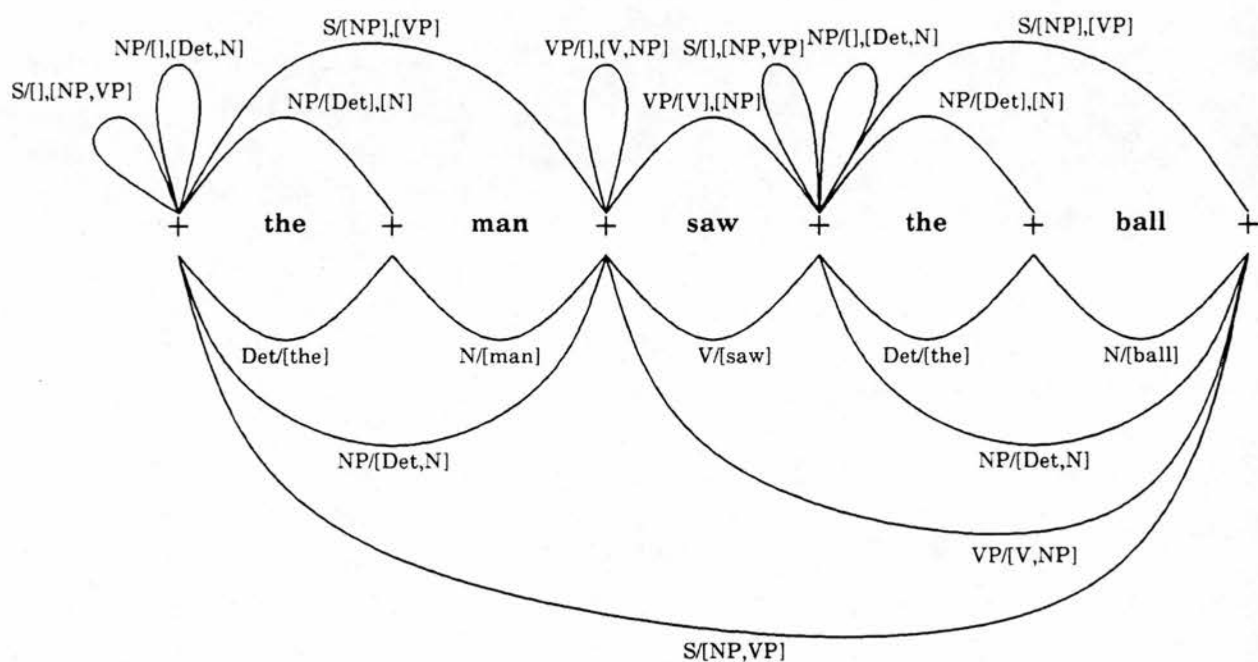


Chart Parsing, Stage 7.
fig 5.12.

Once the chart reaches the state shown in fig 5.12 the parsing process halts because no more new edges can be built either by applying the fundamental rule, or by applying the "bottom-up" policy.

The chart contains an explicit description of all complete and partial parses, shown in the diagram as the arcs (note the partial parse S/[NP],[VP] which is a partial possible explanation of a sentence, and contains "the ball").

5.1.3 Pausing the Parsing Process.

In the above description of how a parse progresses, the order in which edges are added to the chart has not been considered. It is possible to add edges in different orders, this being determined by the chart *strategy*. Two typical strategies are *depth-first* and *breadth-first*. In normal usage of the chart parser the strategy is unimportant, since we need the parse to run to completion. It would be possible to stop or pause the parse before all possible parses have been found, by modifying the conditions for terminating the parse (this terminating condition is explicit in the implementation of the chart parser used, and is at present simply *"stop when there are no further edges to add to the chart"*).

It may be desirable to pause the parsing process, according to some heuristics, to reduce the size of the chart and time taken to parse. An alternative reason for pausing is to enable the user to type more commands before a complete analysis of his previous commands has been made. Determining when to pause the parsing process is difficult since this affects the parse time and edges contained in the chart. Crude methods of performing this task could be used (for example, using elapsed time, or cpu time), but information will inevitably be absent from the chart at the time that it is paused, and this may affect the possible plans that can be recognised. This is a particular problem if edges are removed from the chart whilest the parser is paused (for example, to prune the chart for efficiency), since these edges might have combined in more ways and resulted in the recognition of interesting edges. The chart parser gives an efficient method of analysing data, and an analysis of the consequences of pausing the parsing process is likely to be at least as computationally expensive as the parsing process itself. The alternative is to make a crude assessment based upon simple criteria (for example, pausing when a complete parse has been found), and resume parsing later if necessary. However, if such criteria were used, this would prevent the recognition of competing explanations which are based upon different assumptions. The technique of developing multiple explanations in the chart will be used as a basis for advice generation in

chapter 6. At present, the parsing process is run to completion so that all parses are found. The modification of the parsing process to encompass alternative termination criteria will be left for present, although the chart parser has an explicit method for detecting the termination of the parsing process which would make the modification of the terminating condition easy to achieve in practice.

5.2 Chart Parsing and Plan Recognition.

In Plan Recognition structure needs to be inferred from sequences of actions (ie. the *Plans),* and the aims of the actions need to be determined (ie the *Goals).* This Plan Recognition task can be considered to be a parsing problem, where we are attempting to fit actions into a grammar which describes a goal / sub-goal hierarchy. Grammars have been used to describe the interaction of users with computers (Fountain and Norman, 1985). The ambiguity that occurs in Natural Language also occurs in Plan Recognition, where an action sequence does not necessarily uniquely define the plan being followed. Plan Recognition is typically required to hypothesise about the plans being followed when actions do not fit in to a pre-defined grammar or plan library. Thus the information explicit in the chart as partial parses, is useful in determining the intentions under these circumstances (Ross and Lewis, 1988). In Plan Recognition the goals may not be known, and hypotheses need to be generated which are based upon the actions found. This corresponds to a "bottom-up" parse with the Chart Parser. Another requirement of Plan Recognition is to incorporate actions as they are observed. Such *Incremental* parsing can easily be implemented using a chart parser without involving the re-computation of parses built up so far (by simply storing the chart, and initialising the next parse with that chart rather than an empty chart).

5.2.1 Incremental Parsing.

For example, consider the grammar shown in fig 5.13:

```
plan ::= plan1, plan2.
plan1 ::= a, b.
plan2 ::= c,d.
plan3 ::= b,f.
```

A Simple Plan Grammar.

fig.5.13.

In the grammar, the elements "a", "b", etc are place holders and need not be atoms in the general case. If the action "a" is observed and the parse run to completion using the grammar in fig 5.13, then the chart shown in fig 5.14 will result. At this stage, the best interpretation of the action is made by the partial plan "plan1".

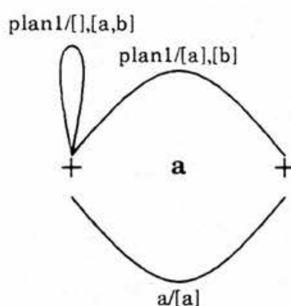


Chart after Observing Action "a".
fig.5.14.

If action "b" is then observed, it can be accounted for by incorporating it into the chart to give the chart shown in fig 5.15. This chart has been formed by using the chart shown in fig 5.14 as the initial starting point for developing the new chart, and adding the new inactive edge "b" to the chart. The parsing process then proceeds incorporating this new edge into the chart as though the parsing process had been performed in one application. By using this facility the actions can be added to the chart as they are observed, and the work involved in generating the previous charts does not need to be duplicated by the following applications of the parse. This shows that "b" could form part of the existing plan "plan1" to form the complete plan "plan1/[a,b]", or action "b" could start "plan3" giving the partial plan "plan3/[b],[f]. Alternatively action "b" could form parts of both plans if the plans are independent.

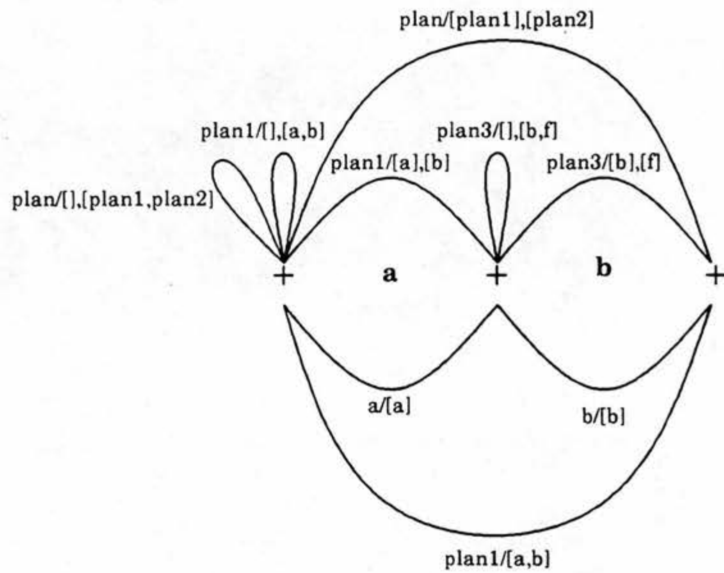


Chart after Observing Action "b".

fig.5.15.

If action "c" is then observed, it can be recognised as starting a new plan "plan2", the chart for this situation is given in fig 5.16.

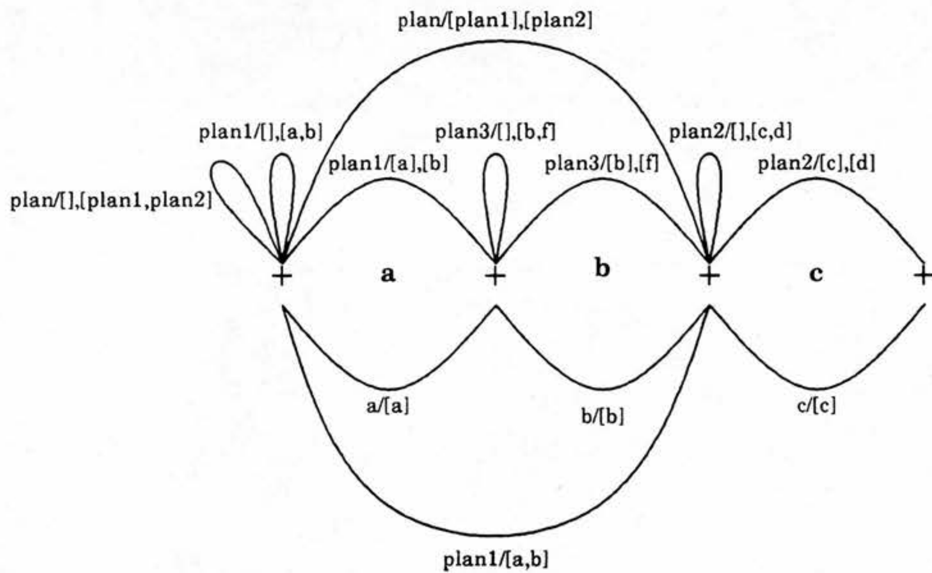


Chart after Observing Action "c".

fig.5.16.

Once action "d" is observed, this completes the recognition of plan "plan" and gives the resulting chart given in fig 5.17. This resultant chart contains all possible partial and complete plans according to the grammar (assuming no errors were made in entering the actions), and was built incrementally so that previous work did not need to be repeated.

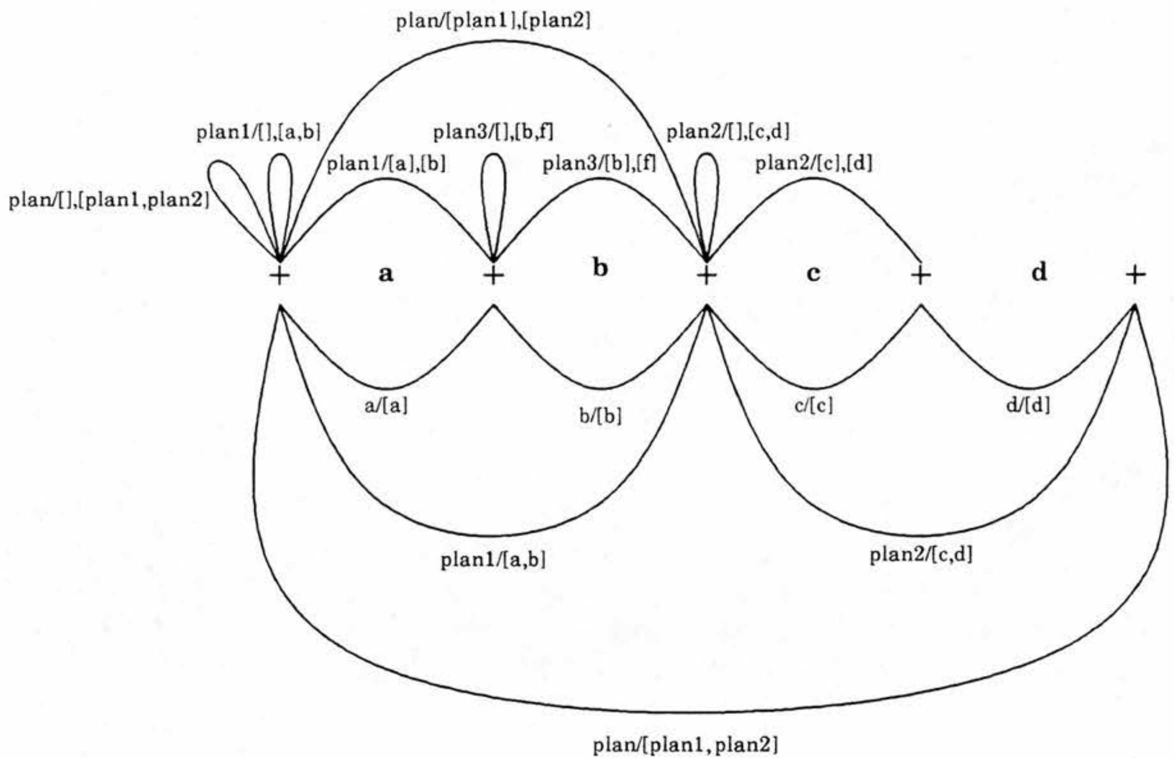


Chart after Observing Action "d".
fig.5.17.

Therefore, given a context-free plan grammar, we are able to detect all possible partial and complete plans that the action sequence might be part of. However, in general it is not possible to write context-free grammars for plan recognition. The actions might not occur in succession and it is not

possible to write a grammar which describes all possible plans that a user might be following. Therefore, account needs to be taken of:

- Context.
- Fragmented Plans
- Multiple, Interleaved Plans.
- Incomplete Grammars.

these will now be discussed, with particular reference to the UNIX domain.

5.2.2. Context.

A context-free grammar is not capable of describing most plans in the UNIX domain. For example, consider the context-free grammar rule shown in fig 5.18 in which A and B can match any argument.

```
group-copy ::= copy, copy.
group-copy ::= copy, group-copy.
copy ::= cp A B.
```

A Simple Context-free Grammar for UNIX.

fig.5.18.

This grammar describes a "group copy" as being a "copy" command followed by another "copy" command. The general case is given by the recursive rule; that a "group-copy" consists of a "copy" followed by a "group-copy". Such a rule might be used to identify that the user is attempting to copy a group of files from one location to a different location. This information could then be used to suggest a better way of achieving the goal, or detect typing slips, misconceptions, etc. For example, observing the command sequence given in

fig 5.19 might well suggest that the user has made a typing slip on line 3 ("dr" for "dir").

1	cp a dir
2	cp b dir
3	cp c dr

An example UNIX Command sequence.

fig.5.19.

Such a "group-copy" rule is reasonable for an action (command) sequence where the destination is constant, but not when the destination address varies (for example, in the command sequence given in fig 5.20).

1	cp file1 directory1
2	cp file2 directory2

An example UNIX Command sequence.

fig.5.20.

The directory arguments should be the same for this rule to apply. The problem is that the "group copy" goal is not independent of the arguments of the constituent commands, so these need to be encoded into the grammar. The context-free grammar given in fig 5.18 could be rewritten as the context-sensitive grammar shown in fig 5.21.

```
group-copy([F1,F2], Dir) ::= copy(F1,Dir), copy(F2,Dir).
group-copy([F1|FL],Dir) ::= copy(F1,Dir), group-copy(FL,Dir).
copy(F,Dir) ::= cp F Dir
```

where; F, F1 and F2 have type "file", and "Dir" has type "directory".
FL is a list of files.

A simple context-sensitive UNIX Grammar.

fig.5.21.

In this grammar the uppercase values are treated as PROLOG variables which can be instantiated to values. Using the context-sensitive grammar ensures that the destination directories are the same for both constituent commands. However, there is still a problem with dependency between actions. For example, consider the action sequence given in fig 5.22.

```
1    cp file1 directory
2    cp file1 directory
```

A UNIX Command Sequence with Dependent Actions.

fig.5.22.

This command sequence causes an overwrite of the new file "file1" located at directory "directory", not the creation of two files (which is suggested by the grammar). Conditions need to be applied to the grammar rules to reject this false plan. The new grammar is shown in fig 5.23., where the condition {F1 <> F2} prevents the incorrect recognition of this plan. An additional condition is required for the general recursive form of the grammar rule.

```
group-copy([F1,F2],Dir) ::= copy(F1,Dir), copy(F2,Dir)
                                          {F1<> F2}
group-copy([F1|FL],Dir) ::= copy(F1,Dir), group-copy(FL,Dir).
                                          {not( member( F1, FL))}
copy(F,D) ::= cp F D
```

where:

{F1 <> F2} is the condition that F1 must not be the same file as F2.

{not(member(F1, FL))} is the condition that F1 must not be the same file as any file in the list FL.

A Grammar for UNIX Incorporating Dependency Conditions.

fig.5.23.

The condition (F1 <> F2) needs to be verified before the new inactive edge "group-copy(Files,Directory)/[copy(F1,Dir), copy(F2,Dir)]" can be added to the chart.

5.2.3. Fragmented Plans.

There is no guarantee that the constituents of the plan occur in an unbroken sequence. A plan might need to extend over actions which do not form part of that plan. For example, consider the grammar given in fig 5.24.

```
plan1 ::= a,b
```

A Simple Plan Grammar.

fig.5.24.

If the action sequence "a,x,b" is observed, then this should be partially explained by plan1 (if the actions are all independent; that is the execution of any action has no effect on the validity or execution of any other action).

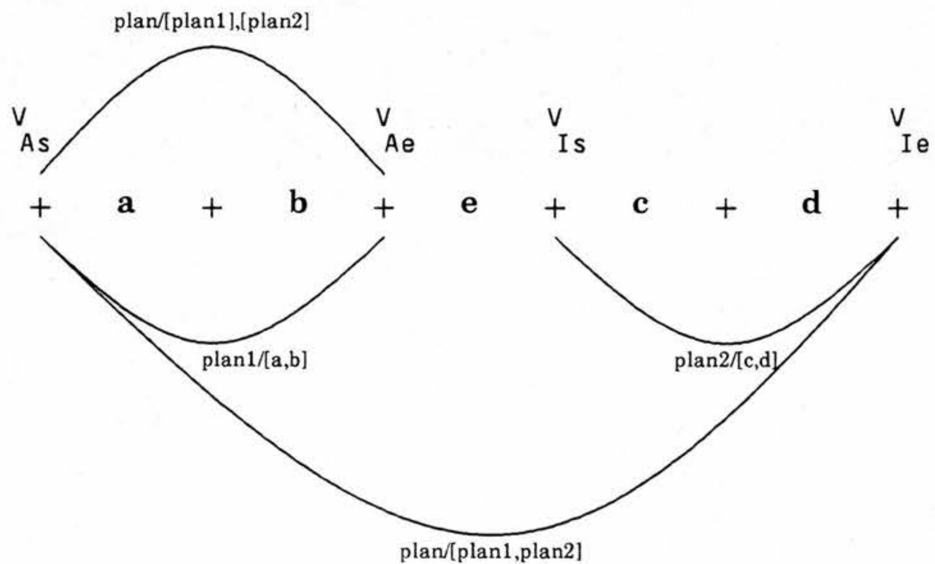
Action "x" might form part of another plan, or no plan at all. The detection of this fragmented plan is achieved by altering the Fundamental Rule to the rule shown in fig 5.25.

An active edge with End Vertex "V_{Ae}", meets an inactive edge with start vertex "V_{Is}" if;

- $V_{Ae} =< V_{Is}$, *and*
- *the category of the inactive edge is the next constituent that the active edge needs. The formation of the contents of the new edge remains the same as before.*

Fundamental Rule for Fragmented Plans.
fig.5.25

These conditions are depicted in fig.5.26.



Extension Conditions for fragmented Edges.
fig.5.26.

If the actions are not independent, then the validity of recognised plans is uncertain. For example, consider the grammar shown in fig 5.27.

---

move(A,B) ::= copy(A,B), remove(A).
copy(A,B) ::= cp A B.

An Example UNIX Grammar.

fig.5.27.

If the action sequence given in fig 5.28 is then given, the parser would incorrectly recognise the plan "move(a,b)" after command 3 (at no point in the command sequence is there a point which would have corresponded to the action "move(A,B)", therefore such an action should not be recognised).

1	copy(A,B)
2	remove(B)
3	remove(A)

An Example Command Sequence.

fig.5.28.

This problem did not arise before, because the post-conditions of the actions were implicit in the ordering of the actions. Now that additional actions can be inserted between constituents of a plan which do not form part of the plan, yet have side-effects which invalidate the plan, the ordering within the grammar is insufficient to ensure the validity of plans. This problem can be overcome by adding explicit post-conditions into the grammar; transforming the grammar to that shown in fig 5.29.

```
move(A,B) ::= copy(A,B), remove(A)
                    {A does not exist,
                     B exists}
copy(A,B) ::= cp A B
```

UNIX Grammar with Post-conditions.

fig.5.29.

These post-conditions are applied before the inactive edge "move(A,B)/[copy(A,B), remove(A)]" is added to the chart. This implies that the current context should be maintained for each of the vertices in the action sequence (since the context needs to be checked before any new edges can be added to the chart). However, because the plan recognition task is achieved incrementally only the current context is needed. This is achieved because the application of the post-conditions are made just before an inactive edge is added to the chart, and any inactive edge being added must end at the present maximum vertex (the right hand end of the sequence so far). This makes the maintenance of all other contexts redundant provided that the parsing is performed incrementally. In the UNIX filestore domain, a model of the current state of the filestore needs to be maintained as the local context.

These post-conditions are not a sufficiently rigorous method of validating the plan, because they only test the *name* of a file and not the *contents* of the file. Tests are needed to ensure that no intervening commands in the command sequence have altered the existence and contents of the file. For example, the command sequence:

1% **cp A B**
2% **rm B**
3% **top B**
4% **rm A**

could be considered to be equivalent to "mv A B" under the present scheme using post-conditions. Whereas, editing the file B ("top B") has created a new file with the same name, but the contents of this new file may be unrelated to the original. However, for simplicity post-conditions which check the existence of files will be used throughout this thesis. The post-condition mechanism can provide more rigorous checks on the intervening commands (for example, by ensuring that the file has not been deleted in intervening commands), but this will not be developed further in this thesis.

5.2.4. Interleaved and Enclosed Plans.

Fragmented plans are a sub-set of the more complex problems of Multiple, Interleaved plans. Consider the grammar in fig 5.30.

```
plan ::= plan1, plan2.
plan1 ::= a,b.
plan2 ::= c,d.
```

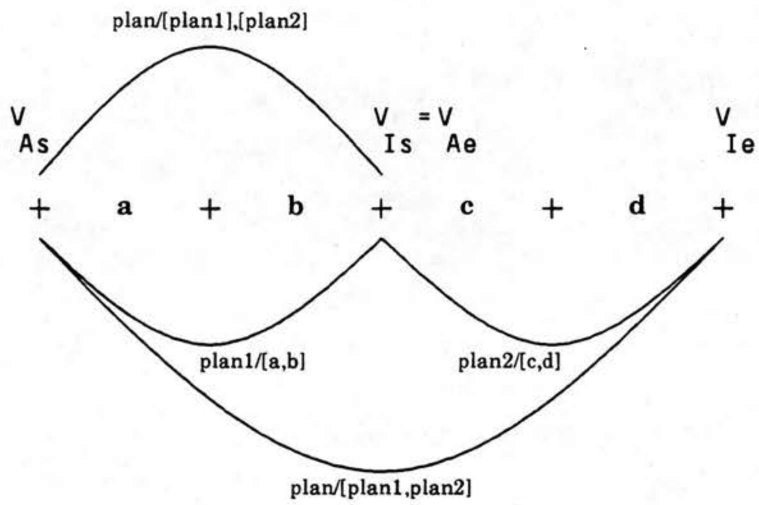
UNIX Grammar.

fig.5.30.

If three different action sequences are observed;

1. a,b,c,d
2. a,c,b,d
3. a,c,d,b

Assuming that the actions are independent, then each of these three action sequences should be recognised as the plan "plan". By using the fundamental rule for Fragmented Plans (fig 5.25) the chart parser can recognise the plans "plan1" and "plan2" in each sequence. However, it can only recognise the complete plan in fig.5.31. where the active and inactive edges are touching ($V_{Is} = V_{Ae}$), allowing the active edge "plan/[plan1],[plan2]" to be extended. In the remaining cases (fig.5.32 and 5.33) the active and inactive edges necessary for producing the edge spanning "plan1" and "plan2" never satisfy the meeting conditions. The conditions which determine when edges meet are $V_{Is} > V_{As}$; that is, the start vertex of the active edge, must come before the start vertex of the inactive edge. In addition to the fragmented case (fig.5.26), the possible extension criteria for an active edge are given in fig.5.31-5.33. For clarity, relevant edges only are shown in figs.5.31-5.33.

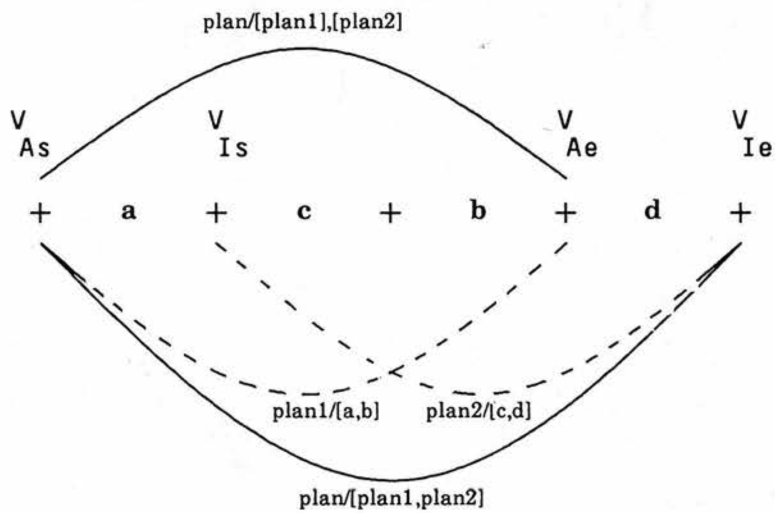


Extension Conditions for Touching Edges.
($V_{Is} = V_{Ae}$)

fig.5.31

———— Represents an Edge containing all elements.

. . . . Represents an Edge containing some elements.

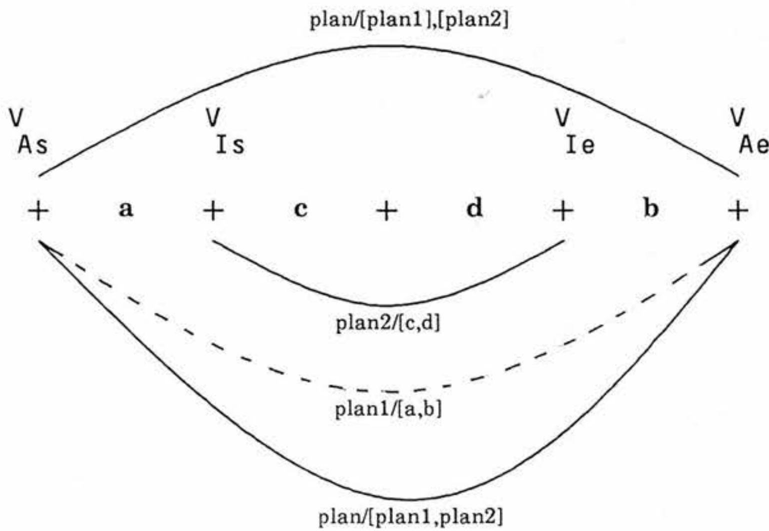


Extension Conditions for Interleaved Edges.
($V_{Is} >= V_{As}$, $V_{Ie} >= V_{Ae}$ and $V_{Ae} >= V_{Is}$)

fig.5.32

———— Represents an Edge containing all elements.

. . . . Represents an Edge containing some elements.



Extension Conditions for Enclosed Edges.

(V_{Is} >= V_{As} and V_{Ie} =< V_{Ae})

fig.5.33

By combining the necessary extension conditions for each case (Touching, Fragmented, Interleaved and Enclosed), the condition for testing pairs of active and inactive edges for possible extension has been reduced to V_{Is} >= V_{As}. That is; the start vertex of the Inactive Edge must occur after the start vertex of the Active Edge.

Consider the grammar given in fig 5.34.

```
groupmove([A,C],B)  ::=  move(A,B), move(C,B).
move(A,B)           ::=  mv A B.
move(A,B)           ::=  copy(A,B), remove(A).
copy(A,B)           ::=  cp A B.
remove(A)           ::=  rm A.
```

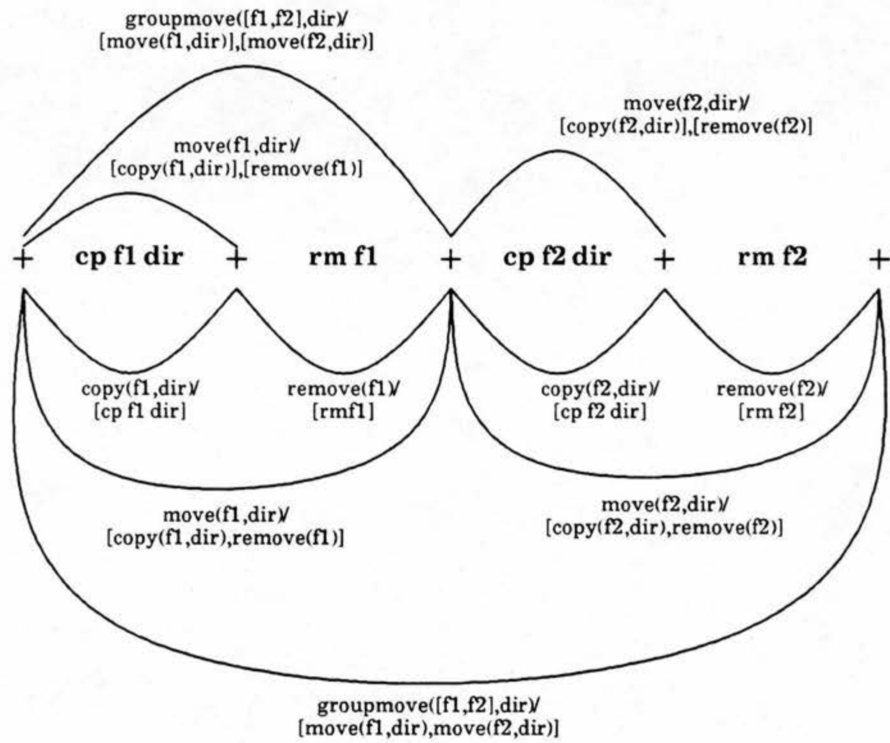
An Example UNIX Grammar.

fig.5.34

The action sequence:

cp f1 dir
rm f1
cp f2 dir
rm f2

can be recognised as "groupmove([f1,f2],dir)" (fig.5.35) by using the "Touching" extension condition (fig.5.31).

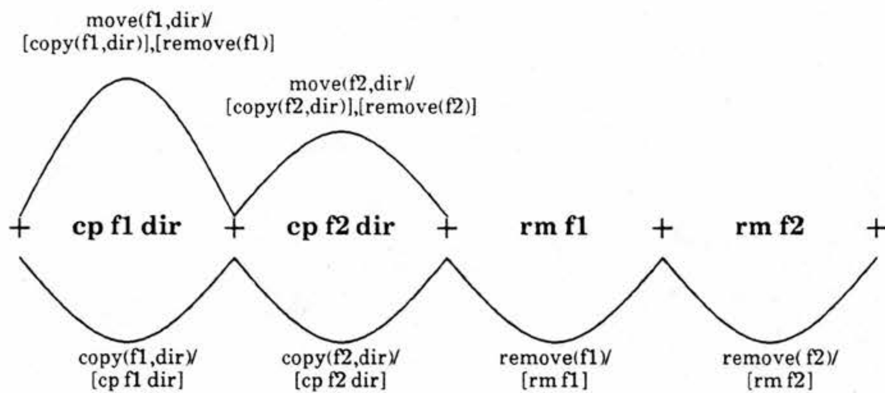


Recognising a Group Move with "touching" extension Conditions.

fig.5.35

However the action sequence;

cp f1 dir
cp f2 dir
rm f1
rm f2



Group Move which Cannot be Recognised using "Touching" Extension Conditions.
fig.5.36

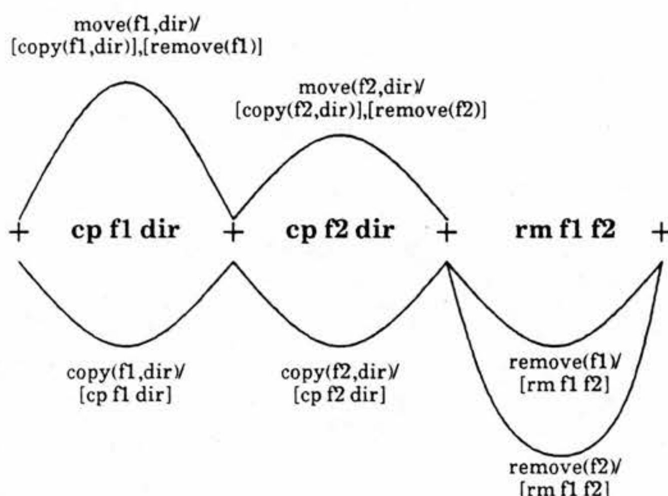
cannot be recognised as "groupmove([f1,f2],dir)" solely by using the "Touching" extension condition (fig.5.36), but the "Interleaved" condition must be used, or new grammar rules written for the different possible valid orderings of the commands.

If the command sequence;

cp f1 dir
cp f2 dir

rm f1 f2

is seen, the third command "rm f1 f2" (equivalent to rmf1; rm f2) forms the part of two partial plans "move(f1 dir)" and "move(f2,dir)", which must combine to enable the recognition of the "groupmove" sequence. Hence the extension requirements must allow an action to be part of two plans. This situation is shown in fig.5.37.



Group Move using a Combined "Remove" Command.
fig 5.37

The current implementation of the chart parser was altered to encompass the recognition of touching, fragmented, enclosed and interleaved edges. However, these alterations were made to determine the feasibility of recognising such edges. The alterations were not used in the resultant UNIX ADVISOR because of the increase in complexity involved in recognising plans, and the increase in computational overheads. It is likely that it would be impractical to extend all edges using the fundamental rule encompassing touching, fragmented, enclosed and interleaved edges because

of the computational overhead. Instead, the fundamental rule could be altered (as described above) as the plan recognition process progresses. For example, the fundamental rule could combine touching and fragmented plans. If the parsing process cannot account for some actions, then the fundamental rule could be altered to look for interleaved and enclosed edges.

5.3. A UNIX Grammar.

A UNIX grammar is a description of how typical goals that users wish to achieve are decomposed into sub-goals. Such a grammar is developed that specifies different possible behaviours that users exhibit while using the UNIX filestore commands. This grammar is based upon the analysis performed in Chapter 3 and it is used to test the chart parsing approach to plan recognition in a real domain. The grammar is not intended to be complete, since the plan recognition technique enables the grammar to be expanded through the recognition of plan cliches. However, it is intended that the grammar is representative of planned behaviour that users exhibit in the UNIX domain. The decomposition results in a goal / sub-goal hierarchy that describes how the UNIX system is used, and contains only correct plans and goals. The grammar does not explicitly mention misconceptions that the user has or errors that the user makes. It is assumed for the purposes of generating advice, that all errors and misconceptions can be modelled as incorrect knowledge of the preconditions and effects of commands. It is also assumed that users combine commands to form correct plans which would achieve their goals, if the command details were correct. For example, the user knows to use the command "rm" to delete a file tree, but uses the wrong form of the command. This grammar must either be determined through the analysis of users' tasks, or by the designers of the system specifying the tasks that they expect users of the system to perform. The task of generating the grammar would require a large effort to decide which goals users wish to achieve and the possible methods for achieving these. For example, the analysis of the experiments performed in Chapter 3 indicated different methods that users' use to solve the "move tree" goal (fig. 5.38). Plans to achieve this goal were chosen because the plans were particularly rich in problems for users. These problems arose due to misconceptions, lack-of-knowledge and errors. The decomposition shown in fig. 5.38 includes "buggy" plans which would not be encoded in the corresponding grammar, nor would parts of the hierarchy which were only used by few individuals (for example the use of "cat" to copy files). The "buggy" parts of the goal hierarchy are shown by dashed lines in fig.5.38. It

so happens that in this goal hierarchy all the "buggy" parts are terminal branches of the tree, although this would not generally be the case. Such plans are not included because, in general, it would be impossible to encode all such plans.

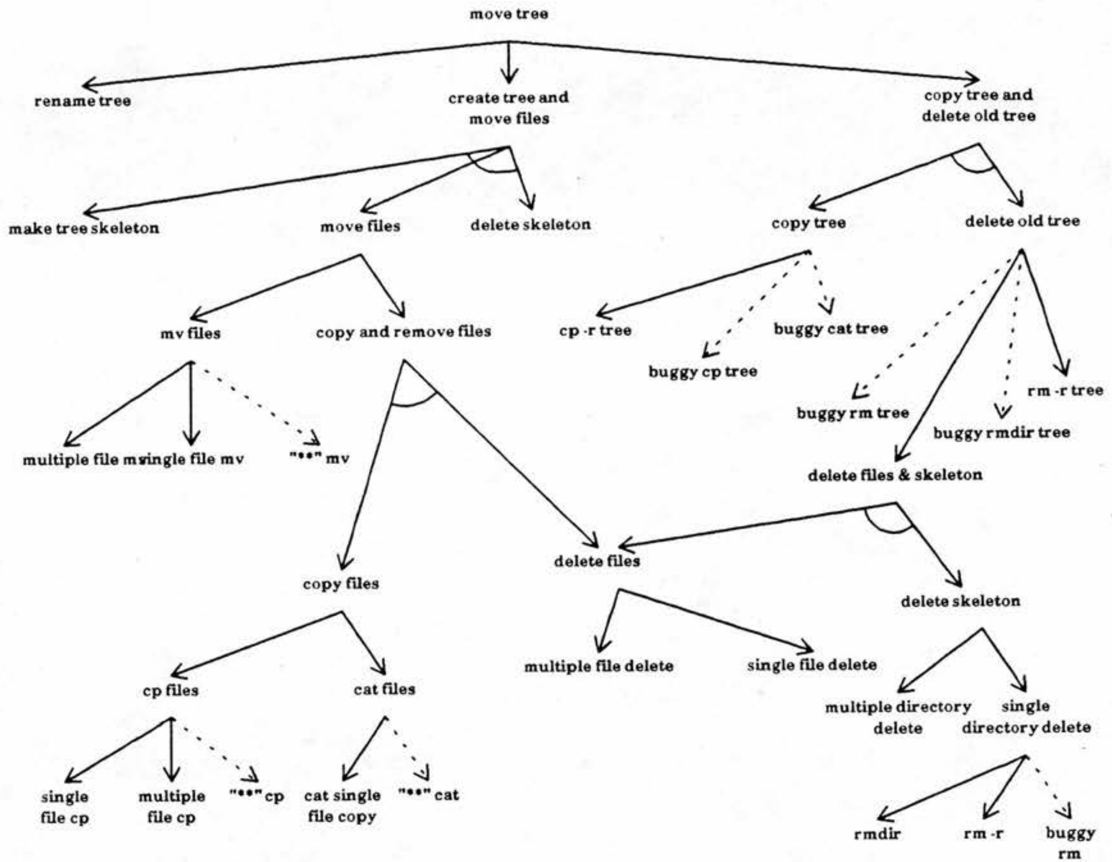


fig.5.38.

A Decomposition of the "move tree" Goal (same as fig.3.6).

The UNIX Advisor does not claim to use a representative grammar for UNIX users, but uses fragments of a possible grammar to show the principles involved in its use.

5.3.1. The Form of the Grammar.

The grammar consists of rules which describe:

i. How goals are decomposed (rules).

ii. Allowable commands (lexical rules).

Rules have the form:

rule(Tag, RuleNumber, Status, Goal, SubGoals, Conditions).

Lexical rules have the form:

lexical(Tag, LexicalNumber, Status, Command, LexicalInterpretation).

where:

- The "Tag" is a name associated with the grammar, enabling multiple grammars to be used on the chart. This facility was not used in this work.
- The "RuleNumber" or "LexicalNumber" is a tag associated with a particular rule. This enables a trace of the rules used in a plan to be easily maintained (this information being of use to determine whether to give advice to the user).
- The "Status" is a simple method of marking whether the user "knows" the rule. The "Status" takes the values "in" (the user knows about the rule), or "out" (the user does not "know" about the rule).
- The "Goal" is a description of a goal state.
- The "SubGoals" consist of a list of goals that must be satisfied to achieve the "Goal".
- "Conditions" are a list of conditions that must be true for the "Goal" to be satisfied.
- The "Command" is the representation of the command typed by the user. This representation uses STRIPS-like "add" and "delete" lists.
- "LexicalInterpretation" is the representation of the command that is used in the chart.

An example grammar is given in fig.5.39. which uses a simplified form of a UNIX grammar for clarity.

```
%	A goal is composed of an action sequence "actseq".
%	An actseq is composed of copying a file to a new directory, changing "working
%	directory" to that directory and editing the copied file.
%	File paths are given from "root".

rule(trial, rule:1, out, goal, [actseq], []).
rule(trial, rule:2, out, actseq, [cp(A/B,C/D), cd(A,C), top(C/D)], []).

%	The lexical entries describe the allowable commands and how these are
%	represented in the chart.

lexical(trial, lexical:1, out, [cp(X,Y),[add(Y)]], [cp(X,Y)]).
lexical(trial, lexical:2, out, [cd(X,Y),[delete(cwd(X)), add(cwd(Y))]], [cd(X,Y)]).
lexical(trial, lexical:3, out, [top(X),[]], [top(X)]).
```

fig.5.39
A Sample UNIX Plan Grammar.

The plan grammar used in the UNIX Advisor is given in Appendix VIII. This grammar has the same form as the grammar in fig.5.39, except that it uses infix notation to make the grammar clearer, and each rule passes information about whether a command has failed. This information is needed by a "redo" rule which is used to represent the user replanning a task that he attempted but which failed. The UNIX grammar is based upon the goal decomposition for moving and copying files shown in fig.5.38. The "buggy" parts of the hierarchy are not built in to the grammar since this would restrict the number of problems that could be recognised. Instead, these "buggy" parts of the decomposition occur through the generation of command misconceptions using the UNIX user command model.

5.3.2. Incomplete Grammars.

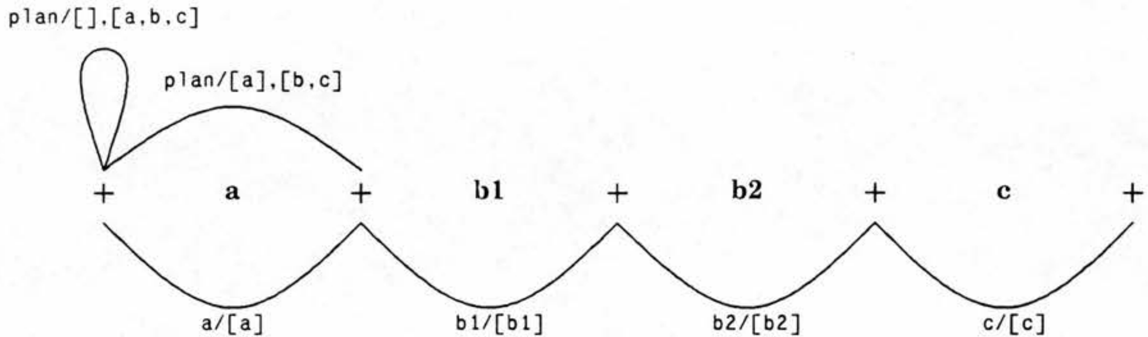
For any non-trivial domain, it is not possible to enumerate all possible goals that users wish to achieve, and their ways for achieving these goals. Therefore, the UNIX Advisor does not assume that the grammar is complete. Instead, the grammar describes the goal / sub-goal hierarchy that the designers intend the users to use, or through evaluation of the system the main goals that users perform and their plans for achieving these. The grammar will then be dynamically extended as a session progresses, to cover any idiosyncrasies that the user might exhibit in a manner explained below. For example, consider the grammar:

plan ::= a, b, c.

If the user types the action sequence;

a, b1, b2, c

the plan cannot be detected because the constituent "b" is missing from the action sequence, and "b1, b2" appears in its place. The resultant chart is shown in fig.5.40:



Example Plan Suitable for Cliche Recognition.
fig.5.40

The plan cannot extend to account for the entire action sequence because the actions "b1, b2" occur instead of the action "b". However, if these two action sequences are *equivalent* (they result in the same net effects), then the command sequence is legitimate. These types of command *cliche* are often exhibited by users of UNIX. For example, a user achieves a "mv" command by using "cp" and "rm"; or "cat file1 > file2" for "cp file1 file2"; or "cd; cd directory" for "cd ..". We define a "cliche" to be a sequence of actions which has the same net effects as some other, different sequence of actions. This definition assumes that there are no other reasons why the user performed the task in the given way, that would affect the plan being recognised.

5.3.2.1. Recognising Cliches.

The recognition of plan cliches depends upon the grammar having a representation of the effects of the action sequence. The cliche detection proceeds by analysing the chart once the parsing process is complete. This analysis consists of five steps:

1. Look for active edges of the form Category / Found, Need.
   where:
   Category is the category of the edge
   Found is a non-empty list of constituents that have been found
   (this must be non-empty since an empty list means that the first element has not been found, and therefore the active edge cannot be generated by the bottom-up addition of edges).
   Need is a list of Constituents, with the form [N1, N2 | Needs], where;
   "N1" has not been recognised,
   "N2" is a category that has been recognised (this restricts the generation of the replanned category to the limits of the size of the hole), and
   "Needs" is the remainder of the constituents needed to complete the edge, which may be an empty list.

   In the chart shown in fig.5.40, edge "plan/[a],[b,c]" satisfies these requirements with;

Category	matching	"plan"
Found	matching	"[a]"
N1	matching	"b"
N2	matching	"c", and
Needs	matching	"[]"

2. Locate the effects of N1. This is achieved by hypothesising N1 as a goal, and then using this to plan the actions necessary to achieve this goal. The chart parser is used to plan by running the parser "top-down" from the hypothesised goal, with uninstantiated actions between the start and end vertex of N1. These actions become instantiated to the possible actions that could have formed the plan. Only valid plans and commands can be used to generate the plan and list of actions to achieve that plan.

3. The net effects of the observed actions are compared with the net effects of the hypothesised actions (we employ a STRIPS-like representation of the effects of actions, see chapter4). If these effects are equivalent (in the above example, if the effects of "b" are the same as "b1, b2") then the relevant edge (in the above case plan/[a,b1,b2],[c]) is added to the chart, and the cliche rule is noted (b ::= b1, b2). For UNIX this means executing the plans in a simulated filestore to ensure that the effects are equivalent. In general, determining the equivalence of commands in this way is not sufficient. However, if the command sequence is not fragmented or interleaved, then this technique is equivalent to reasoning about post-conditions. A more general example of the simulated filestore incorrectly recognising equivalent commands is illustrated by the following fragmented command sequence:

   ```
   1%    cp a b
   2%    rm b
   3%    cp c b
   4%    rm a
   ```

   The equivalence mechanism would recognise commands 1 and 4 being equivalent to a "mv a b" - whereas this is not the case since the contents of file "a" at command 4 arise from file "c" not file "a".
   However, the problem of reasoning about the equivalence of commands is, in general, very complicated and outside of the scope of this thesis.

4. Continue the parsing process as normal until no further edges can be added (these resulting from the addition of the new edge).

5. If the complete inactive edge (plan/[a,b1,b2,c]) is found, then all of the cliches which were used to identify this plan, are added to the grammar as Cliche Rules (causing, where appropriate, the generalisation of other cliches in the grammar). This mechanism will be described in more detail in section 5.3.2.2.

Three important aspects of this cliche recognition process are the generation of plans from a goal, the formation of a specific cliche rule which accounts for the observed behaviour, and the ability to generalise the cliche to account for other similar instances of behaviour. These are discussed in the next section.

5.3.2.2. Generating Plans.

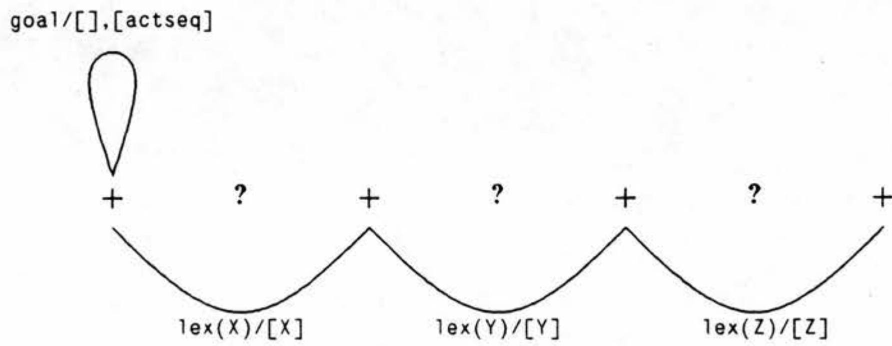
It is necessary to generate plans in order that different possible sequences of actions can be identified which would achieve the expected goal. These plans can then be compared to the actual command sequence to determine whether the user has used a cliche. One method of generating possible plans is to use the chart parser to generate these plans from the expected goal. This is achieved by adding an empty active edge corresponding to the goal that is to be satisfied. Then, uninstantiated inactive edges corresponding to the actions that are needed to satisfy the goal, are added to the chart. The grammar in fig.5.39 must be altered slightly to differentiate edges which are the uninstantiated lexical items, from other edges in the chart (otherwise the goal could be satisfied by simply instantiating the edges to whatever was necessary). Thus, lexical categories in the grammar are prefixed "lex", giving the grammar in fig.5.41. Each unrecognised action in the command sequence typed by the user is added to the chart as a partially instantiated edge (For example, lex(X)/[X]).

```
%       A goal is composed of an action sequence "actseq".
%       An actseq is composed of copying a file to a new directory, changing "working
%       directory" to that directory and editing the copied file.

rule(trial, rule:1, out, goal, [actseq], []).
rule(trial, rule:2, out, actseq, [lex(cp(A/B,C/D)), lex(cd(A,C)), lex(top(C/D))], []).

%       The lexical entries describe the allowable commands and how these are
%       represented in the chart.

lexical(trial, lexical:1, out, [cp(X,Y),[add(X)]], [lex(cp(X,Y))]).
lexical(trial, lexical:2, out, [cd(X,Y),[delete(cwd(X)), add(cwd(Y))]], [lex(cd(X,Y))]).
lexical(trial, lexical:3, out, [top(X),[]], [lex(top(X))]).
```

fig.5.41

A UNIX Plan Grammar.

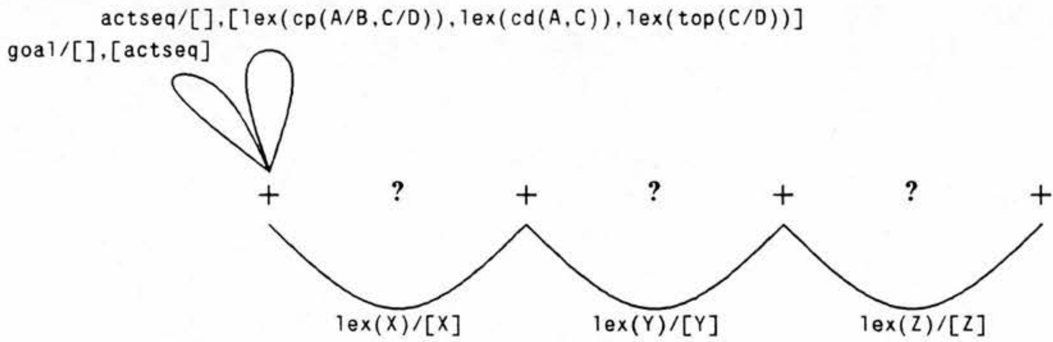
This results in the initial chart shown in fig.5.42. The edges "lex(X)" can combine freely as lexical items.



Initial Chart for Replanning the Goal "goal"

fig.5.42

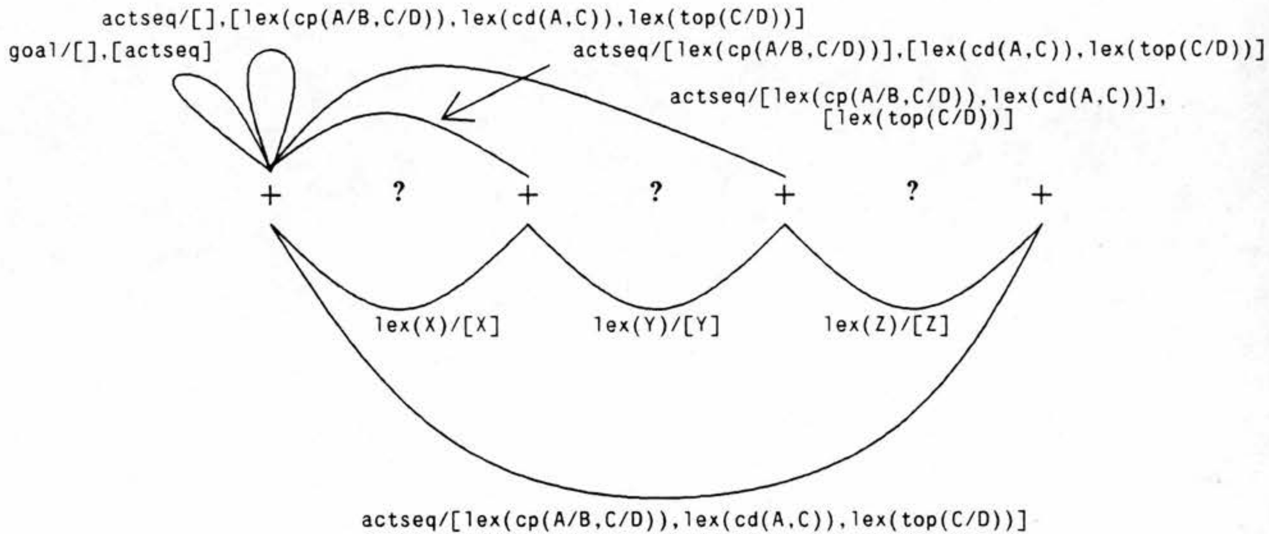
The chart parser is then used by working "top down" from the goal, adding new empty active edges whenever an active edge can be decomposed. Thus the chart will have the intermediate stage as shown in fig.5.43.



Initial Chart for Replanning the Goal "plan"

fig.5.43

The first element that the "actseq" empty active edge needs is "lex(cp(A/B,C/D))" which matches the inactive edge "lex(X)", so the "actseq" active edge is extended to account for this edge. This process continues, until the parsing process halts, with the final chart given in fig.5.44.



Final Chart for Replanning the Goal "plan"

fig.5.44

Once the parsing process has halted, then the chart can be analysed for cliches. If a complete parse has been found, then the edges which could not be accommodated in the plan by using the grammar, are compared with the corresponding edges in the complete plan (in fig.5.45 , "cd c" compared with "cd ~" and "cd c"). The effects of these two command sequences are compared by executing them in a filestore simulation. If the net effects are identical, then a specific cliche rule is added to the grammar. The addition of such a rule is shown in fig.5.45

Using the grammar of fig.5.39, the following command sequence was typed:

```
[cp(b/a,c/a),[add(c/a)]],
[cd(b,~),[delete(cwd(b)), add(cwd(~))]],
[cd(~,c),[delete(cwd(~)), add(cwd(c))]],
[top(c/a), []].
```

This command sequence is not explicitly represented in the plan grammar, and it leads to the following cliche rule to be added to the grammar.

```
rule(trial, cliche:1,cd(b,c), [cd(b,~), cd(~,c)], []).
```

Then, when a similar command sequence is observed (such as the one below), the old cliche rule is deleted and generalised to form a new cliche rule which accounts for this.

```
[cp(x/y,z/y),[add(z/y)]],
[cd(x,~),[delete(cwd(x)), add(cwd(~))]],
[cd(~,z),[delete(cwd(~)), add(cwd(z))]],
[top(z/y), []].
```

```
rule(trial, cliche:2,cd(A,B), [cd(A,~), cd(~,B)], []).
```

An Example of Cliche Recognition.

fig.5.45

The generalisation of the cliche rule is performed by looking for a similar rule before a new cliche rule is added. Similar rules have the same structure of the constituents of the rule. Generalisation of these constituents is then performed by decomposing the structures into their component atoms or variables. If there are two similar atoms then the generalised form has the same atom. If there are two dissimilar atoms, then the atom is generalised to a variable. An atom and a variable take the variable as the generalised form. Care must be taken when performing this generalisation

to ensure that the same variables are substituted throughout an expression for the same atoms.

When applied to the representation used in the UNIX Advisor, the cliche detection mechanism works in the same way but results in more complex cliche rules being added (see Appendix IX for an example).

The cliche detection mechanism can be used at any level in the plan grammar provided that the edges fit the description given above. Therefore, we are capable of detecting cliches at any level of plan abstraction, not just at the level of actions. For example, with the grammar:

```
plan ::= plan1, plan2, plan3.
plan1 ::= a, b.
plan2 ::= c, d.
plan3 ::= e, f.
plan4 ::= w, x.
plan5 ::= y, z.
```

and the observed action sequence;

```
a, b, w, x, y, z, e, f.
```

then, if "wxyz" is indeed equivalent to "c,d", recognition of the general cliche rule;

```
plan2 ::= plan4, plan5.
```

should occur, rather than the specific rule;

```
plan2 ::= w, x, y, z.
```

The cliche detection mechanism offers a powerful technique for user modelling because the grammar need not be complete, rules are generated which are specific to an individual rather than all possibilities having to be

compiled into the grammar. Cliches can be detected by STRIPS using triangle tables to generate MACROPS (Fikes et al, 1972). However MACROPS cannot be detected for interleaved or fragmented plans, whereas the chart parsing cliche detection technique can in principle handle these cases (although this would require a modification to the cliche detection rules). An example of a fragmented plan is:

```
1%      cp a b
2%      ls
3%      rm a
```

In the above example the "cp, rm" cliche for a "mv" is split by the "ls" command. The capability to recognise such fragmented cliches is important since the analysis in chapter 3 indicated that users often type commands which fragment the observation of planned behaviour (for example, by typing "ls" intermittently, reading mail, etc). By using the generalised fundamental rule developed in section 5.2.3 the chart parser could, in principle, recognise this cliche. However, this has not been implemented in this thesis.

5.4. Discussion.

The description of chart parsing indicates that the techniques are powerful and efficient when there is a problem that can be described by a grammar, and all possible solutions and partial solutions are needed. It is the explicit nature of the chart and the potentially flexible control structure which makes the chart appealing for plan recognition. The basic requirements for plan recognition have been exhibited, and there is the potential for the chart to be used in other user modelling tasks; where the grammar can form a part of the user model, the chart parser can be used to re-plan action sequences according to this model to introduce the user to new concepts.

5.4.1. Bidirectional Chart Parsing.

An interesting application of chart parsing techniques was made in a program to check the syntax of PASCAL programs (Elsom-Cook and duBoulay, 1986). The problem of syntax-checking usually involves the use of a left-to-right parse of the program. However, such an approach has limitations when a word is seen that does not fit the syntax because the parser has attempted to fit the sequence to an incorrect part of the grammar. This results in an error being detected too late (For example, a syntax checker giving an error message for "no end of program" when really the error was forgetting to end a comment). The syntax checker can take into account only the words which it has seen, that is the words preceeding the potential error. Therefore a large amount of information about the program is lost (all words coming after the point where an error was found). The program detects bugs in PASCAL programs by arranging for a chart parser to parse the program both left-to-right and right-to-left.

To illustrate this in a very simplified context, consider the grammar:

p1 ::= a,b,w,x,y,z.
p2 ::= a,b,c,d,e,f.

and the observed action sequence

a b w d e f

With Left to Right parsing:
p1 would have found "a b w"
and p2 would have found "a b"
so p1 seems to be the better explanation (explains three tokens, against two tokens for p2).

with Right to Left parsing:
p1 would have found nothing
and p2 would have found "f e d"
suggesting that "p2" is a better explanation.

Combining the information from both parses suggests that p2 is intended (five tokens against three tokens for p1).

They call this parsing technique "Island Parsing", the idea being to recognise areas of correct code throughout the program. The islands are initiated around PASCAL key words (eg. begin, while...), resulting in a chart which contains these partial parses. The key words form the basis of the parse, since the parsing process should be more confident that these actions are correct. Further analysis of these partial parses, using heuristics concerning typical errors and their relative frequency, is used to identify the most likely bugs. The domain of PASCAL syntax checking is suitable for this approach because the PASCAL key words form natural islands from which the chart can be developed in both directions (there is no such equivalent with UNIX). PASCAL also offers a "tight" syntax with a clearly

defined ordering of tokens, which does not occur in the UNIX domain (nearly any UNIX command can follow any other).

Bidirectional chart parsing has also been used in a natural language front-end (Steel and deRoeck, 1987), in which these parsing techniques are used to parse language input from both ends of the utterance. Bidirectional parsing is not used in the UNIX Advisor, although modifications were made to the parser to investigate bidirectional parsing, because the methods described in section 5.2 to detect fragmented, interleaved and enclosed edges made the bidirectional parsing technique redundant.

In FITS3 (Woodroffe, 1990), a chart parser is used to investigate plan recognition. This work proposes the use of techniques such as bidirectional parsing and endorsements (Sullivan and Cohen, 1985) to suggest the plan being followed. The work is currently involved with the temporal representations of plans in grammars (in our implementation, this problem has been overcome by using the parser incrementally), non-linearity of plans and the use of endorsements to rank plan hypotheses.

5.4.2. Comparison with other Plan Recognition techniques.

Most previous attempts at plan recognition have depended upon knowing the goals being followed. A notable exception to this is the POISE program (Carver et al, 1984) which uses a blackboard architecture to form hypotheses about the plan being followed. The blackboard incorporates a reason maintenance system which allows the hypotheses to remain consistent and independent. The chart-based approach also offers these features, since the edges in the chart remain independent and contain the local context in which they apply. Thus, the chart can be thought of as a specialised form of the blackboard architecture for hypothesising plans according to a pre-defined grammar. However, the plan recognition task requires further processing once the partial plans have been recognised. This ability can be incorporated into the blackboard, but lies outside the

scope of the chart parser - further analysis techniques being needed to analyse the resultant chart and select the plan that it is most likely that the user is following. Since the recognition of plan fragments is a computationally demanding task, it needs to be made as efficient as possible. Chart parsing offers this efficiency.

Other programs involving plan recognition use a description of the goal to be attained. This allows the programs to focus the search for the plan being followed, and offers more constraints for determining whether a valid plan is being followed. An example is the MACSYMA ADVISOR program (Genesereth, 1979), where the problem of determining the user's plan is not considered. It is assumed that the user's plan is known by the machine, which is obtained by the user defining his goal to the computer. The ADVISOR can then use the MUSER modeller to develop a graph of actions to achieve the goal. Such approaches offer a method of analysing user's actions by essentially avoiding the Plan Recognition Problem. This is achieved by placing the user in the "analysis loop" and interrogating him. We feel that this is not a sensible action to take with the Human-Computer Interaction problem, since we wish the computer to cooperate with the user rather than hinder him by relying solely upon questions to determine his goals.

5.4.3. Complexity and Efficiency Considerations.

The advantages of using a chart parser over other parsing techniques are that it generates all possible partial and complete parses, without having to rebuild the constituents of any parse. Aho derives a proof that the time to process a sequence with Earley's algorithm increases as the cube of the length of the sequence (Earley, 1972). This section attempts to place an upper bound on the amount of work performed by the parser. Clearly, this will be directly related to the grammar and the actions observed, but the aim is to compare the increase in complexity of the chart produced by relaxing the conditions of the fundamental rule.

An analysis was performed using a simple grammar which contained only two rules. However, the grammar was written such that every action would combine with edges in the chart. The grammar is shown in fig.5.46 and consists of a recursive rule definition which will recognise the action sequences: [a], [a,a], [a,a,a], ...etc as being valid plans. Although this is a very simple grammar, the addition of each action "a" combines with every active edge in the chart causing the chart to grow rapidly.

```
rule( test, p, [a]).
rule( test, p, [a, p]).
lexical( test, a, [a]).
```

Test Grammar.

fig.5.46

Using this test grammar, the chart parser was used to parse action sequences of increasing length, consisting of sequences of "a" (for example, "aaaaa"). For each parse, the number of actions in the sequence, the number of edges generated, the total cpu time taken for the parse, and the cpu time / edge in the resultant chart were calculated. This data is given in appendix X. The data was collected for three cases:

i. The fundamental rule does not allow "holes" in the plan (fundamental rule as given in fig.5.3 where $V_{Ae} = V_{Is}$).
ii. The fundamental rule allows "holes" of an arbitrary size in the plan ($V_{Ae} =< V_{Is}$).
iii. The fundamental rule allows "holes" equal in size to one action ($V_{Ae} = V_{Is}$ or $V_{Ae} + 1 = V_{Is}$).

The data was analysed by plotting graphs for each of the three fundamental rule conditions. Fig.5.47 shows a plot of the number of edges in the resultant chart against the length of the action sequence. All of the

curves show an exponential rise in the number of edges generated with an increase in action sequence length. However, this is dramatically greater for the charts which can contain holes, and there is not a significant difference between allowing holes of an arbitrary size against restricting this to holes of size 1 (this could be attributed to the short length of the action sequences used - restricted by a Local Stack Overflow causing the parser to fail on longer sequences). These results can be easily explained since every new element in the action sequence combines with many more active edges in the "hole" case than in the "no hole" case. The work involved producing the edges is reflected by the total cpu time taken to generate the chart. This is shown in fig.5.48, which shows similar increases of cpu time to the number of edges produced. When a plot of the cpu time per edge contained in the final chart against the action sequence length (fig.5.49), an increase in time taken to generate the edges with increasing action sequence length is shown. This increase is due to the time taken to find candidate edges, which increases as the number of edges in the chart increases. Again, the work required per edge is greater for the "hole" case because there are potentially many more edges to consider in this case. However, when the cpu time per edge is plotted against the number of edges in the chart (fig.5.50), the cpu time per edge appears to flatten out as the number of edges increases. This must be due to the proportion of potential combinations of edges to combinations resulting in an edge becoming constant. Thus for large numbers of edges in the chart, the cpu time seems to increase linearly with the number of edges in the chart.

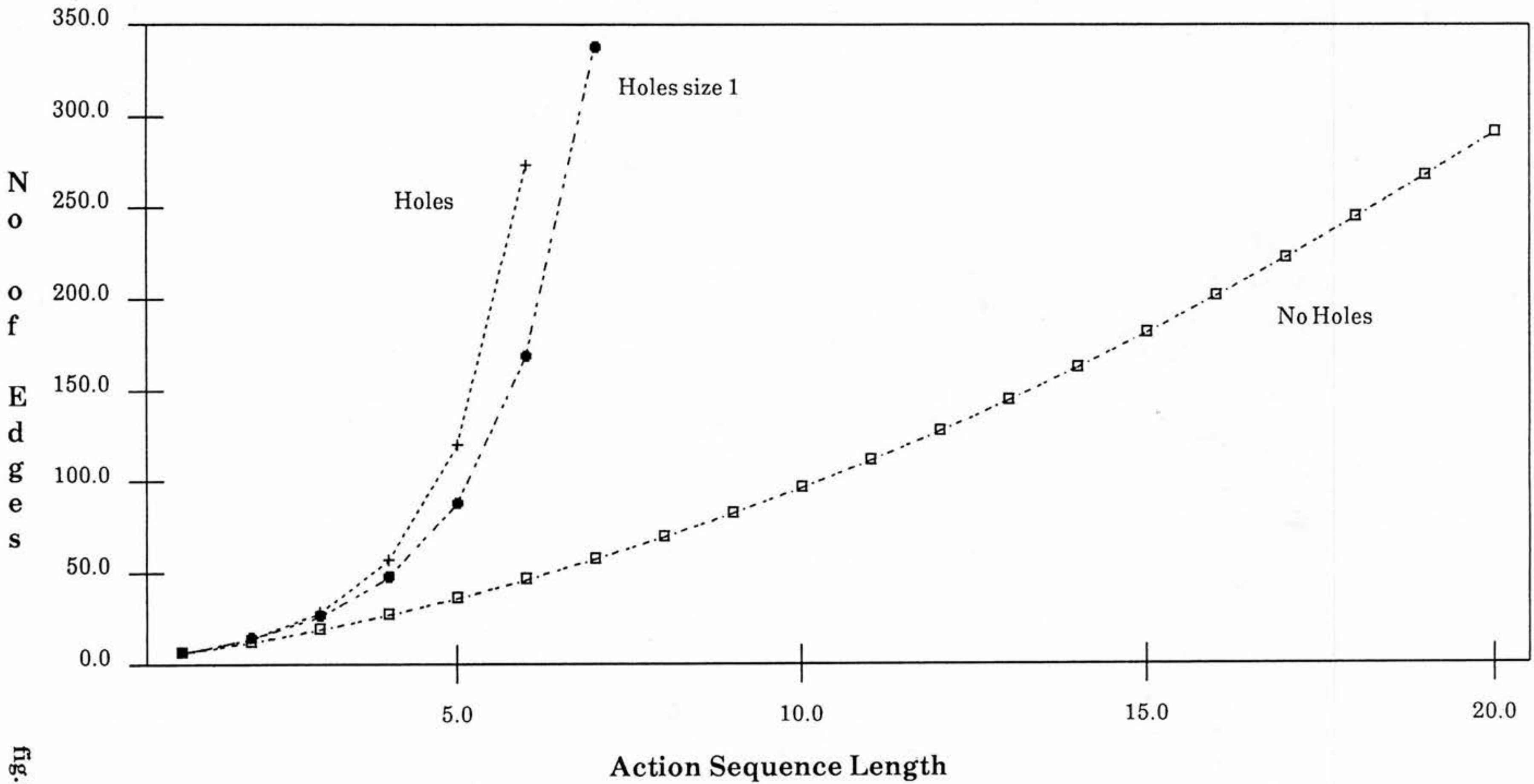


fig.5.47

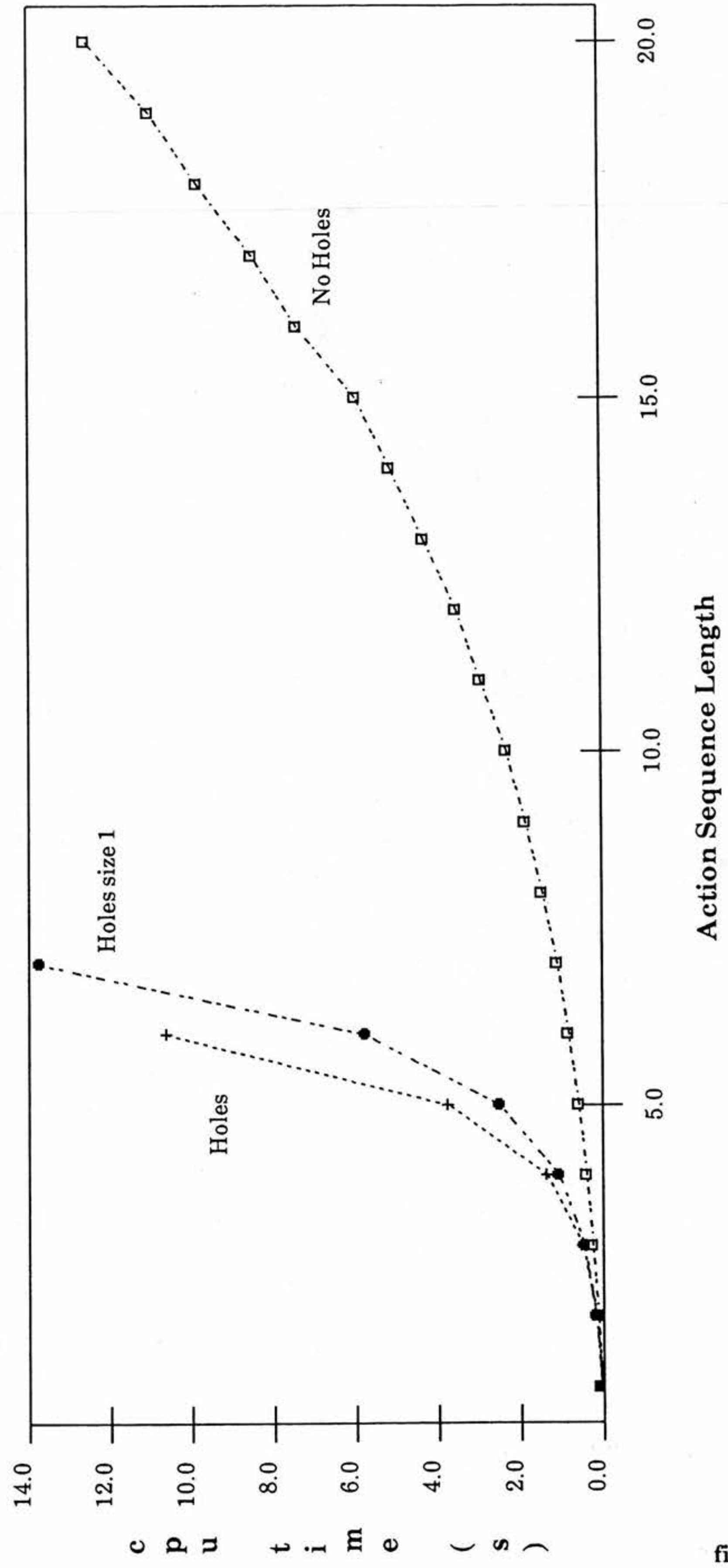


fig.5.48

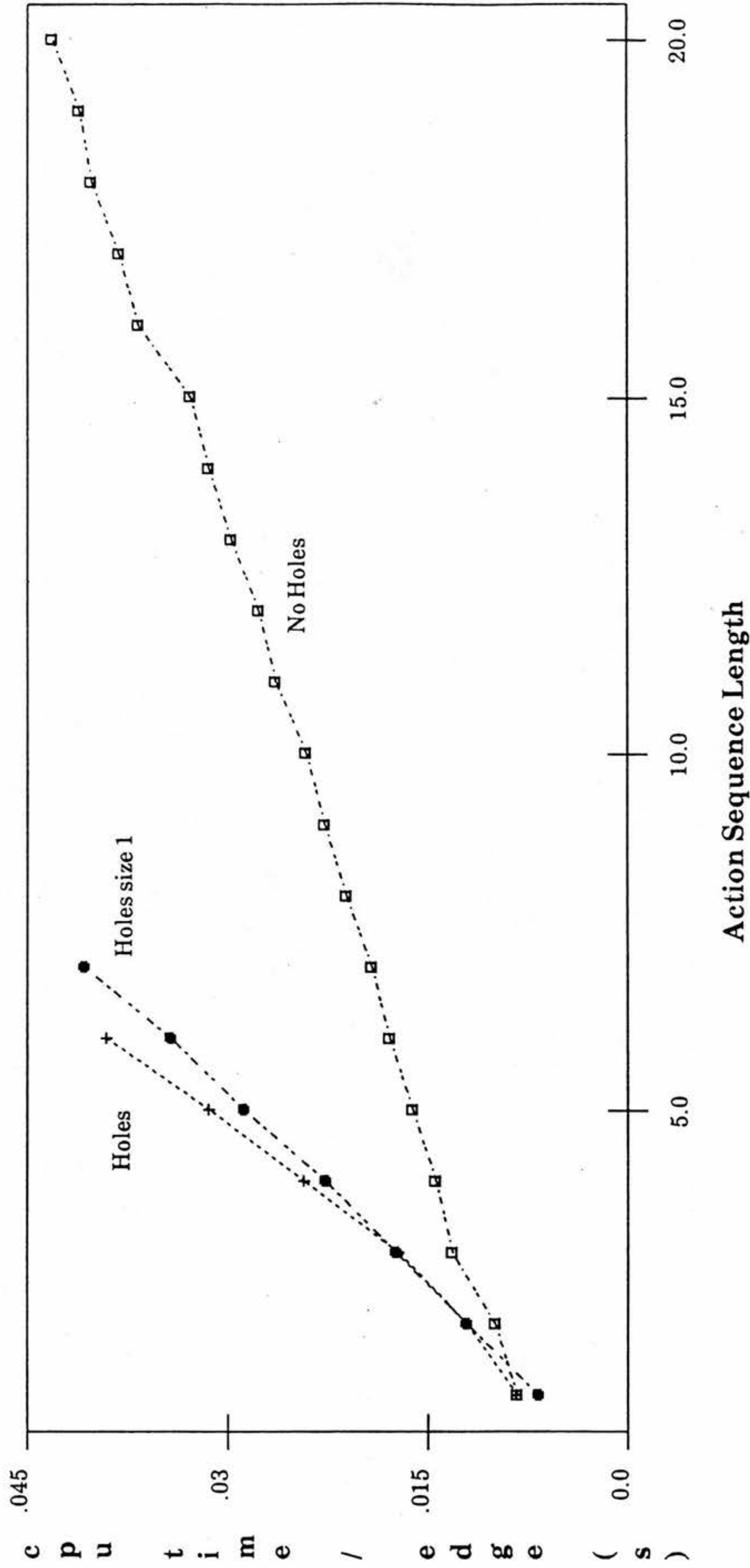


fig.5.49

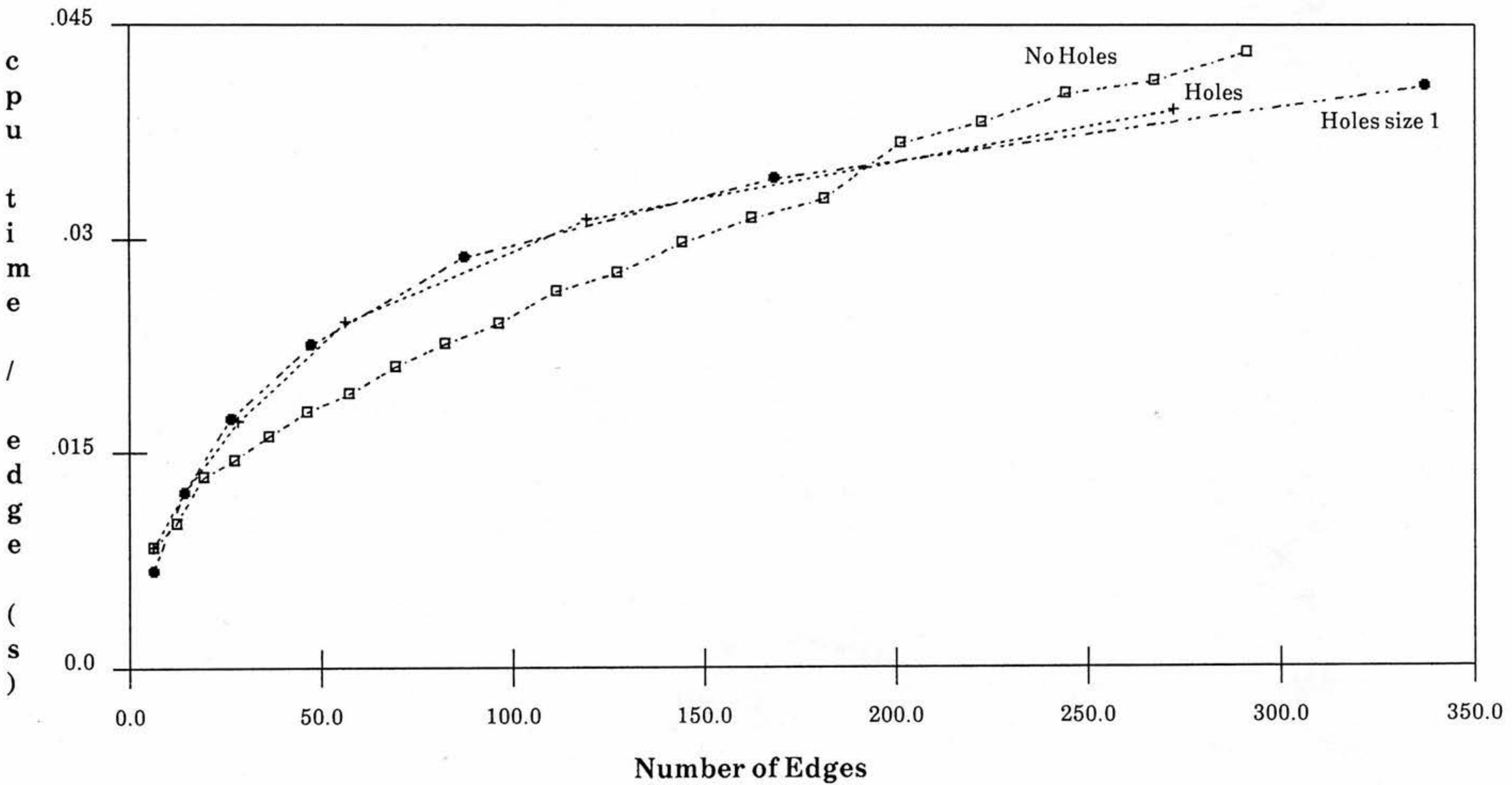


fig.5.50

It can be seen from this analysis that the parsing process becomes much more expensive when holes are allowed in plans. Two possible ways of overcoming this problem (apart from disallowing holes - which would be extremely restrictive) are:

i. Halt the parsing process before completion.
ii. Purge the chart of redundant edges and edges that are thought not to represent the user's intentions.

Both of these approaches require a heuristic analysis of the chart. In the case of halting the parsing process before completion, heuristics would need to determine when to stop the parsing process. This also implies that the parsing strategy should be controlled to give a "best first" recognition of plans, so that the exclusion from the chart of potentially useful edges is kept to a minimum. In general, it is not clear what the nature of the parsing strategy and "halting heuristic" should be, so that the chart is kept to a minimum size without discarding useful information.

The second alternative, of purging the chart, appears to be easier to implement. Edges can be discarded from the chart after a dialogue with the user has established his goals and plans for achieving these. Thus, there will be periodic purges of the chart once the user's intentions are known. There can also be purges to delete edges which ended a certain number of commands ago, and are therefore unlikely to combine with any future actions.

5.4.4. Extending the grammar for User Modelling.

The recognition of cliche rules is important because it enables the grammar to adapt to the characteristics of an individual. Since each individual will cause a unique grammar to be developed by the UNIX Advisor, this grammar must be considered as part of the user model. The grammar rules also convey information about the extent of the user's

knowledge through the "status" tag associated with each rule. In a more sophisticated implementation the two-valued "in" or "out" status could be replaced by a confidence factor indicating a degree of certainty that the UNIX Advisor "believes" that the user "possesses" that rule in his mental model of UNIX.

5.5 Summary.

This chapter has proposed the use of a chart parser for plan recognition and has outlined the main benefits of this approach; these being the detection of all possible partial and complete plans, the flexible nature of the control of the parsing process, and the explicit nature of the resultant parses. In addition to the benefits of chart parsing already described, the chart makes an ideal framework to support hypotheses; since each edge is internally consistent, distinct from all other edges, and retains an explicit account of its context.

The main problems encountered with the plan recognition task are the local context of plans, fragmented, multiple and interleaved plans, and incomplete grammars. These problems have been outlined, and solutions offered, based upon the chart parsing framework.

A solution to the problem of basing plan recognition on an incomplete grammar has also been offered, by the ability to recognise specific cliches that users appear to be following.

The information gained from the plan recognition task must now be combined with hypotheses about typical errors, misconceptions and lack-of-knowledge so that the user's problems can be detected from an analysis of his actions.

Chapter 6

A Chart-Based Approach to Advice Generation

6.0 Introduction.

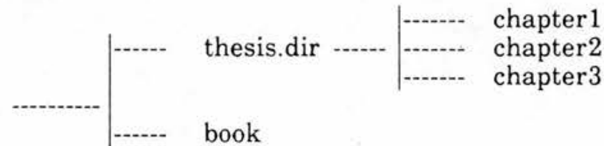
The Chart Parser performs the task of combining observed actions into possible plans that could account for those observations. However, the task of offering intelligent advice involves more than recognising possible plans. There are the additional problems of:

- Determining when the user requires advice.
- Determining what the user's problems are.
- Deciding what (if anything) to say to the user, and how to say it.

This chapter addresses the first two problems by developing multiple hypotheses about the user's intentions in parallel and then, using a heuristic to select between these hypotheses, determine whether the user requires advice. Once this has been established, the nature of the user's problem can be determined through the user model of commands.

6.1 Generating Multiple Hypotheses.

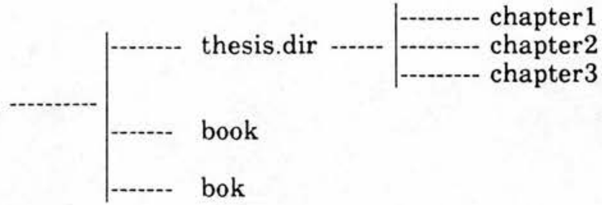
The commands that the user types do not necessarily reflect his intentions, but it is these intentions that must be determined by the UNIX Advisor if "intelligent" advice is to be forthcoming. By manipulating the user model of commands, it is possible to generate a set of hypotheses which describe plausible intended actions made by the user. These intentions cannot be inferred from observation of his actions alone, because of the errors that he makes and the misconceptions that he possesses about UNIX. For example, consider the UNIX filestore shown in fig 6.1. and the user issuing the command "cp thesis.dir bok".



A UNIX filestore.

Fig. 6.1.

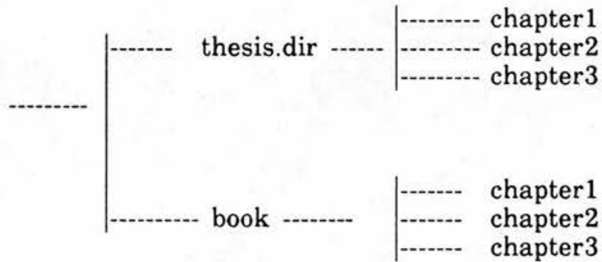
We could infer that the user intends a number of different effects for this command. Three examples of possible behaviours that might have been intended by the user are: a file "bok" to be created containing the literal contents of directory "thesis.dir" (fig6.2); a complete copy of the filestore tree rooted at "thesis.dir" created and rooted at directory "book" (fig6.3); or a complete copy of the filestore tree rooted at "thesis.dir" created and rooted at a new directory "bok" (fig.6.4).



The User's Intended UNIX filestore.

(the user made no mistake and has no misconception).

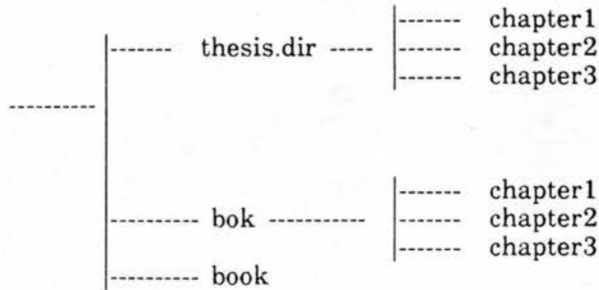
Fig. 6.2.



The User's Intended UNIX filestore.

(the user made the typing error "bok" for "book', and had the misconception that the command would copy a whole filestore tree).

Fig. 6.3.



The User's Intended UNIX filestore.
(the user had the misconception that the command would copy a whole filestore tree).

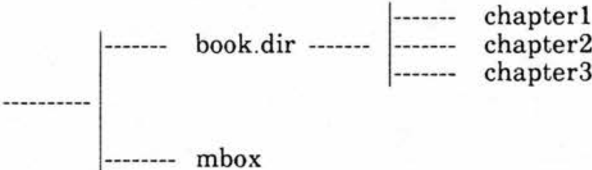
Fig. 6.4.

Without consulting information about the user's beliefs and the context in which the command was typed, it is impossible to select between these alternatives. The information about the user's beliefs is maintained in the user model of commands which describes misconceptions that the user might have, and the extent of the user's knowledge about UNIX commands. The context is the user's beliefs about the current state of the UNIX filestore, not the actual current filestore. The difference between the current filestore and the user's beliefs about the current filestore can arise due to inadequate feedback from UNIX about the effects of commands issued by the user.

Since the user's beliefs about the current state of the filestore cannot be directly measured, some other means of detecting the context is needed. The current state of the actual filestore could be used to generate hypotheses about misconceptions, but this would lead to a sub-set of possible hypotheses being generated which might not include the user's intended actions. When the commands are interpreted, a model of the current filestore is used to determine the possible instantiations of the command. However,

the current filestore might not correspond with the user's beliefs. Therefore, only one possible set of interpretations has been generated (based upon the filestore model). That is, hypotheses are being discarded automatically through the choice of context. For example, consider the command sequence and filestore shown in fig 6.5.

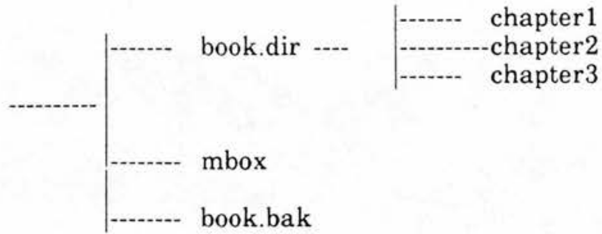
1% ls -F
book.dir/ mbox
2% cp book.dir book.bak
3% cd book.bak
FAIL



An Example UNIX filestore and command sequence.

Fig. 6.5.

In this example command sequence, command 2 ("cp book.dir book.bak") could be interpreted as creating a file, or creating a copy of the filestore tree. At this stage, the command does not fail so it is not clear whether UNIX has performed as the user intended. When command 3 ("cd book.bak") is typed by the user, the command cannot be understood by the system because the interpretations are based upon the current filestore context (fig.6.6) which does not contain a directory "book.bak". The only hypothesis that can be supported at this stage is that "cd" means "change current working directory to a file" ("book.bak"). Such a hypothesis is not plausible and was indeed not observed during the data-gathering phase of this research.



The UNIX filestore after the command sequence has been executed.

Fig. 6.6.

There are two methods of overcoming the problem of generating plausible hypotheses:

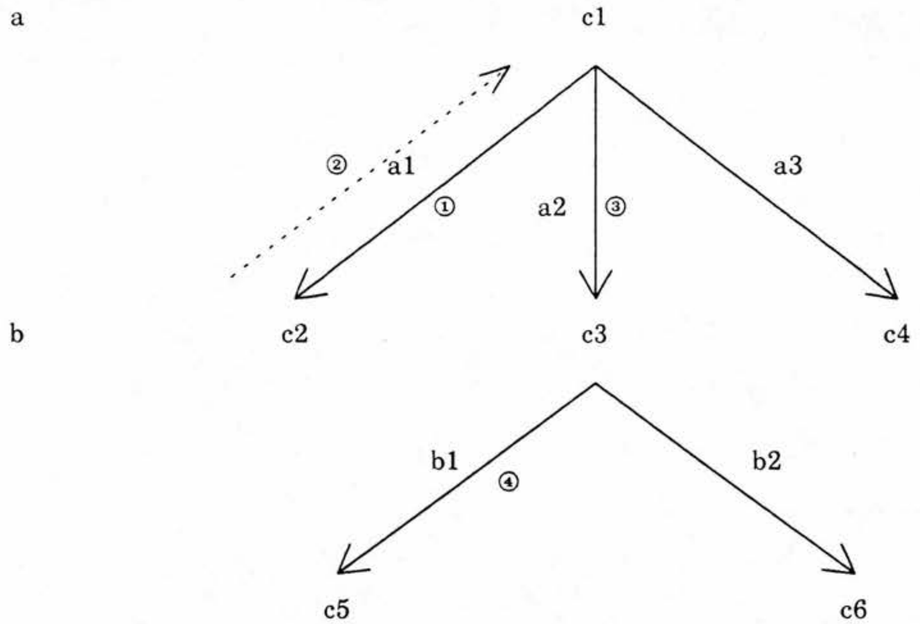
1. When a command cannot be interpreted, backtrack and search for possible reasons why the command failed. Modify the hypothesised description of the offending command and then recompute the command sequence.

2. Maintain multiple filestore contexts, one for each hypothesis generated. Then generate hypotheses for the following commands based upon each context, and selecting the "best" hypotheses and contexts as being those intended by the user.

6.1.1 Backtracking Solution.

The backtracking solution operates by generating hypotheses, based upon the current actual context, at each command typed by the user. In fig 6.7, command "a" is interpreted as three different hypotheses "a1", "a2" and "a3" using the present context "c1". Each hypothesis results in a different context ("c2", "c3" and "c4"). The interpretation "a1" corresponds

with the interpretation made by UNIX, resulting in the context "c2" which will be used to interpret the next command. When command "b" fails, the program attempts to determine why that command failed by first determining if there is a command hypothesis for the present command which would account for the behaviour. If there is such a hypothesis, then this could be the reason for the failure, however command "b" has no possible interpretations in context "c2". Therefore, the program backtracks ② to the previous command to determine whether it could satisfy the failed preconditions of the last command through an alternative hypothesis. If it can achieve this, then this command hypothesis is selected as the intended interpretation and the filestore context altered accordingly ③. The final command is then re-interpreted to ensure that the command makes sense in the new context ④. The backtracking continues until a solution is found, or there are no more commands to be checked (a practical limit of ten commands, say, might be imposed to prevent excessive amounts of backtracking).



Backtracking Solution to Detecting the User's Problems.

Fig. 6.7.

One problem with such an approach is that the mechanism only starts to look for a different interpretation once a command has failed. In the example given in fig.6.5, such an approach does not appear to be restrictive, but in general this leads to problems. It is conceivable that the user may expect that UNIX has achieved one task, when in fact it has achieved another, yet there has been no command failure. Under these conditions the user will never be told about his problem. An example would be if the user typed the command sequence (for the filestore given in fig.6.5):

```
1% ls -F
book.dir/      mbox
2% cp book.dir book.bak
3% cp mbox book.bak
```

Command 3 would not fail, but would overwrite the file "book.bak" with new contents (those from mbox). It might be some time before the user discovers that he has a problem.

In general the backtracking approach does not make use of the information available in the chart as a method of triggering the search for the user's intended plan, or for deciding among possible alternative interpretations.

6.1.2 Multiple Contexts Solution.

The multiple contexts mechanism requires that all possible contexts are maintained in addition to all possible hypotheses. This enables the chart to develop all possible plans in all possible filestore worlds concurrently. Such a solution would enable the system to compare different plan hypotheses and determine the plan that the user was most likely to be following. However, such an approach would not be practical because of the exponential growth in the number of hypotheses and contexts that need to be maintained.

A variation of this approach is to treat the command hypotheses differently from the interpretation made by the UNIX system (the "UNIX interpretation"). The UNIX interpretation of a command (if such an interpretation exists) is entered into the chart as a fully instantiated command. However, the hypotheses are not generated by applying the user model, etc to form fully instantiated hypotheses. Instead, partially instantiated hypotheses are generated and added to the chart. For example,

in fig 6.8 the command "cp mbox book" creates the fully instantiated UNIX interpretation ① in which all the conditions and effects are fully specified; and the hypotheses ② - ④ which have the skeletal form of an interpretation with no detail of the conditions or effects (these being left uninstantiated).

The command "cp mbox book" creates;

The interpretation ;

① copy-file-to-directory([mbox,[john],[],file,not-executable],
[book,[john],[ch1,ch2], directory,executable])
effects([add-node([mbox,[john,book],[],file,not-executable]).

and the hypotheses;

② copy-file-to-directory(A,B) effects(C).

③ copy-file-to-file(A,B) effects(C).

④ copy-tree-to-tree(A,B) effects(C).

where:

A, B and C are uninstantiated.

An Example Command Interpretation and Hypotheses.
Fig. 6.8.

The command type obtained for the hypotheses (eg "copy-file-to-file") is based upon the command issued by the user ("cp") and matched against possible mis-typings of the command (eg. "cd" for "cp"). These commands are then used to generate inactive edges in the chart by matching the commands against all possible lexical items in the grammar, and generating the resultant command types. However, the commands generated must take the same number of arguments as the command issued by the user, thus the command "change-directory(A) effects(B)" cannot be generated because "cd" takes one argument only and two arguments were

given by the user. These hypotheses are not dependent upon the context of the filestore because no detail has been placed in the hypotheses. The only information contained in the hypotheses at this stage is the command type thought to have been intended by the user. Therefore, the necessary command hypotheses are available to combine in the chart irrespective of context, one of which will be the user's intended plan. A modification has to be made to the chart parser to ensure that edges containing hypotheses do not take context into account when combining. This is achieved by annotating each edge as being a "hypothesis" (the edge contained one or more partially instantiated hypotheses) or an "interpretation" (the edge contained no hypotheses and is fully instantiated). If a new edge is an interpretation then the context is checked before it is added to the chart, otherwise the context is not checked at this stage. Although this approach does not require the maintenance of multiple contexts, it does result in meaningless edges being formed in the chart which would not have arisen if the context were taken into account.

It is this "multiple hypotheses" solution which will be developed in this chapter as a method of determining when the user requires help, and the nature of his problem.

6.2 Maintaining Multiple Hypotheses.

The chart parsing discussed in Chapter 5 used only one inactive edge between two adjacent vertices. That is, there was only one lexical entry in the chart for each utterance (eg. "det/[the]" for the utterance "the"). However, there are no restrictions on the number of lexical entries that can be placed in the chart for any action, the chart ensures that the interpretations remain independent and give rise to the relevant extensions. Thus, the chart can accommodate multiple hypotheses for the same action concurrently. For example, consider the grammar given below:

plan1 ::= a,b.
plan2 ::= c,d.

If the user typed "t1", this could be interpreted as either action "a" making no assumptions ie. "a(nil)", or action "c" subject to the assumptions "a1" ie. c(a1). This is achieved by matching the command typed by the user against the corresponding lexical items in the grammar. These lexical items represent the possible intentions that the user could have for "t1", and are added to the chart, giving the chart shown in fig.6.9.

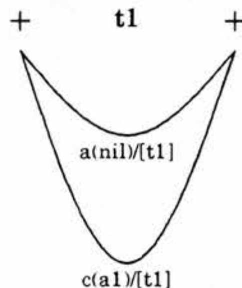


Chart (Stage 1).
fig.6.9.

The addition of these inactive edges causes the bottom-up policy to add empty active edges to the chart, to give the chart in fig.6.10.

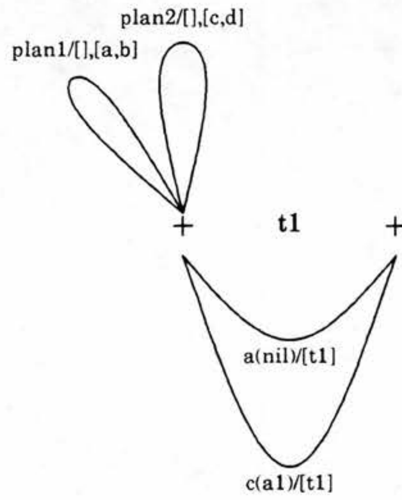


Chart (Stage 2).
fig.6.10.

These combine with the relevant inactive edges to give the chart shown in fig.6.11.

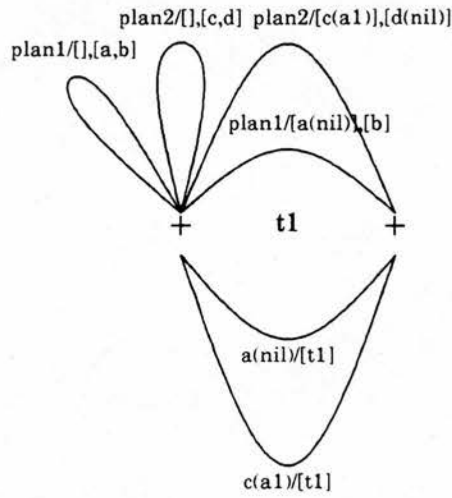
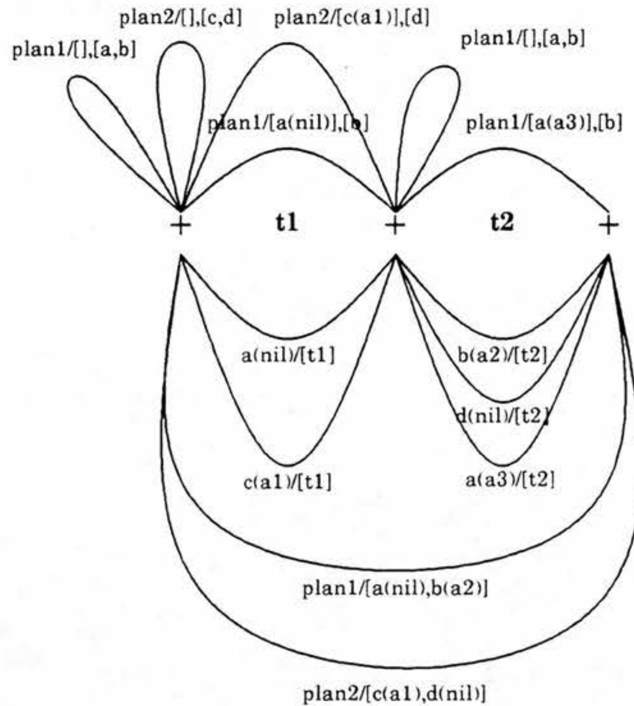


Chart (Stage 3).

fig.6.11.

Now, the action "t2"is observed, which can be interpreted as "d(nil)", "b(a2)", or "a(a3)". These interpretations are added as inactive edges in the chart, and combined according to the grammar, to give the final chart shown in fig.6.12.



Resultant Chart.

fig.6.12.

The resultant chart contains two inactive edges which span the entire action sequence. These edges are "plan1/[a(nil),b(a2)]" and "plan2/[c(a1),d(nil)]" which represent the plans;

"plan1" under assumptions "a2", and

"plan2" under assumptions "a1".

(Assumptions "nil" are defined to be where no assumptions are made).

Therefore the chart can be used to maintain different interpretations independently and the resulting plans retain the information about the interpretations that were made for the individual actions.

6.2.1 Maintaining Partially Instantiated Edges.

One particular problem encountered with the chart parser was how to maintain partially instantiated edges in the chart. Consider the grammar given below;

plan1 ::= cp, rm.
plan2 ::= cd, ls.

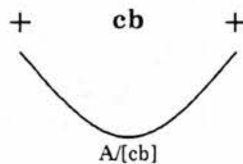
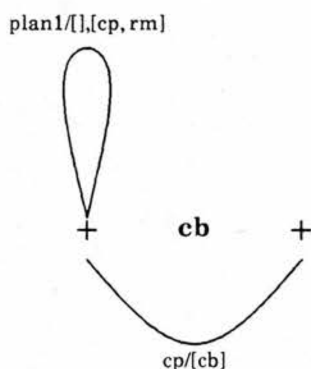


Chart Containing an Uninstantiated Edge.

fig.6.13.

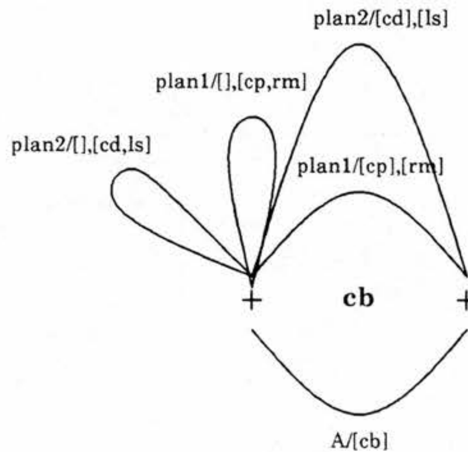
If the user makes a typing error and types "cb", then this does not have a lexical interpretation, but the hypothesis "A" (uninstantiated) could be made and this edge added to the chart (fig.6.13). Addition of this edge causes the parser to look for grammar rules which use this edge as a constituent, and the rule "plan1 ::= cp, rm" is found. Applying this rule instantiates "A" to "cp", and adds in the edge plan1/[][cp,rm] (fig.6.14).



Instantiation of Edges is Forced.

fig.6.14.

This instantiation causes the original inactive edge "A" to be instantiated as well to "cp", which precludes "plan2" from being developed (although this plan was just as valid as plan1). This is due to there no longer being an active edge which could combine with the first element of a plan rule (that is; "cp" will not match "cd"). This problem of the parser forcing instantiations and these propagating through the chart, is overcome by using a copy of an edge which is to be used for extension. The copy is not identical, since it contains different variables. This ensures that the instantiations do not propagate throughout the chart and therefore all possible extensions are found, resulting in the chart shown in fig.6.15 (in which the inactive edge "A" remains uninstantiated, enabling it to combine as necessary).



Resulting Chart with Instantiations Suppressed.

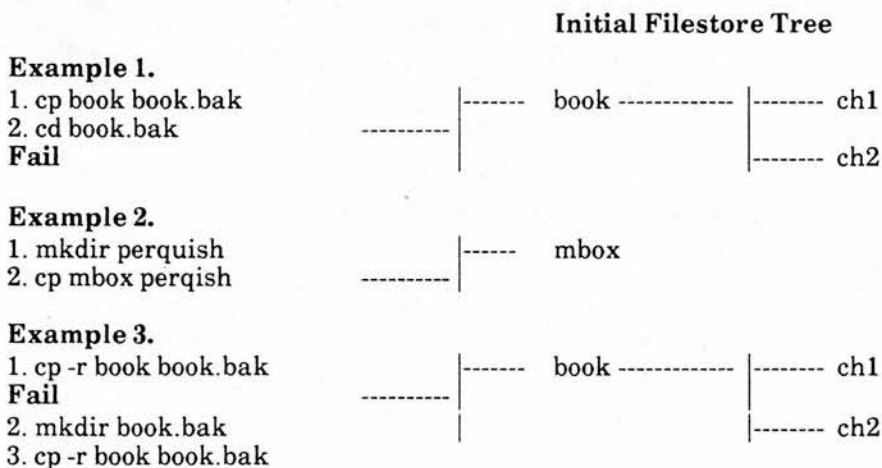
fig.6.15.

6.3 Analysing the Chart.

The chart contains all possible goals and plans that the user might plausibly (in the sense of being derivable from the grammar and lexical information) be following and it must be analysed to determine when the user requires advice, the most likely goal that the user is attempting to achieve, and his plan to achieve this goal.

6.3.1. Detecting When the User Requires Advice.

Detecting when the user requires help is not a simple task. Assuming that the user requires help only when a command has failed is a simplistic solution to the problem. Such a solution means that command failure cannot be anticipated by the system, and a long sequence of commands could be executed before an incorrect command is detected. Consider the command sequences shown in fig.6.16.



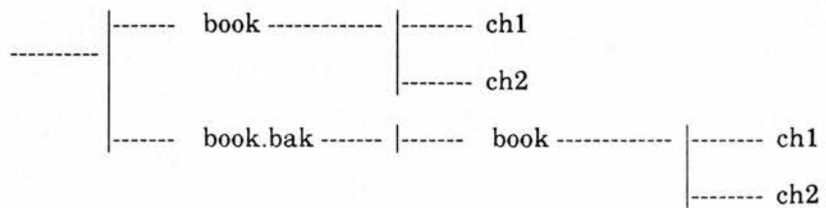
Example Command Sequences and Initial Filestores.

fig.6.16.

Example 1 causes a command failure because 'book.bak' is created as a file, not a directory, therefore the command to change working directory fails.

Example 2 does not exhibit a command failure, yet the sequence has failed to achieve what the user wanted (to copy the file 'mbox' into a new directory 'perquish'). The user made a typing mistake and a copy of the file was made at the original location, but with a different name. After command 2 in this sequence the user is probably not aware of his error.

Example 3 shows the user trying to copy a tree 'book', calling its root a new name 'book.bak'. The first attempt fails because the directory 'book.bak' has to exist before the copy can be achieved, so the user creates the directory and repeats the copy command - which succeeds, except the effects are different to those desired by the user (an extra level in the tree hierarchy is added) resulting in the filestore shown in fig.6.17.



Resultant Filestore after the Command Sequence in Example 3 (fig.6.16).

fig.6.17.

6.3.2. Using the Chart to Detect When a User Requires Advice.

One method of detecting when the user requires advice is to analyse the plan hypotheses contained in the chart and evaluate each plan according to certain criteria. These criteria must give a measure of how well the plan corresponds to the user's intentions. It is assumed that the user's problems with UNIX arise through having an incorrect model of the detail of the commands used to achieve his goals. One measure of the correspondence of the plan hypotheses and the user's intentions is the number of commands that a plan hypothesis contains. The assumption underlying such a measure is that the user will be attempting to achieve goals which consist of several commands. We do not assume that the longest plan that is recognised is necessarily believed by the user. The purpose of the plan recognition process is to attempt to find a plan which is consistent with the actions performed by the user so that advice can be generated. The plan that accounts for the maximum number of the user's actions provides the most information concerning the users actions and is therefore taken to be the best candidate upon which to base advice.

If the longest plan is longer than the plan which corresponds to the sequence of UNIX interpretations placed upon the commands (the plan which corresponds to the combination of commands interpreted in the actual UNIX world), then this suggests that the user's beliefs are best accounted for by that longest plan (since that plan explains the most actions). However, this plan might not be a valid plan because it contains uninstantiated edges, and no reference has been made to the context in forming these edges. Therefore, the plan must be investigated to determine whether there is a "filestore world" in which the plan could be valid. This verification is achieved by extracting the command interpretations upon which the plan is based from the chart, then applying the plan recognition over these commands, except only allowing one interpretation of each command (the UNIX interpretation). In effect this runs a simulation of the command sequence in a possible UNIX world. This interpretation of the commands forces the instantiation of the other commands in the sequence. After each

interpretation the command is parsed using the present filestore context and the resultant parse is analysed to ensure that it contains an instantiation of the original plan hypothesis. This mechanism for detecting whether a user requires help is contained in a heuristic which is used to analyse the chart after each command has been entered by the user. If the longest edge is not unique, then the first edge found is used for generating advice. Other edges are used in the analysis if the user rejects the explanation based upon the first edge.

Each edge in the chart is annotated with information about whether the edge contains hypotheses, and the length of span of the edge. This information can be used to decide if the user requires advice by applying the heuristic shown in fig.6.18.

If the maximum length of inactive edges ending at the present vertex is 'l',

  and there is an inactive edge of length 'l' in chart which contains one or more hypotheses,

  and all inactive edges of length 'l' contain hypotheses,

  and the edge is not a lexical item,

  and the command sequence giving rise to this edge is valid in a 'filestore' world,

then the user requires advice.

Heuristic to Determine Whether the User Requires Advice.

fig.6.18.

This heuristic gives a simple test to see if the user requires advice and is hard-wired into the analysis of the chart. Examples of the use of the heuristics are given in section 6.4.

6.3.3. Determining the User's Problem.

Once it has been established that advice is required by the user, the computer asks whether he is attempting to achieve the goal (the category of the edge) specified by the edge satisfying the above heuristic. If the user confirms the goal, then the command interpretations which were used in order to generate the goal are selected and the user is asked whether the effects produced by the command sequence are those desired. If the goal is not confirmed, then the UNIX ADVISOR searches for another goal that would satisfy the conditions for initiating advice. The command interpretations are generated by using the UNIX model of commands and the user model of commands. The system searches for a sequence of command interpretations which satisfy the goal hypothesised by the chart. If a command in that sequence is generated by the UNIX model then it does not contain a misconception and is marked as an "interpretation". Otherwise the command is interpreted using the user model to generate a

misconception, or typing or path errors for the command are generated (section 4.4.1). This results in a fully instantiated command sequence which is then verified by applying it to the filestore model to ensure that it is a plausible command sequence. Not all edges in the chart will result in meaningful command sequences. Those that are shown to be of no value are removed from the chart to prevent them from combining further.

Once the command sequence has been verified, then the misconceptions and/or errors involved are isolated and communicated to the user. Misconceptions are modelled as the difference between the user model of commands and the UNIX model of commands, and are therefore not totally explicit at present. However, the misconceptions could be asserted as facts in the user model for future use.

The behaviour of the system is given in section 6.4 which shows how the system responds to the user making a typing error, an error in describing a file path, and possessing a misconception about a command. The examples show the raw output of the system, which would need to be improved if it were to be used with "real" users.

6.4. Example Advice Generation.

This section gives the response of the UNIX Advisor to three action sequences that the user types. The grammar used in the analysis is the grammar given in Appendix VIII. The first example gives the response of the system to a potential typing error made by the user; the second due to a potential path error; and the third to a potential misconception that the user possesses. Each of the responses of the system displays the goal that the UNIX Advisor expects the user to be following, and a complete semantic description of the commands that the user typed which would achieve this goal (ie the user's intended semantic interpretation of the commands). At this point the user is asked to confirm that this is what he intended. If he confirms this, then no more solutions are found, and the error or misconception is implicit in the semantic descriptions, which could then be extracted. Otherwise the UNIX Advisor backtracks to find an alternative solution by first changing the semantic interpretations of the command, and then finding alternative goals that the user might have been following. Any plan which is discarded by the user, as not reflecting his beliefs, is purged from the chart. If the user accepts that an explanation of his actions is described by the plan then the alternative interpretations of the commands are purged from the chart (this assumes that one plan is sufficient to explain his actions). In figures 6.19 to 6.21, the lines have been numbered for convenience.

6.4.1 Example Advice Derived from a Potential Typing Error.

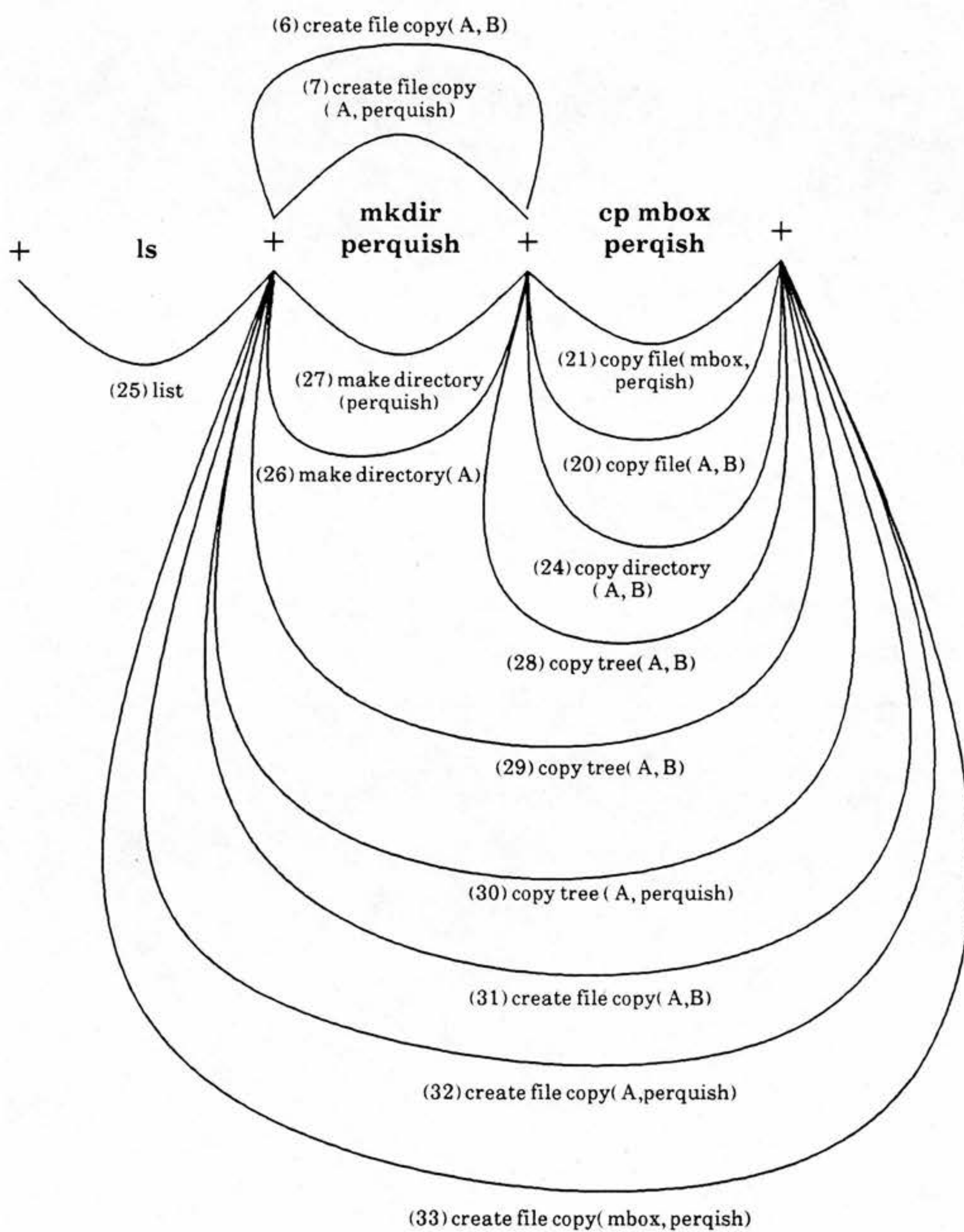
This example (fig.6.19) shows a user making a typing mistake for the name of a directory "perquish" when attempting to copy a file to this directory. No command in the sequence failed, but the advice generation is triggered by the more plausible plan that the user has just created the directory to hold a copy of the file. The corresponding chart is shown in fig.6.20. This chart has been simplified to include the significant edges only, and these edges have been annotated in a simplified manner to aid

readability. Each edge has been annotated with a number enclosed in brackets which corresponds with the edge number given in fig.6.19 and with the complete listing of the edges given in Appendix XI. The edges given in fig.6.19 are fully instantiated versions of the edges in the chart. This instantiation takes place when the command and filestore models are used to determine possible interpretations of the commands. Edge 27 is an interpretation and is fully instantiated in the chart. However, edges 20 and 32 are hypotheses and therefore only partially instantiated in the chart. These edges are instantiated when the command instantiations are made through consultation of the UNIX, user and filestore models.

On line 5 of fig.6.19 the UNIX advisor detects that there are possible plans that account for the user's actions better than the literal interpretation. The literal interpretation for command 3 is that file "mbox" is renamed as "perqish". The possible goal of "create_file_copy" is detected (line6), and the interpretations of the commands needed to achieve this goal are determined (lines 9 to 11 for command 2; lines 12 to 14 for command 3). This new interpretation for command 2 contains no misconceptions, and has the effect of creating the directory "perquish". The hypothesis for command 3 is that the command contains a typing error (line 14), and that the intention is to copy the file "mbox" into the directory "perquish". This has the effect of adding file "mbox" into the directory "perquish" (line 13). The UNIX Advisor asks the user to confirm that this is his intended plan (line 15), and the edge in the chart corresponding to this plan is then shown as confirmation (lines 17 to 23).

```
         1 % ls                                                                   1
         bok   book/  f1   f2   mbox  qwerty                                      2
         2 % mkdir perquish                                                       3
         3 % cp mbox perqish                                                      4

         looking for explanations......                                           5

         Goal: goal create_file_copy mbox at_location [john,jml,forth2,usr,/]     6
         to mbox at_location [perquish,john,jml,forth2,usr,/]                     7
         Sequence contains 1 errors or misconceptions.                            8

edge 27  2/([mkdir,perquish],[mkdir,[perquish,[john,jml,forth2,usr,/],[],unknown, 9
         unknown]],[(add_node,[perquish,[john,jml,forth2,usr,/],[],dir,x])],      10
         _126508,interpretation(2),ok)                                            11
edge 20  3/([cp,mbox,perqish],[cp,[mbox,[john,jml,forth2,usr,/],[],file,o],[perquish, 12
         [john,jml,forth2,usr,/],[],dir,x]],[(add_node,[mbox,[perquish,john,jml,  13
         forth2,usr,/],[],file,o])], _131789,hypothesis(3),typing_error)          14

         Is this what you wanted ? (yes/no): yes                                  15
         Your plan is:                                                            16
edge 32  2-edge(rule 15,[rule 15,lexical 930,lexical 910],goal create_file_copy   17
         mbox at_location [john,jml,forth2,usr,/] to mbox at_location             18
         [perquish,john,jml,forth2,usr,/] status _131789,hypotheses([3]),         19
         [copy_file mbox at_location [john,jml,forth2,usr,/] to mbox              20
         at_location [perquish,john,jml,forth2,usr,/] status _131789=3-4,         21
         make_directory perquish at_location [john,jml,forth2,usr,/]              22
         status _131789=2-3],[],2,4,[2,3],[])                                     23

         4 %                                                                      24
```

fig.6.19
Example Advice Derived from a Typing Error.



Simplified Chart for typing error example (fig6.19).

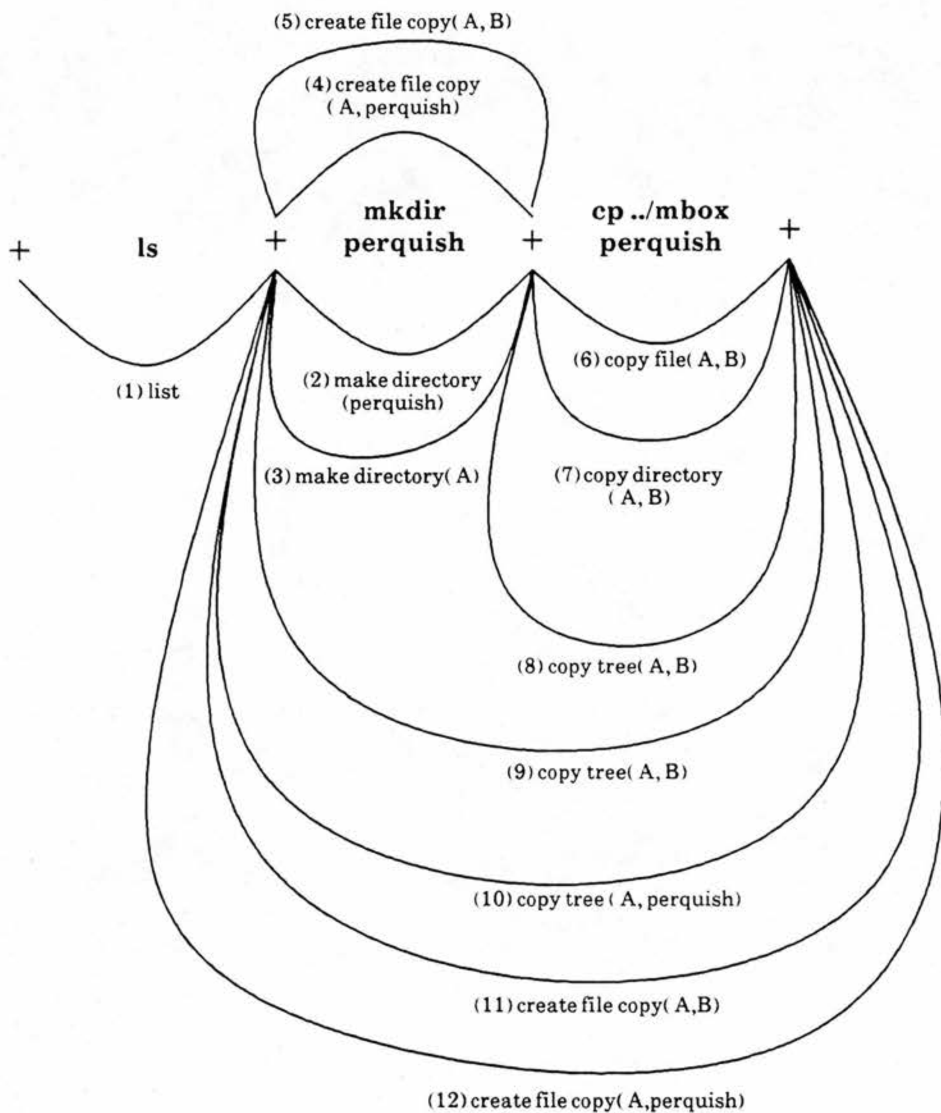
fig.6.20.

6.4.2 Example Advice Derived from a Potential Path Error.

This example (fig.6.21) shows a user attempting to copy a file to a directory that has just been created.

```
         1 % ls                                                                1
         bok   book/  f1   f2   mbox  qwerty                                   2
         2 % mkdir perquish                                                    3
         3 % cp ../mbox perquish                                               4

         looking for explanations......                                        5

         Goal: goal create_file_copy mbox at_location [john,jml,forth2,        6
         usr,/] to mbox at_location [perquish,john,jml,forth2,usr,/]           7
         Sequence contains 1 errors or misconceptions.                         8

edge 2   2/([mkdir,perquish],[mkdir,[perquish,[john,jml,forth2,usr,/],[],      9
         unknown,unknown]],[(add_node,[perquish,[john,jml,forth2,usr,/],       10
         [],dir,x])],_126508,interpretation(2),ok)                             11
edge 6   3/([cp,../mbox,perquish],[cp,[mbox,[john,jml,forth2,usr,/],[],file,o], 12
         [perquish,[john,jml,forth2,usr,/],[],dir,x]],[(add_node,[mbox,[perquish, 13
         john,jml,forth2,usr,/],[],file,o])],failed,hypothesis(3),path_error)  14

         Is this what you wanted ? (yes/no): yes                               15
         Your plan is:                                                         16
edge 12  2-edge(rule 15,[rule 15,lexical 930,lexical 910],goal create_file_copy 17
         mbox at_location [john,jml,forth2,usr,/] to mbox at_location          18
         [perquish,john,jml,forth2,usr,/] status failed,hypotheses([3]),       19
         [copy_file mbox at_location [john,jml,forth2,usr,/] to mbox           20
         at_location [perquish,john,jml,forth2,usr,/] status failed=3-4,       21
         make_directory perquish at_location [john,jml,forth2,usr,/]           22
         status failed=2-3],[],2,4,[2,3],[])                                   23

         4 %                                                                   24
```

fig.6.21
Example Advice Derived from a Path Error.



Simplified Chart for path error example (fig6.21).

fig.6.22.

However, he incorrectly specifies the path for the file and the command fails. The program is able to locate other possible files that the user may have intended and suggest these to him. On line 5 the UNIX Advisor detects that the user may be attempting to follow a different plan to that being performed by UNIX. The "create_file_copy" goal is suggested to the user as being his intended goal (lines 6 to 7). The new interpretations of commands 2 and 3 are given on lines 9 to 14, these indicating that the user might have incorrectly specified the filepath for file "mbox" (line 12). This expansion for the location of "mbox" contains a path error. The user confirms that this is his intended plan (line 15), and the plan is displayed (lines 16 to 23). A simplified form of the chart is shown in fig.6.22. The chart is similar to the chart for the typing error, except that all the edges for the command "cp ../mbox perquish" contain hypotheses since this command failed. Again, the advice shows instantiations of the edges extracted from the chart, these instantiations occurring when the commands are interpreted in the command and filestore models.

6.4.3 Example Advice Derived from a Potential Misconception.

The example shown in fig.6.23 shows the user attempting to copy a filestore tree. A simplified chart for this command sequence is shown in fig.6.24. The user initially tries to achieve this with the "cp -r" command (line 3), which fails because the destination directory does not exist. The program searches through different explanations for this behaviour (lines 4 to 22), but the user does not confirm any of these goals. These goals are different instantiations of the goal "copy_tree" (edge 2 on the chart shown in fig.6.24). The goals are generated by modifying the user model and introducing typing and path errors to determine a plausible sequence of command interpretations and hypotheses that could account for the observed command sequence. For example, the first hypothesis was that the user possessed the misconception that the directory "book" would be created by the command (lines 7 to 13). Eventually one goal is suggested that

corresponds with the user's intentions (lines 22 to 29), and he confirms this goal. An assumption is made that the user is now advised that he must create a directory.

The user creates the directory (line 38), and then attempts to use "cp -r" again (line 39). This latter command does not fail, but the advice generator is invoked because the goal that has been achieved is different to his original goal (he has added another directory level in the filestore tree). The program finds a better plan which is a redo of the user's initial plan (line 41), and this is raised as the user's intentions. Again there is more than one instantiation of the commands for this goal. This results in the various instantiations of edges 3, 5 and 8, the instantiation that fits the user's beliefs is the second plan that is presented (lines 56 to 69). The unix advisor also presents the intended goal in lines 72 to 79, which is an instantiation of edge 11 in the chart in fig.6.24. This "redo copy_tree" goal contains the initial misconception about copying a tree, and the misconception that "cp -r" does not copy the contents of the directory "book", but copies the whole filestore tree "book". This results in the directory "book" being copied as a sub-tree of "book.bak", instead of the intended plan of copying the contents of directory "book" to "book.bak". The UNIX Advisor established this intended plan in lines 23 to 30.

This example shows the flexibility of the system for detecting misconceptions, but it also highlights the problems involved with the computer / user dialogue necessary to isolate the problem.

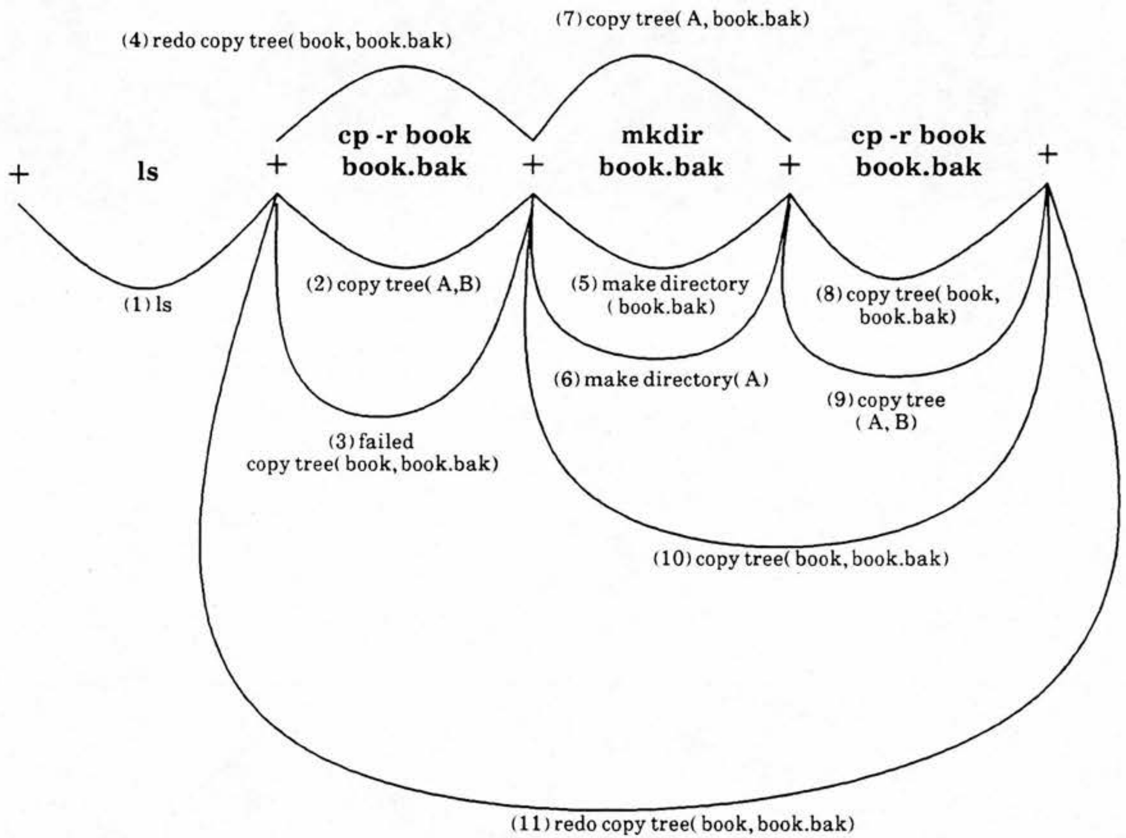
```
1 % ls                                                          1
bok  book/  f1   f2   mbox  qwerty                              2
2 % cp -r book book.bak                                         3

looking for explanations......                                  4
removing copy_tree _126346 at_location _126349 to _126385      5
at_location _126388 status failed                               6
```

```
        Goal: goal copy_tree book at_location [john,jml,forth2,usr,/] to book           7
        at_location [john,jml,forth2,usr,/]                                             8
        Sequence contains 1 errors or misconceptions.                                   9

edge 2  2/([cp,-r,book,book.bak],[cp,[book,[john,jml,forth2,usr,/],                     10
        [chapter1,chapter2],dir,x],[book.bak,[john,jml,forth2,usr,/],[],unknown,        11
        unknown]],[(add_node,[book,[john,jml,forth2,usr,/],[],dir,x])],failed,          12
        hypothesis(2),misconception)                                                    13

        Is this what you wanted ? (yes/no): no                                          14

        Goal: goal copy_tree book at_location [john,jml,forth2,usr,/] to                15
        book.bak at_location [john,jml,forth2,usr,/]                                    16
        Sequence contains 1 errors or misconceptions.                                   17

edge 2  2/([cp,-r,book,book.bak],[cp,[book,[john,jml,forth2,usr,/],                     18
        [chapter1,chapter2],dir,x],[book.bak,[john,jml,forth2,usr,/],[],                19
        unknown,unknown]],[(add_node,[book.bak,[john,jml,forth2,usr,/],                 20
        [],dir,x])],failed,hypothesis(2),misconception)                                 21

        Is this what you wanted ? (yes/no): no                                          22

        Goal: goal copy_tree book at_location [john,jml,forth2,usr,/] to                23
        book.bak at_location [john,jml,forth2,usr,/]                                    24
        Sequence contains 1 errors or misconceptions.                                   25

edge 2  2/([cp,-r,book,book.bak],[cp,[book,[john,jml,forth2,usr,/],                     26
        [chapter1,chapter2],dir,x],[book.bak,[john,jml,forth2,usr,/],[],                27
        unknown,unknown]],[(add_node,[book.bak,[john,jml,forth2,usr,/],                 28
        [chapter1,chapter2],dir,x])],failed,hypothesis(2),misconception)                29

        Is this what you wanted ? (yes/no): yes                                         30
        Your plan is:                                                                   31
edge 2  1-edge(rule 133,[rule 133,lexical 912],goal copy_tree book                      32
        at_location [john,jml,forth2,usr,/] to book.bak at_location                     33
        [john,jml,forth2,usr,/] status failed,hypotheses([2]),[copy_tree book           34
        at_location [john,jml,forth2,usr,/] to book.bak at_location                     35
        [john,jml,forth2,usr,/] status failed=2-3],[],2,3,[2],[])                       36
```

	Usage: cp f1 f2; or cp [-r] f1 ... fn d2	37
	3 % **mkdir book.bak**	38
	4 % **cp -r book book.bak**	39
	looking for explanations......	40
	Goal: redo copy_tree book at_location [john,jml,forth2,usr,/] to	41
	book.bak at_location [john,jml,forth2,usr,/]	42
	Sequence contains 2 errors or misconceptions.	43
edge 3	2/([cp,-r,book,book.bak],[cp,[book,[john,jml,forth2,usr,/],	44
	[chapter1,chapter2],dir,x],[book.bak,[john,jml,forth2,usr,/],[],	45
	unknown,unknown]],[(add_node,[book.bak,[john,jml,forth2,usr,/],	46
	[chapter1,chapter2],dir,x])],failed,hypothesis(2),misconception)	47
edge 5	3/([mkdir,book.bak],[mkdir,[book.bak,[john,jml,forth2,usr,/],[],	48
	unknown,unknown]],[(add_node,[book.bak,[john,jml,forth2,usr,/],	49
	[],dir,x])], _132624,interpretation(3),ok)	50
edge 8	4/([cp,-r,book,book.bak],[cp,[book,[john,jml,forth2,usr,/],	51
	[chapter2,chapter1],dir,x],[book.bak,[john,jml,forth2,usr,/],[],dir,x]],	52
	[(add_node,[book.bak,[john,jml,forth2,usr,/],[],dir,x])],failed,	53
	hypothesis(4),misconception)	54
	Is this what you wanted ? (yes/no): **no**	55
	Goal: redo copy_tree book at_location [john,jml,forth2,usr,/] to	56
	book.bak at_location [john,jml,forth2,usr,/]	57
	Sequence contains 2 errors or misconceptions.	58
edge 3	2/([cp,-r,book,book.bak],[cp,[book,[john,jml,forth2,usr,/],	59
	[chapter1,chapter2],dir,x],[book.bak,[john,jml,forth2,usr,/],[],	60
	unknown,unknown]],[(add_node,[book.bak,[john,jml,forth2,usr,/],	61
	[chapter1,chapter2],dir,x])],failed,hypothesis(2),misconception)	62
edge 5	3/([mkdir,book.bak],[mkdir,[book.bak,[john,jml,forth2,usr,/],[],	63
	unknown,unknown]],[(add_node,[book.bak,[john,jml,forth2,usr,/],	64
	[],dir,x])], _132624,interpretation(3),ok)	65
edge 8	4/([cp,-r,book,book.bak],[cp,[book,[john,jml,forth2,usr,/],	66
	[chapter2,chapter1],dir,x],[book.bak,[john,jml,forth2,usr,/],[],dir,x]],	67
	[(add_node,[book.bak,[john,jml,forth2,usr,/],[chapter2,chapter1],	68
	dir,x])],failed,hypothesis(4),misconception)	69

```
        Is this what you wanted ? (yes/no): yes                                         70
        Your plan is:                                                                   71
edge 11 3-edge(rule 10,[rule 10,rule 133,lexical 912,rule 14,lexical 930,               72
        lexical 912],redo copy_tree book at_location [john,jml,forth2,usr,/]            73
        to book.bak at_location [john,jml,forth2,usr,/] status failed,                  74
        hypotheses([2,4]), [goal copy_tree book at_location                             75
        [john,jml,forth2,usr,/] to book.bak at_location [john,jml,forth2,usr,/]         76
        status failed=3-5,goal copy_tree book at_location                               77
        [john,jml,forth2,usr,/] to book.bak at_location [john,jml,forth2,usr,/]         78
        status failed=2-3],[],2,5,[2,3,4],[failed==failed])                             79

        5 %                                                                             80
```

fig.6.23
Example Advice Derived from a Misconception.



Simplified Chart for misconception (fig.6.23).

fig.6.24.

6.5 Alternative Advice.

The user will require help and advice at times other than those when his plan has deviated from the actual plan. One example of this is where users are restricted to a sub-set of UNIX commands and by doing so, make poor use of the facilities available (Jerrams-Smith, 1986) (for example, using "cat file1>file2" instead of "cp file1 file2", or "cp file1 file2; rm file1" instead of "mv file1 file2"). It might not always be appropriate to substitute the sequence of commands that the user typed, with an alternative command sequence, since they might have different side-effects or have been typed for a specific reason. For example the user might decide to copy a file and then change his mind to "move" it instead. Therefore he will add the extra "remove" command. It would be wrong to frustrate the user by always telling him about better ways to achieve particular goals. A better approach might be to have a more subtle heuristic about advising the user about better ways to achieve goals than the heuristic:

> *"If the user has achieved goal and there is a better (shorter) plan to achieve that goal than the plan he has followed, then advise the user about the better plan".*

Other factors should be taken into account, for example, whether the user knows about the commands contained in the new plan, how many new (to the user) commands are contained in the new plan (so that advice which contains too many new concepts is not offered to the user).

The chart can be analysed and potential goals that the user has achieved (but could be achieved using a shorter plan) are detected. First the completed goals are isolated, then the goal is re-planned (this could be achieved by running the chart parser top-down). Next, an analysis of the resultant chart is made using the heuristic:

"If a plan can be found that contains a shorter plan than the user was following and the plan involves rules from the plan grammar that the user does not know, then advise the user about this plan".

An example of advice generated using this heuristic is given in fig.6.25.

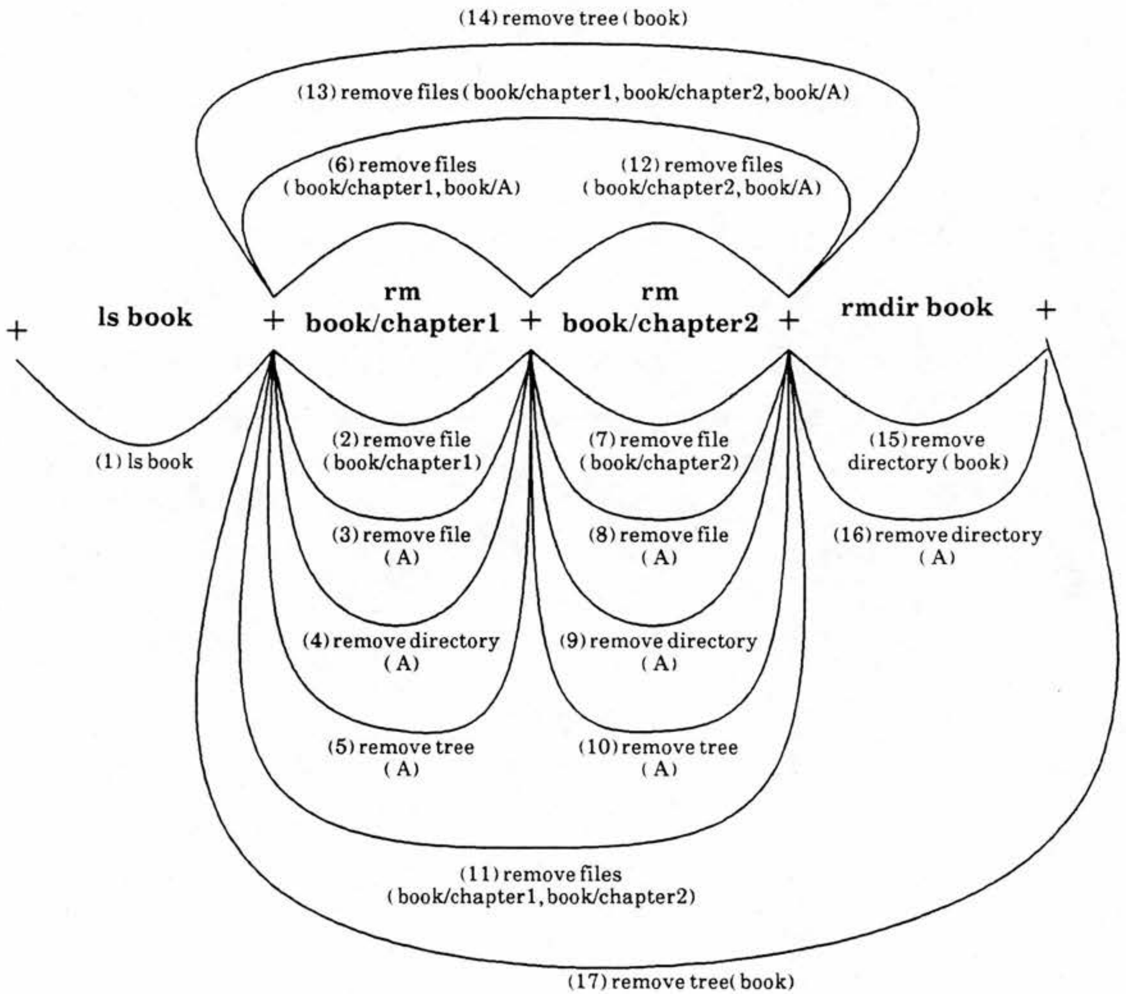
---

	1 % **ls**	1
	bok book/ f1 f2 mbox qwerty	2
	2 % **ls book**	3
	chapter1 chapter2	4
	3 % **rm book/chapter1**	5
	4 % **rm book/chapter2**	6
	5 % **rmdir book**	7
edge 5	remove_tree book at_location [john,jml,forth2,usr,/]	8
	status _141289	9
	Could have been achieved with	10
edge 5'	3-command(rm,[[book,[john,jml,forth2,usr,/],[chapter1,chapter2],	11
	dir,x]],[(delete_node, [book, [john,jml,forth2,usr,/],	12
	[chapter1,chapter2],dir,x])], _141289, _142710)	13
	6 %	14

Extending the user's knowledge.
fig 6.25

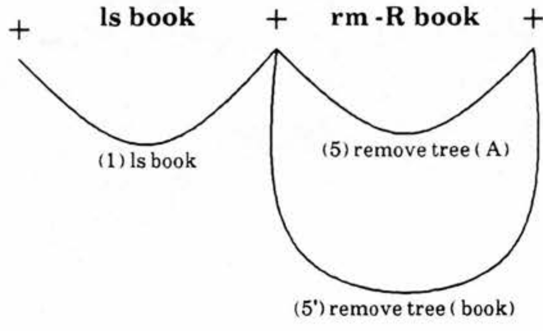
In this example, a shorter plan is detected for the task of deleting a filestore tree. The suggested command is "rm -R book" (lines 11 to 13), which would have had the same effect as clearing the directory with the two "rm" commands ("rm book/chapter1" and "rm book/chapter2"), then deleting the directory ("rmdir book"). This causes the advice to be triggered. Although fig.6.25 does not explicitly mention the command "rm -R", this is encoded in the remove_ tree goal and the effects of the interpretation of the command (lines 12 and 13). These show that the command is deleting a non-empty

directory containing the files "chapter1" and "chapter2" (line 13). The only valid command to achieve such a task is "rm -R book", which is the command that the UNIX Advisor suggests. The chart for this command sequence is shown in fig.6.26. From this chart, edge 5 is used in the replanning, when it is instantiated to the command to delete the tree "book" (the directory "book" and the files that it contains). This enables the "remove tree" goal to be completed in one command, giving the simplified chart shown in fig.6.27.



Simplified Chart for lack-of-knowledge (fig.6.25).

fig.6.26.



Simplified Chart for replanning due to lack-of-knowledge.

fig.6.27.

The recognition of cliches could also be used to trigger the generation of advice, though this is merely a special case of the above heuristic. A cliche rule is added to the grammar when the cliche is observed, and by definition, the cliche is longer than an alternative method of achieving the goal. Therefore, the next time that the cliche is observed, it will trigger the generation of advice.

It is also possible to ensure that the user is not overwhelmed by new information, by limiting the number of new rules that he is told. For example, if a user who did not know about the "mv" command or pattern matching, typed the following command sequence:

1%	**cp f1 d**	% copy file "f1" to directory "d".
2%	**cp f2 d**	% copy file "f2" to directory "d".
3%	**rm f1**	% remove file "f1"
4%	**rm f2**	% remove file "f2"

should he be told that he could have achieved this by typing "mv f? d"? This would probably be bad advice to give since it contains too many new concepts, and the user might not know why the command worked. This raises the question as to what the user should be told. Should he be told how to copy multiple files into a directory, how to use pattern matching, how to delete multiple files, how to move one file to another, how to move a file into a directory, how to move multiple files, ... etc? This choice is not easy, and some kind of tutoring heuristics would be needed such as:

> *"First, Expand the user's knowledge about the commands that he knows"*

Using this heuristic the user would be told about how to copy groups of files into a directory (without using pattern-matching). Then later, when this new rule had been learnt, the user can be told about new commands when a similar situation arises. Alternatively, the tutoring strategy might give advice about new commands first, the problem then being how to select between the different plan rules. A crude way to resolve this conflict would be to encode an estimate of the value to the user of different plan rules, and then advise about the rule which had a maximum value for that user. This has not been implemented in the current system, but it would be trivial to do this by tagging each grammar rule with an estimate of it s value.

6.6 Discussion.

By using a chart to maintain multiple interpretations of UNIX commands, it is possible to compare different plan hypotheses and use these as the basis for offering advice to the user (Lewis and Ross, 1988). The ability to *detect* when the user requires help is of fundamental importance to this approach. Other advice-giving systems either require the user to ask for help (for example, Genesereth, 1982), or wait for a failure to occur before searching for the mistake. Alternatively, the systems work in a tightly constrained environment which makes it easy to determine whether the user is forming a valid plan (for example, Carver, 1984). In the UNIX domain, the user is not always aware that he has made a mistake (for example, copying a directory with "cp d1 d2") and that the consequences of his beliefs deviate from what has actually occurred. Thus, an advice-giving system cannot rely on the user to ask for help at the most appropriate time.

The heuristic used to determine whether advice is needed is very simple, yet the example command sequences show that the heuristic is effective at detecting this point. Other heuristics could be used to trigger the generation of advice. For example, the fact that the user is currently following a plan that has never been used before might be indicative that he has a problem, and that the system should make a more detailed analysis of the chart to verify the situation. Another possible heuristic would be to take into account the number of interleaved plans or the size of the holes in the plan. Alternatively the plan grammar could be annotated with frequencies which reflect the relative usage of plans, and trigger advice on infrequently used plans. However, whichever heuristics are used to analyse the chart for triggering advice, they are independent of the chart parsing process and can therefore be easily modified.

Advice cannot be generated for invalid plans that the user might be following (for example, the user might think that to move a file, you remove the file then copy it). The assumption is made that users can develop plans but have problems with implementing them with the UNIX commands.

The identification of the user's problems is difficult because in the course of generating the possible plans, instantiations of the commands are made. Parts of these instantiations might not be necessary to identify the user's problems, so the same plan might be suggested several times with slight changes in the instantiations (eg. whether the file is executable or not). This leads to a protracted dialogue between the user and the system in an attempt to determine the user's intentions. This dialogue needs to be improved so that the user's intentions are determined more naturally.

In an attempt to improve the efficiency of the plan recognition process, hypotheses are discarded from the chart when they have been rejected by the user, or if no valid plans are found. This ensures that the plans do not continue to extend when they are known to be incorrect or invalid. The complexity of the chart is not dramatically affected by the addition of these hypotheses because there is only one hypothesis for each command variant type. These hypotheses retain their generality when they combine in the chart, and therefore represent a whole class of possible instantiations of the edge, rather than have each of these edges explicitly represented in the chart. Also the periodic deletion of invalid and incorrect edges from the chart ensures that these edges do not continue to combine when they are known not to reflect the user's intentions.

For this analysis, all UNIX commands have been assumed to have equal importance in a plan. Thus, having the "ls" command missing from a plan prevents the plan being recognised in the same manner as if the "rm -R" command were missing from that plan. At present we have no mechanism for representing the relative importance of commands in a plan, and any command can suggest a plan. In BELIEVER (Schmidt et al, 1978) and TRACK (Carberry, 1988), only certain actions are suggestive of goals. In the UNIX environment, commands such as "ls" should not initiate goals being hypothesised, since the command is not suggestive of any particular goals.

This chapter has addressed the problems of detecting when the user requires advice, and how to determine what the user's problems are in achieving his goals. However, little attempt has been made at deciding what to say to the user when a problem has been found, and how this should be articulated. At the moment, if it is thought that the user might have a problem, the system enters into a dialogue with the user about the potential problem. No assessment has been made as to whether it is preferable to enter into this dialogue, simply give a warning message to the user, or suppress the dialogue until later once more information about the user's intentions has been gained or a more appropriate time to discuss the problem is reached. Such an assessment would need to be based upon the seriousness of the problem - for example, if the user has spent some time in creating a series of files, then to allow him to delete these with a single command without asking for confirmation would seem to be inappropriate. A better course of action would be to indicate the potential problem before allowing the command to be executed (and actually intervening between the user and UNIX in this case). It might prove to be most beneficial to always offer the advice when it is realised that there is a potential problem.

An associated problem is how to tell the user about the problem. Do you just state the problem, or enter a tutorial dialogue with the user (posing hypothetical questions, perhaps) in an attempt to give the user a deeper understanding of how UNIX works? Without any model of tutoring or advice-giving, the UNIX Advisor cannot address this problem.

The advice generated by the UNIX Advisor has assumed that the user is not aware that he has a problem. However, there are times when the user wants help. At present UNIX offers help through an on-line manual, which can be accessed through the "man" command. This is unacceptable for two main reasons:

- The user has to know about the "man" command before he can get help.

- The help given is merely a print out of the contents of the manual, and is not adapted to the needs of the individual.

Therefore, both of these problems need to be tackled for the "help" to be improved. The first problem could be removed by having a dedicated "help" key or part of the screen dedicated to advise the user about how to obtain help. The second problem involves an analysis of the user's knowledge about UNIX and his present plans. Again the plan that the user is following could be hypothesised by analysing the chart, but the actual plan would have to be determined by entering into a dialogue with the user to establish what his current plan is. If this can be established, then the chart can be used to complete the plan, and provide help to the user about the plan rules that he does not know. Thus the chart is used to generate expectations about the commands that the user wishes to type. It is likely that a dialogue would also need to be entered into with the user to establish the questions that he wishes to ask, but the use of the contextual information contained in the plan should be useful in constraining this process.

6.7 Summary.

The chart-based approach to plan recognition uses the chart to maintain multiple partially-instantiated hypotheses linking the user's actions to plans. With different possible interpretations of the user's actions available, the chart is analysed to determine the most likely plan that the user is following and whether he requires help. Two simple heuristics are used to perform this analysis, which can be adapted to improve the system if necessary. A possible problem that the user has is identified by instantiating the plan and performing a dialogue with him to ensure that the plan is the one that he thinks that he is following. Dialogue with the user is kept to a minimum and only performed when a divergent plan is found which it is thought that he is following. This dialogue is consistent with the system remaining silent until it has something useful to contribute.

Alternative opportunities for giving advice can be detected by altering the heuristics that analyse the chart. This was illustrated by using a heuristic to detect errors and misconceptions, and a different heuristic to detect lack-of-knowledge. In the latter case the chart parser is used to re-plan goals and offer advice about how the user could improve his performance with the system, thereby extending his knowledge of UNIX. If the user explicitly asks for help then the chart could in principle be analysed to determine the plan that he is following, the next command that he should type, and the rules that he "knows". In this case the user model of commands could also be used to guide the help, providing the interpretations of misconceptions that the user has about commands and his extent of knowledge (although this has not been implemented at present).

Chapter 7

Discussion and Conclusions

7.0 Introduction.

The preceding chapters have described the development of the UNIX ADVISOR and its components. This chapter discusses methods of evaluating the system, the contribution and limitations of the User Modelling, Plan Recognition and Misconception Detection techniques used. Directions for further work are described, and finally conclusions are drawn to the work.

7.1 System Evaluation.

The Misconception Detection, Plan Recognition and User Modelling components of the UNIX Advisor were integrated into a system described in Chapter 6. The work has been based upon an analysis of logged data and experiments performed with users. However, no comprehensive evaluation of the system has been performed due to the problems associated with this task.

At present, the system has a poor user interface which is not intended to tutor or advise the user in any way. The purpose of the interface is for the qualitative evaluation of the system, making it possible to determine whether the program has detected expected misconceptions. However, this information is not presented in a form which can be easily interpreted by the user. Thus it would be impractical to evaluate the program directly with UNIX users. Indeed, it would be difficult to evaluate the system without an advice-generation module interfacing with the user. If such a module existed, it would be possible to evaluate the system by comparing the performance (in terms of speed, concepts learned, etc) of users of UNIX with the Advice-giving system, versus UNIX alone. However, such an approach would introduce many more variables: for example, the nature, frequency, and method of the advice, whether a misconception had been isolated and corrected, the validity of the user model and plan grammar, the method used for triggering advice, etc. Therefore, it would be difficult to decouple these effects from each other to determine their relative merits.

An alternative approach to the analysis is to run the UNIX Advisor with a user, and have a human to interpret the misconceptions and errors suggested by the program. However, the interpreter would have an extremely difficult role in which he would have to interpret the output of the program without making any additional input (which may have an effect upon the user's beliefs).

Logged data concerning the interactions that users have with the UNIX operating system could be analysed by experts to determine places where they believe that the user has exhibited a misconception. These command sequences could then be analysed using the UNIX Advisor to determine whether it also detects the misconception. The problem with this approach is that it is extremely difficult to analyse a command sequence without knowledge of what the user is attempting, since much of the contextual information is missing. This contextual information could be provided by placing the user in an experimental framework (similar to the experiments performed in the data gathering exercise in chapter 3). Comparing advice generated by the computer with that produced by a human expert would provide a measure of the capability of the UNIX ADVISOR. However, there is no guarantee that the expert's analysis is correct, thus a combination of the above methods of determining the UNIX ADVISOR's performance is needed.

The difficulty in evaluating the system effectively, has prevented a detailed analysis of its performance. However, some qualitative evaluations were performed in Chapter 6. The examples used command sequences which had been gained from the analysis of logged sessions and the experiments performed in the data-gathering phase.

7.2 Discussion of the system components.

7.2.1 User Modelling.

The user models employed in the system are the command model and the plan model. Both of these models are more than just overlay models because of their ability to model new behaviours. These models will now be examined individually.

7.2.1.1 Model of Commands.

The model of commands is an essential component of the advice-giving system because of its ability to model misconceptions as well as modelling a sub-set of knowledge. This is achieved through the selection of different parameters in the model to "explain" different behaviours. The system's misconception detection hinges upon this capability, and the types of misconception which can be detected are directly determined by the level of detail used in the model. The model, therefore, is capable of recognising misconceptions through the incorrect use of parameters in the model. Therefore, the representation used for the command model has direct impact upon the type of misconceptions that the resulting system can detect. It could be that this model should be refined, after evaluation, to reflect more accurately the problems that users experience. The model was determined both from an analysis of user's interactions with UNIX, and through a functional modelling of the commands themselves. In this way, an attempt was made to model the problems that users possess.

There are limitations with this model, since ideally each of the parameters within the model should be independent of the others. Therefore, changing the value of one parameter should not affect other parameters in any way. However, there are dependencies between these parameters which can make certain combinations of parameters potentially meaningless (for example, describing a file that contains links to other filestore nodes).

Another consideration is the model's extensibility both to other areas of UNIX, and to other domains. The UNIX filestore sub-domain is relatively easy to model because of the clear preconditions and effects of the commands. Other UNIX commands, for example the commands concerned with filtering data (grep, awk, diff, etc) are not so easy to model because the commands affect the *contents* as well as the existence of the files. Therefore, some way would need to be found to describe the transformation that has been performed upon the files - though it might be sufficient to describe the file's origins and a list of the operators performed on the file (for example, a file might be tagged as being "filtered"). However, such a scheme would not be able to detect the detailed problems that people experience with the use of the command. The plan grammar would also have to be changed to describe how the commands are related to each other.

The model is extensible to other domains for which a causal user model can be written. The performance of the resulting Advice-Giving System will rely upon how accurately the model represents what actually occurs, and whether the modification of parameters within the model generates realistic misconceptions.

The way that the model is altered to take account of the observed actions is important, since this will directly affect the advice that will be generated. In the description of the user model of commands, it was assumed that observation of a valid command was sufficient evidence to attribute the user with knowing that instance of the command. This need not necessarily be the case, since the user may have typed it by mistake, or he may have been told to type that particular command by a friend. The learning algorithm could be modified to take such occurrences into account. Instead, when an instance of a command is observed, it could increase the certainty in the model that the user knows about that instance. Only when the certainty increases over a threshold value will the model assume that the user knows about the command instance. The inclusion in the model of misconceptions, could be a useful method for trapping further occurrences of

the misconception. It could also be used in detecting when the user no longer believes in the misconception.

The user model of commands learns about the user's beliefs by generalising over instances of commands. However, there is no mechanism by which a user can be modelled as forgetting about the use of a command (or more generally an instance of a command). Such a mechanism would result in a specialisation of the model. The specialisation could be achieved in a simple way, by marking a command variant as being unknown. This could be done when the command variant has not been used for a certain period of time. However, within a command variant the specialisation could not be achieved easily. This should alter the parameters in the model such that the instance of the command is removed from the model, without affecting the commands that the user does know about. However, it is not clear how to decide which parameters to alter in order that such an effect is achieved.

7.2.1.2 Plan Grammar.

The plan grammar itself forms a part of the user model. This model is involved with describing the extent of the user's knowledge in planning goals and the particular cliche rules that he uses to achieve his task. With the grammar, the assumption is made that the user does not possess misconceptions about formulating plans, but that misconceptions are related to the details of the commands themselves. This assumption is made to simplify the plan recognition and misconception detection tasks, and the assumption appears to have the support of the analysis performed in Chapter 3. If such an assumption is not valid, then the introduction of buggy rules and associated cliche rules into the grammar would go some way to remedy this problem. However, a method of generating buggy rules has not been addressed in this work (the cliches which are detected are valid plans for achieving the task).

The use of a grammar to describe plans in the UNIX domain appears to be effective, though only a small part of the domain has been considered in this work. Extending the grammar to other areas would require considerable analysis to determine the tasks which users follow (but the grammar does not have to be complete because cliche rules can be generated).

The grammar can be extended to model the confidence that rules are "known" by the user. This information can be used to modify the content of the advice generated by the system. One of the main benefits of the grammar is that it need not be complete (though it must be correct). This relaxation makes the problem of writing the grammar feasible.

7.2.2 Plan Recognition.

The importance of this work to plan recognition is through the ability to:

- recognise plans without a goal being explicitly stated by the user.
- recognise plans which contain errors and misconceptions.
- recognise multiple, interleaved plans and suspended plans.
- operate in a "real" domain.
- recognise plans set in a background of "uninteresting", unplanned actions.

Each of these aspects has been attempted in isolation by other researchers, but the only other integrated approach is that of Kautz (Kautz, 1987). Chart parsing offers a different mechanism to Kautz's circumscription-based approach. Kautz shows how his method can operate in several different domains, including UNIX. Within these domains a complete representation of specialisation, decomposition and temporal constraints is needed. This is a severe drawback for the approach, not

required by the UNIX Advisor which uses an incomplete plan grammar and the recognition of cliches. Kautz also requires that all actions are purposeful. This is not required by the UNIX Advisor since actions can be treated as errors, misconceptions, or simply ignored as being uninteresting for the purpose of giving advice. Kautz's technique embraces non-linear plans, interleaved plans, shared subactions and temporal constraints in an elegant and natural way. However, non-linear plans are recognised in the UNIX Advisor through multiple grammar rules, and interleaved plans and shared subactions can be recognised by altering the fundamental rule. Although Kautz used a theory of circumscription as the basis for his technique, he still uses heuristics to reduce the number of possible plans that are recognised. He identifies the following heuristic:

> When several actions are observed, it is often a good heuristic to assume that the actions are all part of the same top level act, rather than each being a step of an independent act. (Kautz, 1986 P.33)

A similar assumption is made in the UNIX Advisor's heuristics for analysing the chart in which it is assumed that the user's plan is the plan containing most actions.

As with the UNIX Advisor, Kautz's program enables the plan recognition to occur incrementally, building upon work done previously. But unlike the UNIX Advisor, the actions can be observed in any order. Kautz does not consider errors or misconceptions concerning the observed actions or recognised plans, nor does he present a possible method for detecting errors or misconceptions. He assumes that all observed actions are correct and intended. One possible method of extending the method to cope with errors and misconceptions at the level of actions is to tag alternative interpretations of the actions. The plan recognition process could then recognise different possible plans accordingly and heuristics could be used to analyse the resultant plans. These could then be used to identify errors or misconceptions. If the misconceptions were about "the form of the plan hierarchy", then the plan hierarchy would need to contain such generalisations and specialisations to recognise such plans.

Circumscription enables the runtime effort of plan recognition to be minimised, since the relationships between actions are determined before the actions are observed. Therefore, when the user's actions are observed these actions can be incorporated directly into the circumscribed hierarchy. However, this can only be achieved if the hierarchy is complete and consistent. If the hierarchy needs to be modified (for example, in a similar way to adding cliches to the grammar in the UNIX Advisor) then the circumscription work needs to be repeated to incorporate this new information. This extra work would mean that Kautz's approach no longer provides efficiency benefits over chart parsing.

Although Kautz's circumscription-based approach is powerful and generalised it suffers from requiring a complete and correct plan hierarchy, which is not required by the UNIX Advisor. Kautz's program still requires heuristics to simplify the plan recognition process, and there is as yet no ability to deal with errors or misconceptions. Indeed, it is not clear how misconceptions and errors can be incorporated into his framework. Without the ability to accomodate misconceptions and errors the advice that can be generated will be severely limited.

The use of chart parsing for plan recognition has also been researched by Woodroffe. He takes ideas from planning and applies them to the plan recognition task, implementing these in the FITS3 program (Woodroffe, 1990). FITS3 uses a simple hypothetical domain, with the intention of applying the ideas to the UNIX domain. Like the UNIX Advisor it uses STRIPS-like operators to describe observable actions; these consisting of preconditions, actions and postconditions. However, FITS3 differs from the UNIX Advisor by allowing rules which describe non-linear plans (plans where the ordering of actions can be varied). This is achieved by specifying all the ordering relationships between the actions within a rule, and it enables several valid sequences of actions to be recognised with a single rule. Although the UNIX Advisor does not allow the sequencing of constituents of a plan to be altered, the same effect is achieved with multiple

rules in the grammar. Such rules can be added by the UNIX Advisor as cliches, whereas FITS3 requires a complete grammar.

FITS3 incorporates condition ranges into the grammar rules. These condition ranges must be satisfied for the actions to combine legitimately into a plan. An example plan rule used by FITS3 (Woodroffe, 1990) is given in fig.7.1. The range information in this plan rule shows the condition for moving from home to the university via an intermediate location (the shop). The condition states that the location after moving from home to the shop must be "at the shop" before a move can then be made to the university.

```
plan( move(home, university),                                              -plan name
        [move( home, shop), move( shop, university)],                      - expansion
        [move( home, shop) < move( shop, university)],                     - ordering
        [range( at( shop)), move( home, shop), move( shop, university)]    - range
).
```

fig 7.1.

An example plan rule used in FITS3 (Woodroffe, 1990).

The UNIX Advisor also uses constraints that must be satisfied for the plan to be recognised (section 4.2.2). For example, there is a constraint that the source and destination files for the "cp" command are different. The condition ranges and non-linear plan recognition in FITS3 enables interleaved plans and shared plan steps to be recognised. This is achieved in the UNIX Advisor by modifying the fundamental rule. However, without a direct comparison it is unclear as to the performance benefits of the two schemes.

Woodroffe proposes (Woodroffe, 1990) to detect errors by tagging edges in the chart with endorsements. These endorsements provide "clues" that an error has occurred (but the endorsement itself may be insufficient

evidence to report the error). He proposes to use four types of error endorsements:

- Unsatisfied prerequisites. This is when the conditions necessary for the observed action are not true in the world model.
- Violated range. This is when the conditions necessary for partial plans to combine are violated.
- Phantom. This is when an action has not changed the world model.
- Unnecessary. This is when there is no need to add a partial plan because its effects are already true in the world model.

As the parsing proceeds, FITS3 combines these endorsements. Woodroffe suggests that error heuristics could be applied to the chart to determine whether an error has occurred. As an example, Woodroffe suggests the following heuristic:

If an action has an unsatisfied prerequisite
and an earlier action was a phantom
then it is possible that the student believes that the action satisfies these prerequisites.
(Woodroffe, 1990)

Such a scheme of endorsements could also be used to detect misconceptions that a student possesses. However the approach taken in the UNIX Advisor, with multiple partially instantiated plans, gives a much richer description of possible plans that the user might be following. Therefore, the UNIX Advisor should be able to detect errors and misconceptions more accurately than FITS3 and provide more evidence for the advice generator. Also Woodroffe does not explain how FITS3 would initiate advice, although this could be achieved by using heuristics in a similar way to the UNIX Advisor.

The technique of chart parsing is extremely powerful because the chart contains a representation of all possible plans and plan fragments which conform to the grammar. In addition, the plan recognition process can be performed in an incremental manner, which allows previous work to be used to generate new plans. TRACK (Carberry, 1988) uses heuristics to isolate goals and perform plan recognition. However, unlike the chart parsing approach TRACK cannot recognise multiple, interleaved and suspended plans, or plan cliches.

The main problems involved with chart parsing for plan recognition are:

- The development of a grammar which describes the user's activities in sufficient detail. This task would, in practice, require a major analysis and coding effort. However, due to the ability to recognise plan cliches, in principle the grammar need not be complete.
- limiting the generation of edges in the chart to prevent the combinatorial explosion.

The first problem; that of developing the grammar is a problem common to all plan recognition systems and there does not appear to be any feasible alternative to this approach at present.

The problem of limiting the generation of edges is a major limitation of the system at present since no such action is performed. The analysis in Chapter 5 showed how the number of edges in the chart grew exponentially as commands were incorporated. Clearly, this is unacceptable and must be prevented in some way. There are some simple methods in which this may be achieved:

- Forget commands and plans started more than a certain number of commands ago.

- Purge the chart when a top level plan has been recognised and completed by the user. The problem is knowing when the user is following a certain plan without asking him. But it may be reasonable to assume that the probability that the user is following that plan increases with the length of the observed plan.
- Plans which have been rejected (for example, in consultation with the user) can be removed from the chart, and any plans which depend upon this plan, and support just this plan can be removed also.
- Edges which represent incomplete plans, and which have been suspended for a certain number of commands can be assumed to be invalid and deleted.
- Determine a rule to stop the parsing process prematurely.
- Determine a strategy for developing the plans in the chart "best first"

The best method for controlling the number of edges generated would have to be evaluated through experimentation, and it would be reasonable to expect a tradeoff between accuracy and size of the chart.

The techniques appear to be extensible to other domains, provided that a plan hierarchy can be determined for those domains. Extension to other parts of UNIX would be difficult because of the apparent lack of structure in the use of other UNIX commands - however any plan recognition system would experience similar problems.

7.2.3 Misconception Detection.

The ability to detect automatically misconceptions and errors has been achieved by very few existing programs. Those that do achieve this, have an explicit representation of the user's goal (for example: Carver et al, 1984; Genesereth, 1982). An explanation-based approach to recognising and

responding to user's misconceptions is used by Quillici et al (Quillici et al, 1988). Their system is based on the UNIX domain, but assumes a question and answer dialogue between the user and the automated advisor. This dialogue is used to determine the user's goal. The advisor determines whether it shares the user's beliefs, and if it does not it determines why it does not share these beliefs (for example, the belief in incorrect preconditions). The UNIX Advisor does not have the user's intended goal available to it. Instead, it relies upon the development of multiple plan hypotheses as a method of assessing the possibility that the user possesses a misconception or has made an error. Simple heuristics are used to trigger the search for instantiations of these misconceptions. By keeping generalised edges in the chart, large numbers of instantiated edges are avoided. However, these edges must be instantiated before they are used for generating advice.

The misconception detection is based upon the assumptions that:

- Misconceptions are concerned with the commands themselves, not with the plans.
- Misconceptions can be modelled with the command model.

These assumptions, whilst not totally accurate, appear to be a good approximation for the system, enabling the recognition of a wide variety of errors and misconceptions The extensibility of the misconception detection technique is dependent upon the generality of its component parts (ie. the plan grammar and command model). If both of these can be determined for another domain, then the system should offer the same capability as in the UNIX filestore domain.

One particular problem with the detection of misconceptions, is identifying the user's particular misconception through the instantiation of the plan containing the misconception. The instantiations have to be verified by the user to validate their belief in the misconception. Typically, the partially instantiated misconception can give rise to many

instantiations - each of which must be validated in turn. This results in long question and answer sessions (shown, for example, in fig.6.22). An improved method of presenting these misconceptions is needed so that the user can discard large blocks of misconceptions (rather than each instantiation being posed as an individual question). For example, the user could be asked about the generalised form of the misconception, and perhaps asked to fill in his beliefs about the command. This would instantiate the command's parameters, which could then be used to generate the advice.

7.2.4 Integrating Plan Recognition, User Modelling and Advice Generation.

One of the major contributions of this work is the integration of plan recognition with user modelling techniques, which enables the detection of misconceptions and errors. This is an important step forward since it provides Advice Giving Systems with the ability to recognise misconceptions without relying upon the user to provide a description of his task. Instead, the recognition of the user's beliefs are determined from his actions.

The ability to develop multiple plans in the chart, each containing different hypotheses, provides a powerful mechanism for detecting users' mistakes and problems. This ability is particularly important for advice-giving systems, since the advice can be triggered by the computer rather than waiting for the user to request help. The nature of the plan recognition and advice-generation mechanisms, makes the process transparent to the user. He does not have to supply information specifically for these tasks. Only when the program detects a point where it could possibly advise, does it interact with the user. Indeed the method of this interaction should depend upon the tutoring strategy used. It may be that the strategy employed does not always suggest interaction at these moments. Although the UNIX Advisor does not address the problems of

when to offer advice, nor how to offer that advice, it does produce information upon which these decisions can be based.

There are many problems associated with automated advice generation. Two such problems are:

- Determining the best advice to give an individual.
- How to alter the model of the user once a misconception has been detected and advice given to correct this.

One possible solution to the latter problem, is to assume that the user still possesses the misconception until such a time that he exhibits behaviour to support the belief that he no longer possesses that misconception. At this stage the misconception could be retracted from the user model. More generally, the user model should be capable of the user forgetting as well as learning. This would require a reason maintenance system to support the user model.

The problem of articulating advice to users with different knowledge has been addressed in the TAILOR program (Paris, 1988). This program uses two different discourse strategies to generate advice, one strategy for novices and one for experts. Such a treatment could be used in addition to user models which describe the user's knowledge. An analysis was performed of the advice generated by humans (McCoy, 1988). She found that people tend to perform three steps in their advice: They refute the user's incorrect information, make a statement of correct information, and justify the correct information. The nature of the user model used in the UNIX Advisor would make generating such advice possible.

7.3 Further Work.

There are several ways in which this work can be extended. These have been divided into two groups - short term and long term goals.

7.3.1. Short Term.

Misconceptions are not represented explicitly in the user model. This could be rectified easily, and would provide a mechanism for checking commands against known misconceptions that the user possesses. At present, misconceptions are represented as the difference between an over-generalised user model of commands and the corresponding UNIX model of commands.

An improved user interface to the UNIX Advisor needs to be constructed. The primary importance of the interface, at this stage, is to enable the computer to determine the user's beliefs through an improved interaction with the user. This could involve asking the user more general questions which arise through a partial instantiation of a plan, rather than fully instantiating the plan before communicating with the user. This approach should reduce the amount of dialogue necessary between the user and computer, and the amount of backtracking performed by the program. Such an interface could discard whole families of solutions rather than stepping through individual solutions.

A grammar needs to be written which accurately describes the activities that users perform in the UNIX environment. This grammar need only deal with a sub-domain of UNIX, but it should be representative of user's performance within that sub-domain so that the system can be evaluated.

The various parts of the UNIX Advisor are not all fully integrated. The cliche detection, plan fragment recognition, and updating of the user

model have been developed in isolation from each other. These three aspects of the system should be collected together within a single system and tested.

The UNIX Advisor needs to be evaluated with users (and possibly an interpreter) to determine whether the system can accurately determine users' misconceptions and errors on time.

7.3.2. Long Term.

An advice strategy should be implemented and used to interface with the user. This would take into account when and how to advise the user. This could be as simple as "Advise the user when you think that you have found a misconception" and "Tell the user how to correctly perform the goal". The development of such a strategy would involve the use of considerable experimentation to determine the best strategy to adopt.

The grammar and user model could be extended to account for more UNIX commands and features such as pattern matching (a plentiful source of misconceptions in UNIX). The learning algorithm for the user model should be altered so that it does not modify the model immediately that a command is observed, but awaits for evidence to confirm that the user understands that use of the command.

The learning capability of the grammar should be improved so that cliche rules are not formed which are over-generalisations (a problem with the present cliche recognition). The grammar should also be able to make specialisations, which would represent the user forgetting about aspects of the system. Such a model would need to take into account the certainty of the user knowing about an aspect of a command.

Methods for deciding which edges can be deleted from the chart are needed so that the combinatorial explosion is avoided. Also, a mechanisms for controlling the order of the generation of edges, and when to stop parsing

could be used to have the same effect. For example, such a strategy might look for touching and fragmented edges first, and only continue to look for other edges if no useful plan is found.

Before in-field evaluations of the system can be attempted, a mechanism is required for converting the output of the UNIX Advisor into a meaningful form for users. The form that this mechanism would take is not clear. For example, it could consist of a menu-driven interface, or straight text. Either way, the output from the program must be processed to obtain the desired form of the interface.

In-field evaluation of the system is needed to provide quantitative and qualitative analyses of the benefits of such an advice-giving system. Finally, the techniques discussed in this thesis should be applied to other domains to determine the portability of the techniques.

7.4 Conclusions.

This work has developed a mechanism for detecting plans that provide accounts of users' actions in a command-driven environment. The mechanism uses chart parsing, which is capable of detecting fragmented, interleaved and suspended plans. The recognition of such plans cannot be achieved by POISE (Carver et al, 1984) or the MACSYMA ADVISOR (Genesereth, 1982). The recognition process does not require a description of the user's intended goal to be supplied, as in the MACSYMA ADVISOR. Instead, the UNIX Advisor infers the goals from observed actions. A plan grammar is used to describe possible plans that a user might be following. Kautz's approach (Kautz, 1986) requires that the grammar be both complete and correct. However, the grammar used in the chart parsing approach need not be complete, because plan cliches (alternative, correct methods for achieving the same goal) can be recognised and automatically added to the chart. Since the chart contains all possible complete and partial plans, the parsing process can be performed incrementally as commands are observed. It also allows the plan recognition process to be suspended and resumed at a later date. Multiple interpretations of commands can be included in the chart. These combine independently to generate partial and complete plans. This allows hypotheses to be made and allowed to combine freely in the chart, enabling misconceptions to be included. Kautz's approach, and the approach used in POISE cannot include misconceptions in the recognised plans. Advice is triggered by analysing the chart with heuristics, to determine the relative merits of the competing plans. This enables the generation of the advice to be initiated automatically, rather than the user needing to make a request for advice.

The advice generation process uses a user model of the functionality of UNIX commands. This model is developed according to the observation of the user's actions. Thus, any misconceptions that are proposed by the plan recognition process must be consistent with this model. In this way, the advice generated by such a system can be based upon a specific model of the user, and take in to account the context in which the command was issued.

References

Aho A.V., Ullman J.D., 1972. *The Theory of Parsing, Translation and Compiling. Vol 1.* Publised by Prentice-Hall.

Allen J.F., Perrault C.R., 1980. Analysing Intention in Dialogue. In *Artificial Intelligence 15(3)* PP.143-178.

Bannon L., Cypher A., Greenspan S., Monty M.L., 1983. Evaluation and Analysis of User's Activity Organisation. In *proceedings of the Conference on Human Factors in Computer Systems.*

Boggild V., 1986. Graphical Specification of Keyword Queries to Videotex Databases. *University of Waterloo Report CS-86-63.*

Burton R.R., 1982. Diagnosing Bugs in a Simple Procedural Skill. In *Intelligent Tutoring Systems* (Editors: Sleeman D. and Brown J.S.), published by Academic Press PP.157-183.

Burton R.R., Brown J.S., 1982. An Investigation of Computer Coaching for Informal Learning Activities. In *Intelligent Tutoring Systems* (Editors: Sleeman D. and Brown J.S.), published by Academic Press PP.79-98.

Buxton W., 1986. There's more to Interaction than Meets the Eye: Some Issues in Manual Input. In *User Centred System Design* (Editors: Draper S.W., Norman D.A.) published by Lawrence Erlbaum Associates. PP.319-337.

Carberry S., 1988. Modeling the User's Plans and Goals. In *Computational Linguistics 14(3).* PP.23-37.

Carr B., Goldstein I.P., 1977. Overlays: a theory of modeling for computer-aided instruction. *AI Lab Memo 406.* Massachusetts Institute of Technology.

Carver N.F., Lesser V.R., McCue D.L., 1984. Focussing in Plan Recognition. In *proceedings of the National Conference on Artificial Intelligence.* PP.42-48.

Charniak E., McDermott D., 1985. *Introduction to Artificial Intelligence.* Published by Adison-Wesley PP.557-567.

Clancey W.,J., Letsinger R., 1981. NEOMYCIN: Reconfiguring a Rule-Based Expert System for Application to Teaching. In *proceedings of the Seventh International Joint Conference on Artificial Intelligence.* PP.829-836.

Cohen P.R., 1978. On Knowing what to say: Planning Speech Acts. *Department of Computer Science Report TR118* University of Toronto.

Cooper, M., 1988. Interfaces that Adapt to the User. In *Artificial Intelligence and Human Learning* (Editor J.Self) Published by Chapman and Hall Computing. PP. 300 - 309.

Davenport C., Weir G., 1986. PRIAM: Plan Recogniser for Intelligent Advice and Monitoring. *Report No. AMU8602/01S* Heriot-Watt / Strathclyde MMI Unit.

Doyle J., 1979. A Truth Maintenance System. *Artificial Intelligence 12* PP.231-272.

Draper S., 1983. Developing Monitoring Methods for User Centered Design . *Papers from the UCSD Human Machine Interface Project.* Institute for Cognitive Science, University of California, San Diego. Working Documents.

Draper S.W., Norman D.A., 1986. *User Centred System Design.* Published by Lawrence Erlbaum Associates.

Earley J., 1970. An efficient context-free parsing algorithm. In *Communications of the ACM 6(8).* PP.451-455.

Elsom-Cook M., du Boulay J.B.H., 1986. A Pascal Program Checker. In *proceedings of the Research Workshop on Intelligent Computer Aided Instruction*, Windermere, UK.

Fikes R.E., Nilsson N.J., 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Artificial Intelligence 2.* PP.189-209.

Fikes R.E., Hart P.E., Nilsson N.J., 1972. Learning and Executing Generalized Robot Plans. In *Artificial Intelligence 3*. PP.251-288.

Finin T.W., 1982. Help and Advice in Task Oriented Systems. *Computer Science Department Report MS-CIS-1982-22* University of Pennsylvania.

Fountain A.J., Norman M.A., 1985. Modelling User Behaviour with Formal Grammar . In *People and Computers: Designing the Interface* (Editors: Johnson P. and Cook S.) published by Cambridge University Press.

Genesereth M.R., 1979. The Role of Plans in Automated Consultation. In *proceedings of the Sixth International Joint Conference on Artificial Intelligence*. PP.311-319.

Genesereth M.R., 1982. The Role of Plans in Intelligent Teaching Systems. In *Intelligent Tutoring Systems* (Editors: Sleeman D. and Brown J.S.) published by Academic Press. PP.137-155.

Goldstein, I. 1982. The Genetic Graph: A Representation for the Evolution of Procedural Knowledge. In *Intelligent Tutoring Systems* (Editors: Sleeman D. and Brown J.S.) published by Academic Press. PP.51-77.

Hanson J.H., Kraut R.E., Farber J.M., 1984. Interface Design and Multivariate Analysis of UNIX Command Use. In *ACM Transactions on Office Automation Systems 2(1)*. PP. 42-57.

Hartley, R., Pilkington, R., 1987. Software Tools for Supporting Learning. In *Artificial Intelligence Tools in Education* (Editors: Ercoli P. and Lewis R.) Published Elsevier Science. PP. 39-65.

Hayes-Roth B., 1985. Blackboard Architecture for Control. In *Artificial Intelligence 26*. PP.251-321.

Hollnagel, E., 1983. What do we know about man-machine systems?. In *International Journal of Man-Machine Studies 18(2)*. PP135-144.

Jackson P., Lefrere P., 1984. On the application of rule-based techniques to the design of advice-giving systems. In *International Journal of Man-Machine Studies 20(1)*. PP.63-86.

Jerrams-Smith J., 1986. The Application of Expert Systems to the Design of Human-Computer Interfaces. *PhD Thesis* University of Birmingham, UK.

Johnson W.L., Soloway E., 1985. PROUST: An Automatic Debugger for Pascal Programs. In *BYTE magazine 10(4)*. PP.179-190.

Jones J., 1985. A Review of Some User-Modelling Techniques. *Department of Artificial Intelligence Working Paper 177*. University of Edinburgh.

Kautz H.A., 1985 Toward a Theory of Plan Recognition. *Computer Science Department Report TR162* University of Rochester.

Kautz H.A., Allen J.F., 1986. Generalised Plan Recognition. In *proceedings of the National Conference on Artificial Intelligence*. PP.32-37.

Kautz H.A., 1987. A Formal Theory of Plan Recognition. *Computer Science Department Report TR215* University of Rochester.

Kernighan B.W., Pike R., 1984. *The UNIX Programming Environment.* Published by Prentice-Hall.

Lewis J.M., Ross P.M., 1988. Detecting and Modelling User's Beliefs about UNIX. In *proceedings of Institution of Electrical Engineers coloquium on Intelligent Tutoring Systems*. IEE Digest No. 1988/69

Litman D., Allen J., 1984. A Plan Recognition Model for Clarification Subdialogues. *Report TR141* University of Rochester.

London B., Clancey W.J., 1982. Plan Recognition Strategies in Student Modelling: Prediction and Description. In *proceedings of the National Conference on Artificial Intelligence*. PP.335-338.

Mason M.V., 1986. Adaptive Command Prompting in an On-Line Documentation System. In *International Journal of Man-Machine Studies 25*. PP.33-51.

Matz M., 1982. Towards a Process Model for High School Algebra Errors. In *Intelligent Tutoring Systems* (Editors: Sleeman D. and Brown J.S.), published by Academic Press PP.25-50.

McCoy K.F., 1988. Reasoning on a Highlighted User Model to Respond to Misconceptions. In *Computational Linguistics 14(3)*. PP.52-63.

Norman D.A., 1981. The Trouble with UNIX. In *Datamation 27(11)*. PP.139-150.

Paris C.L., 1988. Tailoring Object Descriptions to a User's Level of Expertise. In *Computational Linguistics 14(3)*. PP.64-78.

Quilici A., Dyer M., Flowers M., 1986. AQUA: An Intelligent UNIX Advisor. In *proceedings of the Seventh European Conference of Artificial Intelligence. Vol.2*. PP.33-38.

Quilici A., Dyer M., Flowers M., 1988. Recognising and Responding to Plan-Oriented Misconceptions. In *Computational Linguistics 14(3)*. PP.38-51.

Raymond D.R., 1986. A Survey of Research in Computer-Based Menus. *Computer Science Department Report CS-86-61* University of Waterloo.

Rich E., 1979. User Modelling via Stereotypes. In *Cognitive Science 3(1)*. PP.329-354.

Ross P.M., Jones J., Millington M., 1985. User-Modelling in Command-Driven Systems. *Department of Artificial Intelligence Research Paper 264* University of Edinburgh.

Ross P.M., Lewis J.M., 1988. Chart Parsing for Plan Recognition. In *Artificial Intelligence Tools in Education*. (Editors: Ercoli P., Lewis R.) Published by Elsevier Science.

Ross P.M., 1989. Advanced Prolog - Techniques and Examples. Published by Addison-Wesley.

Schank R.C., 1975. SAM - A Story Understander. *Research Report No.43* Yale University, Department of Computing.

Schmidt C.F., Sridharan M.S., Goodson J.L., 1976. Recognising Plans and Summarising Actions. In *proceedings of the Conference on Artificial Intelligence and Simulation of Behaviour.* PP.291-306.

Schmidt C.F., Sridharan M.S., Goodson J.L., 1978. The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence. In *Artificial Intelligence 11* PP.45-83.

Self, J.A., 1974. Student Models in Computer-aided Instruction. In *International Journal of Man-Machine Studies 6.* PP.261-276.

Self, J.A., 1988a. Student Models: What are they? In *Artificial Intelligence Tools in Education* (Editors: Ercoli P., Lewis R.) published by Elsevier Science. PP.73-86

Self, J.A., 1988b. Bypassing the Intractable Problem of Student Modelling. In *proceedings of Conference on Intelligent Tutoring Systems* ITS-88 Montreal, Canada. PP.18-24.

Shrager J., Finin T., 1982. An Expert System that Volunteers Advice. In *proceedings of the National Conference on Artificial Intelligence.* PP.339-340.

Sidner C.L., 1985. Plan Parsing for Intended Response Recognition in Discourse. In *Computational Intelligence 1.* PP.1-10.

Sleeman D., 1984. An Attempt to Understand Students' Understanding of Basic Algebra. In *Cognitive Science 8.* PP.387-412.

Spark-Jones K., 1984. User Models and Expert Systems. *Computer Laboratory Technical Report No.61* University of Cambridge.

Sridharan M.S., Schmidt C.F., 1977. Knowledge-Directed Inference in BELIEVER. *Report CBM-TR-75* Rutgers University.

Steel S., de Roeck A. 1987. Bidirectional Chart Parsing. In *Advances in Artificial Intelligence* (Editors: Hallam J., Mellish C.) published by John Wiley and Sons Ltd. PP.223-235.

Stevens A., Collins A., Goldin S.E., 1982. Misconceptions in Student's Understanding. In *Intelligent Tutoring Systems* (Editors: Sleeman D. and Brown J.S.), published by Academic Press PP.13-24.

Suchman L., 1987. *Plans and Situated Actions. The problem of human-machine communication.* Published by Cambridge University Press.

Sullivan M., Cohen P.R., 1985. An Endorsement-Based Plan Recognition Program. In *proceedings of the Nineth International Joint Conference on Artificial Intelligence* PP.475-479.

Thompson H., 1981 . Chart Parsing and Rule Schema in GPSG. *Department of Artificial Intelligence Research Paper 165* University of Edinburgh.

University of California, 1983. *UNIX programmer's manual. 4.2 Berkeley Software Distribution Virtual VAX-11 Version.* Computer Science Division, Department of Electrical Engineering and Computer Science. University of California, Berkeley.

Van Lehn K., 1987. Learning One Subprocedure per Lesson. In *Artificial Intelligence 31(1)*. PP.1-40.

Wenger E., 1987. *Artificial Intelligence and Tutoring Systems.* Published by Morgan Kaufmann Publications Inc.

Wilensky R., 1983. *Planning and Understanding. A Computational Approach to Human Reasoning*. Published by Addison Wesley.

Wilensky R., 1984. Talking to UNIX in English: An Overview of an On-Line UNIX Consultant. In *Artificial Intelligence 5(1)* PP.29-39.

Wilkins D.C., Clancey W.J., Buchanan B.G., 1986. Overview of the Odysseus Learning Apprentice. In *Machine learning: A guide to Current Research* (Editors: Mitchel T.M., Carbonell J.G., and Michalski R.S.) published by Kluwer Academic Publishers. PP.369-373.

Winograd T., 1983. *Language as a Cognitive Process: Volume I Syntax.* Published by Adison-Wesley PP.116-127.

Woodroffe M.R., 1990. *Plan recognition and error analysis in the UNIX domain*. PhD Thesis. University of Essex, Department of Computer Science.

Appendices

Appendix I

Example Logging Script

```
# ###### Start of log: Mon Dec 9 12:24:02 1985
# cd ~jackson/msc
cd ~jackson/msc
## 0
# ls
ls
## 0
# top
top
## 1
# top X1
top X1
## 1
# cp -r X1 ~fred/rep.d
cp -r X1 ~fred/rep.d
## 0
# cd ~fred/rep.d
cd ~fred/rep.d
## 0
# ls
ls
## 0
# lpr PR
lpr PR
## 0
# lpr X1
lpr X1
## 0
```

In the above script:

- The user has a login of "fred".
- The session starts with a banner giving the date and time of the start of the logging session.
- Each command consists of the following three lines:

  1. The command line as issued (prefixed by a "#").

  2. The command line as expanded by the shell.

  3. The Status code (prefixed by a "##"): 0 indicates that the command has been performed without an error, other codes are given if an error is encountered, which depend upon the command.

Appendix II

Text of Questions for Data Gathering Experiment

Problem 1

You have been writing a book, and have written a number of chapters and appendices. At the moment your file area looks like:

```
              |-appendix1
              |-appendix2
              |-appendix3
              |
              |-book-------------|-chapter4
|-john-       |-chapter1
              |-chapter2
              |-chapter3
              |
              |-mine------------ |-mbox
```

Now you want to tidy your file area, and have decided to collect all of the chapters (chapter1 to chapter4) into a new directory "chapters", the appendices (appendix1 to appendix3) into a new directory "appendices", and delete the directory "book". This will give a file tree which looks like:

```
                                             |-appendix1
              |-appendices--------------     |-appendix2
              |                              |-appendix3
              |
              |                              |-chapter1
|-john-       |-chapters------------------   |-chapter2
              |                              |-chapter3
              |                              |-chapter4
              |
              |-mine----------------------- |-mbox
```

Please make these changes to the file store.

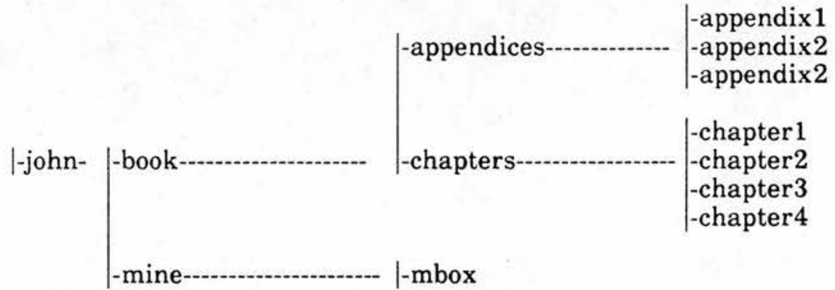
Your home directory is "john", which is also your current working directory.

Please wait for a "%" prompt before continuing.
When you have completed the problem please press " ↑ Z".

Problem 2.

The files remain as you left them at the end of problem 1.

Some time later you decide to collect together the appendices and chapters in a new directory "book", making the File Tree look like:



Please make these changes to the file store.

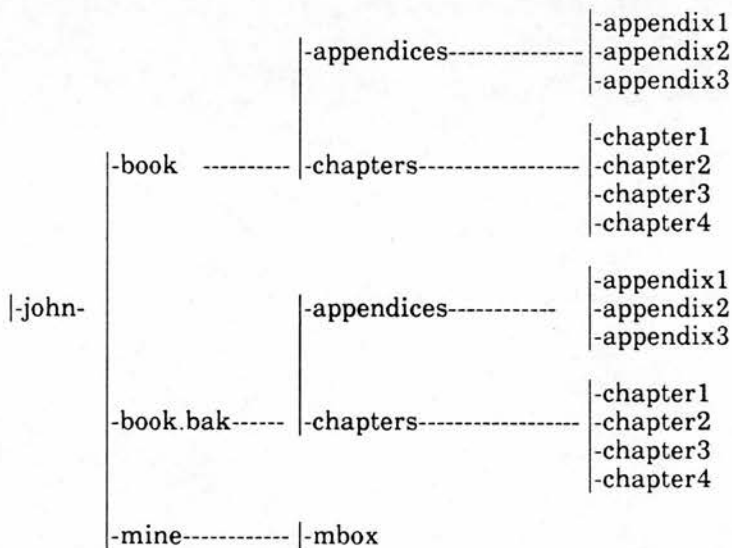
Your current working directory is "john".

Please wait for a "%" prompt before continuing.
When you have completed the problem please press " ↑ Z".

Problem 3.

The files remain as you left them at the end of problem 2.

You are worried about deleting the files by mistake and decide to make a back-up copy of the book. This will result in the file tree looking like:



Please make these changes to the file store.

Your current working directory is "john".

Please wait for a "%" prompt before continuing.
When you have completed the problem please press " ↑ Z".

Appendix III

Part of a Typical Data Script

The following text is a script of keys typed by the user during the data gathering experiment, and the response provided by UNIX. The file tree is generated automatically and presents, pictorially, the state of the user's filestore. This record is generated before each of the three stages of the experiment.

Certain character sequences have specific meanings, these are:

- **↑[↑H↑H ↑H↑H** is the "Escape" automatic name completion character sequence.
- **↑G** is the bell character (this occurs when automatic name completion fails).
- **↑H ↑H** is the delete character sequence.
- Thus the character sequence:

  mv appendix1 appe↑[↑H↑H ↑H↑Hndi↑G↑[↑H↑H ↑H↑H↑Gces/append ↑[↑H↑H ↑H↑H↑Gix1↑M

  will be translated to:

  mv appendix1 appendices/appendix1

  before being used by the *mv* command.

START of Experiment

```
##############################################

                  ##################
Thu Jan 23 12:07:06 BST 1986


Directory structure and contents of /tmp/john
(excluding entries that begin with '.')

          |-appendix1
          |-appendix2
          |-appendix3
          |
|-john-   |-book------|-chapter4
          |-chapter1
          |-chapter2
          |-chapter3
          |
          |-mine------|-mbox

Script started on Thu Jan 23 12:08:11 1986

% ls ↑M
appendix1 appendix3 chapter1 chapter3 ↑M
appendix2 book    chapter2 mine ↑M
% mkdir appendices ↑M
% ls -F ↑M
appendices/ appendix2  book/    chapter2   mine/ ↑M
appendix1   appendix3  chapter1  chapter3 ↑M
% mv appendix1 appe ↑[ ↑H ↑H  ↑H ↑Hndi ↑G ↑[ ↑H ↑H
↑H ↑H ↑Gces/append ↑[ ↑H ↑H  ↑H
                  ↑H ↑Gix1 ↑M
% mv appendix2 appendices/appendix2 ↑M
% mv appendix3 appendices/appendix3 ↑M
% ls -F ↑M
appendices/ book/    chapter1   chapter2   chapter3   mine/ ↑M
% ls appendices ↑M
appendix1 appendix2 appendix3 ↑M
% mv ca ↑H  ↑Hhapter1 book/chapter1 ↑M
% mv chapter2 book/ch ↑[ ↑H ↑H  ↑H ↑Hapter ↑G2 ↑M
% mv ch ↑[ ↑H ↑H  ↑H ↑Hapter33 b ↑[ ↑H ↑H  ↑H ↑Hook/ch ↑[ ↑H ↑H
↑H ↑Hapter ↑G3
                  ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H
↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H
                  ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H
↑H ↑H  ↑H ↑H  ↑H ↑H  ↑H ↑[ ↑H
                  ↑H  ↑H ↑Hapter3 b ↑[ ↑H ↑H  ↑H ↑Hook/ch ↑[ ↑H ↑H
↑H ↑Hapter ↑G3 ↑M
% ls -F ↑M
appendices/ book/    mine/ ↑M
% ls -F book ↑M
chapter1 chapter2 chapter3 chapter4 ↑M
% ls mine ↑M
mbox ↑M
```

```
% mkdir chapters ↑M
% lsa ↑H ↑H↑H ↑H↑H ↑Hm-↑H ↑H↑H ↑Hs -F↑M
appendices/ book/    chapters/  mine/↑M
% cd chaptersa↑H ↑H↑M
% mv book/ch↑[↑H↑H ↑H↑H↑Gapter1 chapter1↑M
mv: cannot access book/chapter1↑M
% mv ~john/book/chapter1 ch↑[↑H↑H ↑H↑H↑Gapter1↑M
mv: cannot access /u2/john/book/chapter1↑M
% pwd↑M
/tmp/john/chapters↑M
% cd ..↑M
% ls↑M
appendices book     chapters   mine↑M
% cd book↑M
% ls↑M
chapter1 chapter2 chapter3 chapter4↑M
         .
         .
         .

% ↑Z↑H↑H
script done on Thu Jan 23 12:18:42 1986
```

##################

Appendix IV

An Example showing the Development of a Misconception

This example shows a user attempting to achieve a task of moving a block of files to a new location. The user started to achieve this goal by moving one file at a time (lines 1 to 22), but then attempting to progress to a "block move" syntax. The user consulted the manual (lines 24 and 25) and copied the syntax (line 30), but was misled by the manual syntax for options, so the command fails. Next, an attempt was made to correct the command (lines 31 to 35), ending with "mv -" syntax (line 36), which was the first syntax that worked. After this, the "mv -" syntax is used for the "block move" case.

1	**ls**
2	**lisp**
3	**man directory**
4	**mkdir**
5	**mkdir lisp**
6	**cd lisp**
7	**ls**
8	**cp ctxt ctxt1**
9	**cd**
10	**ls**
11	**cp assoc lisp assoc1**
12	**cp assoc lisp/assoc1**
13	**mv c1 lisp/c1**
14	**cd lisp**
15	**ls**
16	**cd**
17	**ls**
18	**mv ctxt lisp/ctxt**
19	**cd**
20	**ls**
21	**cd lisp**
22	**ls**
23	**lisp**
24	**man mv**
25	**man mv**
26	**ls**
27	**cd**
28	**ls**

```
29   mv {
30   [mv [-] cx dcg lisp
31   mv[-] -cx -dcg lisp
32   mv[-] cx dcg lisp
33   rm lisp
34   mv - -cx -dcg lisp
35   mv [ - ] cx dcg lisp
36   mv - cx dcg lisp
37   ls lisp
38   cat fils
39   rm fils
40   ls
41   ls lisp
42   rm lisp/assoc1
43   ls
44   mv - assoc c4 index let.l macro.ex mapsoc newvar prop sol2 sub
             lisp
45   ls
46   mkdir plog
47   mv - plex ploglec plog
48   cd plog
49   ls
50   cd lisp
```

Appendix V

An Example of Planning and Re-planning in Error Recovery

The following is an example gained from an experiment script for problem 3. The user followed four distinct stages while attempting to achieve the task. The actions that he performed are given below.

1.

Make directory book.bak

Copy contents of directory book to book.bak (cp book book.bak)

Check

Checking the directory book.bak shows that it has no directory entries (book directory copied to form a file).

2.

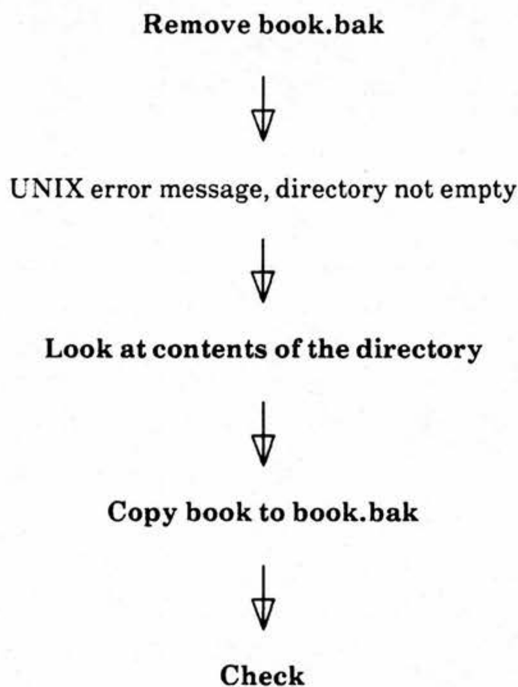
Remove book.bak

Copy book to book.bak

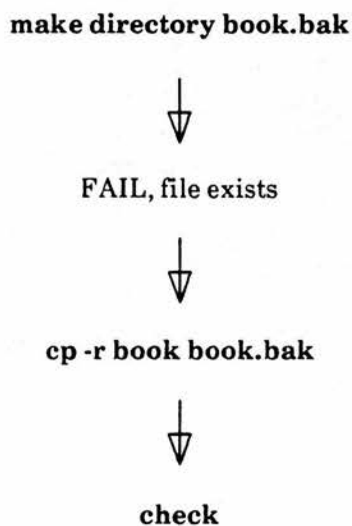
Check

Plan 2 had to be amended due to the unexpected problem of the directory needing to be empty before it can be removed (giving rise to sub-plan 2a). At the end of plan 2, a file had been created, at which point the user looks in the manual for "cp".

2a.



3.



Plan 3 must be amended to cope with the action to make the directory failing. The plan is updated to plan 3a.

3a.

make directory book.bak

FAIL, file exists

remove file book.bak

cp -r book book.bak

check

At the end of plan 3, the user discovers that there is an extra directory, "book.bak".

4.

mv directories chapters and appendices up the tree

Remove extra directory book

Check

Finally, the goal is completed.

Appendix VI

Model of UNIX Commands

```
%       File                :       ~/u2/unix_commands
%       Author              :       J.Lewis
%       Purpose             :       Representation of UNIX commands

%
%       Last update         :       25.7.86

%*************************************************************************
%*************************************************************************
%
%
%
%
%
%       *****************************************************************
%       *                                                               *
%       *                     UNIX COMMAND MODEL                        *
%       *                                                               *
%       *****************************************************************
%
%       Note:
%               Parameter XMode is equivalent to the executable mode of the filestore node.
%               Executable files have parameter x.
%               Non-executable files have parameter o.
%
%*************************************************************************
%*************************************************************************



%       *****************************************************************
%       *                                                               *
%       *                     CP command                                *
%       *                                                               *
%       *****************************************************************

%       Variant 1:          copy file to existing file

%       cp file file

unix_command(active,cp_dorf_to_file,cp,
        args([
                        [
                        Flags1          range [[]]
                        ],
                        [
                        Name1           range [_],
                        Location1       range [_],
                        Links1          range [_],
                        Type1           range [file,dir],
                        XMode1          range [_]
                        ],
                        [
                        Name2           range [_],
                        Location2       range [_],
                        Links2          range [_],
```

```
                    Type2               range [file],
                    XMode2              range [_]
                    ]
        ]),

    preconds([[Name1|Location1]\=[Name2|Location2]
        ]),

    actions([
        (add_node,[Name2,Location2,[],file,XMode1])
        ])
    ).

%       ******************************************************

%       Variant 2:      copy file to new file

%       cp file unknown

unix_command(active,cp_dorf_to_unknown,cp,
    args([
                    [
                    Flags1              range [[]]
                    ],
                    [
                    Name1               range [_],
                    Location1           range [_],
                    Links1              range [_],
                    Type1               range [file, dir],
                    XMode1              range [_]
                    ],
                    [
                    Name2               range [_],
                    Location2           range [_],
                    Links2              range [_],
                    Type2               range [unknown],
                    XMode2              range [_]
                    ]
        ]),

    preconds([
        ]),

    actions([
        (add_node,[Name2,Location2,[],file,XMode1])
        ])
    ).

%       ******************************************************

%       Variant 3:      copy file into a directory

%       cp file dir

unix_command(active,cp_dorf_to_directory,cp,
    args([
                    [
                    Flags1              range [[]]
                    ],
                    [
                    Name1               range [_],
                    Location1           range [_],
                    Links1              range [_],
                    Type1               range [file,dir],
                    XMode1              range [_]
                    ],
                    [
                    Name2               range [_],
                    Location2           range [_],
                    Links2              range [_],
```

```
                        Type2           range [dir],
                        XMode2          range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[Name1,[Name2|Location2],[],file,XMode1])
                ])
        ).

%       ****************************************************

%       Variant 4:      copy file tree into a directory

%       cp -r file dir

unix_command(active,cp_tree_to_directory,cp,
        args([
                        [
                        Flags1          range [[r]]
                        ],
                        [
                        Name1           range [_],
                        Location1       range [_],
                        Links1          range [_],
                        Type1           range [file, dir],
                        XMode1          range [_]
                        ],
                        [
                        Name2           range [_],
                        Location2       range [_],
                        Links2          range [_],
                        Type2           range [dir],
                        XMode2          range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[Name1,[Name2|Location2],Links1,Type1,XMode1])
                ])
        ).

%       ************************************************************
```

```
%         ***************************************************************
%         *                                                             *
%         *                   MV command                                *
%         *                                                             *
%         ***************************************************************

%         Variant 1:          move file to existing file

%         mv file file

unix_command(active,mv_file_to_file,mv,
        args([
                        [
                        Flags1              range [[]]
                        ],
                        [
                        Name1               range [_],
                        Location1           range [_],
                        Links1              range [_],
                        Type1               range [file],
                        XMode1              range [_]
                        ],
                        [
                        Name2               range [_],
                        Location2           range [_],
                        Links2              range [_],
                        Type2               range [file],
                        XMode2              range [_]
                        ]
                ]),

        preconds([[Name1|Location1]\=[Name2|Location2]
                ]),

        actions([
                (add_node,[Name2,Location2,[],file,XMode1]),
                (delete_node,[Name1,Location1,Links1,Type1,XMode1])
                ])
        ).

%         ****************************************************************

%         Variant 2:          move file to new file

%         mv file unknown

unix_command(active,mv_dorf_to_unknown,mv,
        args([
                        [
                        Flags1              range [[]]
                        ],
                        [
                        Name1               range [_],
                        Location1           range [_],
                        Links1              range [_],
                        Type1               range [file, dir],
                        XMode1              range [_]
                        ],
                        [
                        Name2               range [_],
                        Location2           range [_],
                        Links2              range [_],
                        Type2               range [unknown],
                        XMode2              range [_]
                        ]
                ]),

        preconds([
                ]),
```

```
        actions([
                (add_node,[Name2,Location2,Links1,Type1,XMode1]),
                (delete_node,[Name1,Location1,Links1,Type1,XMode1])
                ])
        ).

%       ****************************************************************

%       Variant 3:          move file to directory

%       mv file dir

unix_command(active,mv_dorf_to_directory,mv,
        args([
                        [
                        Flags1          range [[]]
                        ],
                        [
                        Name1           range [_],
                        Location1       range [_],
                        Links1          range [_],
                        Type1           range [file,dir],
                        XMode1          range [_]
                        ],
                        [
                        Name2           range [_],
                        Location2       range [_],
                        Links2          range [_],
                        Type2           range [dir],
                        XMode2          range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[Name1,[Name2|Location2],Links1,Type1,XMode1]),
                (delete_node,[Name1,Location1,Links1,Type1,XMode1])
                ])
        ).

%***********************************************************************
```

```
%       *************************************************************
%       *                                                           *
%       *                   MKDIR command                           *
%       *                                                           *
%       *************************************************************

%       Variant 1:          create directory

%       mkdir dir

unix_command(active,mkdir,mkdir,
        args([
                        [
                        Flags1    range [[]]
                        ],
                        [
                        Name1               range [_],
                        Location1           range [_],
                        Links1              range [[]],
                        Type1               range [unknown],
                        XMode1              range [unknown]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[Name1,Location1,[],dir,x])
                ])
        ).

%**********************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                    TOP command                               *
%       *                                                              *
%       ****************************************************************

%       Variant 1:          edit existing file

%       top file

unix_command(active,top_dorf,top,
        args([
                        [
                        Flagsl              range [[]]
                        ],
                        [
                        Name1               range [_],
                        Location1           range [_],
                        Links1              range [_],
                        Type1               range [file,dir],
                        XMode1              range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (delete_node,[Name1,Location1,Links1,Type1,XMode1]),
                (add_node,[Name1,Location1,Links1,Type1,XMode1])
                ])
        ).

%       ****************************************************************

%       Variant 2:          edit new file

%       top unknown

unix_command(active,top_unknown,top,
        args([
                        [
                        Flagsl    range  [[]]
                        ],
                        [
                        Name1   range [_],
                        Location1           range [_],
                        Links1  range [_],
                        Type1   range [unknown],
                        XMode1  range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[Name1,Location1,[],file,o])
                ])
        ).

%*********************************************************************
```

```
%        ***************************************************************
%        *                                                             *
%        *                     RM command                              *
%        *                                                             *
%        ***************************************************************

%        Variant 1:          remove existing file

%        rm file

unix_command(active,rm_file,rm,
        args([
                        [
                        Flagsl              range [[]]
                        ],
                        [
                        Namel               range [_],
                        Locationl           range [_],
                        Linksl              range [[]],
                        Typel               range [file],
                        XModel              range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (delete_node,[Namel,Locationl,[],Typel,XModel])
                ])
        ).

%        ***************************************************************

%        Variant 2:          remove file tree

%        rm tree

unix_command(active,rm_tree,rm,
        args([
                        [
                        Flagsl              range [[r]]
                        ],
                        [
                        Namel               range [_],
                        Locationl           range [_],
                        Linksl              range [_],
                        Typel               range [file,dir],
                        XModel              range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (delete_node,[Namel,Locationl,Linksl,Typel,XModel])
                ])
        ).

%***********************************************************************
```

```
%       ***************************************************************
%       *                                                             *
%       *                     RMDIR command                           *
%       *                                                             *
%       ***************************************************************

%       Variant 1:          remove directory

%       rmdir directory

unix_command(active,rm_directory,rmdir,
        args([
                        [
                        Flags1          range [[]]
                        ],
                        [
                        Name1           range [_],
                        Location1       range [_],
                        Links1          range [[]],
                        Type1           range [dir],
                        XMode1          range [x]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (delete_node,[Name1,Location1,[],dir,x])
                ])
        ).


%*************************************************************************
```

```
%       ***************************************************************
%       *                                                             *
%       *                    LS command                               *
%       *                                                             *
%       ***************************************************************

%       Variant 1:          list contents of directory

%       ls dir

unix_command(active,ls_arg,ls,
        args([
                        [
                        Flags1          range [[]]
                        ],
                        [
                        Name1           range [_],
                        Location1       range [_],
                        Links1          range [_],
                        Type1           range [dir],
                        XMode1          range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([])
        ).

%       ***************************************************************

%       Variant 2:          list contents of directory with -F flag (specifies directory /
%                           executable file

%       ls -F dir

unix_command(active,lsf_arg,ls,
        args([
                        [
                        Flags1          range [['F']]
                        ],
                        [
                        Name1           range [_],
                        Location1       range [_],
                        Links1          range [_],
                        Type1           range [dir],
                        XMode1  range [_]
                        ]
                ]),

        preconds([
                ]),

        actions([])
        ).

%       ***************************************************************

%       Variant 3:          list contents of default directory

%       ls

unix_command(active,ls_no_args,ls,
        args([
                        [
                        Flags1          range [[]]
                        ]
                ]),

        preconds([
```

```
		]),

	actions([
		])
	).

%	*********************************************************************

%	Variant 4:		list contents of default directory with -F flag

%	ls -F

unix_command(active,lsf_no_args,ls,
	args([
			[
			Flags1			range [['F']]
			]
		]),

	preconds([
		]),

	actions([
		])
	).

%***************************************************************************
```

```
%       ***************************************************************
%       *                                                             *
%       *                    CD command                               *
%       *                                                             *
%       ***************************************************************

%       Variant 1:          change current directory

%       cd dir

unix_command(active,cd_arg,cd,
        args([
                        [
                        Flags1          range [[]]
                        ],
                        [
                        Name1           range [_],
                        Location1       range [_],
                        Links1          range [_],
                        Type1           range [dir],
                        XMode1          range [x]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (change_directory,[Name1,Location1,Links1,Type1,XMode1])
                ])
        ).

%       ***************************************************************

%       Variant 2:          change current directory to HOME directory

%       cd

unix_command(active,cd_no_args,cd,
        args([
                        [
                        Flags1          range [[]]
                        ]
                ]),

        preconds([home([Name|Location])
                ]),

        actions([
                (change_directory,[Name,Location,_,_,_])
                ])
        ).

%***********************************************************************
```

```
%          ****************************************************************
%          *                                                              *
%          *                    PWD command                               *
%          *                                                              *
%          ****************************************************************

%          Variant 1:          show current directory path

%          pwd dir

unix_command(active,pwd_no_args,pwd,
           args([
                              [
                              Flags1              range [[]]
                              ]
                  ]),

           preconds([
                  ]),

           actions([
                  ])
           ).


%***************************************************************************
%***************************************************************************
```

Appendix VII

User Model of UNIX Commands

```
%       File            :       ~/u2/user_commands
%       Author          :       J.Lewis
%       Purpose         :       Representation of User model of UNIX commands
%
%       Last update     :       25.7.86
%**************************************************************************
%**************************************************************************
%
%
%
%
%       ******************************************************************
%       *                                                                *
%       *                   USER COMMAND MODEL                           *
%       *                                                                *
%       ******************************************************************
%
%       Note:
%               Parameter XMode is equivalent to the executable mode of the filestore node.
%               Executable files have parameter x.
%               Non-executable files have parameter o.
%
%**************************************************************************
%**************************************************************************


%       ******************************************************************
%       *                                                                *
%       *               CP command                                       *
%       *                                                                *
%       ******************************************************************

%       Variant 1:      copy file to existing file


%       cp file file

user_command(active,cp_dorf_to_file,cp,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
```

```
                        Links1
                                range ?
                                possible [[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[o],[x],[x,o]]
                        ],
                        [
                        Name2
                                range ?
                                possible [[Name1],[_]],
                        Location2
                                range ?
                                possible [[Location1],[_]],
                        Links2
                                range ?
                                possible [[Links1],[_]],
                        Type2
                                range ?
                                possible [[file],[Type1]],
                        XMode2
                                range ?
                                possible [[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                        [Name1|Location1]\=[Name2|Location2]
                ]),

        actions([
                (add_node,[
                        ? possible [Name1,Name2],
                        ? possible [Location1,Location2,[Name1|Location1],
                                                        [Name2|Location2]],
                        ? possible [[],Links1,Links2],
                        ? possible [file,dir,Type1,Type2],
                        ? possible [x,o,XMode1,XMode2]
                        ]
                )
                ])
        ).


%       ****************************************************

%       Variant 2:      copy file to new file

%       cp file unknown

user_command(active,cp_dorf_to_unknown,cp,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[[]],[_]],
```

```
				Type1
					range ?
					possible [[file],[dir],[file,dir]],
				XMode1
					range ?
					possible [[x],[o],[x,o]]
				],
				[
				Name2
					range ?
					possible [[Name1],[_]],
				Location2
					range ?
					possible [[Location1],[_]],
				Links2
					range ?
					possible [[[]],[Links1],[_]],
				Type2
					range ?
					possible [[unknown]],
				XMode2
					range ?
					possible [[unknown],[x],[o],[x,o]]
				]
		]),

	preconds([
		]),

	actions([
		(add_node,[
				? possible [Name1,Name2],
				? possible [Location1,Location2,[Name1|Location1],
							[Name2|Location2]],
				? possible [[],Links1,Links2],
				? possible [file,dir,Type1],
				? possible [x,o,XMode1,XMode2]
				]
		)
		])
	).


%	***********************************************************

%	Variant 3:		copy file into a directory

%	cp file dir

user_command(active,cp_dorf_to_directory,cp,
	args([
				[
				Flags1
					range ?
					possible [[[]]]
				],
				[
				Name1
					range ?
					possible [[_]],
				Location1
					range ?
					possible [[_]],
				Links1
					range ?
					possible [[[]],[_]],
				Type1
					range ?
					possible [[file],[dir],[file,dir]],
				XMode1
```

```
                                        range ?
                                        possible [[o],[x],[x,o]]
                                ],
                                [
                                Name2
                                        range ?
                                        possible [[Name1],[_]],
                                Location2
                                        range ?
                                        possible [[Location1],[_]],
                                Links2
                                        range ?
                                        possible [[[]],[Links1],[_]],
                                Type2
                                        range ?
                                        possible [[dir]],
                                XMode2
                                        range ?
                                        possible [[o],[x],[x,o]]
                                ]
                        ]),

        preconds([
                        ]),

        actions([
                (add_node,[
                                ? possible [Name1,Name2],
                                ? possible [Location1,Location2,[Name1|Location1],
                                                                        [Name2|Location2]],
                                ? possible [[],Links1,Links2],
                                ? possible [file,dir,Type1,Type2],
                                ? possible [x,o,XMode1,XMode2]
                                ]
                )
                ])
        ).

%       ********************************************************

%       Variant 4:              copy file tree into a directory

%       cp -r file dir

user_command(active,cp_tree_to_directory,cp,
        args([
                                [
                                Flags1
                                        range ?
                                        possible [[[]],[[r]]]
                                ],
                                [
                                Name1
                                        range ?
                                        possible [[_]],
                                Location1
                                        range ?
                                        possible [[_]],
                                Links1
                                        range ?
                                        possible [[[]],[_]],
                                Type1
                                        range ?
                                        possible [[dir]],
                                XMode1
                                        range ?
                                        possible [[x]]
                                ],
                                [
                                Name2
```

```
                        range ?
                        possible [[_]],
                Location2
                        range ?
                        possible [[_]],
                Links2
                        range ?
                        possible [[[]],[_]],
                Type2
                        range ?
                        possible [[unknown],[dir],[unknown,dir]],
                XMode2
                        range ?
                        possible [[x],[unknown]]
                ]
        ]),

preconds([
        ]),

actions([
        (add_node,[
                ? possible [Name1,Name2],
                ? possible [Location2,[Name2|Location2]],
                ? possible [[],Links1,Links2],
                ? possible [dir,Type1,Type2],
                ? possible [x,XModel]
                ]
        )
        ])
).

%       *****************************************************************
```

```
%        **************************************************************
%        *                                                            *
%        *                   MV command                               *
%        *                                                            *
%        **************************************************************

%        Variant 1:          move file to existing file

%        mv file file

user_command(active,mv_file_to_file,mv,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[o],[x],[x,o]]
                        ],
                        [
                        Name2
                                range ?
                                possible [[Name1],[_]],
                        Location2
                                range ?
                                possible [[Location1],[_]],
                        Links2
                                range ?
                                possible [[Links1],[_]],
                        Type2
                                range ?
                                possible [[file],[Type1]],
                        XMode2
                                range ?
                                possible [[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                        [Name1|Location1]\=[Name2|Location2]
                ]),

        actions([
                (add_node,[
                        ? possible [Name1,Name2],
                        ? possible [Location1,Location2],
                        ? possible [[],Links1,Links2],
                        ? possible [file,Type1,Type2],
                        ? possible [XMode1,XMode2]
                        ]
                ),
                (delete_node,[
                        ? possible [Name1,Name2],
                        ? possible [Location1],
                        ? possible [[],Links1,Links2],
                        ? possible [file,Type1,Type2],
```

```
                        ? possible [XMode1,XMode2]
                        ]
                )
                ])
        ).


%       *****************************************************************

%       Variant 2:          move file to new file

%       mv file unknown

user_command(active,mv_dorf_to_unknown,mv,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[[]],[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[o],[x],[x,o]]
                        ],
                        [
                        Name2
                                range ?
                                possible [[Name1],[_]],
                        Location2
                                range ?
                                possible [[Location1],[_]],
                        Links2
                                range ?
                                possible [[[]],[Links1],[_]],
                        Type2
                                range ?
                                possible [[unknown]],
                        XMode2
                                range ?
                                possible [[unknown],[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[
                        ? possible [Name1,Name2],
                        ? possible [Location1],
                        ? possible [[],Links1,Links2],
                        ? possible [file,Type1,Type2],
                        ? possible [XMode1,XMode2]
                        ]
                ),
                (delete_node,[
```

```
                        ? possible [Name1,Name2],
                        ? possible [Location1],
                        ? possible [[],Links1,Links2],
                        ? possible [file,Type1,Type2],
                        ? possible [XMode1,XMode2]
                        ]
                )
                ])
        ).


%       *****************************************************************


%       Variant 3:      move file to directory

%       mv file dir

user_command(active,mv_dorf_to_directory,mv,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[o],[x],[x,o]]
                        ],
                        [
                        Name2
                                range ?
                                possible [[Name1],[_]],
                        Location2
                                range ?
                                possible [[Location1],[_]],
                        Links2
                                range ?
                                possible [[Links1],[_]],
                        Type2
                                range ?
                                possible [[dir]],
                        XMode2
                                range ?
                                possible [[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[
                        ? possible [Name1,Name2],
                        ? possible [[Name2|Location2]],
                        ? possible [[],Links1],
                        ? possible [file,dir,Type1,Type2],
```

```
                ? possible [x,o,XMode1,XMode2]
                ]
        ),
        (delete_node,[
                ? possible [Name1,Name2],
                ? possible [Location1],
                ? possible [Links1,Links2],
                ? possible [file,dir,Type1,Type2],
                ? possible [x,o,XMode1,XMode2]
                ]
        )
        ])
).


%***************************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                    MKDIR command                             *
%       *                                                              *
%       ****************************************************************

%       Variant 1:          create directory

%       mkdir dir

user_command(active,mkdir,mkdir,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[[]],[_]],
                        Type1
                                range ?
                                possible [[unknown],[dir],[file,dir],[unknown,file,dir]],
                        XMode1
                                range ?
                                possible [[unknown],[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (add_node,[

                        ? possible [Name1],
                        ? possible [Location1],
                        ? possible [[],Links1],
                        ? possible [dir,Type1],
                        ? possible [x,o,XMode1]
                        ]
                )
                ])
        ).

%****************************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                      TOP command                             *
%       *                                                              *
%       ****************************************************************

%       Variant 1:          edit existing file

%       top file

user_command(active,top_dorf,top,
        args([
                        [
                        Flagsl
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[[]],[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (delete_node,[
                        ? possible [Name1],
                        ? possible [Location1],
                        ? possible [Links1],
                        ? possible [Type1],
                        ? possible [x,o,XMode1,XMode2]
                        ]
                ),
                (add_node,[
                        ? possible [Name1],
                        ? possible [Location1],
                        ? possible [[],Links1],
                        ? possible [file,dir,Type1],
                        ? possible [XMode1]
                        ]
                )
                ])
        ).


%       ****************************************************************

%       Variant 2:          edit new file

%       top unknown

user_command(active,top_unknown,top,
        args([
                        [
                        Flagsl
                                range ?
```

```
                possible [[[]]]
            ],
            [
            Name1
                range ?
                possible [[_]],
            Location1
                range ?
                possible [[_]],
            Links1
                range ?
                possible [[[]],[_]],
            Type1
                range ?
                possible [[unknown],[file],[dir],[file,dir]],
            XMode1
                range ?
                possible [[unknown],[o],[x],[x,o]]
            ]
        ]),

preconds([
        ]),

actions([
        (add_node,[
            ? possible [Name1],
            ? possible [Location1],
            ? possible [[],Links1],
            ? possible [file,dir,Type1],
            ? possible [x,o,XMode1,XMode2]
            ]
        )
        ])
).


%****************************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                     RM command                               *
%       *                                                              *
%       ****************************************************************

%       Variant 1:          remove existing file

%       rm file

user_command(active,rm_file,rm,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (delete_node,[
                        ? possible [Name1],
                        ? possible [Location1],
                        ? possible [Links1],
                        ? possible [Type1],
                        ? possible [x,o,XMode1,XMode2]
                        ]
                )
                ])
        ).

%       ****************************************************************

%       Variant 2:          remove file tree

%       rm -r tree

user_command(active,rm_tree,rm,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]],[[r]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
```

```
                    possible [[_]],
            Links1
                    range ?
                    possible [[[]],[_]],
            Type1
                    range ?
                    possible [[file],[dir],[file,dir]],
            XMode1
                    range ?
                    possible [[o],[x],[x,o]]
            ]
        ]),

preconds([
        ]),

actions([
        (delete_node,[
            ? possible [Name1],
            ? possible [Location1],
            ? possible [Links1],
            ? possible [Type1],
            ? possible [x,o,XMode1,XMode2]
            ]
        )
        ])
).

%****************************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                 RMDIR command                                *
%       *                                                              *
%       ****************************************************************

%       Variant 1:        remove directory

%       rmdir directory

user_command(active,rm_directory,rmdir,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[[]],[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[x],[o],[x,o]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (delete_node,[
                        ? possible [Name1],
                        ? possible [Location1],
                        ? possible [Links1],
                        ? possible [Type1],
                        ? possible [XMode1]
                        ]
                )
                ])
        ).


%*************************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                  LS command                                  *
%       *                                                              *
%       ****************************************************************

%       Variant 1:         list contents of directory

%       ls dir

user_command(active,ls_arg,ls,
        args([
                           [
                           Flags1
                                   range ?
                                   possible [[[]]]
                           ],
                           [
                           Name1
                                   range ?
                                   possible [[_]],
                           Location1
                                   range ?
                                   possible [[_]],
                           Links1
                                   range ?
                                   possible [[],[_]],
                           Type1
                                   range ?
                                   possible [[file],[dir],[file,dir]],
                           XMode1
                                   range ?
                                   possible [[o],[x],[x,o]]
                           ]
                ]),

        preconds([
                ]),

        actions([
                ])
        ).

%       ****************************************************************

%       Variant 2:         list contents of directory with -F flag (specifies directory /
%                          executable file

%       ls -F dir

user_command(active,lsf_arg,ls,
        args([
                           [
                           Flags1
                                   range ?
                                   possible [[['F']]]
                           ],
                           [
                           Name1
                                   range ?
                                   possible [[_]],
                           Location1
                                   range ?
                                   possible [[_]],
                           Links1
                                   range ?
                                   possible [[],[_]],
                           Type1
                                   range ?
                                   possible [[file],[dir],[file,dir]],
```

```
                    XModel
                            range ?
                            possible [[o],[x],[x,o]]
                    ]
            ]),

        preconds([
            ]),

        actions([
            ])
        ).


%       ****************************************************************

%       Variant 3:          list contents of default directory

%       ls

user_command(active,ls_no_args,ls,
        args([
                    [
                    Flags1
                            range ?
                            possible [[[]]]
                    ]
            ]),

        preconds([
            ]),

        actions([
            ])
        ).


%       ****************************************************************

%       Variant 4:          list contents of default directory with -F flag

%       ls -F

user_command(active,lsf_no_args,ls,
        args([
                    [
                    Flags1
                            range ?
                            possible [[['F']]]
                    ]
            ]),

        preconds([
            ]),

        actions([
            ])
        ).

%************************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                     CD command                               *
%       *                                                              *
%       ****************************************************************

%       Variant 1:          change current directory

%       cd dir

user_command(active,cd_arg,cd,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ],
                        [
                        Name1
                                range ?
                                possible [[_]],
                        Location1
                                range ?
                                possible [[_]],
                        Links1
                                range ?
                                possible [[[]],[_]],
                        Type1
                                range ?
                                possible [[file],[dir],[file,dir]],
                        XMode1
                                range ?
                                possible [[o],[x],[x,o]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                (change_directory,[
                        ? possible [Name1],
                        ? possible [Location1],
                        ? possible [Links1],
                        ? possible [file,dir,Type1],
                        ? possible [x,o,XMode1,XMode2]
                        ]
                )
                ])
        ).

%       ****************************************************************

%       Variant 2:          change current directory to HOME directory

%       cd

user_command(active,cd_no_args,cd,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ]
                ]),

        preconds([home([Name,
                        Location,
                        _,
                        _,
                        _,
```

```
                _])
        ]),

actions([
        (change_directory,[
                        ? possible [Name],
                        ? possible [Location],
                        ? possible [Links],
                        ? possible [Name],
                        ? possible [Type],
                        ? possible [XMode]
                        ]
        )
        ])
).

%**************************************************************************
```

```
%       ****************************************************************
%       *                                                              *
%       *                     PWD command                              *
%       *                                                              *
%       ****************************************************************

%       Variant 1:          show current directory path

%       pwd dir

user_command(active,pwd_no_args,pwd,
        args([
                        [
                        Flags1
                                range ?
                                possible [[[]]]
                        ]
                ]),

        preconds([
                ]),

        actions([
                ])
        ).

%***************************************************************************
%***************************************************************************
```

Appendix VIII

A Sample Plan Grammar

```
%       File            :       ~/u3/plan_sample
%       Author          :       J.Lewis
%       Purpose         :       plan grammar
%                               for UNIX user users
%
%       Last update     :       9.10.86

%*******************************************************************************

%       ****************************************************
%       *                                                  *
%       *                  PLAN SAMPLE                     *
%       *                                                  *
%       ****************************************************


%       Define goals and sub-goals as PROLOG infix operators

:-op(70,fy,[     goal,
                 redo,
                 move_file_to_file,
                 move_files,
                 copy_tree,
                 create_file_copy,
                 change_directory,
                 make_directory,
                 copy_file,
                 copy_directory,
                 copy_directory_to_file,
                 remove_file,
                 remove_files,
                 remove_directory,
                 remove_tree,
                 edit_file,
                 list
                 ]).


%       Define the toplevel category (goal) for UNIX - this is just defined as
%       being any goal.

initial_category(unix,toplevel).


%       ****************************************************
%       *                                                  *
%       *                 GRAMMAR RULES                    *
%       *                                                  *
%       ****************************************************

%       Define the grammar rules.

rule(unix,rule 1,out,
        toplevel,
                [
                goal Goal status Status,
                toplevel
                ],
                        []
        ).
```

```
rule(unix,rule 2,out,
        redo Goal status Status,
                [
                goal Goal status Failed,
                goal Goal status Status
                ],
                        [
                        Failed = = failed
                        ]
        ).

rule(unix,rule 3,out,
        goal move_files FileList to Location status Status,
                [
                move_files FileList to Location status Status
                ],
                        []
        ).

rule(unix,rule 4,out,
        goal copy_tree Tree to Name at_location Location status Status,
                [
                copy_tree Tree to Name at_location Location status Status,
                change_directory Name at_location Location status Status
                ],
                        []
        ).

rule(unix,rule 5,out,
        goal copy_tree Tree to NewTree at_location Location status Status,
                [
                copy_tree Tree to NewTree at_location Location status Status
                ],
                        []
        ).

rule(unix,rule 6,out,
        goal copy_tree Tree to Name at_location Location status Status,
                [
                make_directory Name at_location Location status Status,
                copy_tree Tree to Name at_location Location status Status
                ],
                        []
        ).

rule(unix,rule 7,out,
        goal create_file_copy F1 to F2 at_location [D|L] status Status,
                [
                make_directory D at_location L status Status,
                copy_file F1 to F2 at_location [D|L] status Status
                ],
                        []
        ).

rule(unix,rule 8,out,
        goal create_file_copy F1 to F2 at_location [D|L] status Status,
                [
                make_directory D at_location L status Status,
                move_file_to_file F at_location D1 to F at_location [D|L] status Status,
                list D at_location L status Status
                ],
                        []
        ).

rule(unix,rule 9,out,
        move_files [F1,F2] to [F11 at_location D, F22 at_location D]
                        status Status,
                [
                move_file_to_file F1 to F11 at_location D status Status,
                move_file_to_file F2 to F22 at_location D status Status
```

```
                ],
                        [
                        F1\=F2,
                        [F1,F2] exist_at D
                        ]
        ).


rule(unix,rule 10,out,
        move_files [F1 at_location D1|FL] to [F11 at_location D, F22 at_location D|FL] status
Status,
                [
                move_file F1 at_location D1 to F11 at_location D status Status,
                move_files [F22 at_location D|FL] status Status
                ],
                        [
                        not(member(F1 at_location D,[F22 at_location D|FL])),
                        ]
        ).


rule(unix,rule 11,out,
        copy_files [F1,F2] to [F11 at_location D, F22 at_location D]
                        status Status,
                [
                copy_file_to_file F1 to F11 at_location D status Status,
                copy_file_to_file F2 to F22 at_location D status Status
                ],
                        [
                        F1\=F2,
                        [F1,F2] exist_at D
                        ]
        ).


rule(unix,rule 12,out,
        copy_files [F1 at_location D1|FL] to [F11 at_location D, F22 at_location D|FL] status
Status,
                [
                copy_file F1 at_location D1 to F11 at_location D status Status,
                copy_files [F22 at_location D|FL] status Status
                ],
                        [
                        not(member(F1 at_location D,[F22 at_location D|FL])),
                        ]
        ).


rule(unix,rule 13,out,
        remove_files [F1 at_location D, F2 at_location D]
                        status Status,
                [
                remove_file F1 at_location D status Status,
                remove_file F2 at_location D status Status
                ],
                        []
        ).


rule(unix,rule 14,out,
        remove_files [F1 at_location D, F2 at_location D|FL]
                        status Status,
                [
                remove_file F1 at_location D status Status,
                remove_files [F2 at_location D | FL] status Status
                ],
                        []
        ).


rule(unix,rule 15,out,
        remove_tree D status Status,
```

```
                [
                remove_files [F at_location D | FL] status Status,
                remove_directory D status Status
                ],
                        []
        ).

%**********************************************************


%       ******************************************************
%       *                                                    *
%       *               LEXICAL RULES                        *
%       *                                                    *
%       ******************************************************

%       Define the Lexical entries for the grammar.
%       These are representations of the UNIX commands, which will combine
%       according to the grammar

lexical(unix,lexical 1,out,
        command(cd,
                [
                        [N,B,_,dir,_]
                ],
                [
                        (change_directory,[N,B,_,dir,_])
                ],
                Status,Text),
        [change_directory N at_location B status Status]
        ).

lexical(unix,lexical 2,out,
        command(cd,
                [
                ],
                [
                        (change_directory,[N,B,_,dir,_])
                ],
                Status,Text),
        [change_directory N at_location B status Status]
        ).

lexical(unix,lexical 3,out,
        command(cp,
                [
                        [N1,B1,[],file,_],
                        [_,_,_,_,_]
                ],
                [
                (add_node,[N2,B2,[],file,_])
                ],
                Status,Text),
        [copy_file N1 at_location B1 to N2 at_location B2 status Status]
        ).

lexical(unix,lexical 4,out,
        command(cp,
                [
                        [N1,B1,[],dir,_],
                        [_,_,_ _,_]
                ],
                [
                (add_node,[N2,B2,[],dir,_])
                ],
                Status,Text),
        [copy_directory N1 at_location B1 to N2 at_location B2 status Status]
        ).

lexical(unix,lexical 5,out,
```

```
        command(cp,
                [
                        [N1,B1,_,dir,_],
                        [_,_,_,_,_]
                ],
                [
                        (add_node,[N2,B2,_,dir,_])
                ],
                Status,Text),
        [copy_tree N1 at_location B1 to N2 at_location B2 status Status]
        ).

lexical(unix,lexical 6,out,
        command(ls,
                [
                ],
                [
                ],
                Status,Text),
        [list N at_location B status Status]
        ).

lexical(unix,lexical 7,out,
        command(ls,
                [
                        [N,B|_]
                ],
                [
                ],
                Status,Text),
        [list N at_location B status Status]
        ).

lexical(unix,lexical 8,out,
        command(mkdir,
                [
                        [N,B,[],unknown,_]
                ],
                [
                        (add_node,[N,B,_,dir,_])
                ],
                Status,Text),
        [make_directory N at_location B status Status]
        ).

lexical(unix,lexical 9,out,
        command(mv,
                [
                        [N1,B1,[],file,_],
                        [_,_,_,_,_]
                ],
                [
                        (add_node,[N2,B2,[],file,_]),
                        (delete_node,[N1,B1,[],file,_])
                ],
                Status,Text),
        [move_file_to_file N1 at_location B1 to N2 at_location B2 status Status]
        ).

lexical(unix,lexical 10,out,
        command(mv,
                [
                        [N1,B1,[],dir,_],
                        [_,_,_,_,_]
                ],
                [
                        (add_node,[N2,B2,[],dir,_]),
                        (delete_node,[N1,B1,[],dir,_])
                ],
                Status,Text),
        [move_directory_to_directory([N1,B1],[N2,B2]) status Status]
        ).
```

```
lexical(unix,lexical 11,out,
        command(mv,
                [
                        [N1,B1,_,dir,_],
                        [_,_,_,_,_]
                ],
                [
                        (add_node,[N2,B2,_,dir,_]),
                        (delete_node,[N1,B1,_,dir,_])
                ],
                Status,Text),
        [move_tree_to_tree([N1,B1],[N2,B2]) status Status]
        ).

lexical(unix,lexical 12,out,
        command(pwd,
                [],
                [],
                Status,Text),
        [print_working_directory status Status]
        ).

lexical(unix,lexical 13,out,
        command(rm,
                [
                        [N,B,[],file,_]
                ],
                [
                        (delete_node,[N,B,[],file,_])
                ],
                Status,Text),
        [remove_file N at_location B status Status]
        ).

lexical(unix,lexical 14,out,
        command(rm,
                [
                        [N,B,[],dir,_]
                ],
                [
                        (delete_node,[N,B,[],dir,_])
                ],
                Status,Text),
        [remove_directory N at_location B status Status]
        ).

lexical(unix,lexical 15,out,
        command(rm,
                [
                        [N,B,L,dir,_]
                ],
                [
                        (delete_node,[N,B,L,dir,_])
                ],
                Status,Text),
        [remove_tree N at_location B status Status]
        ).

lexical(unix,lexical 16,out,
        command(top,
                [
                        [N,B,_,_,_]
                ],
                [
                        (add_node,[N,B,_,_,_])
                ],
                Status,Text),
        [edit_file N at_location B status Status]
        ).
```

Appendix IX

Example UNIX Cliche Detection

This example shows a new cliche rule being added to the plan grammar. The form of the rule is basically the same as for other grammar rules, except that the *cliche* tag is used at the start of the rule, and the rule is given a unique identifying number (1 in this case) on line 6. The cliche rule describes that to *move* the specific file /usr/forth2/jml/john/mbox to /usr/forth2/jml/john/fred/mbox can be achieved by the two actions of *copying* the file /usr/forth2/jml/john/mbox to /usr/forth2/jml/john/fred/mbox (lines 8 to 9) and then *deleting* the file /usr/forth2/jml/john/mbox (lines 10 to 11). The status part of the rule is left uninstantiated (line 11), and no conditions are applied to the rule (the conditions field being [] on line 11).

1 % **mkdir fred**	1
2 % **cp mbox fred**	2
3 % **rm mbox**	3
4 % **ls fred**	4
adding new cliche rule:	5
rule(unix,cliche 1,in,move_file_to_file mbox at_location	6
[john,jml,forth2,usr,/] to mbox at_location [fred,john,jml,forth2,usr,/]	7
status _127360,[copy_file mbox at_location [john,jml,forth2,usr,/]	8
to mbox at_location [fred,john,jml,forth2,usr,/] status _127360,	9
remove_file mbox at_location [john,jml,forth2,usr,/]	10
status _127360],[])	11
mbox	12
5 %	13

Appendix X

Chart Analysis Data

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Test Grammar

strategy(test,bu).
policy(test,bf).
initial_category(test,goal).

rule(test,p,[a]).
rule(test,p,[a,p]).

lexical(test,a,[a]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       The following are descriptions of charts which result from word lists for an action
%       sequence a,a,a,a. The charts are given for:
%
%                   *       a parse which does not allow holes.
%                   *       a parse containing holes of length 1.
%                   *       a parse containing holes of any length.

%       RESULTANT CHART FOR PARSE WITHOUT HOLES


Word list: [a,a,a,a]. (27 edges)

 Active edges: edge(p,[],[a],0,0)
        edge(p,[],[a],1,1)
        edge(p,[],[a],2,2)
        edge(p,[],[a],3,3)
        edge(p,[],[a,p],0,0)
        edge(p,[],[a,p],1,1)
        edge(p,[],[a,p],2,2)
        edge(p,[],[a,p],3,3)
        edge(p,[a=0],[p],0,1)
        edge(p,[a=1],[p],1,2)
        edge(p,[a=2],[p],2,3)
        edge(p,[a=3],[p],3,4)
        edge(user,[],[goal],0,0)
Inactive edges: edge(a,[word(a)=0],[],0,1)
        edge(a,[word(a)=1],[],1,2)
        edge(a,[word(a)=2],[],2,3)
        edge(a,[word(a)=3],[],3,4)
        edge(p,[a=0],[],0,1)
        edge(p,[a=1],[],1,2)
        edge(p,[a=2],[],2,3)
        edge(p,[a=3],[],3,4)
        edge(p,[p=1,a=0],[],0,2)
        edge(p,[p=1,a=0],[],0,3)
        edge(p,[p=1,a=0],[],0,4)
        edge(p,[p=2,a=1],[],1,3)
        edge(p,[p=2,a=1],[],1,4)
        edge(p,[p=3,a=2],[],2,4)
```

```
%          RESULTANT CHART FOR PARSE WITH HOLES OF SIZE 1

Word list: [a,a,a,a]. (40 edges)

 Active edges: edge(p,[],[a],0,0)
        edge(p,[],[a],1,1)
        edge(p,[],[a],2,2)
        edge(p,[],[a],3,3)
        edge(p,[],[a,p],0,0)
        edge(p,[],[a,p],1,1)
        edge(p,[],[a,p],2,2)
        edge(p,[],[a,p],3,3)
        edge(p,[a=0-0],[p],0,1)
        edge(p,[a=0-1],[p],0,2)
        edge(p,[a=1-1],[p],1,2)
        edge(p,[a=1-2],[p],1,3)
        edge(p,[a=2-2],[p],2,3)
        edge(p,[a=2-3],[p],2,4)
        edge(p,[a=3-3],[p],3,4)
        edge(user,[],[goal],0,0)
Inactive edges: edge(a,[word(a)=0],[],0,1)
        edge(a,[word(a)=1],[],1,2)
        edge(a,[word(a)=2],[],2,3)
        edge(a,[word(a)=3],[],3,4)
        edge(p,[a=0-0],[],0,1)
        edge(p,[a=0-1],[],0,2)
        edge(p,[a=1-1],[],1,2)
        edge(p,[a=1-2],[],1,3)
        edge(p,[a=2-2],[],2,3)
        edge(p,[a=2-3],[],2,4)
        edge(p,[a=3-3],[],3,4)
        edge(p,[p=1-1,a=0-0],[],0,2)
        edge(p,[p=1-1,a=0-0],[],0,3)
        edge(p,[p=1-1,a=0-0],[],0,4)
        edge(p,[p=1-2,a=0-0],[],0,3)
        edge(p,[p=1-2,a=0-0],[],0,4)
        edge(p,[p=2-2,a=0-1],[],0,3)
        edge(p,[p=2-2,a=0-1],[],0,4)
        edge(p,[p=2-2,a=1-1],[],1,3)
        edge(p,[p=2-2,a=1-1],[],1,4)
        edge(p,[p=2-3,a=0-1],[],0,4)
        edge(p,[p=2-3,a=1-1],[],1,4)
        edge(p,[p=3-3,a=1-2],[],1,4)
        edge(p,[p=3-3,a=2-2],[],2,4)
```

% RESULTANT CHART FOR PARSE WITH HOLES OF ANY SIZE

```
Word list: [a,a,a,a]. (45 edges)

 Active edges: edge(p,[],[a],0,0)
        edge(p,[],[a],1,1)
        edge(p,[],[a],2,2)
        edge(p,[],[a],3,3)
        edge(p,[],[a,p],0,0)
        edge(p,[],[a,p],1,1)
        edge(p,[],[a,p],2,2)
        edge(p,[],[a,p],3,3)
        edge(p,[a=0-0],[p],0,1)
        edge(p,[a=0-1],[p],0,2)
        edge(p,[a=0-2],[p],0,3)
        edge(p,[a=0-3],[p],0,4)
        edge(p,[a=1-1],[p],1,2)
        edge(p,[a=1-2],[p],1,3)
        edge(p,[a=1-3],[p],1,4)
        edge(p,[a=2-2],[p],2,3)
        edge(p,[a=2-3],[p],2,4)
        edge(p,[a=3-3],[p],3,4)
        edge(user,[],[goal],0,0)
Inactive edges: edge(a,[word(a)=0],[],0,1)
        edge(a,[word(a)=1],[],1,2)
        edge(a,[word(a)=2],[],2,3)
        edge(a,[word(a)=3],[],3,4)
        edge(p,[a=0-0],[],0,1)
        edge(p,[a=0-1],[],0,2)
        edge(p,[a=0-2],[],0,3)
        edge(p,[a=0-3],[],0,4)
        edge(p,[a=1-1],[],1,2)
        edge(p,[a=1-2],[],1,3)
        edge(p,[a=1-3],[],1,4)
        edge(p,[a=2-2],[],2,3)
        edge(p,[a=2-3],[],2,4)
        edge(p,[a=3-3],[],3,4)
        edge(p,[p=1-1,a=0-0],[],0,2)
        edge(p,[p=1-1,a=0-0],[],0,3)
        edge(p,[p=1-1,a=0-0],[],0,4)
        edge(p,[p=1-2,a=0-0],[],0,3)
        edge(p,[p=1-2,a=0-0],[],0,4)
        edge(p,[p=1-3,a=0-0],[],0,4)
        edge(p,[p=2-2,a=0-1],[],0,3)
        edge(p,[p=2-2,a=0-1],[],0,4)
        edge(p,[p=2-2,a=1-1],[],1,3)
        edge(p,[p=2-2,a=1-1],[],1,4)
        edge(p,[p=2-3,a=0-1],[],0,4)
        edge(p,[p=2-3,a=1-1],[],1,4)
        edge(p,[p=3-3,a=0-2],[],0,4)
        edge(p,[p=3-3,a=1-2],[],1,4)
        edge(p,[p=3-3,a=2-2],[],2,4)
```

Action Sequence Length	No of Edges	cpu time secs.	cpu/edge secs.
No Holes.			
1	6	0.05	0.0083
2	12	0.12	0.0100
3	19	0.25	0.0131
4	27	0.39	0.0144
5	36	0.58	0.0161
6	46	0.82	0.0178
7	57	1.09	0.0191
8	69	1.45	0.0210
9	82	1.85	0.0225
10	96	2.30	0.0239
11	111	2.92	0.0263
12	127	3.50	0.0275
13	144	4.28	0.0297
14	162	5.08	0.0313
15	181	5.93	0.0327
16	201	7.37	0.0366
17	222	8.47	0.0381
18	244	9.80	0.0401
19	267	10.98	0.0411
20	291	12.55	0.0431
Holes of length 1.			
1	6	0.04	0.0066
2	14	0.17	0.0121
3	26	0.45	0.0175
4	47	1.06	0.0225
5	87	2.50	0.0287
6	168	5.75	0.0342
7	337	13.73	0.0407
Holes of any length.			
1	6	0.05	0.0083
2	14	0.17	0.0121
3	28	0.48	0.0171
4	56	1.35	0.0241
5	119	3.73	0.0313
6	272	10.6	0.0389

Table giving the length of the action sequence used, total number of edges generated, total cpu time to build the chart, and the cpu time per edge in the chart.

Appendix XI

Chart Listing for a UNIX Command Sequence Containing a Typing Error

The following listing is for the edges of a chart for the comand sequence given in section 6.4.1. A simplified diagram of the chart is shown in fig.6.20, and the command sequence is given in fig.6.19.

Active edges:

1. 1-edge(rule 1,[rule 1,rule 133,lexical 912],toplevel,hypotheses([3]),[goal copy_tree _140259 at_location _140260 to _140244 at_location _140245 status _140229 = 3-4],[toplevel],3,4,[3],[])

2. 1-edge(rule 10,[rule 10,rule 133,lexical 912],redo copy_tree _140485 at_location _140486 to _140470 at_location _140471 status _140547,hypotheses([3]),[goal copy_tree _140485 at_location _140486 to _140470 at_location _140471 status _140455 = 3-4],[goal copy_tree _140485 at_location _140486 to _140470 at_location _140471 status _140547],3,4,[3],[_140455 = = failed])

3. 1-edge(rule 14,[rule 14,lexical 912],goal copy_tree _133886 at_location _133887 to _133871 at_location _133872 status _133858,hypotheses([3]),[copy_tree _133886 at_location _133887 to _133871 at_location _133872 status _133858 = 3-4],[change_directory _133871 at_location _133872 status _133858],3,4,[3],[])

4. 1-edge(rule 14,[rule 14,lexical 930],goal copy_tree _127740 to _127692 at_location _127693 status _127682,hypotheses([2]),[make_directory _127692 at_location _127693 status _127682 = 2-3],[copy_tree _127740 to _127692 at_location _127693 status _127682],2,3,[2],[])

5. 1-edge(rule 14,[rule 14,lexical 930],goal copy_tree _128220 to perquish at_location [john,jml,aipna3,usr,/] status _128159,interpretation,[make_directory perquish at_location [john,jml,aipna3,usr,/] status _128159 = 2-3],[copy_tree _128220 to perquish at_location [john,jml,aipna3,usr,/] status _128159],2,3,[2],[])

6. 1-edge(rule 15,[rule 15,lexical 930],goal create_file_copy _127890 to _127893 at_location [_127842|_127843] status _127832,hypotheses([2]),[make_directory _127842 at_location _127843 status _127832 = 2-3],[copy_file _127890 to _127893 at_location [_127842|_127843] status _127832],2,3,[2],[])

7. 1-edge(rule 15,[rule 15,lexical 930],goal create_file_copy _128055 to _128058 at_location [perquish,john,jml,aipna3,usr,/] status _127994,interpretation,[make_directory perquish at_location [john,jml,aipna3,usr,/] status _127994 = 2-3],[copy_file _128055 to _128058 at_location [perquish,john,jml,aipna3,usr,/] status _127994],2,3,[2],[])

8. 1-edge(rule 100,[rule 100,lexical 910],move_file_to_file _136431 at_location _136432 to _136416 at_location _136417 status _136403,hypotheses([3]),[copy_file _136431 at_location _136432 to _136416 at_location _136417 status _136403 = 3-4],[remove_file _136431 at_location _136432 status _136403],3,4,[3],[])

9. 1-edge(rule 100,[rule 100,lexical 910],move_file_to_file mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _136641,interpretation,[copy_file mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _136641 = 3-4],[remove_file mbox at_location [john,jml,aipna3,usr,/] status _136641],3,4,[3],[])

10. 2-edge(rule 1,[rule 1,rule 14,lexical 930,lexical 912],toplevel,hypotheses([2,3]),[goal copy_tree _138351 at_location _138352 to _138336 at_location _138337 status _138321 = 2-4],[toplevel],2,4,[2,3],[])

11. 2-edge(rule 1,[rule 1,rule 14,lexical 930,lexical 912],toplevel,hypotheses([3]),[goal copy_tree _139120 at_location _139121 to perquish at_location [john,jml,aipna3,usr,/] status _139090 = 2-4],[toplevel],2,4,[2,3],[])

12. 2-edge(rule 1,[rule 1,rule 15,lexical 930,lexical 910],toplevel,hypotheses([2]),[goal create_file_copy mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _143116 = 2-4],[toplevel],2,4,[2,3],[])

13. 2-edge(rule 1,[rule 1,rule 15,lexical 930,lexical 910],toplevel,hypotheses([2,3]),[goal create_file_copy _140764 at_location _140765 to _140740 at_location [_140743|_140744] status _140725 = 2-4],[toplevel],2,4,[2,3],[])

14. 2-edge(rule 1,[rule 1,rule 15,lexical 930,lexical 910],toplevel,hypotheses([3]),[goal create_file_copy _141599 at_location _141600 to _141575 at_location [perquish,john,jml,aipna3,usr,/] status _141560 = 2-4],[toplevel],2,4,[2,3],[])

15. 2-edge(rule 10,[rule 10,rule 14,lexical 930,lexical 912],redo copy_tree _138577 at_location _138578 to _138562 at_location _138563 status _138639,hypotheses([2,3]),[goal copy_tree _138577 at_location _138578 to _138562 at_location _138563 status _138547 = 2-4],[goal copy_tree _138577 at_location _138578 to _138562 at_location _138563 status _138639],2,4,[2,3],[_138547 = = failed])

16. 2-edge(rule 10,[rule 10,rule 14,lexical 930,lexical 912],redo copy_tree _138847 at_location _138848 to perquish at_location [john,jml,aipna3,usr,/] status _138912,hypotheses([3]),[goal copy_tree _138847 at_location _138848 to perquish at_location [john,jml,aipna3,usr,/] status _138817 = 2-4],[goal copy_tree _138847 at_location _138848 to perquish at_location [john,jml,aipna3,usr,/] status _138912],2,4,[2,3],[_138817 = = failed])

17. 2-edge(rule 10,[rule 10,rule 15,lexical 930,lexical 910],redo create_file_copy _141008 at_location _141009 to _140984 at_location [_140987|_140988] status _141079,hypotheses([2,3]),[goal create_file_copy _141008 at_location _141009 to _140984 at_location [_140987|_140988] status _140969 = 2-4],[goal create_file_copy _141008 at_location _141009 to _140984 at_location [_140987|_140988] status _141079],2,4,[2,3],[_140969 = = failed])

18. 2-edge(rule 10,[rule 10,rule 15,lexical 930,lexical 910],redo create_file_copy _141302 at_location _141303 to _141278 at_location [perquish,john,jml,aipna3,usr,/] status _141376,hypotheses([3]),[goal create_file_copy _141302 at_location _141303 to _141278 at_location [perquish,john,jml,aipna3,usr,/] status _141263 = 2-4],[goal create_file_copy _141302 at_location _141303 to _141278 at_location [perquish,john,jml,aipna3,usr,/] status _141376],2,4,[2,3],[_141263 = = failed])

19. 2-edge(rule 10,[rule 10,rule 15,lexical 930,lexical 910],redo create_file_copy mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _142932,hypotheses([2]),[goal create_file_copy mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _142825 = 2-4],[goal create_file_copy mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _142932],2,4,[2,3],[_142825 = = failed])

Inactive edges:

20. 1-edge(lexical 910,[lexical 910],copy_file _130890 at_location _130893 to _130929 at_location _130932 status _130884,hypotheses([3]),[word(command(cp,[[_130890,_130893,[],file,_130902],[_130908,_130911,_130914,_130917,_130920]],[(add_node,[_130929,_130932,[],file,_130941])],_130884,[cp,mbox,perqish]))=3],[],3,4,[3],[])

21. 1-edge(lexical 910,[lexical 910],copy_file mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _130371,interpretation,[word(command(cp,[[mbox,[john,jml,aipna3,usr,/],[],file,o],[perqish,[john,jml,aipna3,usr,/],[],unknown,unknown]],[(add_node,[perqish,[john,jml,aipna3,usr,/],[],file,o])],_130371,[cp,mbox,perqish]))=3],[],3,4,[3],[])

22. 1-edge(lexical 911,[lexical 911],copy_directory _130791 at_location _130794 to _130830 at_location _130833 status _130785,hypotheses([3]),[word(command(cp,[[_130791,_130794,[],dir,_130803],[_130809,_130812,_130815,_130818,_130821]],[(add_node,[_130830,_130833,[],dir,_130842])],_130785,[cp,mbox,perqish]))=3],[],3,4,[3],[])

23. 1-edge(lexical 912,[lexical 912],copy_tree _130692 at_location _130695 to _130731 at_location _130734 status _130686,hypotheses([3]),[word(command(cp,[[_130692,_130695,_130698,dir,_130704],[_130710,_130713,_130716,_130719,_130722]],[(add_node,[_130731,_130734,_130737,dir,_130743])],_130686,[cp,mbox,perqish]))=3],[],3,4,[3],[])

24. 1-edge(lexical 913,[lexical 913],copy_directory_to_file _130593 at_location _130596 to _130632 at_location _130635 status _130587,hypotheses([3]),[word(command(cp,[[_130593,_130596,_130599,dir,_130605],[_130611,_130614,_130617,_130620,_130623]],[(add_node,[_130632,_130635,_130638,file,_130644])],_130587,[cp,mbox,perqish]))=3],[],3,4,[3],[])

25. 1-edge(lexical 920,[lexical 920],list _125584 at_location _125585 status _125570,interpretation,[word(command(ls,[],[],_125570,[ls]))=1],[],1,2,[1],[])

26. 1-edge(lexical 930,[lexical 930],make_directory _126684 at_location _126687 status _126678,hypotheses([2]),[word(command(mkdir,[[_126684,_126687,[],unknown,_126696]],[(add_node,[_126684,_126687,_126711,dir,_126717])],_126678,[mkdir,perquish]))=2],[],2,3,[2],[])

27. 1-edge(lexical 930,[lexical 930],make_directory perquish at_location [john,jml,aipna3,usr,/] status _126519,interpretation,[word(command(mkdir,[[perquish,[john,jml,aipna3,usr,/],[],unknown,unknown]],[(add_node,[perquish,[john,jml,aipna3,usr,/],[],dir,x])],_126519,[mkdir,perquish]))=2],[],2,3,[2],[])

28. 1-edge(rule 133,[rule 133,lexical 912],goal copy_tree _134108 at_location _134109 to _134093 at_location _134094 status _134080,hypotheses([3]),[copy_tree _134108 at_location _134109 to _134093 at_location _134094 status _134080=3-4],[],3,4,[3],[])

29. 2-edge(rule 14,[rule 14,lexical 930,lexical 912],goal copy_tree _132481 at_location _132482 to _132431 at_location _132432 status _132418,hypotheses([2,3]),[copy_tree _132481 at_location _132482 to _132431 at_location _132432 status _132418=3-4,make_directory _132431 at_location _132432 status _132418=2-3],[],2,4,[2,3],[])

30. 2-edge(rule 14,[rule 14,lexical 930,lexical 912],goal copy_tree _133125 at_location _133126 to perquish at_location [john,jml,aipna3,usr,/] status _133059,hypotheses([3]),[copy_tree _133125 at_location _133126

to perquish at_location [john,jml,aipna3,usr,/] status _133059 = 3-4,make_directory perquish at_location [john,jml,aipna3,usr,/] status _133059 = 2-3],[],2,4,[2,3],[])

31. 2-edge(rule 15,[rule 15,lexical 930,lexical 910],goal create_file_copy _134831 at_location _134832 to _134772 at_location [_134775|_134776] status _134759,hypotheses([2,3]),[copy_file _134831 at_location _134832 to _134772 at_location [_134775|_134776] status _134759 = 3-4,make_directory _134775 at_location _134776 status _134759 = 2-3],[],2,4,[2,3],[])

32. 2-edge(rule 15,[rule 15,lexical 930,lexical 910],goal create_file_copy _135577 at_location _135578 to _135515 at_location [perquish,john,jml,aipna3,usr,/] status _135502,hypotheses([3]),[copy_file _135577 at_location _135578 to _135515 at_location [perquish,john,jml,aipna3,usr,/] status _135502 = 3-4,make_directory perquish at_location [john,jml,aipna3,usr,/] status _135502 = 2-3],[],2,4,[2,3],[])

33. 2-edge(rule 15,[rule 15,lexical 930,lexical 910],goal create_file_copy mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _136885,hypotheses([2]),[copy_file mbox at_location [john,jml,aipna3,usr,/] to perqish at_location [john,jml,aipna3,usr,/] status _136885 = 3-4,make_directory john at_location [jml,aipna3,usr,/] status _136885 = 2-3],[],2,4,[2,3],[])

Papers in Support of Candidature

Detecting and Modelling User's Beliefs about UNIX

In *proceedings of Institution of Electrical Engineers colloquium on Intelligent Tutoring Systems*. IEE Digest No. 1988/69

Detecting and Modelling Users' Beliefs about UNIX.

J.M.Lewis and P.M.Ross.
Department of Artificial Intelligence, Edinburgh University.

Abstract.

Users of the UNIX[1] operating system have misconceptions about UNIX commands, make errors in specifying commands, and have incomplete knowledge of the domain. Thus, their beliefs about the task in hand can differ from what is actually occurring. Our aim is to automatically detect and model the user's errors, misconceptions and extent of knowledge, so that this can be used to offer him individualised help or advice.

Our approach is to infer users' intentions from the commands that they issue, and use this information to detect their problems. An Active Chart Parser generates all possible parses according to a grammar of typical plans that users follow. The points when the user might need advice are detected from an heuristic analysis of the chart, and the goal that he is attempting to achieve is inferred. This goal is verified against a STRIPS-like model of the user's beliefs about UNIX commands, which determines the misconceptions and errors that the user could possess. This model of the user's "command beliefs" is specific to an individual, and is dynamically altered as the session progresses to account for the commands he issues.

Introduction.

People experience problems when using computer systems, making errors and possessing misconceptions about these systems. They also tend not to realise the potential of the systems because of their incomplete knowledge. Automated advice is required to help them to achieve their goals when they have a problem, or to improve their performance with the system. One such system is the UNIX operating system, with which users have particular problems [Norman 1981, Lewis 1985].

The detection and modelling of users' beliefs about UNIX is a prerequisite for offering advice. These beliefs must for the most part be inferred from the commands issued by the user. This paper is based upon the premises that:

i. There is a grammar which describes users' goals within the UNIX domain and their plans to achieve these goals. That is, users' behaviour can be described as a hierarchical goal / sub-goal tree which terminates in leaves corresponding to the semantic interpretation of UNIX commands themselves. We shall refer to this as the "plan grammar".
ii. Users experience problems in *achieving* their goals rather than in *formulating* them. Thus, all errors and misconceptions can be attributed to users incorrect beliefs about UNIX commands, rather than their beliefs about the plans and goals.

A program (the "UNIX Advisor") has been developed to detect and model the beliefs that users have about UNIX, determine when to give advice and the basis for this advice.

---

1. UNIX is a trademark of Bell Laboratories.

Plan Recognition.

The advice is derived by analysing different possible plans that a user could be following. These plan hypotheses are inferred from the commands that he has typed. First, multiple semantic interpretations are made for each command, which correspond to different possible expectations that the user could have. There are two types of interpretation:

i. A description of what would actually occur in the UNIX system, this being a fully instantiated semantic interpretation of the command (the " interpretation").
ii. Partially instantiated hypotheses which correspond to alternative semantic interpretations of the command.

For example, the command "cp mbox book" could be interpreted as:
The interpretation (to copy a particular file to a particular directory with fully instantiated preconditions and effects);

```
        copy-file-to-directory([mbox,[john],[],file,not-executable],
                               [book,[john],[ch1,ch2],directory,executable])
            effects([add-node([mbox,[john,book],[],file,not-executable]).
```

and the hypotheses (which have unspecified preconditions and effects);

```
        copy-file-to-directory(A,B) effects(C).
        copy-file-to-file(A,B) effects(C).
        copy-tree-to-tree(A,B) effects(C).
```

where: A, B and C are uninstantiated.

These interpretations are parsed using a chart parser and the plan grammar which recognises all possible plans and partial plans that the user could be following [Ross 1988]. The plan grammar must be context-sensitive so that invalid plans are not recognised. In the case of the UNIX Advisor this involves maintaining a record of the current filestore. However, it would be impractical to generate different contexts for each of the plan hypotheses in the chart. Therefore, only chart plans which do not contain hypotheses ("UNIX plans") take the contextual information into account, other plan hypotheses are allowed to combine freely. This leads to an over-generation of plan hypotheses in the chart, but these can be weeded-out later when they are found to be invalid accounts of the user's intentions.

The parsing process halts when no new plans or partial plans can be added to the chart, which is then analysed to determine whether the user requires advice, and the nature of that advice.

Giving Advice.

The user need not be aware that he would benefit from advice, so we cannot always rely on him to ask for it. Two instances of when a user might require advice are considered:

i. The user's plan deviates from what is actually occurring in the UNIX system, which may cause one or more of the commands to fail. In this case he has an underlying problem. Either he has made errors or he has been working with misconceptions about the preconditions or effects of the commands.
ii. The user has achieved a goal but could have done so more efficiently by using a plan of which he is unaware.

By analysing the chart using simple heuristics, the UNIX Advisor automatically detects these points at which the user requires advice.

Advice based upon an Underlying Problem.

A measure of the likelihood that the user is following a particular plan contained in the chart ("chart plan") is the number of user-typed commands occuring in that chart plan (that is, the "length" of the plan). Thus the longer the chart plan, and hence the number of user-typed commands accounted for, the more likely it is that this particular chart plan corresponds to the plan that the user is following ("user plan"). The longest UNIX plan is compared against the plan hypotheses. If a plan hypothesis is found which is longer, it is investigated to determine whether it is the user plan. This investigation involves the extraction of the semantic interpretations of the commands from the chart and their validation to ensure that they can be generated by the model of the user's beliefs about UNIX commands (the "user model"). This user model consists of a STRIPS-like representations of UNIX commands, which are manipulated to change the preconditions and effects of those commands and thereby account for the postulated interpretations. These interpretations must also be validated to ensure that there is a possible world in which such a command sequence could exist.

Once this stage has been reached, a decision must be made as to whether to enter a dialogue with the user to verify if this is what he was attempting to achieve. At present the UNIX advisor always enters a dialogue

with the user to identify his problems. However, in a full implementation of the system account would need to be taken about the importance of interrupting the user, whether to offer advice and how best to achieve this.

Advice based upon Re-Planning.

Again this heuristic uses the length of the chart plans to determine whether the user requires advice. If there is a completed goal which has been achieved by a UNIX plan, and there is a shorter plan hypothesis which could achieve this goal *about which the user is unaware*, then this plan hypothesis is investigated. The plan hypotheses are generated by applying the chart parser to develop plans "top-down" from the goal, and with uninstantiated command interpretationsin the chart. Partially instantiated command interpretations arise from this planing process, but they must be verified (by consulting the user model) to ensure that they are valid. Next, the command sequence is parsed with the context of a possible world to ensure that such a command sequence could exist.

Conclusion.

Using chart parsing and multiple semantic interpretations of commands, provides a powerful technique for recognising users' possible plans. Heuristics analyse the resulting chart to detect when the user might require advice, and the content of this advice is determined from consulting the user model.

References.

Norman D.A., 1981 "The Trouble with UNIX", Datamation, November 1981, vol.27 no.12.

Lewis J.M., 1985 "Analysing the Actions of UNIX Users". Edinburgh University. DAI Working Paper 188.

Ross P.M., Lewis J.M., 1988 "Plan Recognition for Intelligent Tutoring Systems" in Ercoli and Lewis (Eds) "Artificial Intelligence Tools in Education" PP.29-37. Elsevier Science Publishers B.V.

Plan Recognition for Intelligent Tutoring Systems

In *Artificial Intelligence Tools in Education.*
(Editors: Ercoli P., Lewis R.)
Published by Elsevier Science.

PLAN RECOGNITION FOR INTELLIGENT TUTORING SYSTEMS

Peter Ross and John Lewis
Department of A.I.
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN

ABSTRACT

Nearly all intelligent tutoring systems (ITSs) share the educational initiative with the user, to some degree. Often, the nature of the topic and of the tutorial strategy is such that the system could benefit from trying to recognise the intent behind the sequence of the user's actions so far. The actions might be natural language utterances, or commands to some simulation system where it is the ITS's job to comment on the exploratory use of the simulation, or commands to some other command-driven system whose underlying concepts it is the ITS's job to convey. A learner may find it as difficult, or harder, to tell the system what he is trying to do as to actually do it.

There have been many attempts to incorporate plan recognition algorithms into AI systems, but most have been based on rather limiting assumptions as far as an ITS is concerned, such as that the user has a single known goal driving his actions. More generally, a learner who is exploring his topic may have several goals, or none; he may suspend or abandon goals, sometimes without warning; his plans to acheive those goals may be bad, because of carelessness or misconceptions; and he may use unnecessarily complex plans. This paper gives a brief survey of existing methods of plan recognition that can contribute to run-time user modelling, and describes some work done at Edinburgh on such modelling of UNIX users that provides a convenient setting for exploring the general problem.

1. INTRODUCTION

Virtually all tutoring involves plan recognition of some kind. In a dialogue, it is obviously important to try to recognise the purposes and assumptions behind the sequence of the student's utterances, whether they are in natural language or in some more formalised language such as mathematical operations or operating system commands. In less highly interactive tutoring, where the student submits a completed piece of work for criticism (for example, a proof or a program), it is important to be able to recognise why the student chose one particular way of expressing himself, since this suggests his strengths and weaknesses.

Clearly, nearly all intelligent tutoring systems are likely to benefit from having some kind of plan recognition component. Most ITSs, considered from a mechanistic viewpoint, execute the following cycle after some initialisation:

- get some input from the user
- analyse it.

It may be input that stands alone, but has considerable internal structure, e.g. a Pascal program, or it may be part of a sequence of inputs that, by themselves, have little internal strcuture but collectively have a lot. Part of the analysis consists of trying to decide why the input(s) have one particular structuring rather than another. - comment on it, if appropriate and feasible to do so. This may involve setting new problems, or making Socratic comments or asking for clarification, for instance.Plan recognition is an important part of the analysis. It features in many other kinds of system too, for instance: - dialogue systems and 'story understanding' systems. See e.g. [Carberry 83 and 85], [Sidner 85], [Litman 86], [Pollack 86] and a very large number of others. The aim here is usually to recognize a speaker's intentions, in a local and/or global context within the dialogue, so as to be able to formulate a good reply. Surprisingly little of the work has been concerned with recognising inadequate plans. - strategic planning systems, where the aim is to recognise the purpose of other agents so as to plan to co-operate or to allow for the effects of those agents. The Intelligent Agents work at Stanford is addressing this problem, although there are many planning problems to be solved which affect the recognition task (see, for example, [Georgeff 84]). One way of avoiding the difficult job of plan recognition is, of course, just to ask the user. However, an intelligent tutoring system may not wish to trust a learner to articulate his plans. A monitoring or advice-giving system may not want to bother the user by asking for the information.

The prototypical task consists of trying to recognise the structure of some observed sequence, of actions or utterances, and so discern the purpose. The sequence may be complete or not, and may be guaranteed to have one purpose (perhaps known in advance), to have one or more purposes, or may not be guaranteed to have any purpose. However, most workers have assumed at least one purpose, and have also in general assumed competence on the part of producer of the sequence (a speaker, or an agent doing actions or issuing commands). At present the typical approach to plan recognition is either - generate plans, see which fit. This requires a plan generator (typically a grammar). The existence of multiple plans or of inadequate plans can complicate the matching stage tremendously. Shuffle grammars have been used to handle some aspects of these [Huff 82]. - try to parse the sequence according to some grammar. This is very sensitive to the nature of the grammar and of the parsing algorithm. Intention recognition then builds on the results of parsing or plan fitting. Little has yet been done to try to formalise plan recognition methods; see[Kautz 85, 86] for a survey and suggestions. See below for some further general suggestions.

2. SOME EXISTING SYSTEMS

In the following descriptions, some knowledge of simple ideas in planning systems is taken for granted. Unfortunately the AI literature on planning is still a mess, without even a consistent terminology. Only recently has there been much attempt at a proper formalisation - see, for instance, [Drummond 86]. For the most part, nothing more sophisticated than the traditional but limited STRIPS-like approach has been used. This has serious flaws. In particular, it does not extend well to handling mental operations such as deductions and decisions to gather information; nor does it cope well with representing beliefs, for instance that something may be believed until there is a need to accept something contradictory.

The systems mentioned below are suggestive, but the literature is too big for this to be remotely representative.

2.1 BELIEVER

The early work at Rutgers University, on BELIEVER, was unusual because it did not take the user's goal as something given. The goal was inferred either from the context of the action sequence (in a kitchen, cooking is the normal goal) or from certain suggestive aspects of things involved (there's only a few things you might be doing with a frying pan). This does limit the system to looking for 'typical' goals - considering the use of a frying pan as a paperweight or as a makeshift hammer could only come from considering a rather fine level of detail, and the space to be searched there would be prohibitively big.

The data supporting plan recognition included a world description involving objects and relations betwen them, collections of likely goals and general plan fragments and so on, and a separate user's model of the world. The user's model resembled the actual world model, although using beliefs rather than unequivocally true data. This user's model was used as follows. Assuming a goal, generate a plan by the usual depth-first search methods. Any precondition, however, was checked against the user's model to see if the user was supposed to believe it true already; if so, no further planning was needed for that precondition. Thus the generated plan fitted the user's beliefs rather than the truth. The plan was kept as general as possible, and matched against the user's actions. Any unmatched action might be handled in one of several ways:

- it might be demonstrably irrelevent to the goal, and so could be 'explained' as a self-contained extraneous action.
- it might be possible to make the generated plan more specific in such a way as to bring in the need for the action.
- it might be possible to amend the user's model of the world suitably, so that the action was entailed by a newly-generated plan.

Clearly, there are many details omitted from this superficial account, such as what the initial user's model was and how amendments to it might be suggested. See [Schmidt 78, Sridharan 77, 83a, 83b] for more details.

2.2 The MACSYMA Advisor

This system [Genesereth 82] is intended to offer advice to novice users of the MACSYMA system for manipulating algebraic expressions. The Advisor is given information about the user's goal, and his sequence of actions to achieve that. It assumes that the actions are rational and that there is a plan to be found, and tries to determine what misconceptions might have led to the user's use of an inadequate plan. At the lowest levels, the system consults a database of what it is thought the user knows; the database is accessed by a single procedure 'fetch'. Misconceptions are assumed to be of a single form, namely that the analogous 'fetch' in the user's own head is returning the wrong sort of result. Plan recognition is used to try to comprehend the user's actions as far as possible, by a combination of top-down and bottom-up methods. An inbuilt planner, MUSER, is used top-down to expand from the general goal down to a more detailed level, though still a level with many unbound variables in it. Bottom-up analysis is used to try to fill in bindings as well as to suggest plan fragments; the latter is possible because certain actions in MACSYMA can be strongly suggestive of certain local intentions. A choice between competing candidates for the user's actual plan can be resolved simply by asking the user. Once plan recognition has reached a level of detail that mentions calls to 'fetch', the misconception detection stage can start.

2.3 POISE

POISE [Carver 84] is an intelligent user interface system; the prototype was for an intelligent assistant for an office automation system. It is based on a blackboard architecture, underpinned by a truth maintenance system (TMS), and uses application-specific heuristics for rapid pruning of the potentially vast number of plan interpretations for a given series of user actions. The TMS makes it possible to reason about the assumptions underpinning a possible interpretation of the actions. In this blackboard framework, there are explicit 'focus of attention' and 'prediction' databases as well as the customary plan library or planner. The example heuristics given in [Carver 84] give details such as 'this action can start/end plan X', 'this is the mostlikely explanation for action A in context C' and so on; in particular they give a partial ordering on the plans that an action might be part of. There are also control heuristics such as 'the user is less likely to start a new plan than to continue an existing one' and 'a single action is more likely to be part of one plan than several'.

2.4 PROUST

PROUST [Johnson 85] commented on semantic flaws in novices' Pascal programs written in response to chosen programming exercises; such programs were guaranteed syntactically correct by a standard compiler. Its knowledge of the specific problem was encapsulated as a set of goals and constraints entered by the teacher; it was of course sensitive to the correctness and form of this data, as expressed in a simple formal language. Given this

data, it used a library of plans to generate possible ways of writing a program to solve the problem. Rather than generate a fully detailed program and then match it against the user's one, the matching began at a very general level and proceeded as long as there was a reasonably good fit, in order to prune the size of the search space to manageable levels. Having found the best fit, a library of common bug types was searched in an attempt to account for any discrepancies. This gave a basis for making comments to the user. PROUST is not a genuinely interactive program, although work continues to make it part of a proper ITS.

3. A GENERAL APPROACH

Our own interest in plan recognition stemmed from a recent project to model aspects of users of UNIX as they worked, in which one of our interests was in trying to find the best and the possible explanations for a user's commands so far. The model consisted of mutually consistent sets of hypotheses on a blackboard - see [Ross 87] for more information. The unusual aspects of plan recognition in this area are that a user may have a plan, may have a bad plan, may be trying to form a plan (or recover from one) by gathering information and exploring, may switch goals without warning, either postponing or abandoning earlier goals, and may also give essentially purposeless commands as far as his goals are concerned (e.g. "fortune" on UNIX prints a random joke from a large library of them; many users run the command for relaxation or for a break in effort).

3.1 Using a chart parser: the ideas

Our starting point is, as with other systems, a grammar-based method, with possible user goals appearing at the top of the grammar. However, we employ an active chart parser to drive the recognition. A short account of chart parsing can be found in [Winograd 83] or [OShea 84]. The grammar (not limited to being context-free) can be explicit, or implicit in semantic information, and may or may not be complete. The advantages of a chart parser for plan recognition include: - the edges of the chart show the recognizable fragments of a complete parse, and carry complete state information about that fragment so that a dependency record system is unnecessary for this stage. Because of this independence of edges it is easy to weed out moribund edges or edges deemed unfruitful on semantic or other grounds. Henry Thompson has pointed out, in conversation, that this independence makes the approach apt for use in parallel or distributed systems. By using chart parsing, you have already paid the price of providing a local context. - the parsing strategy can be changed while parsing, to focus effort on parts of the sequence. For example, there may be good reasons to switch between bottom-up or top-down, breadth- or depth-first, left-right or right-left, or the many variants of these. - the 'fundamental rule' can be modified so that edges need not actually meet at a member of the sequence in order to combine. This gives a simple way of handling suspended goals resumed later. For example, you might require edges to be no more than "N" tokens apart if they are to be considered for combination; this might be weakly justified on grounds of human attention span. As yet we have not tried run-time modification of the 'fundamental rule', with the attendant problems of guaranteeing that no parses are missed. This problem may be less serious than for linguists, since in plan recognition there is usually no unequivocal 'winning' parse, just a currently most credible one, since there is usually no marker that delimits plan sequences; and presumably the rule modifications would be guided by credibility considerations as well as complexity ones. - gaps in recognising a fragment can guide the detection of plans with missing steps, and can guide the detection of essentially purposeless actions. For example, suppose that a plan rule is (to simplify greatly, and consider a trivial context-free rule)

task(a) = > a1, a2, a3

and the parser has recognised a1 and a3 as complete fragments in the sequence so far. What intervenes between the edges for a1 and a3 may not concern the task(a) task or its parents at all, suggesting a missing step and an intervening goal if that subsequence parses suitably or a purposeless step or misconception if it doesn't. On the other hand, what intervenes may, on analysis, turn out to be semantically equivalent to the 'missing' a2. Thus the parsing process can help the recognition of a valid plan without needing to contain all the equivalents of a step explicitly. If, on the other hand, the parser has recognised a1 and a2 but not a3 so far, this is nevertheless suggestive of the user intending to do task(a) step; the connection can easily be extracted from the edges in the chart so far.

- multiple edges stemming from one element of the sequence can suggest that the element has multiple purposes, if the user has interleaved plans for separate goals.
- incremental parsing is possible.

The plan recognition process can therefore develop as new elements are added to the sequence (for instance, when the user gives another command to the system). The search strategy and halting criterion can be varied to focus recognition effort appropriately.

A chart parser could be viewed as a specialised form of blackboard system,but the independence of edges makes for easier control decisions. The blackboard-based methods of [Carver 84] can be transplanted directly into chart form.

It is important to realise that the chart parsing does not do the whole job of plan recognition, but it provides a suitably digested form of the data to facilitate the recognition. That the process does depend on the nature of the grammar is clear; in the example above, if parsing is going from left to right within the rule and a1 is a missing step, then the connection between a2 and a3 is harder to establish since they will not appear within one edge. Using a binary grammar, and/or a grammar in which every right-hand side begins with a terminal, would avoid this at the expense of weakening the correlation between rules and 'natural' plans; indulging in grammar conversions requires substantial bookkeeping to maintain those correlations. A purely syntactic grammar also fails to capture

the notion of the relative significance of a step, yet this is very important. The grammar cannot by itself be the bottom line of the recognition process, although it may be a large part.

3.2 Using a chart parser: the current system

We are using an active chart parser written in Prolog, currently with an explicit grammar - every rule is tagged by an arbitrary Prolog term, so providing a simple rule filtering mechanism. The symbols used in the grammar rules can be arbitrary Prolog terms; use of compound non-ground terms is the main method of providing context sensitivity. Edge independence is maintained by term copying, using structure smashing rather than the common assert/retract hack; the parser is free of side effects. There are checks against duplication of edges, whether from duplicate rules or from different 'evolutionary paths'. Bidirectional parsing could thus be added at trivial cost, essentially by tagging each edge with an expansion direction and trivially modifying the fundamental rule.

We are using the parser as part of a system to detect misconceptions underlying a UNIX user's commands. The commands are represented in a STRIPS-like way. The current system uses the method suggested above for recognising cliches not explicitly encoded in the grammar, by looking for context-dependent equivalence of an actual subsequence to a subsequence suggested by the grammar but not found. To give a simple example, it is capable of recognising that the sequence

.... {active in directory 'old'}
cd {change to login directory}
cd new {change to directory 'new'}

is a cliche for 'change directory from old to new'. The system initially notes this specific cliche, in the context of 'old' and 'new', but can generalise it to cover all instances if further examples in different contexts occur later. We have logged volunteers' commands for several years; cliches such as this or using 'copy then delete original' instead of 'rename' are common, but it would be somewhat limiting to encode them specifically within the recognition system. There are other actions which are not cliches to the user, but which are to the plan recognition process - a trivial example would be that 'copy', 'edit' or numerous other commands could do the job of file creation required by a plan step.

One of the aims of the current system is to recognise a user's misconceptions by spotting bad plans and seeing what changes to the correct representation of commands could make that bad plan into a good one but in a new world (cf[Genesereth 82]). So far, it works in a small subdomain of UNIX commands. This is, however, still a rather limited interpretation of what a misconception is, although able to cope if the user rectifies his beliefs. The wider issues here are outside the scope of this paper.

4. CONCLUSIONS

Using modified forms of chart parsing seems to be a very promising way of driving the plan recognition process. The approach seems capable of handling previously troublesome aspects such as inadequate plans or unnannounced changes or suspensions of goals, or multiple goals. It is also conceptually simpler than the circumscription-based approach of [Kautz 86].

As we said earlier, the chart provides a suitably digested form of data for plan recognition. There are many issues left for us to explore; some concern control, some concern efficiency and some are more domain-dependent matters. So far, it has proved very worthwhile.

5. REFERENCES

[Drummond 86] Drummond M.E., 'A representation for action and belief in automatic planning systems', Proceedings of CSLI/AAAI Worshop on Planning and Action, Oregon USA, 1986 (also AI Applications Institute TR-16)

[Carberry 83] Carberry S., 'Tracking User Goals In An Information-Seeking Environment', Proceedings of AAAI-83, 59-63, 1983

[Carberry 85] Carberry S., 'Pragmatic Modelling in Information Systems', University of Delaware PhD thesis, 1985

[Carver 84] Carver N.F., V.S.Lesser and D.L.McCue, 'Focussing in plan recognition', Proceedings of AAAI-84, 42-48, 1984

[Genesereth 82] Genesereth M.R., 'The role of plans in intelligent teaching systems', in Sleeman (ed), Intelligent Tutoring Systems, 137-156, Academic Press, 1982

[Georgeff 84] Georgeff, M., 'A Theory Of Action For MultiAgent Planning', Proceedings of AAAI-84, 121-125, 1984

[Huff 82] Huff K. and V.Lesser, 'Knowledge based command understanding: an example for the software development process', TR82-6, COINS, University of Massachusetts at Amherst, 1982

[Johnson 85] Johnson W.L., 'Intention-Based Diagnosis of Errors in Novice Programs', PhD Thesis, Yale University, 1985. Also published by Morgan Kaufman, 1986

[Kautz 85] Kautz H., 'Toward a theory of plan recognition', Computer Science Dept. TR162, Rochester University, 1985

[Kautz 86] Kautz H. and J.Allen, 'Generalized plan recognition', Proceedings of AAAI-86, 32-37, 1986

[Litman 86] Litman D., 'Understanding Plan Ellipsis', Proceedings of AAAI-86, 619-624, 1986

[OShea 83] OShea, T. and M.Eisenstadt, Artificial Intelligence: Tools, Techniques and Applications, Harper and Row, 1983

[Pollack 86] Pollack M., 'Inferring Domain Plans In Question Answering', University of Philadelphia PhD thesis, MS-CIS-86-40, 1986

[Ross 87] Ross P.M., J.G.JOnes and M.Millington, 'User modelling in intelligent teaching and tutoring', in Trends in Computer Assisted Education, Lewis and Tagg (eds), 32-46, Blackwell, 1987

[Schmidt 78] Schmidt C.F., N.S.Sridharan and J.L.Goodson, 'The plan recognition problem: an intersection of psychology and artificial intelligence', Artificial Intelligence, 11, 45-83, 1978

[Sidner 85] Sidner C.L., 'Plan parsing for intended response recognition in discourse', Comp. Intelligence 1, 1-10, 1985

[Sridharan 77] Sridharan N.S. and C.F.Schmidt, 'Knowledge-directed inference in BELIEVER', in Pattern-directed inference systems, Waterman and Hayes-Roth (eds), Academic Press, 1977 (also available as report CBM-TR-75, Rutgers University)

[Sridharan 83a] Sridharan N.S. and J.L.Bresina, 'Knowledge structures for a modular planning system', Report CBM-TR-133, Rutgers University, 1983

[Sridharan 83b] Sridharan N.S., J.L.Bresina and C.F.Schmidt, 'Evolution of a plan generation system', Report CBM-TR-128, Rutgers University, 1983

[Winograd 82] Winograd T., Language As A Cognitive Process, Addison-Wesley, 1982