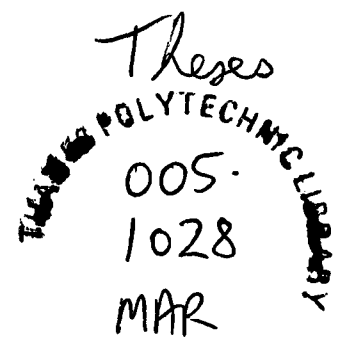
TUTORING SYSTEMS BASED ON USER-INTERFACE DIALOGUE SPECIFICATION

FRANK MARTIN

A thesis submitted in partial fulfilment of the requirements of the Council for National Academic Awards for the degree of Doctor of Philosophy

August 1990

Thames Polytechnic, London



CONTENTS

Acknowledgements

I wish to thank Dr. du Boulay for all his support during the protracted period of this research and especially for agreeing to supervise it in the first place. Thanks are due also to Professor Cross for suggesting that earlier tutoring work on the Polytechnic Prime computers could be turned into a research project.

Lastly I must thank both my wife Jean for her perceptive insights on the first draft and colleagues who read and commented upon it.

Abstract

Tutoring Systems based on User-Interface Dialogue Specification

F. A. Martin

This thesis shows how the appropriate specification of a user interface to an application software package can be used as the basis for constructing a tutorial for teaching the use of that interface. An economy can hence be made by sharing the specification between the application development and tutorial development stages. The major part of the user-interface specification which is utilised, the task classification structure, must be transformed from an operational to a pedagogic ordering. Heuristics are proposed to achieve this, although human expertise is required to apply them. The reported approach is best suited to domains with hierarchically-ordered command sets.

A portable rule-based shell has been developed in Common Lisp which supports the delivery of tutorials for a range of software application package interfaces. The use of both the shell and tutorials for two such interfaces is reported. A computer-based authoring environment provides support for tutorial development.

The shell allows the learner of a software interface to interact directly with the application software being learnt while remaining under tutorial control. The learner can always interrupt in order to request a tutorial on any topic, although advice may be offered against this in the light of the tutor's current knowledge of the learner. This advice can always be over-ridden.

The key-stroke sequences of the tutorial designer and the learner interacting with the package are parsed against an application model based on the task classification structure. Diagnosis is effected by a differential modelling technique applied to the structures generated by the parsing processes.

The approach reported here is suitable for an unsupported software interface learner and is named LIY ("Learn It Yourself"). It provides a promising method for augmenting a software engineering tool-kit with a new technique for producing tutorials for application software.

List of abbreviations used

ACT	Adaptive Control of Thought
AI	Artificial Intelligence
ASCII	American Standard Code for Information Interchange
ATN	Augmented Transition Network
CAL	Computer-Assisted Learning
CASE	Computer-Aided Software Engineering
CLG	Command Language Grammar
DYCAL	DYnamic Computer-Assisted Learning
I/O	Input/Output
ITS	Intelligent Tutoring System
LIY	Learn-It-Yourself
OOPS	Object-Oriented Programming System
TICCIT	Time-shared Interactive Computer-Controlled Information Television
VDM	Vienna Development Method

Chapter 1

Introduction

1.1 Background

Computer users are becoming increasingly sophisticated. As they operate ever-improving hardware they provide software suppliers with a market for new products which is evolving continuously. The increase in the number of computers in use and the number of software products to run on them has led to an explosive growth in training requirements. The future is clear: computers can and will provide the tutorial means for users to learn how to use unfamiliar software. If this seems to be rather a sweeping statement let us consider the alternatives. The traditional approach has been to send the learner on a commercial course. If the learner is being sent by his or her employer then that organisation has to meet not only the very significant expense of the course itself but also the cost of losing the services of the employee for the duration of the course. Many potential users will not be able to seek funds from an employer to go on a course: professionals in non-computing disciplines, for example, learning a new package in their own time, or someone at home improving their "computer literacy" skills on a domestic computer. For them, self-tuition will be the only way. Self-tuition may not mean the use of a computer: books provide a traditional way of disseminating information by self-study. When learning a skill, however - and using a software interface is principally a cognitive skill with a small motor element - learning by doing is superior to learning from written material, at one remove from the subject matter.

The work portrayed below describes this "learning by doing" in terms of presenting the learner with a structured view of the domain which is to be mastered - the *target package* - coupled with appropriate tutoring material. The learner interacts with the user interface of the software package being learnt and it is the structure of this *software interface* which underlies the view of the target package which is presented to the learner. The approach is called **"Learn-It-Yourself"**, or **LIY** for short. For a different class of learner - the child in school - the idea of presenting a relatively unstructured learning environment has been proposed (Papert 1980). The motivating features for the child - using a simple graphical programming language called LOGO - are assumed to rest on the fun involved and the child's natural inquisitiveness. Learning objectives

relate to developing simple arithmetic and spatial ability. At present such an unstructured approach is usually inappropriate to the learner of a new software interface - particularly if it is text-based - due principally to the complexity of such interfaces. This could change somewhat in the future as user interfaces become more heavily based on graphical paradigms. These provide a relatively small number of tools of universal applicability which can be put together by the user to provide powerful facilities. This "putting together" of a small number of tools leads to a large number of features, many of which are best learnt by experiment. An example of this is the drawing tool Microsoft Windows Paint which provides a huge range of facilities for drawing, since the user can combine different styles, fonts, palettes, brushes and so on. The manual for this tool is only 37 pages long; although it is a powerful package it is best learnt by exploratory trial-and-error.

Microsoft Windows Paint is simply "graphical interface"; the graphics facilities dominate the package and apart from filing system features there is very little else. This type of software is at one extreme compared with a purely text-based interface which manipulates a complex computer system, for example, the concepts of which the learner must acquire through a training sequence. It is plausible to suggest that future systems are likely not be at either of these two extremes but somewhere in between. It will thus be appropriate to present the learner of such a system with a structured view of the domain to be learnt, with opportunities to experiment with the software in a protected environment which will not permit damage to occur to the machine's software systems as a whole.

LIY is a method, based upon a software tool-kit, for engineering the type of courseware that is specifically designed to teach the use of a limited class of *software interfaces*. The tool-kit comprises courseware authoring and delivery environments. As an example, LIY could be used to teach the use of a new word-processing or database management system. Software interfaces are usually *task-oriented*, in that operating them can be viewed as carrying out a sequence of actions to achieve a task. Thus the LIY approach is appropriate for task-oriented domains. Considered from a methodological viewpoint, LIY would not be appropriate for more open-ended domains such as history or geography.

Furthermore, LIY is only designed for dealing with text-based interfaces. At present it normally requires that all application input (i.e. semantic input to the application rather than command input which interrogates or controls it) be terminated by a recognisable character, such as *enter* or *escape*. LIY cannot usually deal with fixed-length input not followed by a recognisable terminator although this can occur on occasion as discussed in chapter 3. The current version of LIY has no method of managing "hot keys". These are certain pre-determined key-stroke sequences which always suspend the current task and invoke some standard associated service. The most common example is the constant availability of a certain key - often F1 in PC-based software - which enables the user to seek help. LIY will not handle software interfaces incorporating direct-manipulation devices such as mice. O'Shea has pointed out some of the difficulties associated with modelling the users of such devices (O'Shea 1989). As software interfaces become increasingly graphics-based these difficulties will assume more importance. They are discussed further in section 5.4.1.

It is proposed above that the computer itself is the natural delivery medium for tutoring the learner of a new software interface. This idea is not new. The LEARN system of UNIX (Kernighan and Lesk 1979) and the DYCAL system for Prime computers running the PRIMOS operating system (Martin 1983) both provided a tutorial environment with controlled embedded access to the user interface. That is to say, learners could be set assignments in which they were requested to manipulate the interface to an actual program rather than, for example, a simulation of that interface. The tutorials were designed to teach the use of the operating systems themselves. More recently application packages such as Lotus 1-2-3 and WordPerfect have been released with built-in tutorial assistance. These tutorials allow the learner to interact with what appears to be the genuine application software. Typically, only a restricted subset of the operations that can normally be performed is available, making learner control difficult and browsing by the learner impossible. A more fundamental problem is that there appears to be a very tight coupling between the application and its tutorial. Tight coupling precludes the development of a tutoring system for software interfaces which is portable in the sense of being applicable over a

range of interfaces. LIY adopts a "loose coupling" approach in order not to preclude portability. Finally, existing systems for commercially-available application software do not use any ITS technology such as sophisticated student modelling or diagnosis. Diagnosis, when it occurs, is very much at the level of matching character-strings.

It is not intended that the reader should infer that written material has no place in tutorials for software interfaces. On the contrary, written text and graphics provide extremely valuable input to the whole self-teaching process since currently it is easier for most people to find a required page in a book than to find a particular screen. This may change as a result of research into hypertext systems. It is a moot point whether the book supports the computer-based tutorial or the tutorial supports the book. Let it be said that they complement one another.

Other workers have attempted different approaches to producing tutorial material for software interfaces. The DOMINIE system has a knowledge-elicitation phase which captures static screen-dumps from the application (Spensley and Elsom-Cook 1988). These can then be displayed to the learner as part of an appropriate teaching operation. The DOMINIE work focuses upon the representation of multiple teaching strategies and the selection of the most appropriate such strategy. It does not however permit the learner to interact directly with the software being taught, unlike the LIY approach described here.

For software interfaces, alternatives to tutoring systems are *advice* systems. These can permit the user - perhaps a novice - to interact directly with a program but allow him or her to interrupt in order to seek advice. This is exemplified by the EMACS editor (Stallman 1979). An alternative design is for systems which themselves give advice at what are considered appropriate moments. The possible design of one such system is outlined in the context of a tutorial for WordStar (Jackson and Lefrere 1984). The approach proposed is based on the maintenance of plan representations of hypotheses concerning the user's state. These plans could then be revised dynamically. Greenfield describes an approach to plan generation based on Definite Clause Grammars

(Greenfield 1988), a representation formalism particularly suited to processing by a Prolog interpreter. This technique is used to represent pre-defined user plans and to parse command line input, in this case to UNIX. Also for UNIX, the EUROHELP system is a 100 person-year project which is proposing an intelligent help system for UNIX mail (Breuker 1988). This important undertaking is examining many different aspects of ITSs for advice systems, such as plan generation, discourse and - obviously - aspects of explanation. The SINIX Consultant is an intelligent help system for SINIX - a UNIX derivative developed by Siemens AG (Kemke 1987). It is reported to be a command-based taxonomic hierarchy, similar to that of LIY, and permits the user to ask questions in natural language. Knowledge for answering these questions is held in frame-like representations at the nodes in the taxonomy. Woodroffe describes the FITS system which is a tutor for the UNIX command line interface (Woodroffe 1988). The thrust of this work again focuses upon planning with the program maintaining a hierarchy of increasingly abstract possible plans. These are hypothetical representations of the learner's actual plan.

Jackson and Lefrere provide an interesting analysis of some of the difficulties of matching a hypothetical plan to users' actions and revising such a plan if necessary. These include the user: (i) changing goal; (ii) adopting an alternative strategy, not in the plan, to achieve the same goal; (iii) incorporating another task into the original plan; (iv) making an error, for example typing the wrong command or typing a series of commands in the wrong sequence.

The TOTS system (Rickel 1988) shares some similarities with the LIY work described here in that it attempts to provide a domain-independent intelligent tutoring shell for task-oriented domains. Both the FITS and TOTS approaches base their plan representations upon Sacerdoti's *procedural network* (Sacerdoti 1977). LIY is aimed at a subset of such domains: user interfaces to software. Like LIY, TOTS is weak in the area of identifying learner misconceptions, principally because these are particularly domain-specific. It is unclear whether TOTS could be used for teaching the use of software interfaces. Rickel does not report any evidence that it would be able to do so in a manner which would support direct interaction with the target software.

It can be argued that to learn a programming language is also to learn the use of a software interface. The best-known work in this area is PROUST (Johnson and Soloway 1987), a tutoring system for teaching Pascal. The Pascal compiler is simply a "black box" which takes program statements as input and produces machine-code and error messages as output. Like the compiler, PROUST processes a complete (though syntactically correct) Pascal program. It attempts to identify and report semantic errors by comparing such mistakes with a "bug catalogue" of known possible errors. It is a non-interactive program, whereas the work described in this thesis is suited to tutoring *interactive* software interfaces. PROUST incorporates knowledge both about Pascal and about the typical bugs learners make when developing Pascal programs.

The Lisp Tutor, based on the ACT* theory of learning (standing for *Adaptive Control of Thought*), is also concerned with teaching a programming language (Anderson and Reiser 1985). However, unlike PROUST which operates post-hoc after the student has submitted a complete program, Anderson's tutor deals with the interactive environment of a Lisp interpreter. Errors are detected and reported immediately they are committed. Further, the learner is required to repair such errors at once. The LIY approach described below can be applied to a wide range of software interfaces, admittedly of less complexity than a Lisp interpreter. For a given cost of implementation, there appears to be a trade-off between the power of a tutor and its complexity on the one hand and its generality over a range of domains on the other. The Lisp Tutor is towards the high end of the implementation cost scale. Figure 1.1 shows how implementation costs change with respect to distance from a *line of constant cost* and attempts to position PROUST and Anderson's approach as used in building the Lisp Tutor in relation to LIY, which exhibits low cost and high generality but only moderate power.

LIY's aims are set out fully in the next section. It has so far been described in terms of computer-based delivery of tutorial material concerned with software interfaces; it also attempts to provide an *authoring* environment for building such tutorials.

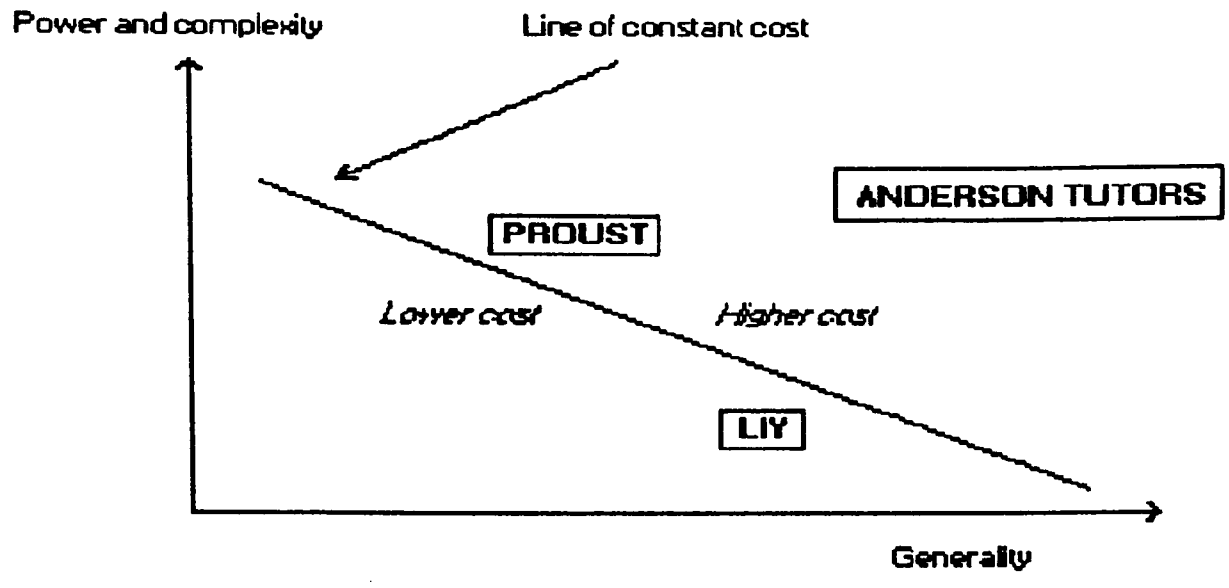


Fig. 1.1

In the authoring field, Tang *et al.* propose object-oriented tools for modelling users and dialogues, with a mapping between the two (Tang *et al.* 1989). These tools would then prompt the designer for domain-specific information.

Woolf describes an authoring system being built to enable teachers who are not familiar with AI programming to construct intelligent tutorials (Woolf 1987). The interface to the system is predominantly graphical. Woolf identifies the main problem in the building of such authoring systems as being that of domain knowledge representation. As is described in succeeding chapters, the LIY representation of the domain is based on the specification of the user interface to the application software. LIY assumes that this will have been defined at the software design stage.

The TEACHER'S APPRENTICE system once more proposes an authoring environment with a highly graphical interface (Lewis *et al.* 1987). The chosen domain is the familiar one of school algebra. This tutor, like the Lisp Tutor (Anderson and Reiser 1985), is based on the ACT* theory of cognition in which

pre-stored fine-grained production rules model all aspects of the learner's behaviour, both correct and "buggy". These rules must be elicited from the designer. The tutoring strategy is said to be "induced" from the behaviour of the designer playing the roles both of teacher and ideal student. In fact the designer must specify correct and incorrect protocols in order for the system to generate the productions required.

SCALD uses a script-based expert system to support the tutorial designer which incorporates knowledge about how to build CAL systems (Nicolson and Scott 1986). It does not support an intelligent tutorial delivery environment, nor is it specifically aimed at software interfaces.

The systems described above all focus on one or more of the accepted issues in ITS design as a means of investigating and refining approaches to handling those issues: domain and learner modelling, planning, teaching strategies, problem generation, natural language interfaces, representation of teaching expertise and so on. In contrast the LIY research reported here is concerned specifically with software interfaces and their representation and asks the question "How can this representation - which will already exist - be exploited in the cause of tutorial creation and delivery?".

1.2 Aims and rationale

Section 1.1 above describes the area of this research and the background in terms of related research. This section provides a more focused view of the aims of the research and explains the rationale for carrying it out.

The aims are considered first. There were two principal aims, along with four more general ones. The principal aims were:

1. **To investigate the extent to which a *specification* of the user interface can be used as the basis for building a tutorial for that interface.**

If it were the case that a user-interface specification could contribute to construction of a tutorial for that interface, then this would save effort: for some given project the specification would be contributing to both the software engineering and the *courseware engineering* stages. Such an economy could only occur when the project development provides not only for the software but also its tutorial courseware. No economy would occur if the tutorial were to be implemented retrospectively as an afterthought.

2. **To demonstrate the feasibility of a portable shell for supporting the production and delivery of tutorials for software interfaces.**

This aim tests the domain-independence of the LIY approach. Most of the tutors being discussed in the current literature are for single domains such as the teaching of algebra. The work described here is concerned with developing a method with wide applicability. If the research had been focused on aspects of design of a tutor for just one particular software interface then the issues the work would have addressed would have been those confronting the ITS research community in general. These issues have not gone away simply because of the portability aim of LIY. Some of them are considered in LIY's design and are discussed in detail in chapters 3 and 4. Others are discussed in the concluding

chapter. Please note, though, that the desire for portability pervades the entire LIY conception.

There were four subsidiary, or more general, aims which were kept in mind as being desirable:

(a) The learner should be allowed to interact directly with the software interface being taught.

The reasoning behind this is that learning by "doing" is very effective. Learning by interacting with a simulation could be as good provided that the simulated interface was as good as the real thing - a situation appropriate on cost grounds to teaching airline pilots, but hardly to teaching software interfaces. The learning experience of interacting with static screen dumps, as for example in DOMINIE, is likely to be of lower quality (Spensley and Elsom-Cook 1988).

(b) The learner should be able to interrupt at any time.

On the face of it learner control does not seem to have provided the breakthrough in CAL acceptability which was hoped of it and it would be instructive in the future to analyse the reasons. Merrill points out in his study of learner control in the TICCIT system that distinctions can be made between learner control of strategy, presentation and content (Merrill 1980). In LIY the learner essentially has control over strategy, and the case for using it rests on the high level of motivation anticipated of the learner. In a study by Hartley it was found that a group of students offered learner control reported a greater degree of stimulation and satisfaction than a similar group learning the same material under program control, factors which are clearly concomitant with maintaining a high degree of learner motivation (Hartley 1981). More recently Hartley and Tait report experiments with a system offering both learner-control and advice in the domain of mathematics for biology students (Hartley and Tait 1986). While the system was liked by the students, there was some concern as to whether it met the particular requirement of stimulating thought and reflection in this particular domain. The authors propose a design incorporating a

knowledge base to support the advisor which would enable it to probe deeper issues concerning the student's understanding.

Anderson has pointed out the importance of control to the learner even if this control is illusory (Anderson 1989). He described an informal experiment in which a lecturer was giving the last lecture of a course. He divided the students into two groups (group 1 and group 2). He needed to teach only one of two possible topics. Group 1 were allowed to choose the topic while group 2 were to be denied a choice although they were taught the same topic as group 1. The result was that group 1 performed better in post-tests than group 2, who were denied even the illusion of having some control over what was to be taught.

MATILDA, a system for teaching Lisp to novices, was apparently not as successful as the system used in Hartley and Tait's study (Elsom-Cook 1983). The learners, who were students on a taught M.Sc. course in cognition, computing and psychology, were largely computer-illiterate, and were inhibited about trying things to find out how MATILDA worked. It can be inferred that the cognitive load associated with learner control was relatively large compared with the cognitive content of the learning task. The learner perceived "learning the task" as being more important and therefore opted to minimise cognitive load by avoiding learner control. For software interfaces the LIY approach should help to overcome this problem for the following reasons. First, a high level of motivation on the part of the learner is assumed. Second, many learners will already be computer-literate even if they lack knowledge concerning the software interface that they are learning (package-illiterate). Third, some learners will be transferring skills from another not dissimilar interface (computer-literate and package-literate). As an example of the last point consider a learner who is familiar with WordStar and who is learning to use an alternative word processor such as WordPerfect. Having control over navigation within the task domain would allow the learner to capitalise on existing knowledge of word processors. This is discussed further in the context of *learner acceptance* in the rationale below.

(c) LIY should comprise not only a delivery system but also an authoring system.

This requirement is necessary so that tutorials can be built for a wide range of different software interfaces. It follows as a result of the second of the principal aims described above: the feasibility of a portable shell.

(d) LIY should incorporate intelligent tutoring system technology where possible.

The standard ITS concepts are discussed more fully later on, together with other ideas which have not yet been fully developed in the implementation. These include a variation in certain of the learner model attributes to include both a *characterisation profile* and a *performance profile*, so that longer-term attributes of the learner could be preserved across a range of tutorials. Additionally, an idea put forward by Pask concerning feedback systems has led to the suggestion of a general architecture for those tutoring systems which include learner control (Ogborn and Johnson 1982). This is described in detail in section 4.2.3.

The rationale for this research is based on the proposition, stated earlier, that the computer is the natural medium for delivering training material for software interfaces. A further step along this path is to consider the stage at which courseware should be produced. ("Courseware" here is specifically limited to mean training material for software interfaces.) Associated with the fact that courseware is difficult and time-consuming to produce is the fact that, like documentation - an analogy which will recur later - there is a tendency for courseware production to be an afterthought. Some of the work which is described here is concerned with examining ways of building the courseware at the same time as the software. It is hoped that this will lead to an overall reduction in the effort - and hence cost - required to produce both software and courseware compared with a more conventional, separate approach. The desire to seek ways of reducing the cost of producing courseware permeates the LIY approach. As with any creative undertaking, courseware production provides many challenges.

These include:

- problems of courseware creation;
- problems of courseware maintenance;
- problems of learner acceptance.

Good *courseware creation*, like writing a good book, is perhaps more of an art than a science. Nevertheless the aspiring author of a book can learn techniques and approaches to apply to the craft (art?) which will result in a higher quality product. The same is doubtless true of courseware production, but in the case of courseware for software interfaces it is clear that the computer itself could provide added support. This is because objects which exist in the user interface to the application software are also those objects about which knowledge is to be taught by the courseware. Such support is certainly highly desirable: some writers claim that the ratio of courseware production time to student usage time can be upwards of 40:1, which represents a working week for every hour of running time (Kearsley 1982). Experiences with the DYCAL system indicated a much higher ratio than this (Martin 1983).

Courseware maintenance is perhaps more of a problem when considering tutorials for software interfaces in comparison with other types of computer-based tutorial: when the software changes in such a way that the user interface is affected, then the courseware must change also (Mayer 1967). It is not obvious that in this instance courseware re-writing can be avoided - you cannot simply change one or two identifiers and recompile! - but if the software changes in a way which impacts upon the conceptual objects which the software manipulates then by comparing old and new versions of structures representing objects in the user interface it would be possible to predict those courseware elements in need of updating. The similarity, mentioned above, between courseware for software interfaces and documentation is that in both cases they can get out of step with software versions. This can be very misleading and quite possibly worse than having no tutorial courseware at all. An implementation approach which keeps the development of software and its tutorial courseware locked in step should be of help in obviating this problem.

The third of the problems mentioned in this section, *learner acceptance*, is perhaps the most important. The two major goals of the application of AI techniques to tutoring systems are the production of more effective courseware on the one hand and exploring the cognitive processes involved in learning and teaching on the other. Both approaches use techniques which are based largely on the architecture laid down by Hartley in which he considered an *adaptive* teaching system (Hartley 1973). At the present time one of the best-known approaches to the second goal concentrates on very fine-grained modelling of the learner in order to force him or her to stay on the learning path of some ideal learner who would become an expert in the tutored domain. This "expert paradigm" is best exemplified in Anderson's Lisp and geometry tutors. While it can be argued that these tutors are adaptive - indeed, the ACT theory of learning is acronymic for *Adaptive Control of Thought* - they do not adapt to the *will* of the learner. Thus the learner cannot exert any influence over *what* to learn or over the *sequence* in which to learn it. For software interfaces this is particularly important for two reasons. The first is that a learner may well not desire nor need to know everything about a software interface. As an example the installation of software may well only be done by a particular member of a department, while other users need not know the installation procedures. On the other hand, some time later it might be that the installation procedure is the *only* topic that a particular user wishes to learn from a tutorial. Secondly, users who are bringing skills from other similar user interfaces - so-called *transfer of training* - possibly only need to be taught a restricted subset of skills in order to be productive with the target application software. While LIY is not as adaptive - in a fine-grained sense - as the Anderson tutors are to each input from the learner, the learner-control capability described earlier in this section does allow tutorials to be adapted to the needs of the software interface learner.

1.3 Conclusions in brief

This section provides a short summary of the conclusions, set out more fully in chapter 5, with respect to the principal aims.

With regard to the first aim - that of investigating the use of a user-interface specification as the basis of a tutorial - the outcome is positive but there are some reservations. Task classification[1] leads in the first instance to an *operational ordering* of user commands which defines the order in which tasks should be carried out to achieve an objective. The LIY tutor requires a *pedagogic ordering*, in which commands are laid out in a sequence which is logical for the learner. It appears possible to transform from operational to pedagogic ordering by applying heuristics. These are rather heavily dependent upon knowledge of the domain, for example: "prompt the designer for any SETUP functions and teach these last". They would not appear to be tractable in the sense of encoding as rules into a program to carry out the transformation. Nonetheless they have been used with success for the transformation by hand of task classifications in three separate domains.

The second principal aim is concerned with demonstrating the feasibility of a portable shell for software interface tutorials. The LIY work described here shows that such a shell can be built; it has been used successfully in the construction of two tutorials. The first is for teaching the use of a DIALLER program to control a modem: in fact this program is simply a front end, with no modem control implemented. The other is for teaching the use of the LIY authoring system: this is a "real" program which updates files on disk.

---

[1] The term *task classification* is used in preference to *task analysis* as the latter term currently has a more overtly psychological connotation than is desired. Human factors workers use *task analysis* to refer to inferred users' tasks rather than operations in the task domain of a user interface.

1.4 Implementation and environment

The LIY implementation broadly follows the proposals set out in an earlier paper (Martin 1987). As described in the previous section, two "application" programs were developed in order to test the LIY approach. These were a phantom DIALLER - the front end of a program to control a modem - and the authoring sub-system of LIY itself. Tutorials were successfully developed for these programs. A delivery environment for LIY tutorials was also built incorporating:

(i) a graphical interface to the learner;

(ii) domain and learner models;

(iii) teaching strategy encoded as a set of rules;

(iv) rule interpreter;

(v) a set of teaching operations.

It was decided to implement all the software in the same programming language in order to minimise interface problems between the various programs. The language used was Golden Common Lisp 286 Developer version 2.2 - an almost complete Common Lisp implementation. It implemented the Common Lisp *package* feature which was used to separate the name-spaces of the various software components. This was both desirable from the implementation point of view, and essential in being able to demonstrate LIY tutorials running with real software. No modification to either of the application software packages was necessary in order to get them to run with the tutorials, although some of the standard Common Lisp input-output routines which these packages used were replaced - only when being used for tutorials - with special-purpose versions of increased functionality. The interface of these routines to the application software remained transparent and in accordance with the Common Lisp standard at all times.

The hardware used was a Tulip AT running MS-DOS with 2.5mB of RAM and a 40mB hard disk. Although it incorporated a Hercules monochrome graphics card the graphics implementation was confined to the so-called IBM graphics characters.

My interest in tutoring systems for software interfaces was kindled when, in 1980, the ageing Thames Polytechnic ICL 1902A was replaced by Prime computers running the PRIMOS operating system, which is quite similar to UNIX. At about the same time I came across UNIX itself and the LEARN system (Kernighan and Lesk 1979). The latter is a set of computer-based tutorials for learning about UNIX, in particular its filing system and the editor *ed*. I implemented the somewhat similar DYCAL system for PRIMOS which gave several generations of students an introduction to the Polytechnic computing environment and was also distributed to a handful of other academic Prime users (Martin 1983). A developing interest in A.I. focused my attention on ITSs, and a determination to develop better tutoring systems than LEARN and DYCAL resulted in my registration for a research degree in January 1984. I initially considered a tutoring system for a financial application which is described in an earlier paper (Martin 1987). In the event the DIALLER, with a much simpler user interface, and its tutorial were developed instead, followed by the tutorial for LIY's authoring sub-system. The financial application was not implemented and it is not reported here, although a pedagogic task classification tree was evolved for it. The transformation heuristics described in section 3.5 were applied to the original tree, in operational ordering, and it was pleasing to discover that they produced the same tutorial ordering as that which had earlier been worked out empirically. Progress was sporadic, but a half-sabbatical for the academic year 1988-89 enabled me to complete the programming.

Although no formal evaluation of LIY has been attempted with learners, it *has* been used by a handful of people and their suggestions noted. In consequence, changes were made which strengthen the diagnostic messages to the learner and which generally improve the user interface of the feedback component. This is described more fully in section 3.7.

1.5 Structure of this thesis

The contents of this chapter are principally concerned with background, aims and rationale. Because of the significance of specification methods, particularly in user-interface design, chapter 2 is devoted to this topic. The last section of the chapter (2.4) describes the interface representation elements used by LIY. Chapter 3 portrays the LIY method for producing tutorials and describes how it works. It also sets out the approach taken to the transformation of an operational task classification to pedagogic ordering. (Note that appendices B and C describe the complete development of the pedagogic structures for the DIALLER and ELICITOR tutorials. These are the two LIY tutorials which have so far been built and which are described in sections 3.3 and 3.4 respectively.) Chapter 4 outlines the components of an intelligent tutoring system and focuses on certain ITS aspects of LIY. It also proposes an architecture for ITSs incorporating learner-control. Chapter 5, "Discussion and Conclusion", assesses the extent to which LIY achieves the aims - both principal and subsidiary - set out in section 1.2. It also outlines further research and development work which might be appropriate.

Chapter 2

Specification methods

This chapter starts by examining specification methods used in software engineering. It then discusses and contrasts methods of user interface specification, going on to describe one of them in relation to the requirements of a tutoring system. The final section justifies the selection of various user interface attributes for incorporation in the LIY system.

2.1 Specification methods in software engineering

Specification methods are increasingly being used in software engineering, principally as a means of reducing the incidence of errors. In addition to specification methods, *design* methods are evolving - very often involving a specification technique - which aim both to reduce the cost and to increase the reliability of a software design.

Many of these design methods owe a considerable debt to the ideas of structured programming. This is particularly true of Structured Design (Yourdon and Constantine 1979) and Jackson Structured Design (Jackson 1983), also known as JSD. Yourdon and Constantine offer a method of structuring by breaking up a large problem into a number of smaller, more manageable units. Jackson Structured Design (JSD) grew out of Jackson Structured Programming (JSP) - a program design method - but now encapsulates it. JSD starts by building a model of the environment in which the proposed system is to operate - the "real world". This model is described in terms of entities and their actions. (Note that a JSD entity is not the same as a database entity.) The functions expected of the proposed system are then added. Timing considerations lead to what is known in JSD as dynamic modelling, in which each JSD entity is modelled as a sequential process. The JSD entities have to be connected by a scheduler. The last phase of the JSD method is to convert the specification into a set of executable programs. A notation is used for specifying entities and actions. Diagrams are used for modelling the real world and the proposed system.

The Structured Analysis school (De Marco 1978, Gane and Sarson 1979) both use data analysis, data dictionaries, data flow diagrams and a "formalism" for representing algorithms known as "pseudocode" or "structured English". While these techniques are not as mechanistic as JSD they have found a wide degree of acceptance although the notation for describing algorithmic specification lacks conciseness. Neither of these Structured Analysis methods go as far as JSD: they both stop short of implementation whereas JSD considers both specification and implementation.

A rather more formal approach is found in USE - User Software Engineering - which is a method for building interactive information systems based on the use of a formal specification method and various automated tools (Wasserman 1984). The user interface is modelled as a set of transition diagrams and there is a graphics editor to maintain them. An interpreter can execute them as dialogue descriptions for prototyping. Originally, algorithmic specification was to be in a specially-designed Pascal-like programming language called Plain, with the idea that a Plain interpreter could be built to offer rapid system prototyping. Subsequently systems were formally specified in BASIS (Leveson *et al.* 1983) which used an abstract model based on Hoare's ALPHARD language (Hoare 1972).

The computer support for the implementation of software directly from a specification is referred to under the umbrella heading of computer-aided software engineering ("CASE"). CASE is currently targeted at automating the production of business systems. The methodologies it supports are those based on data analysis and data flow rather than those based on set theory and logic. CASE *tools* focus on one of the stages of systems development, typically business system analysis and design, database and file design, programming - often generating code in Cobol - system maintenance and project management. CASE *workbenches* are more powerful, offering a complete set of CASE tools for system implementation based upon a single design methodology. It could remain simply a dream, but might not one day a CASE workbench contain also a *user-interface tutorial generator* tool?

Returning now from design to specification, perhaps the best-known method of formal (program) specification is the Vienna Development Method (VDM) which is described by Jones (1980). This proposes a concise method for specifying data objects and their processing based on logic and set theory. Specifications using this method can then be transformed into actual programs. Because the specification method is sufficiently formal, Jones' method allows the designer to reason about specifications and programs. Thus designers using VDM are encouraged to satisfy themselves that the design is correct: they can prove it to be so. The specification methods of Structured Analysis in particular are

insufficiently rigorous to allow this. It would appear that the utility of a software specification system is proportional to the individual effort required to master its use (and unfortunately inversely proportional to its degree of current acceptance within the computing community at large). Complete specifications for systems using techniques such as VDM and BASIS are arduous to produce. Specification of a fragment of a university administration system in BASIS formed part of Leveson's Ph.D. thesis (Leveson 1980).

Another example is the formal specification of a text editor (Sufrin 1982). A number of points are made below concerning this particular work. This is because it is the specification of an application which is more like the software systems for which tutorial approaches such as that of LIY would appear to be useful. The notation used is the Schema Notation developed by the Programming Research Group at Oxford (Morgan 1985). Schema has evolved more recently into the better-known **Z** notation (Spivey 1989).

The first point to make about Sufrin's specification is that it is quite long - the journal article is 46 pages, of which the formal specification takes up perhaps 30 - whilst an *informal* specification is provided in four pages as an appendix. Secondly, no attempt has been made to prove particular properties about the editor: although there is a formal description of each of the editor functions there is no consistent set of pre- and post-conditions. There is no implementation detail, therefore no transformation from specification to implementation, and therefore no argument concerning the validity of assertions during transformation. But then what Sufrin has attempted to do is

> *"to permit exploration of the consequences of our design and to provide an unambiguous definition against which the correctness of implementation strategies might be proven".*

Nor has this been easy: he acknowledges a serious flaw in an earlier formalisation. As in most examples of creative work, at the end Sufrin suggests improvements, here in the form of abstractions which would enable the editor to be enhanced. Although this fits in with his "exploration" justification quoted

above, there seems to be a danger of the tail wagging the dog in that there could be grave difficulties with enhancement if a suitable abstraction could not be found. This is not meant to be a specific criticism of Sufrin's work since this latter problem is present in all design and specification systems. The point to note is that the problem of dealing with enhancements does not simply go away even with a formal specification approach. Lastly, the user interface of Sufrin's editor is particularly straightforward: every editor function can be implemented with a single key depression. Since most systems have more complex user interfaces than this it follows that specifications for such systems would be even longer and require even more effort than that for Sufrin's editor.

Elsewhere Sufrin describes how the specification language **Z** might be applied to the design of the user interface to an electronic mail system set in an office context (Sufrin 1986). As before, a modeless command set is assumed, so that one key-stroke is all that is necessary to accomplish any particular function. The creation and editing of documents on the screen is to be done through the editor Sufrin specified earlier, discussed above. It follows that the concerns expressed earlier about the editor are felt even more deeply about this larger system. Although the specifications describe the functional behaviour of the interface they need to be supported by more tangible views of its appearance. Perversely, the formal specifications represent a triumph of function over style; no essence of the aesthetic element of the interface is conveyed. A specification in this form could not become the basis of a contract of acceptance between client and system designer - a claim often made in favour of the formal specification approach - since the client would not have any interface to envisage. It would be necessary to provide mock-ups of the proposal but this could pose the problem of inconsistencies arising - possibly later - between the mock-ups and the specification. A better strategy would be to derive a prototype from the specification itself although this could pose so much effort for the designer, *before* a contract had been signed, as to render the approach economically infeasible.

The observations made above are not in any way meant to imply criticism of Sufrin's achievements. Indeed, they are especially valuable in that they show the

great effort required to produce formal specifications of real systems. Nonetheless, formal approaches have been found to be of value in producing correct specifications which can be agreed with clients and which enable correct implementations to be produced. The barrier to the wider acceptance of formal specification approaches appears to be the cost in relation to the short life of the final product. The portion of costs devoted to procuring the expertise and effort for specification is especially significant. Many writers, including Sufrin, advocate the adoption of the formal specification approach since it is used in other, more mature, engineering disciplines. Such an approach would be more viable when depreciated over a longer product life-time of perhaps twenty years, say. Over a four-to-six year lifetime the formal specification approach appears at present for most applications - but not all - to be simply too costly.

Although software specification has been the target of considerable research and development, it does not appear to provide a suitable "handle" for building a tutorial for some arbitrary software product. There is too large a gap between, on the one hand, the functional behaviour of the software system and, on the other, both the users' perceptions of the system through its user interface and the psychological requirements - particularly with respect to structure - of a tutorial. Maybe there is a parallel with Clancey's observations concerning the shortcomings of the GUIDON tutorial for the MYCIN expert system (Clancey 1987). This research attempted to turn a rule-based expert system, incorporating an explanation facility, into a tutoring system. It was found to be unsuccessful for tutoring since MYCIN's knowledge was too "compiled" to suit the needs of the learner. From a functional viewpoint the rule-base drove the system successfully and it could provide meaningful explanations in terms of rule-traces. These explanations, however, were meaningful only to those already familiar with MYCIN's domain. Clancey goes on to describe NEOMYCIN - an attempt to incorporate epistemological meta-knowledge into MYCIN - which he hopes will be more successful as a tutoring system.

With a view to moving closer to the hypothetical learner as the user of a software system, we turn next to considerations of user-interface specification.

2.2 Specification methods for user-interface design

The previous section was concerned with specification methods used in the design and implementation stages of the engineering of reliable software. This section considers some of the difficulties, associated with the user interface, which are posed by *software* specification techniques. It also discusses interface specification methods in their own right.

VDM (Jones 1980) is one of the most rigorous and best-known techniques but has a rather restrictive way of specifying input-output. This restriction becomes apparent when one considers the context of a highly interactive system, perhaps executing on a personal computer with a sophisticated windowing and graphics capability. It is this type of system, running mass distribution software, for which the greatest need for computer-based tutorial support has been identified. Yet VDM doesn't have an easy way of representing the complex input-output interactions of such a system. VDM defines input-output in terms of lists, which one may assume normally to be of text characters. An interactive system would therefore need to define many such lists to describe interactive I/O. There is no obvious way in VDM to handle the temporal characteristics of overlapped input and output. Anderson, discussing the properties of a formally specified interactive system, notes the lack of a mechanism for handling temporal characteristics as being a particular problem for user-interface specification methods (Anderson 1986).

Other techniques have been used for describing - and perhaps modelling - complex user interfaces; they are discussed below. Currently, formal program specification methods provide powerful data abstraction and procedural specification capabilities but are weak on user-interface representation; conversely, methods designed for representing and modelling complex user interfaces do not address the problems of data and procedural specification.

No single method, or even class of methods, has emerged as pre-eminent for user-interface specification. A number are discussed including Backus-Naur Form (BNF), transition diagrams, the Command Language Grammar or CLG

(Moran 1981) and path algebras (Alty 1984). It is important to distinguish between *user representation methods* and *user-interface representation methods.* The techniques examined here are all examples in the latter category. Other workers such as Reisner and Payne are interested in modelling the user *per se* during interaction with a system. They are attempting to develop theories of user behaviour and of users' representations of interfaces, such as Reisner's Formal Grammar (Reisner 1981), Payne's Task-Action Grammar or TAG (Payne 1984) and Johnson *et al.*'s TAKD (Johnson *et al.* 1984). For an interesting discussion of classes F(X) of user models, see (Whitefield 1987). F(X) represents agent F's model of X, where F could be one of program, user, researcher, designer; X could be one of system or program, user, designer. Note that nobody is interested in modelling the researcher!

Jacob contrasted the BNF and transition diagram approaches to representing the user interface of a small part of a military message system (Jacob 1983). He was interested in a complete formal specification for such a system, both as a design and implementation aid and for rapid prototyping of its user interface. His view is that transition diagrams provide a more readable specification of the user interface than that offered by BNF. Although the two approaches can be shown to be formally equivalent, Jacob maintains that surface differences can have an important effect on comprehensibility. This idea is appealing: as an example, one has only to think of the ease of doing arithmetic in the Roman compared with the Arabic number representations. Jacob points out that transition diagrams explicitly embody the concept of a state and the transition rules associated with it. These states have a fixed temporal relation (e.g. State2 cannot be reached until State1 has been reached) which is essential in specifying an interactive dialogue. In BNF the temporal relation between events is implicit which makes it much harder to use for dialogue specification. In contrast to declarative specification methods, transition diagrams comprise a procedural element which goes some way to overcoming the problems, mentioned earlier, concerned with the temporal aspects of interface representation. Jacob describes tools which allow textual descriptions of transition diagrams to be input and transformed into an equivalent graphical representation. The USE system mentioned in the previous section is also based on similar tools (Wasserman

1984). Of course, BNF or transition diagrams are fine for describing the syntax of a user interface, but what of its semantics? Jacob doesn't really deal adequately with semantics in detail, but proposes that semantic actions should be described in some high-level programming language-like constructs. Numeric labels attached to the arcs of the transition diagrams are used to refer to code sections which define semantic actions associated with the given syntactic elements. Figure 2.1 illustrates a possible representation of the MS-DOS "cd" command. "\" is the subdirectory operator.

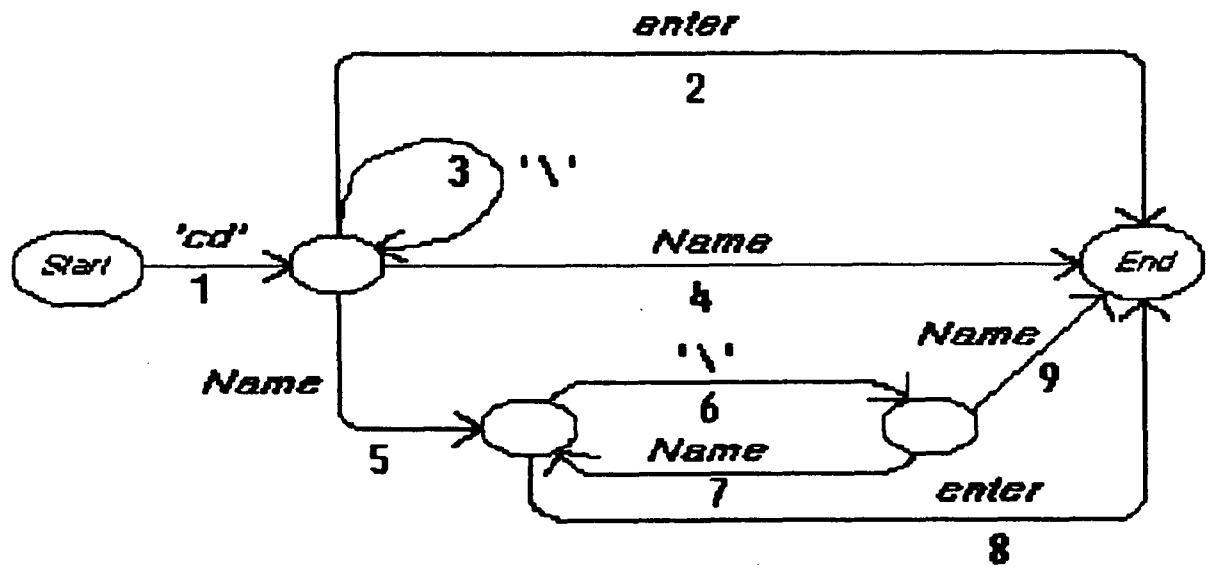


Fig. 2.1

The semantics of the command would be specified by action or condition-action sequences for the numbered arcs. The dollar sign in "$Name" signifies de-referencing of the symbol "Name" which has been passed from some suitable lexical analyser. For the "cd" command of figure 2.1 the sequence would be:

(1) action: TempDir := CurrentDir

(2) action: CurrentDir := TempDir

(3) action: TempDir := Root

(4) condition: not exists-dir($Name)
action: response(TempDir '\' $Name 'not found')

(5) condition: exists-dir($Name)
action: TempDir := TempDir '\' $Name

(6) no associated semantics

(7) condition: exists-dir($Name)
action: TempDir := TempDir '\' $Name

(8) action: CurrentDir := TempDir

(9) condition: not exists-dir($Name)
action: response(TempDir '\' $Name 'not found')

It is thought that the first attempt to use transition diagrams for user interfaces was due to Parnas (Parnas 1969). An improvement was subsequently made which allowed transition diagrams to invoke other diagrams in a fashion similar to the familiar program subroutine principle, i.e. non-terminal input symbols could appear on the transition arcs (Woods 1970). A more general approach based on Woods' augmented transition networks (ATNs) but which allows hierarchies of transition states has been proposed (Kieras and Polson 1985). The

subroutine nesting idea mentioned above is generalised not only to conditions but also to actions and states. As before, in comparison with VDM the semantics of the actions lack rigour.

Transition diagrams appear therefore to be a promising representation method for that class of user interfaces which lends itself to this approach. There are unresolved problems with their use in situations which are non-deterministic such as occur in certain graphics windowing displays. Consider an arcade game in which the display on the screen shows the hunter and its quarry, both in motion, represented internally as two objects but notionally as two transition diagrams. Assume that the transition diagram for the quarry will indicate termination of the hunted object if caught by the hunter. Some suitable transition is similarly indicated concerning the state of the hunter if this event occurs. "Caught" here means that hunter and quarry occupy the same place on the screen. It is possible for the transition diagrams of each party to encode a transition for "Am I at the same point as the other party?", though this would be a poor approach if many hunters and quarries were to be represented. Instead it would be better to have a third object, or agent - the screen manager. This meta-process would be able to detect adjacency and send appropriate messages to the other parties. Thus the interaction is resolved by encoding a state-transition in the hunter and quarry based on reception of such an appropriate message. The difficulty with this as a means of user-interface representation is that the message from the screen-manager doesn't model anything in the interface. The message is not the output of systems analysis, but is merely introduced to support the animation of objects which *are* in the interface.

There are also difficulties with the use of transition diagrams to represent those user interfaces which permit the use of "hot keys". The asynchronous control behaviour of hot keys poses problems for modelling the domain by a tutoring system and a facility to do this has not been incorporated into the current version of LIY. Chapter 5 describes a possible approach to this problem.

The Command Language Grammar, or CLG, is a representation method for exploring the concept of the user interface (Moran 1981). Moran adopts three

perspectives in CLG: the linguistic view, the psychological view and the design view. Further, user-interface components are stratified into four distinct levels: task level, semantic level, syntactic level and interaction level. The task level imposes a structure over the set of tasks which the user wishes to carry out with some hypothetical system. This is very much in the style of *user representation* mentioned earlier as a basis for developing a psychological model of the user. The semantic level defines the conceptual entities and operations of this hypothetical system together with the methods for accomplishing the tasks from the task level in terms of these entities and operations. Thus the semantic level refines the task level - the pattern for all the adjacent levels. The syntactic level recodes the methods from the semantic level in terms of the syntactic level commands, while the interaction level describes the user's physical actions associated with the syntactic elements.

Moran's *linguistic view* of CLG provides an analysis of the structure of command language systems and is relatively brief. He compares CLG with the state-transition and augmented transition network approaches but finds the state-transition representation lacks a sufficient analysis of the functions associated with the states. This finding accords with the general view expressed earlier that user-interface specification methods are weak in the area of procedural specification. However, it is as well to remember that CLG is designed as a representation for investigating user interfaces in general, whereas LIY requires a specific representation method for the engineering of courseware. This is a more pragmatic objective which doesn't therefore necessarily rule out a state-transition representation.

The *psychological view* sees CLG as a means of representing a user model, i.e. a model of the user's view of some interactive system. Due to the lack of a method for representing knowledge in CLG, it is unsurprising that Moran states that the four levels of CLG can only represent a part of what the user knows about a system. The problem from a courseware engineering viewpoint is that any model of the user's knowledge provided by CLG is static. As a representation method CLG cannot provide support for modelling the user's interaction with a system in a way which would intelligently support delivery of

a tutorial for that system. On the other hand one of the strengths of an interface representation system like CLG is that it forces the system designer to consider the user's conceptual model of the system. Moran asserts that this is defined in CLG by the semantic level. It is naturally a desirable objective of any courseware engineering method that it should provide support for the learner to assimilate or induce the underlying conceptual model. On learning, Moran suggests consideration of Rumelhart and Norman's modes of learning: accretion, tuning and restructuring (Rumelhart and Norman 1978). Of these CLG can only address the simplest two: accretion and tuning. Since any representation system could claim to be able to model learning by accretion - an additive process - it is not obvious that CLG is offering any outstanding advantages for modelling learning compared to other interface representation methods. As regards tuning, the learner shifts his or her focus over the subject domain, subsuming lower levels into higher-level concepts. Yet any interface representation method which would enable the learner to forge a link between an objective (task) and its means of accomplishment (action) - and which in some way structures the objectives - would encourage this learning mode.

The *design view* regards CLG as a tool for helping the designer generate and evaluate alternative system designs. The sequence of levels in CLG proceeds from abstract to concrete, providing a pathway for design by successive refinement. Moran proposes the addition of design aids - design principles, design operations and design rules - for helping with design decisions. Unfortunately there is no reported experience of using CLG as a design aid. Moran exemplifies CLG by reference to a model mail system called EG. However, EG is sufficiently small that its whole design can be held "in the mind". Thus in a sense the EG example shows how CLG can be used as a representation method rather than a design method since it would appear that EG has not been designed using CLG. This is not a failing: indeed, Moran stresses that CLG *is intended as* a representation method. Merely, caution needs to be exercised in making claims for CLG as a design aid. Experiences with CLG in this role are reported by Sharratt (1987), who describes some possible improvements and extensions.

CLG represents a system in terms of its entities and operational characteristics at various levels. The top-most task level, while providing a "first cut" means of structuring a system's operational domain, provides only a weak separation from the semantic level. Further, as Moran admits, decisions as to whether details should be admitted to the task level or the semantic level are arbitrary. In the LIY system the output of task classification, required both for the implementation of the application software and of its tutorial, is a representational level broadly equivalent to the semantic level of CLG.

Foley has proposed an Interface Definition Language (IDL) which is an object-oriented high-level description language for user interfaces (Foley 1987). IDL describes the user interface at the conceptual and semantic level, rather than the syntactic and lexical levels, and could thus be used to implement any particular user interface through a user-interface management system. This approach has recently been reported, using the UIDE User Interface Design Environment (Foley 1988). IDL enables the construction of a knowledge base concerning the proposed interface. Algorithms have been developed for possible transformations which can be made to the knowledge base while preserving internal consistency. These transformations enable the designer to transform one proposed interface into another, at the same time maintaining functional equivalence, so as to permit the exploration of the consequences of different designs. UIDE is reported as not only implementing the knowledge base which represents the conceptual design of the user interface (subsuming IDL), but also the transformation algorithms and a user-interface management system to implement any application's user interface. It could be that, within UIDE, transformations may be possible towards a pedagogic orientation for a user interface. Such an approach would parallel the LIY transformations described in the next chapter.

Waddington and Johnson propose relating a family of task models to user-interface specifications so as to be able to explore the consequences of adopting differing user interfaces (Waddington and Johnson 1989). The approach is hierarchical in a manner somewhat similar to CLG, involving a "generalised task model", a "specific task model" and a "specific interface model". To strengthen the procedural aspect of the specification, the generalised task model can

decompose tasks into procedures, which decompose again into actions. The specific interface model uses a representation based upon pre- and post-conditions. However, from a formal specification viewpoint, a great deal more remains to be said about the syntax and semantics of the mappings between the components.

Alty has proposed an interesting application of algebra to networks (Alty 1984). His path algebra technique provides a powerful means of analysing the complex dialogues of an interactive system. In particular, path algebras can be used for detecting redundant paths, loops, etc. which can arise in a less-than-perfect command language. Alty claims that path algebras are quite general and have applicability in CAL as a design tool, but while they can obviously be used for network analysis their use as a design *aid*, particularly for CAL, is not so apparent. Others (Ferraris *et al.* 1984) have proposed alternative network disciplines - Petri nets in this case - for direct application to CAL as a means of modelling the semantics of the domain being taught and the conditions under which the learner is allowed to make transitions between nodes, or sub-goals, within the domain.

2.3 Jacob's specification method revisited

In his original paper Jacob proposed a complete specification method for user interfaces based upon state transition diagrams (Jacob 1983). As discussed in the previous section, both syntax and semantics were considered. Figure 2.2 relates a tutoring system to an application software package through a user interface.

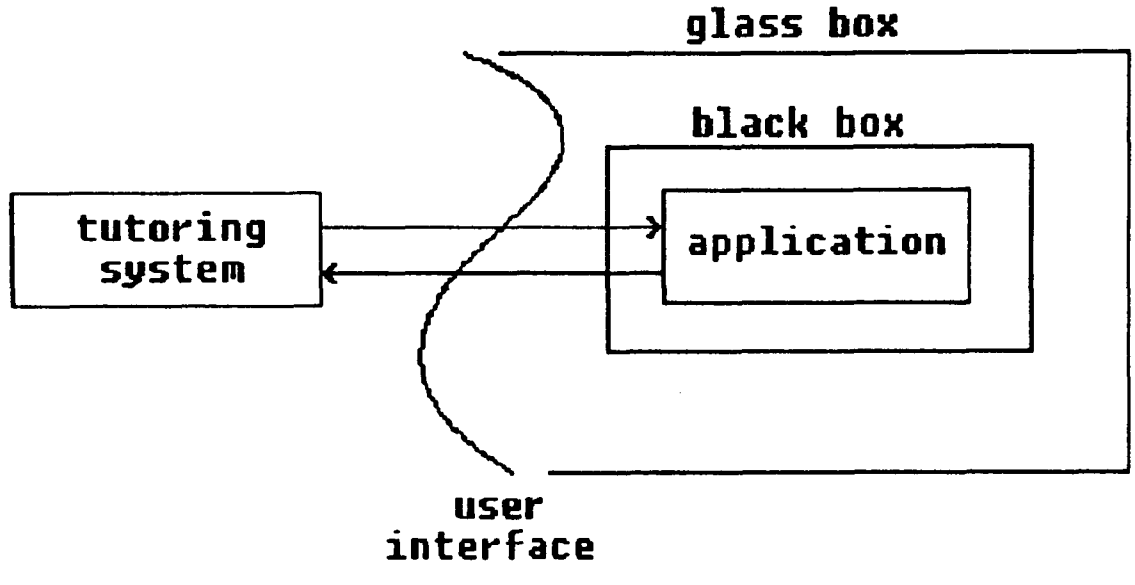


Fig. 2.2

From the perspective of the tutoring system the application is perceived as a "black box" so that any of the tutor's knowledge concerning it - particularly necessary for learner diagnosis - must be represented to the tutor by the user interface. (Learner diagnosis in this context enables a tutoring system to provide an analysis and commentary concerning a learner's interaction with the application software.) The user interface behaves somewhat like a "glass box" enveloping the application black box - a concept proposed in the slightly different context of the teaching of programming languages (du Boulay *et al.* 1981).

The power of the user interface to represent the application to the tutoring system will vary with the extent to which a specification of the interface is available; certain syntactic and semantic definitions of components of the interface may or may not be present. An analysis follows showing the effects of the presence or absence of various specification elements in the interface. The cases are considered in order of decreasing interface power, so that the first case offers the most powerful interface representation and thus provides the tutoring system with the greatest capability for performing learner diagnosis. A distinction is made between commands to the application representing control input and other application input conveying semantic information to the application. Consider as an example the user interface to an "application" which is in fact an operating system. Printing a file might be accomplished by a command with two components. The first component, perhaps *print*, is a control command - one of a limited set of possibilities - whereas the second component, usually a file-name, is application input.

"Jacob's ladder"

(i) Complete user interface specification in Jacob style, i.e. syntax of commands and application input using transition diagrams; semantics of commands and application input.

(ii) Syntax of commands and application input; semantics of commands only.

(iii) Syntax of commands and application input; semantics of application input only.

(iv) Syntax of commands and application input; no semantics specified.

(v) Syntax of commands only.

(vi) Syntax of application input only.

(vii) No specification components.

Case (i), at the top of Jacob's ladder, permits a tutoring system to infer a complete model of the application and thus in principle to perform optimal learner diagnosis.

Case (ii) is weaker in that, for example, an analysis by the tutor would be incomplete for a sequence in which a learner attempted to access a non-existent file.

Case (iii) would mean that, again for example, if a learner were requested to rename a file, then the alternative strategy of copying followed by deletion of the original file could not be detected as being equivalent.

Case (iv) clearly combines the restrictions of cases (ii) and (iii).

Cases (v) and (vi) are more restrictive still and, to be meaningful, require some mechanism in the syntactic structure to enable the tutor to discriminate between application input representing commands and that which represents other semantic information to the application. Some learner diagnosis would still be possible with these cases.

Case (vii) permits no learner diagnosis, although obviously simple right/wrong assessment is possible, based on detection of perfect performance by string-matching.

The next section discusses the requirements of a tutoring system in more detail and proposes case (v) as being appropriate for testing, with an implementation, the utility of the transition diagram technique.

2.4 Selection of a specification technique

An ideal formal specification for most programs would consist of a functional component - what the program is actually to do - and an interface component - how the program is to conduct a dialogue with a user. In addition there could be a further interface specification to describe a program's interaction with other machine elements. These might be device interfaces to sensors, for example, or possibly interfaces to other programs. Since this research is concerned with tutorials for the user interface, the link to equipment and other programs will not be considered. Also inappropriate would be a full functional specification; this research is not concerned with examining the binding between software specification and implementation. What is of interest are the elements of an interface specification which could be exploited in the building of a tutorial for that interface. Both input and output would need to be considered in order that an application interface be completely specified. The research described here is only concerned with the input side; a tutor needs to focus on learner input to an application in order to attempt interpretation of it in a meaningful way. It might be possible for a tutor to manage interpretation of learner interactions with software if application output be considered in addition to input; detection of an error message, for example, could act as a powerful trigger to tutorial action of some kind. However, it is not clear that a tutor's "black box" view of software would permit it to infer very much from consideration of error messages. An approach to learner diagnosis is proposed which incorporates a *model* of the application software, against which recorded learner input can be interpreted by the tutor. Application output is not considered.

In the context of this research, specification can be regarded as serving essentially two purposes. Firstly, as a specification of the software it describes, it could be rendered executable. Thus it could be used as a prototype for all or part of a program. It would be perfectly possible for the specification of the input side of a user interface to be used as the application front-end, displaying appropriate screens, handling correct input, guarding against incorrect input and dealing with error messages. This would in principle be possible for cases situated towards the top of Jacob's ladder, particularly case (i). Such an

approach has not been followed for LIY following consideration of the implementation effort necessary: in order that LIY remain portable it would require the building of part of a general-purpose application generator, capable of handling front-end input-output. Nor has back-end specification been considered. The current so-called "fourth-generation" approach typically allows high-level specification of back-end processing, largely in terms of database access, using structured English. This is subsequently transformed into a structured high-level-language program.

Secondly, specification can support the design and delivery of a tutorial. It can be used for tutorial construction as an aid to the designer, for example ensuring that courseware is built for every command in the interface. It can also be used during tutorial delivery, both for learner diagnosis based upon a model of the domain when evaluating learner input, and as a means of providing a conceptual representation of the interface to the learner, possibly in graphical form.

It is appropriate now to turn back to Jacob's ladder and select a "rung" which would appear to support the aims of this research. The top of the ladder offers the most power but, as has been pointed out earlier, appears to be somewhat ambitious. Not unusually it is the semantic definitions which pose the biggest problems. Jacob's semantic definitions may or may not be sufficiently formal to be understood by an interpreter. To build such an interpreter, however, is not all that would be required. A tutoring system would need to find a method for interpreting the learner's intentions in order to provide effective diagnosis. The requirements for implementing a tutoring system at the top of Jacob's ladder would be rather like having to implement PROUST (Johnson and Soloway 1987) with the additional tasks of needing to define Pascal and implement an interpreter for it.

It would appear to be useful, therefore, to turn to the other end of the ladder and see what a weaker specification could offer a tutoring system. Case (vi), in which only application input syntax is defined, appears to be problematic in that not all user interactions with software require application input. Consider, taking an operating system interface as an example, the actions of navigating to, or

listing, a directory. With knowledge only of application input a tutoring system would not be able to perform diagnosis based upon all types of learners' interactions.

Moving up the ladder to case (v) provides a tutorial with knowledge about the syntax of command input but not of application input. Thus a tutor should be able to *model* the learner's use of commands and perform a measure of diagnosis. Application input, as opposed to command input, could be handled (but in a somewhat simple-minded fashion) in the manner of case (vii), seeking a strict match between known correct input and the learner's input.

Case (v) from Jacob's ladder, i.e. specification of input command syntax, has thus been selected as the basis for an LIY implementation.

The specification elements used for the tutorial in LIY are two-fold. Firstly there is an operational task or command hierarchy. This represents the output of the systems analysis task classification stage. Figure 2.3 shows an example taken from LIY's DIALLER. It illustrates such an operational hierarchy for the top level of the program and should be read as "DIALLER *consists_of* DIAL_DIRECT *and* DIAL_FROM_MEMORY *and* SETUP *and* QUIT". The dotted continuation marks indicate that each sub-operation (DIAL_DIRECT etc.) is itself recursively decomposed in the same way.

The *operational ordering* shown here is not particularly appropriate for supporting a tutorial. This is discussed in the next chapter together with a view of the task classification transformed into *pedagogic ordering*.

Secondly the specification contains a mapping of the input command syntax of the interface on to the nodes in the pedagogic ordering. As an example, for the DIALLER this means attaching "D" to the DIAL_DIRECT node, "M" to the "DIAL_FROM_MEMORY" node, and so on, "D" and "M" being two of DIALLER's top-level commands. This command representation of the domain forms a model which is interpreted by a deterministic transition tree parser during the learner diagnosis phases of tutorial delivery. The domain model also

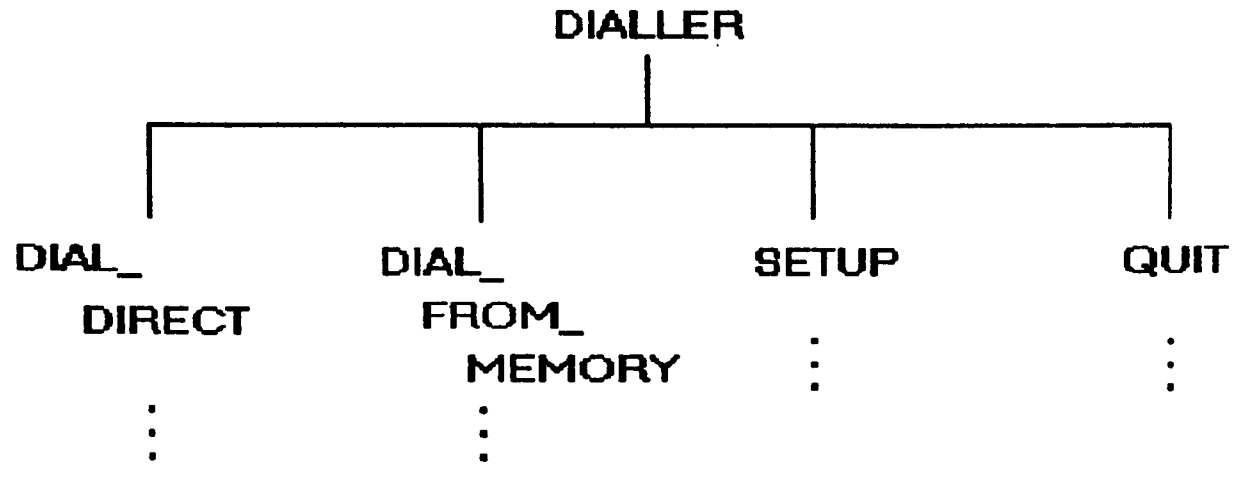


Fig. 2.3

needs to know the navigating sequence followed by the actual domain. Specifically, this is necessary so that the parser can be reset at the appropriate point in the hierarchy after the execution of a bottom-level leaf command. Strictly speaking, this is a semantic consideration which moves LIY slightly above case (v) on Jacob's ladder.

No syntax of direct application input is represented although LIY's parser recognises the termination symbols for this type of input. Such values - typically either *enter* or *escape* - are, like the commands, attached to the appropriate nodes of the pedagogic task classification.

Thus for LIY, only a proportion of a user-interface dialogue specification has been exploited: the task command hierarchy, the control routing following leaf processing, the syntax of input commands and the terminators for application input. Yet this is sufficient for the construction of a domain model capable of being used for learner diagnosis.

Chapter 3

LIY : The "Learn-It-Yourself" approach

This chapter starts with an overview of the LIY method. It then describes LIY's principal components in relation to the four elements of the Hartley and Sleeman model (Hartley and Sleeman 1973). LIY is described from the viewpoint of the learner and then of the tutorial designer, in each case drawing on appropriate examples. A section is devoted to the technique for transforming the task classification structure to yield a pedagogic ordering. Although the earlier sections of this chapter are illustrated by reference to existing LIY tutorials, the transformation technique is exemplified through references to the well-known operating system MS-DOS. The development of the two existing LIY tutorials from the interface specification elements discussed in chapter 2 is reported not in this chapter but in appendices B and C. A complete description of the pedagogic task classification structure is provided. There follows a section setting out LIY's control behaviour and the chapter is summed up with some closing remarks in the final section.

3.1 Overview of the LIY method

LIY consists of both a system for delivery of tutorial material and a system for authoring it. The delivery system is the more fully developed. It uses domain and learner representations and performs diagnosis using a form of differential modelling which has some similarities with the use of issues in the WEST system (Burton and Brown 1982). The authoring system is only partly implemented at present; all its aspects, whether currently implemented or not, are straightforward but time-consuming to program. The following description therefore emphasises the delivery system.

Figure 3.1 shows how LIY teaches the potential user of a software application by permitting interaction with it while the tutorial maintains control.

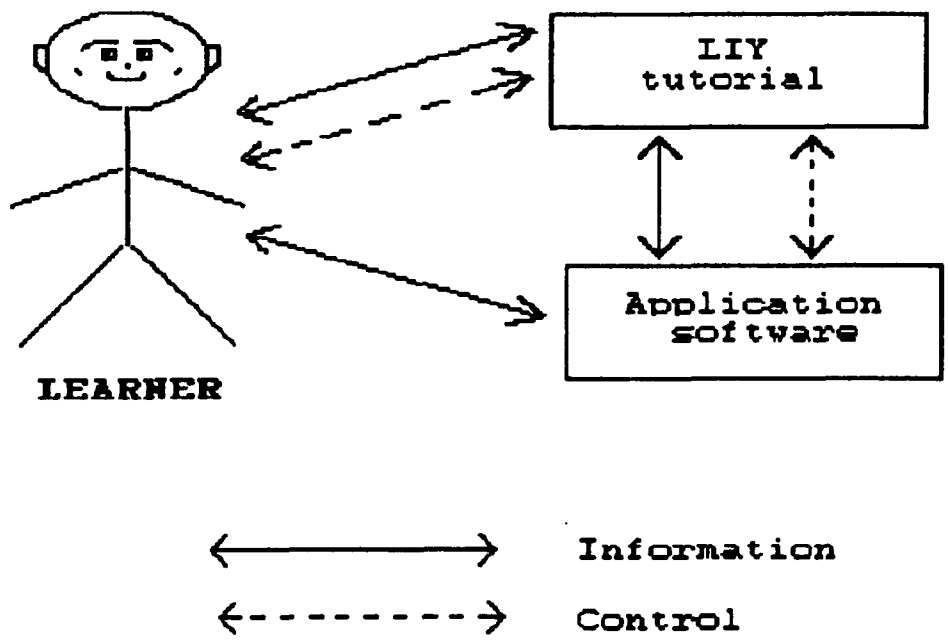


Fig. 3.1

Two LIY tutorials have been written so far. One teaches the use of a DIALLER program which in principle controls a modem installed in a computer. A complete implementation would allow the user to connect to the telephone

system through the keyboard, and then to a remote computer, for example. The other teaches the use of the ELICITOR program which is the authoring system for building LIY tutorials. Rather than teach the use of existing applications it was decided to develop software with, of course, a particular specifiable interface. Thus the DIALLER and ELICITOR programs have been built. This approach offered the following advantages:

(i) LIY could be tried within the scope and limitations - text-based input, etc. - set out in chapter 1;

(ii) the LIY method could be applied to a very simple interface in the first instance (that of the DIALLER program);

(iii) using a common development environment (Lisp) would facilitate the capture for tutorial diagnosis of the learner's input to the application. Note, though, that the two implementations - tutoring system and application - are segregated in separate name-spaces by the Lisp *package* feature. This means that applications can run quite independently of the tutoring system and in particular that the latter does not need to be loaded into memory to run an application.

It can be seen that the ELICITOR "application" is a tool in the LIY system. The DIALLER program is a cut-down version of what the real thing might be: it presents an appropriate interface to the user but doesn't connect to a modem nor to the outside world. In the passages describing the learner's and the tutorial designer's views of LIY (sections 3.3 and 3.4), the examples are drawn from the DIALLER and the ELICITOR respectively.

The LIY tutor contains a representation of the application domain imported in a modified form from the systems analysis and design stage for development of the application itself. The objective is to utilise some of the work done during this early phase later on, at the tutorial design and delivery stage.

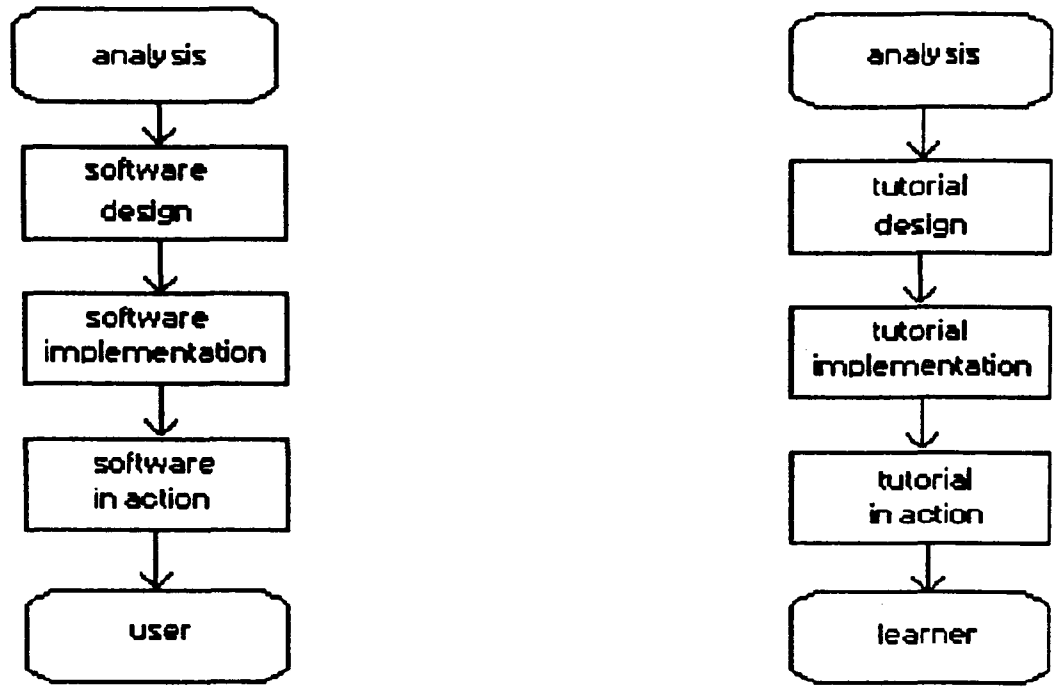


Fig. 3.2

Figures 3.2 and 3.3 contrast the conventional and LIY approaches to application and tutorial design. In figure 3.3 the reference to "shared interface representation" is not meant to imply an actual shared machine representation. Rather, it implies that a proportion of the systems analysis effort, devoted to developing the task classification structure, can serve at both the software implementation and tutorial implementation stages. Note that this task classification structure represents an *operational* sequence. In other words, it represents the way in which operations in the hierarchy are constructed from those at a lower level. The operational sequence must be transformed to a *pedagogic* sequence, as discussed below in section 3.5.

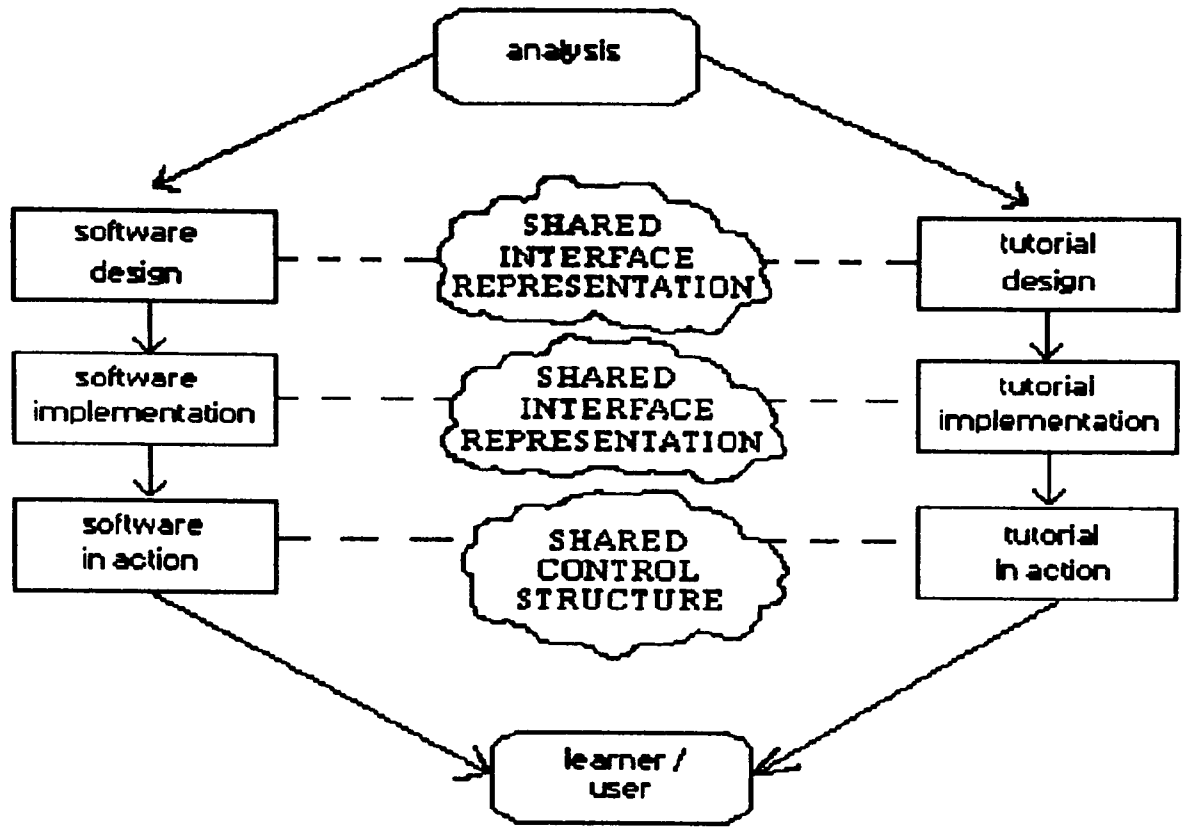

analysis
software design
SHARED INTERFACE REPRESENTATION
tutorial design
software implementation
SHARED INTERFACE REPRESENTATION
tutorial implementation
software in action
SHARED CONTROL STRUCTURE
tutorial in action
learner / user

Fig. 3.3

3.2 LIY's principal components

This section describes LIY in relation to Hartley and Sleeman's four-component architecture for an ITS (Hartley and Sleeman 1973). In chapter 4 an alternative architecture is proposed, able to incorporate learner-control, which is a development of the five-ring model (O'Shea *et al.* 1984).

Hartley and Sleeman describe the four components of an ITS as being:

(a) representation of the task;

(b) representation of the learner;

(c) teaching strategy expressed as a set of means-ends guidance rules;

(d) set of teaching operations.

3.2.1 Representation of the task

The representation of the task is elicited from the tutorial designer as a tree. Leaf nodes in the tree typically correspond to an internal command in the application. The tree is almost the *only* application-dependent part of an LIY tutorial, the only other application-dependent objects being path-names loaded in at LIY top-level. Besides structuring the task domain of the application, the tree contains much other information attached to each node in the structure. Examples of this information include teaching operations such as slide-shows or exercises using the application. It also models the domain in terms of its control structure so that during the diagnosis phase, when the learner's key-stroke sequence is being parsed, it can be used as a transition tree. It is described more fully in the sections that follow, particularly section 3.5.

3.2.2 Representation of the learner

LIY builds representations of the learner as profile information, as well as computing an assessment of the learner's performance during the diagnosis phase. Global information inferred about the learner's characteristics and performance is used to maintain a *characterisation profile* and a *performance profile.*

Characterisation Profile

This is used to determine the advice given when the learner attempts to take control in order to navigate to an alternative topic. The advice is adapted to an assessment of the learner's interaction style in a set of rules ("L-C-ADVICE"). These rules consider equally three qualitative variables: *COMPETENCE*, *DUCKER*, and *FLITTER*. *COMPETENCE* really belongs in the performance profile but is considered here since it contributes - equally with the other two variables - to advice given to the learner by the tutor. The *COMPETENCE* variables - *WEAK*, *NORMAL* and *STRONG* - together with *DUCKER* and *FLITTER* are in fact coded as boolean functions which examine the value of associated variables (*COMPETENCE*, *DUCK-CNT* and *FLIT-CNT* respectively).

(a) COMPETENCE

More specifically, the rules consider *WEAK*, a particular range of values of this variable. *COMPETENCE* is scored on a continuous scale from 0 to 10, with an initial value of 5. Depending upon the outcome of assignments set, it is modified by an increment for a correct answer or a decrement for an incorrect one, bearing in mind that it is restricted to the range 0 to 10. *COMPETENCE* is not referenced directly in the advice rules, but there are three qualitative variables based on its value. These are *WEAK*, *NORMAL* and *STRONG*, corresponding to values of *COMPETENCE* in the ranges 0 to 2.4999, 2.5 to

7.4999, and 7.5 to 10. The normal value of the increment applied to *COMPETENCE* is 0.5, but there is an amplification effect at the start of the tutorial, the first three increments (or decrements) applied having values 2, 1, and 0.66667. The idea of this is to decrease the sensitivity of the advice rules with the passage of time so as to avoid apparently significant random movements around the mean. No particular claim is made for this technique and it has yet to be evaluated.

(b) DUCKER

This qualitative variable is used in the rules to indicate a learner who habitually avoids set assignments. To duck an assignment means that the learner, having failed with it on two or more successive occasions, has elected to abandon it (thus avoiding it) and to move on to the next topic. Such a learner is considered to be a *DUCKER* if this has happened with more than two assignments.

(c) FLITTER

A learner is deemed to be a *FLITTER* if, on three or more occasions, he or she has forced a move to a new topic under learner-control in the face of advice from the tutorial against such action. Note that LIY's philosophy is that, if the learner is sufficiently determined, such moves should always be possible.

If both *DUCKER* and *FLITTER* occur together only the *DUCKER* variable is updated.

Three levels of advice are offered against a move. The strongest is reserved for the learner whose characterisation profile indicates that all three of *WEAK*, *DUCKER* and *FLITTER* apply, and that there is more than one prerequisite topic associated with the learner's target move. (There is further discussion of prerequisites and LIY's control behaviour in section 3.5.) The next level down

in strength of advice applies to the same situation but where there is just one prerequisite, or alternatively where only one or two of *WEAK*, *DUCKER* and *FLITTER* apply. The weakest advice against a move to the learner's target is reserved for situations in which either none of the three qualitative variables apply although there is more than one prerequisite topic, or one or two apply but there is only one prerequisite associated with the learner's target topic. A move is permitted with no contrary advice if there are no outstanding prerequisites (whatever the state of the qualitative variables) or in the situation in which none of these variables apply and there is just one prerequisite.

Performance Profile

Nodes in the tree are marked to indicate that a topic has been taught when the learner has completed all the teaching operations associated with it. This represents one aspect of the learner's performance. The other aspect of the performance profile is *COMPETENCE*, a score representing the learner's ability to handle the assignments set by the tutorial. *COMPETENCE* is considered above, rather than in this section, for clarity.

Diagnosis

During diagnosis a comparison is made between the effect of running the learner's key-stroke sequence and a "correct" sequence through a model of the application. The matters addressed by the correct sequence will normally be a subset of those addressed by the learner. There is a fuller discussion of the diagnosis module in chapter 4.

3.2.3 Teaching strategy

The teaching strategy which LIY uses is described in detail in section 3.6. Briefly, there are five sets of rules labelled arbitrarily with the letters "a" to "e". Each rule in a given rule-set is named by a combination of rule-set letter and a number, based on increments of ten, for example *a10*, *a20* and so on. Figure 3.27, at the end of this chapter, illustrates the relationship between the rule-sets.

"a" rules are LIY's top-level rules and connote a teaching strategy as follows:

> compute the "next" untaught topic in the task representation and teach it;
> permit learner-controlled interruption under certain circumstances;
> if there is other knowledge about a topic - represented as *designer rules* - then apply that other knowledge. (There is a description of *designer rules* in section 3.4.3.)

"b" rules are concerned with control behaviour following a learner interruption.

"c" rules determine the outcome of such an interruption in terms of advice as described in the previous section;

"d" rules conventionally describe *designer rules*;

"e" rules select the next teaching operation.

The forward-chaining interpreter for these rules is very straightforward. It avoids the problems of conflict resolution by firing the first rule it finds with a matching antecedent. The consequent of a rule can include a call to the interpreter to run another rule-set or to exit from interpretation of the current rule-set. The interpreter normally exits from a rule-set (or halts at the top-level) when it can find no more rules to fire. On occasion it is useful to set the rule interpreter global variable *LOOPLIMIT* to a numeric value - typically 1 - which indicates a limit on the number of passes the interpreter should make over a "called" rule-set.

3.2.4 Set of teaching operations

LIY teaching operations are described in detail in section 3.4 - "How the Courseware Designer sees LIY". In brief, the operations include:

(i) slide-show;

(ii) create an application environment: setup the application in some particular way;

(iii) watch and record learner input (when interacting with the application: implies subsequent diagnosis);

(iv) place the learner at some chosen point in the application;

(v) get learner input directly ("immediate" assessment);

(vi) free learner exploration of the application (no diagnosis).

3.2.5 Other LIY components

The ELICITOR is an LIY program which interacts with the courseware designer to enable the construction of LIY tutorials. It allows the designer to specify the appropriate task classification structure, and then permits enhancement to selected nodes in this structure by letting the designer point with the mouse at a target topic.

Teaching material is presented to the learner in the form of "slides". These are in fact simple ASCII files which can be created by the tutorial designer using any suitable text-editor.

LIY captures the key-stroke sequence of the learner interacting with the application. This is done, transparently to the application, by substituting the normal Lisp input-output routines used by applications with replacement routines of the same names. These routines are contained in a module (actually, a file) along with the slide-show delivery routine.

There are many LIY utility functions and they are grouped together logically as initialisation routines, mouse-driving routines, further input-output routines and "others" - the latter being quite a large file!

3.3 How the learner sees LIY

3.3.1 Teaching

The following discussion is based on a learner's interaction with the DIALLER tutorial. Figure 3.4 shows a typical screen from a slide-show: the very first screen of the tutorial, in fact. The banner at the bottom of the screen indicates that the learner may get a re-run of the sequence of slides forming a slide-show by pressing the "home" key. The space bar moves the tutorial on to the next teaching operation, while the learner can interrupt by pressing the "control + break" combination. On the right of the banner is indicated the title of this current topic.

A learner-control interruption displays the screen which is illustrated in figure 3.5 If the learner quits then the environment is saved to the extent that he or she can subsequently continue without having to cover topics already learnt. Option "E" permits the learner to interact directly with the application, from its top level, as if the tutorial were not present; no diagnosis is performed but, on quitting the application, control reverts to the appropriate place in the tutorial. Option "B" permits the learner to browse over the task classification tree and to use the mouse to select a topic to learn. Alternatively if the learner knows the topic's name then it can be typed in directly to the menu.

Figure 3.6 illustrates a typical screen from the DIALLER program; here, the learner has been placed in the application and asked to carry out some assignment with it. Figure 3.7 demonstrates that the learner can interrupt in the application as well as in a slide-show.

If the learner selects option "B" to browse then a plan of the (partial) task classification tree is displayed, as illustrated by figure 3.8. The current node in fact flashes. The learner can see more of the tree by clicking on the arrows at

the edges of the screen, can select a topic to learn by clicking on it[1], or can quit - reverting to the original topic being taught - by selecting the "quit" lozenge at the top-left of the screen. The previous section described how topic selection is mediated by advice from LIY, based on the learner's current state, although the learner can over-ride this advice if necessary.

---

[1] The *proceed-n* nodes, necessary for the transformation from general tree to binary, are not selectable (see section 3.5).

```
WELCOME TO THE LIY TUTORIAL FOR THE PHANTOM PHONE DIALLER!

If you follow this tutorial you will learn how to use a very simple
interface to a program which can dial, through a modem, to the
telephone numbers of remote services such as time-sharing systems,
bulletin boards and other electronic mail systems.

There is one surprising feature of this DIALLER, though: it doesn't
actually dial any numbers!  It is simply a hollow shell - an interface
which connects to the user but not to the telephone system.

The reason for its existence is to test out certain ideas about
producing tutorials (like this one) to teach the users of software
products such as, in this case, a DIALLER.  Other possible types of
software for which this approach might be useful could include a word-
processor or spread-sheet program, for example.

When you are ready, pressing the space bar will move you on from
the current slide, whether to the next slide or to some other
activity such as interacting with the application (the DIALLER).

Space:NextScreen|Home:SeeItAgain|Ctrl-Break:Interrupt.  Current:DIALLER-TA
```

Fig. 3.4

```
The reason for its existence is to test out certain ideas about
producing tutorials (like this one) to teach the users of software
products such as, in this case, a DIALLER.  Other possible types of
software for which this approach might be useful could include a word-
processor or spread-sheet program, for example.

When you are ready, pressing the space bar will move you on from
the current slide, whether to the next slide or to some other
activity such as interacting with the application (the DIALLER).
```

```
Space:NextScreen|Home:SeeItAgain|Ctrl-Break:Interrupt. Current:DIALLER-TA
Returning from slide show...

OK - What would you like to learn?

Press RETURN to continue with your original topic.
Type  Q to quit LIY
      B to browse
      E to explore DIALLER freely
      or the topic's name.
All end with RETURN
>
```

Fig. 3.5

```
DIAL DIRECT FROM KEYBOARD

Type your number, which should be followed by RETURN

If using numeric keypad, ensure NumLock is on

The following characters may be embedded :-

    (   )   -   <space>

    T   (switch to Tone dialling)
    P   (switch to Pulse dialling)
    @   (to pause dialling {Any key restarts})

        To return to previous menu, press       Esc

Number dialled..
```

Fig. 3.6

```
        If using numeric keypad, ensure NumLock is on

        The following characters may be embedded :-

            (   )   -   <space>

            T   (switch to Tone dialling)
            P   (switch to Pulse dialling)
            @   (to pause dialling {Any key restarts})

        To return to previous menu, press       Esc

Number dialled..
Returning from application...
OK - What would you like to learn?

Press RETURN to continue with your original topic.
Type  Q to quit LIY
      B to browse
      E to explore DIALLER freely
      or the topic's name.
All end with RETURN
)
```

Fig. 3.7

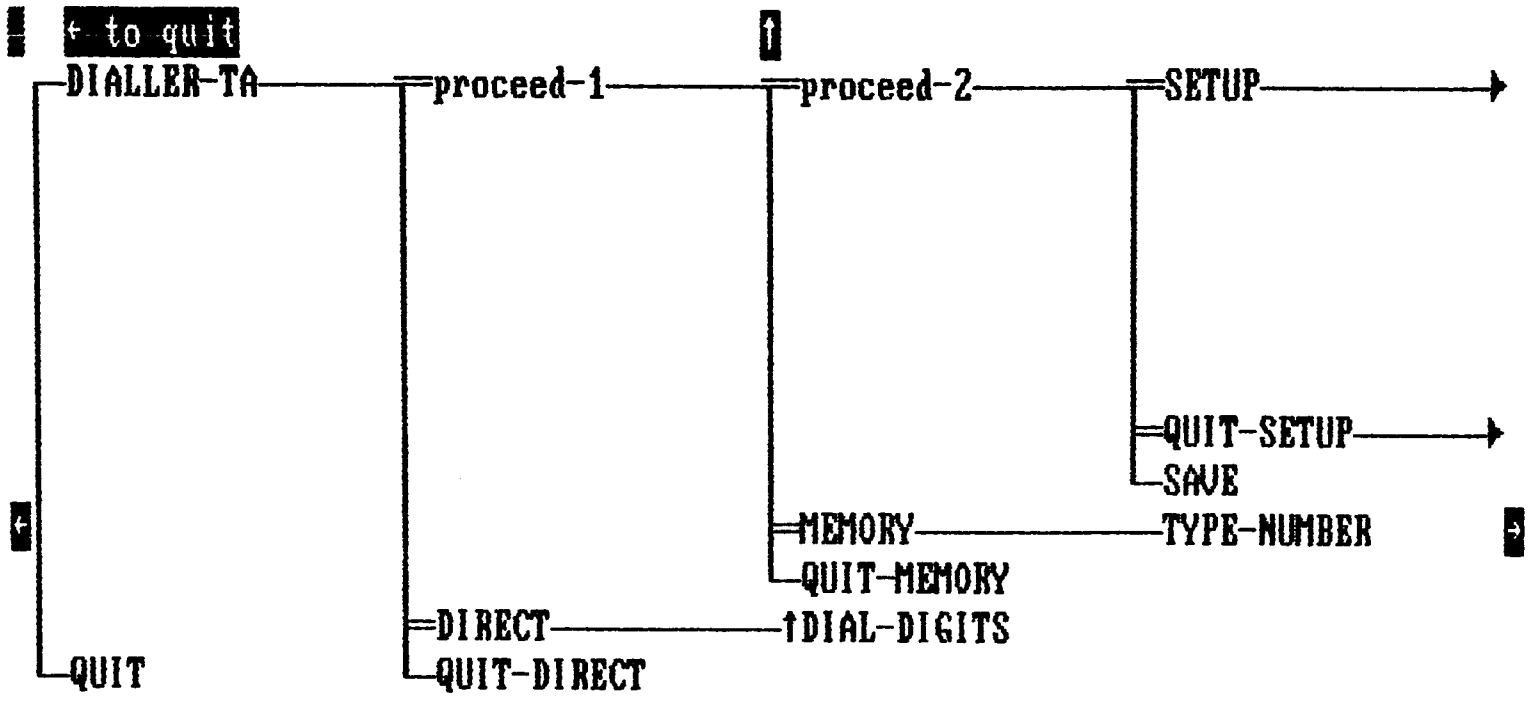

← to quit
DIALLER-TA
proceed-1
proceed-2
SETUP
QUIT-SETUP
SAVE
MEMORY
TYPE-NUMBER
QUIT-MEMORY
DIRECT
DIAL-DIGITS
QUIT
QUIT-DIRECT

Fig. 3.8

3.3.2 Assessment

Suppose that the tutorial asks the learner to interact with the application in some way. There is no standard screen which illustrates this interaction; the tutorial, in a slide-show, makes clear what the learner is required to do and then control is switched to the application. As an alternative to this kind of assessment, the tutorial can request a direct response form the learner. Figure 3.9 shows an example of this. The tutor issues the ">>" prompt. Here, and in what follows, the learner's responses are underlined.

```
>> a:/top
 I'm afraid that doesn't seem right.
Press any key to continue..
```

Fig. 3.9

The feedback is independent of the context, and the learner is shown a replay of the associated slide-show before being given the assignment again.

In the unfortunate instance that the learner enters incorrect input a second time the feedback of figure 3.10 appears:

>> a:/top/next

I'm afraid that STILL doesn't seem right.

Press any key to continue..

Type NEXT if you would like to move on, or press RETURN to try again: next
NEXT
The correct response should have been:
A:\TOP\NEXT

(... next topic is selected ...)

Fig. 3.10

The learner can then decide whether to persevere with the current assignment or move on. In the latter case the correct response is shown. This is adequate for "immediate" assessments like the one shown in figures 3.9 and 3.10. It leaves something to be desired when the assessment involves the learner interacting with the application since all that is shown is a string corresponding to a correct key-stroke sequence. It would be better if the tutorial could "walk" the learner in "show me" mode through the application at this point, making the learner follow with the correct key-strokes.

As a result of the diagnosis stage applied following a learner's interaction with the application, the feed-back can be improved to a certain extent. The differing messages in figures 3.11 allude to the different issues detected when running the learner's and tutorial designer's key-stroke sequences through the domain model. These messages are more fully explained in the next chapter which contains a precise description of the diagnosis process.

Welcome to ELICITOR - the Task Analysis structure (TA) creation program

Please type the name of the application which
the tutorial is to teach (or Q to quit) :q
Quitting...
I'm afraid that doesn't seem right.

You appear to have quit ELICITUT in an abnormal way.
The exit command "ESCAPE" associated with the topic QUIT was expected.
Press any key to continue..

(a)

I'm afraid that doesn't seem right.
Possibly you left out some of the commands,
or used them in the wrong order.

The command "ENTER" associated with the topic DIAL-DIGITS was expected.

Press any key to continue..

(b)

I'm afraid that STILL doesn't seem right.
Possibly you misused one or more of the commands which alter the state of DIALLER.

The command "S" associated with the topic SAVE should be avoided for this assignment.

Press any key to continue..

(c)

I'm afraid that STILL doesn't seem right.
Possibly one or more of the character strings which you typed into DIALLER was incorrect.

The input 123 4567 was expected.

Press any key to continue..

(d)

Fig. 3.11

Note that figure 3.11 (a) is taken from the ELICITOR authoring system tutorial (ELICITUT) since abnormal exit cannot normally occur in DIALLER.

Usually the learner does the right thing, as shown in figure 3.12:

Well done!
Press any key to continue..

Fig. 3.12

The limitations illustrated by figure 3.11 concerning the quality of the feedback to the learner result from the positioning of this LIY implementation on Jacob's ladder, as discussed in chapter 2. They also relate to the domain-independence required of a portable tutoring shell. More powerful diagnosis and error feedback would follow from situating LIY further towards the top of the ladder, where it could take in more syntactic and semantic information from the user interface specification.

3.3.3 Feedback in the form of advice

Section 3.2.2 above described how the characterisation profile part of LIY's representation of the learner distinguishes between the four possible outcomes of a learner's interruption and request to learn an alternative topic. One possible outcome is that the request is enabled immediately; otherwise one of three messages advising against the move is displayed. These messages are graded in terms of the strength of their exhortation. Figure 3.13 shows the weakest of these.

OK - What would you like to learn?

Press RETURN to continue with your original topic.
Type Q to quit LIY
B to browse
E to explore DIALLER freely
or the topic's name.
All end with RETURN
>set-pause
SET-PAUSE
***It might be better for you not to move to** SET-PAUSE*
at this stage because you have not yet mastered
the following prerequisites :-

ABANDON
SAVE
QUIT

You may inspect the structure of prerequisite information by selecting the BROWSE option following a Ctrl-Break interruption, which you can type right away:

Alternatively, you may type F to force a move to SET-PAUSE, or press RETURN to continue with your original topic:

Fig. 3.13

The emboldened line in figure 3.13 can on occasion be replaced by either of two stronger messages:

"You are advised AGAINST moving to"

or

"You are VERY STRONGLY advised AGAINST moving to"

The ways in which qualitative variables in the characterisation profile - *DUCKER*, *FLITTER*, and so on - combine with the number of outstanding prerequisites to produce these messages was described in the previous section. Further discussion of prerequisites is held back until section 3.5.

3.4 How the courseware designer sees LIY

Whereas the previous section - describing the learner's view - concentrated on the DIALLER application, this section draws on examples from the ELICITOR program and occasionally the ELICITUT tutorial which teaches its use. ELICITOR supports the tutorial designer, particularly when the task classification tree is being built.

3.4.1 ELICITOR, ELICITUT and its domain model

ELICITOR asks the tutorial designer for the name of the application for which the tutorial is being constructed. It creates a sub-directory of this name in which it will build representations of the pedagogic task structure. This representation is first built as an ASCII text file, eliciting the structure from the designer. Then, when the designer is satisfied that the structure is correct, ELICITOR creates a Lisp version of it. Finally, the designer can interact with ELICITOR using the mouse to select individual topics from a graphical representation of the structure on the screen. This step is necessary in order to add further information to individual nodes. Examples of this information include teaching operations such as slide-shows and exercises using the application. A complete description is given in section 3.5.3.

Figure 3.14(a) illustrates the situation when the designer is creating a tutorial application for the first time, while in figure 3.14(b) ELICITOR reports that there already exists a Lisp version of the task structure for the named application.

Welcome to ELICITOR - the Task Analysis structure (TA) creation program.

Please type the name of the application which
*the tutorial is to teach (or Q to quit) :*learn

Fig. 3.14 (a)

Welcome to ELICITOR - the Task Analysis structure (TA) creation program.

Please type the name of the application which
*the tutorial is to teach (or Q to quit) :*learn

TA.LSP already exists. It will be renamed
to TA.BAK and a new version created.
Do you wish to go ahead? (Y or N)

Fig. 3.14 (b)

Figure 3.15 shows ELICITOR detecting the existence of the ASCII text form of the task structure, TA.TXT. From the point of view of the ELICITOR program all this is very straightforward. It is significant, however, when the construction of the ELICITUT tutorial is considered. This tutorial teaches the use of the ELICITOR program and has to adapt to the different semantics associated with the presence or absence of the files TA.LSP and TA.TXT. In addition to these different semantics there is also a different allowable syntax, since the "Y/N" or "Q/R/C" input of figures 3.14(b) and 3.15 may or may not be required.

A version of TA.TXT already exists.

If you would like to keep it for editing with a text
editor, please quit by typing Q.

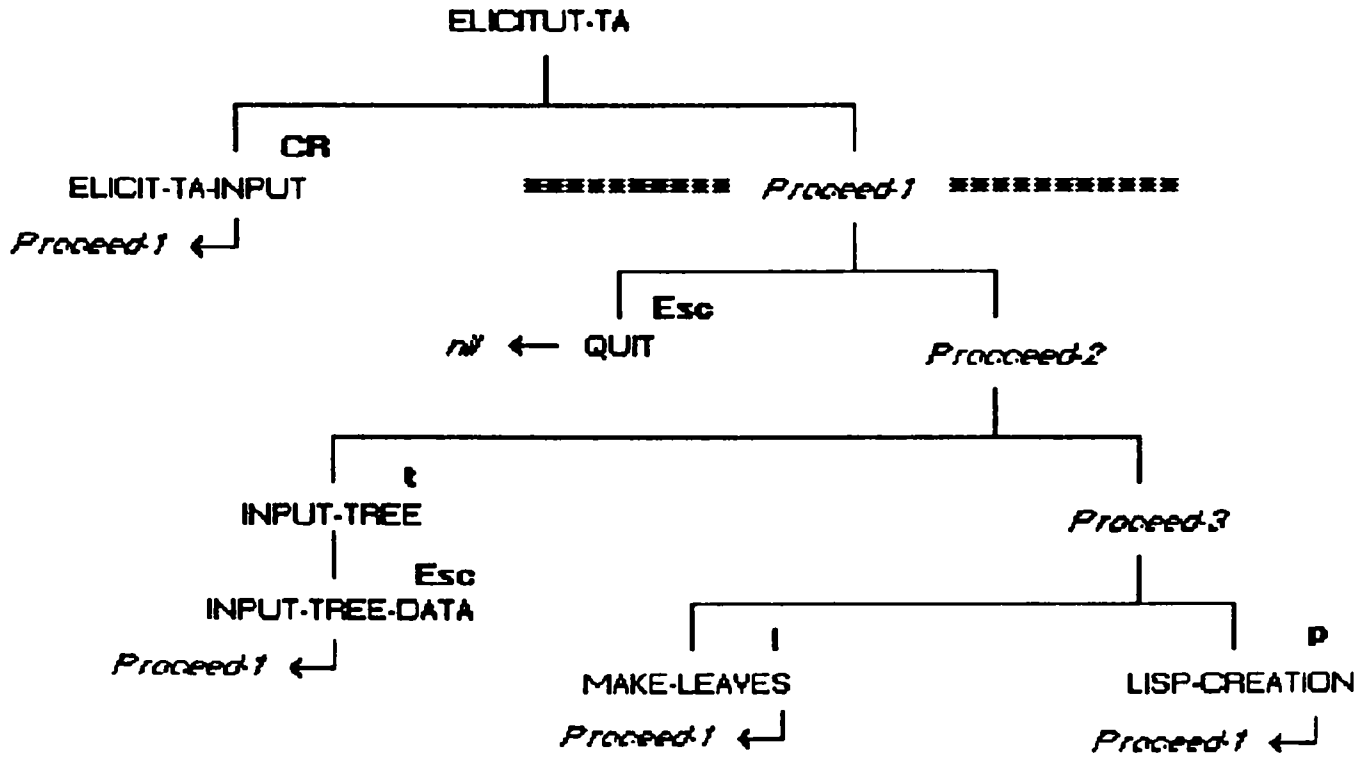
If you would like to recreate a new version, (the old version will be renamed TA.BXT), please type R.

If you would like to continue with the existing TA.TXT,
adding to it if you wish,
please type C.

Your choice..

Fig. 3.15

This impacts upon the domain model, used for the diagnosis phase. This model, represented as a transition tree built on the nodes of the task classification structure elicited from the tutorial designer, is exemplified for ELICITUT by the diagram in figure 3.16.



Esc, **t**, etc. are the invoking syntax tokens;

<-- indicates *leads_to* information: the subsequent menu item to be displayed;

******* indicates a barrier. At ELICITUT-TA, syntax tokens below the barrier are "invisible". This means that if e.g. LEARN is typed as a directory name, the "L" is not thought by the model to mean MAKE_LEAVES. Everything then *leads_to* Proceed-1.

Fig. 3.16

The diagram above represents the relationships between the individual nodes in the pedagogic task structure of the ELICITUT tutorial which teaches the use of

the ELICITOR program. This tree is not part of the ELICITOR run-time environment, only being used by ELICITUT. LIY application domain models parse the learner's key-stroke sequence against syntax tokens in a tree like that of figure 3.16. Character strings in this sequence which match the syntax tokens of nodes on sub-trees "visible" from the current node are recognised by the parser. It then designates the associated node as the new current node. If a leaf contains *leads_to* information then this is used instead to indicate the new current node. *Proceed-n* nodes are introduced merely to enable the structure to be represented as a binary tree (rather than a general tree) and are transparent. Thus tokens visible at *Proceed-1* are **Esc** at *QUIT*, **t** at *INPUT-TREE*, **l** and **p**. Any learner input which is not identifiable as a syntax token is taken off the learner's key-stroke string and treated as application input (rather than application commands), about which the model knows nothing explicitly. The parsing process then repeats. In this way LIY builds up a picture of the learner's interaction which can be compared with the result of running the designer-supplied key-stroke sequence through the same model. (There is a full discussion of this differential modelling technique in chapter 4.)

Initially the current node is set to the root of the tree. *ELICIT-TA-INPUT* is the node which deals with obtaining the name of the application, as shown in figure 3.14. The model expects the name to be followed by *CR* (carriage return, or *enter*). The difficulty here is that, if the name contains character-strings which correspond to syntax tokens visible from the current node, such input would be interpreted as application commands with the associated wrong behaviour of the model. This is not normally such a problem in menu-driven systems since application input usually occurs at leaves of a tree like that of figure 3.16. The tree is arranged so that, at a particular leaf, all input except for the terminating symbol can be considered as application input. There is an example of this on the node *INPUT-TREE*. When **t** is detected at *Proceed-1* for example, *INPUT-TREE* becomes the current node. All further input is then considered to be application input until **Esc** is detected, whereupon *Proceed-1* is reinstated as the current node by virtue of the *leads-to* symbol attached to *INPUT-TREE-DATA*.

User input to the top-level of ELICITOR does not in this instance result from a menu choice. In other words the user has not issued a command to select the option - involving the input of an application name - which could be followed by the parser on the tree. Thus the technique used for *INPUT-TREE* of routing to a leaf for application input is inappropriate. From the top node of the tree the parser cannot distinguish between commands such as **l** and **p** for *MAKE-LEAVES* and *LISP-CREATION*, say, and **l** or **p** appearing as characters of the application name input to *ELICIT-TA-INPUT*. To overcome this problem the node *Proceed-1* is designated as a barrier to the "view" from *ELICITUT-TA*, from which none of the children of *Proceed-1* are visible. Following the input of the application name the model routes control to *Proceed-1*, from which all its children *are* visible. Recall from chapter 2 that the LIY implementation is based on a rung of Jacob's ladder which includes specification of command input but not of other application input. To distinguish between the two, LIY uses the known terminators of the latter application input. This poses no problems for leaves of the tree, such as *INPUT-TREE-DATA* in figure 3.16, since from such nodes no further command input is "visible". The use of the barrier handles the case where application input is required *and* command input is possible, but no command exists to signal the start of the input and so no routing is possible to a "safe" leaf node, such as *QUIT*. (From *QUIT* no further commands are visible.) No command selection signals the start of *ELICIT-TA-INPUT*; the first thing the user must do is to type in the application name, and so the barrier is appropriate in this case. In general, it is probably only such situations, in which the dialogue starts with application input rather than a command, for which the barrier might be necessary. However, the barrier is not always required in this situation as is illustrated by considering a word processor. Application input is accepted at the top level of the tree for such a product while at the same time the command set also has to be "visible" from this point. The difference between this example and that in the ELICITUT case is that, for the word processor, there is an empty intersection set between key-strokes corresponding to data typed into a document and the command-set for the software itself. Thus word processors usually use function- or control-key sequences as commands. For ELICITUT, characters in the input file-name cannot be guaranteed to be different from all of the visible commands and thus the barrier is necessary.

There is, in the case of ELICITUT, further predictable application input in the form of the user's answers to the questions posed by ELICITOR, as illustrated in figures 3.14(b) and 3.15. The set of possible responses does not intersect with the set of commands "visible" from *Proceed-1* and so ELICITUT treats them correctly as application input. For a non-empty intersection set the problem could be overcome by adding an extra node to the tree below *ELICIT-TA-INPUT* to absorb this application input before routing to *Proceed-1*.

Application input is distinguished from command input by its terminator (often *ENTER*). It *is* possible to accept fixed-length application input in situations in which, even if command input is possible, control within the application will always transfer to the same node. Such a situation has been exploited with certain single-character inputs to the DIALLER but should not in general be thought of as a standard feature of LIY.

```
PLEASE CHOOSE ONE OF THE FOLLOWING OPTIONS :-

For describing the task tree, press                    T

For finishing the tree, declaring
the remaining nodes to be leaves,                      L

For turning the task analysis file .TXT
into a runnable .LSP file,                             P

To exit the LIY-MAKE program, press                    Esc

Your choice..
```

Fig. 3.17

Figure 3.17 shows the main menu screen of the ELICITOR program. (On the ELICITUT domain model of figure 3.16 it corresponds with the "view" from the node *Proceed-1*.) Initially the tutorial designer will use option "T" to build the pedagogic task tree structure. A screen rather like that of figure 3.18 will then appear. In fact this diagram illustrates the situation after the designer has already input some data. The program performs a certain amount of error-checking: other than at top level, "parents" can only be declared if they have already been made known as "children". Figure 3.18 illustrates an error whereby the child symbol *learn-ta* is input although it has already been declared (as a parent).

To quit this TA creation phase at any time, please
press Escape

Please enter each node in the task structure and follow it with ENTER :

> *LEARN-TA CONSISTS-OF* > part1 > part2

Is this OK? :

LEARN-TA CONSISTS-OF PART1 PART2 (Y or N) Yes

> part1 ***CONSISTS-OF*** > learn-ta

Error - LEARN-TA - already declared as a parent

>

Fig. 3.18

Input to the ELICITOR program to build the tree is stored as a text file which could be edited with any text editor. The format of the structure requires that all nodes eventually be declared as parents. If they have no children then the *CONSISTS-OF* parts must be declared explicitly to be *nil*. Such leaf nodes, declared only as children, can be generated (as parents with no children) by use of option "L". A correct text file can be generated as a file of Lisp code by use of option "P". Subsequently, nodes in this file can be enhanced in order to add Lisp code to designate teaching operations associated with the node, for example. To do this, the structure input by the designer is displayed on the screen and individual nodes can be selected with the mouse. The user interface has not been further developed, so that at present this enhancement requires the addition of statements, from a small Lisp repertoire, representing data elements in list form.

3.4.2 LIY's teaching operations

LIY offers the tutorial designer nine low-level operations which can in general be categorised as being associated either with teaching or with assessment. These teaching operations can be attached to any individual node in the pedagogic task structure. The six principal operations were listed earlier and are discussed here. As a Lisp data structure, a set of teaching operations is represented as a list, within which there exists in list form a code for each particular teaching operation followed by any associated arguments. For example, the code for a slide-show is "S" and its associated argument is the path-name of a file containing the text to be displayed for the slide-show.

The six principal operations are:

(i) slide-show : "S"

This is simply text concerning a particular topic to be displayed to the learner. There is a particular property attaching to a slide-show: it will always precede other operations for a given node (although there can be more than one slide-

show per topic). If the learner's response is diagnosed as being in error then LIY backs up to the previous slide-show for re-display. A slide-show and its subsequent teaching operations up to the next slide-show (if any) make up a ***fragment***. In other words, a set of teaching operations for a particular topic consists of a set of fragments; each fragment consists of a slide-show followed by zero or more other teaching operations.

(ii) create an application environment : "E"

This sets up the application in some particular way. It is very similar to placing the learner at some desired point in the application (type "P"), the difference being that this operation causes the application to terminate. It is thus invisible to the learner although its effect is to change the state of the application's environment.

(iii) watch and record learner input : "W"

This is used by the designer when the learner is to interact with the application software. It implies that diagnosis will subsequently take place and causes the learner's key-stroke sequence to be recorded. The argument is the minimal correct string to be used in the diagnosis phase.

(iv) place the learner at some chosen point : "P"

This causes the application to be invoked and the learner to be presented with a screen which normally corresponds with the topic which is being currently taught. The argument to this command can be nil, in which case LIY generates a command sequence which is appropriate for driving the application to the screen associated with the currently-taught node. If it is a node-name then the screen associated with this node is displayed. Alternatively the argument can be a command-string which will simply be executed as if it were application input from the keyboard. In both this operation and the application environment option "E", screen-formatting output is suppressed by the replacement output routines where possible.

(v) get learner input directly : "G"

This is the "immediate" assessment mode illustrated in figures 3.9 and 3.10. Rather than watching the learner interacting with the application software, LIY seeks a response to some question directly. The argument for this teaching operation is simply the correct response.

(vi) free learner exploration of the application : "F"

This operation enables the learner to explore the application to some purpose, although there is no diagnosis of the learner's performance.

In addition there are a further three operations which are tutorial-oriented rather than learner-centred:

(vii) re-run designer rules : "D"

- only when the learner is repeating a set of teaching operations following diagnosis of an error. There are no arguments. Designer rules are discussed below.

(viii) re-run "set-up" operations : "R"

- only when the learner is repeating a set of teaching operations following diagnosis of an error. This operation re-runs a concatenation of "E", "W" and "P" operations for *previous* fragments on this node, so that the application environment for the learner is re-created in exactly the desired way. There are no arguments.

(ix) execute Lisp code directly : "X"

This has been useful on just one occasion to invoke execution of a program other than the application: ELICITUT uses it to run the program which lets the

designer select with the mouse a node to enhance. Its argument is a Lisp expression to be evaluated.

The tutorial delivery system processes teaching operations sequentially within a fragment and for computational reasons expects them to be in the sequence:

S, D, E, R, W, P, G, F, X

Thus a fragment starts with a slide-show and then, broadly speaking, sets up the learner's environment in the application, preparing for assessment if necessary, and launches the learner into the application.

3.4.3 Designer rules

The behaviour of an LIY tutorial is governed by the teaching strategy represented by the fixed sets of rules introduced earlier in section 3.2. The format of the rules is described in section 3.6. LIY permits the designer to over-ride this fixed behaviour by adding extra rules in one of two categories: control rules and teaching operation rules.

The existence of designer control rules is probed by the rule interpreter. If such a rule-set exists then it is interpreted. Whether or not it exists, control then passes to LIY's standard control rules. This facility is used in ELICITUT in order to probe for, and delete if necessary, any directory already in existence for the application *LEARN* which the learner is required to cause ELICITOR to create. Attempting to create such a directory if it were already in existence would cause the operating system to report an error - an event to be avoided at all costs.

Designer teaching operation rules take over entirely from the default rule-set. Normally, rules in the set TEACHING-OP-RULES (the "e" rule-set) are used to process the designer's teaching operations, described above, which are attached to the nodes in the task classification tree. If this rule-set were to detect, at the outset, the existence of a set of designer teaching operation rules then it would invoke the rule interpreter upon this set. It would not regain control until the teaching of the particular node was complete. As an example, this facility could be used by the tutorial designer to replace the standard rule-set with a slightly edited version. This latter rule-set could be augmented with output messages so that, following detection of a learner error, very specific information about the current task could be provided to the learner. Further Lisp development would probably be required in order to utilise fully the under-documented Lisp functions invoked by the standard teaching operation rules or to include designer-written functions. Although tested, this facility has not been used so far in either of the two LIY tutorials.

3.4.4 Further aspects

In addition to the teaching operations which the designer will need to attach to each node, there are a number of other features of a node that will require consideration. They include, for example, the barrier and *leads_to* information shown in figure 3.16. Section 3.5.3 below contains a complete description of all such information.

3.5 Operational and pedagogic task/subtask hierarchies

One of the outputs from the systems analysis stage of a software implementation is an operational task classification tree. This structure contains the elements (nodes) representing operations that the learner or user of an interface can invoke. (Rather than using the DIALLER or ELICITOR programs to illustrate this section, a subset of MS-DOS has been chosen since it provides wider-ranging examples, exercising a higher proportion of the transformation heuristics proposed below.) Thus for a subset of MS-DOS the structure might look initially like figure 3.19. *dos*, *execute* and *park* are italicised to emphasise that they are not actual MS-DOS commands. *dos* is included in order to give a name to the domain, while *park* is inserted to illustrate the subsequent discussion of *exit*-type commands; being an operating system for a personal machine, there is no explicit exit mechanism for leaving MS-DOS. Application programs in MS-DOS are run by typing their name. In the diagram, *execute* serves the purpose of naming a node in the tutorial space to describe this function. If an LIY-type tutorial existed for MS-DOS and the learner invoked such an application program from within the operating system tutorial, it would be desirable for the application to be treated as a "black box" from the perspective of the MS-DOS domain model in the tutor. However, it would also be desirable for a tutorial to be available for the application program's user interface, in which case invoking the application from the operating system could be represented as a suspension of the MS-DOS tutorial followed by *free exploration* of the application. The exploratory mode is always available following a learner interruption and was described in section 3.3.1.

There are several important features to note about figure 3.19. First, the tree represents a constructive operational partial ordering in terms of the hierarchy: it could be read as "*dos* **consists-of** the operations *cd* ... *park*". Secondly, the tree is a general tree in the sense that a node may have an arbitrarily large number of "children". However, algorithms are generally less complex for processing binary trees, in which the number of children at any node is limited to not more than two. General trees are transformed into binary trees according to the tree transformations numbered 1 to 7 in figure 3.21 of section 3.5.1. To

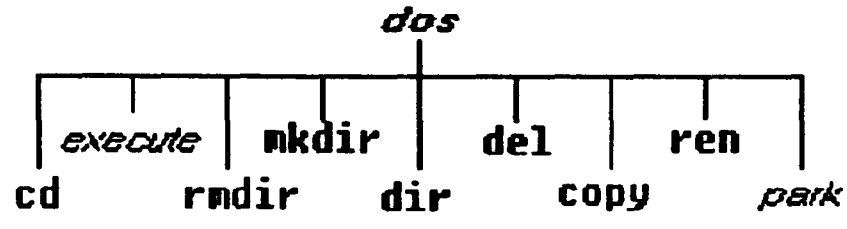


Fig. 3.19

do this, "dummy" nodes labelled *proceed-1*, *proceed-2* etc. are introduced. Note that these transformations are distinct from the heuristic rules labelled (a) to (g) in section 3.5.2.

From the point of view of the tutorial, a structure is required which incorporates not only nodes corresponding to the operations in figure 3.19 but also nodes concerned with extracted information which might be common to more than one node, together with dependency information to help prescribe a tutorial ordering over the task tree. This type of structure is referred to as a pedagogic task classification structure. It is produced by applying heuristic rules and transformations.

Furthermore, the tutorial viewpoint requires a different perspective on the classification tree. Instead of the operational ordering of figure 3.19 (*dos* **consists-of** ...) a tutorial emphasis dictates a structure which reflects knowledge: *knowledge about dos* **consists-of** *knowledge* ... This difference in emphasis cannot be achieved by LIY alone; it requires the application of the heuristic rules

described below. Knowledge about taxonomies of teaching topics is incorporated into the rules, which have been successfully applied "by hand" to a number of small domains. However, in no way is it claimed that the transformations could be done automatically by machine. This is precluded by the semantic content of the information needed; taking rule (a) applied to the MS-DOS example, "identify groupings" requires that the tutorial designer be aware of the fact that the *directory* commands should be grouped together.

3.5.1 Dependency and binary tree transformation

LIY, in teaching an application domain, traverses the tree in pre-order (root, left sub-tree, right sub-tree), but uses dependency information in determining advice to give to the learner on receipt of a learner-initiated navigation request. This was described earlier in this chapter. Dependency between sibling sub-trees at the same level results from the application of the heuristics given in the next section and is denoted by marking the arcs as shown in figure 3.20. Pre-order traversal as an LIY teaching strategy appears to be very similar to the cognitive apprenticeship strategy of DOMINIE (Spensley *et al.* 1990).

Figure 3.20 (a) represents the fact that knowledge about topic ***a*** consists of knowledge about topic ***b*** and topic ***c***. Figure 3.20 (b) also represents this, but additionally that knowledge about ***c*** is dependent upon knowledge about ***b***; ***b*** is a prerequisite for ***c***. (In the definition below it would be said that "***b*** is linked by DEPEND to ***c***".) Thus the learner would be advised to learn ***b*** before ***c***; without any learner-control intervention LIY would teach the topics in the sequence ***a***, ***b***, ***c***. A learner interrupting in the case of figure 3.20 (a) would be permitted to learn ***c*** without comment from the tutor, irrespective of the state of learner knowledge of ***b***.

Fig. 3.20

Prerequisite topics of a particular node are defined as follows:

> *All leaf-nodes in a left sub-tree which is linked by DEPEND to some node X, and leaf-nodes in left sub-trees which are linked by DEPEND to ancestors of X, are prerequisites of X.*

The focus upon leaf nodes in this definition attempts to reduce the set of possible prerequisite topics to those which are crucial: i.e. to those topics concerned directly with commands which manipulate the application software interface being taught.

Figure 3.21 shows a subset of the possible transformations, developed by trial-and-error, from general to binary trees. Arrows show specific dependencies, e.g. in transformation (1) ***c*** and ***d*** both depend on ***b***, but ***d*** does not depend on ***c***.

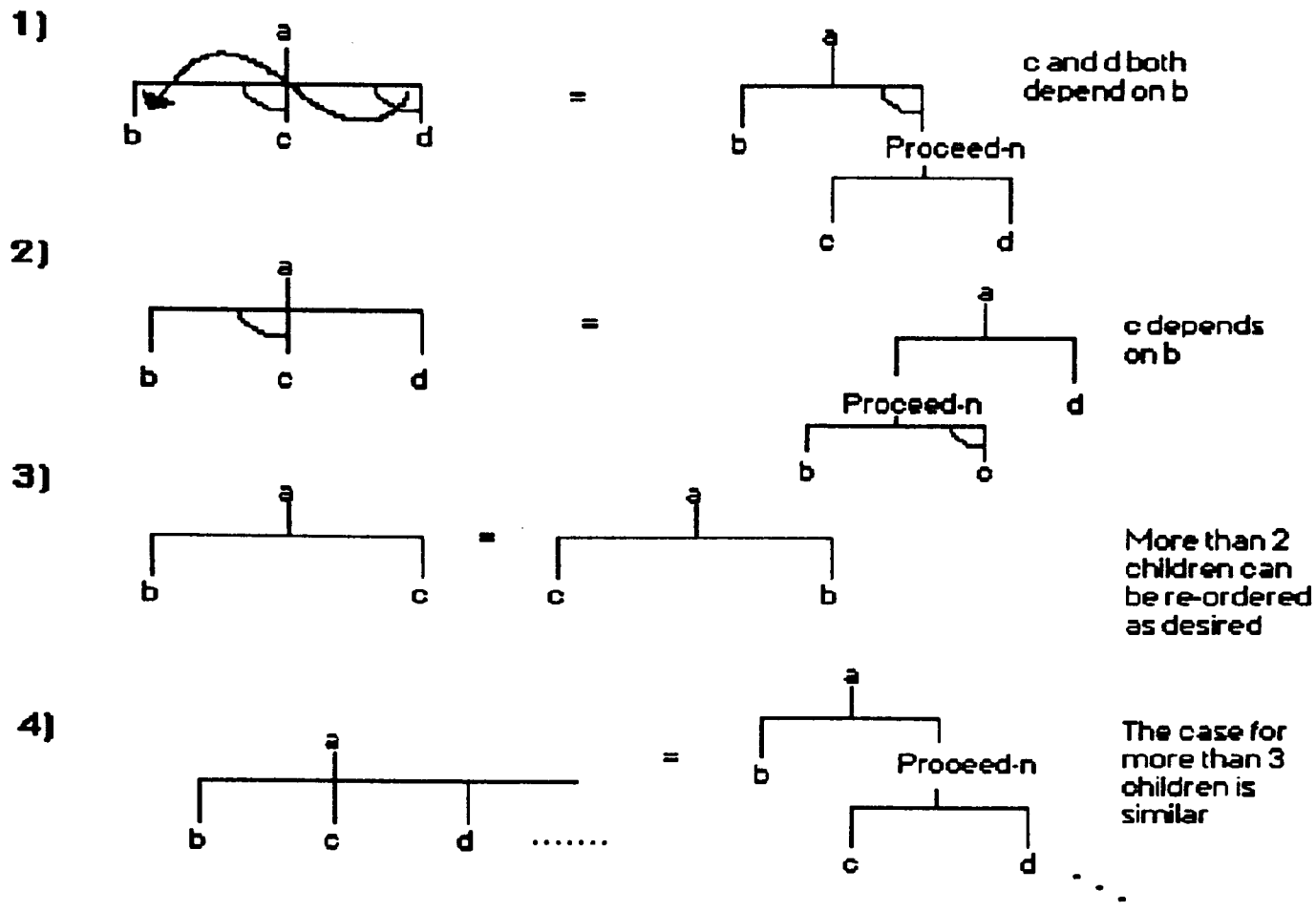

1)
a
b
c
d
=
a
b
Proceed-n
c
d
c and d both depend on b
2)
a
b
c
d
=
a
Proceed-n
d
b
o
c depends on b
3)
a
b
c
=
a
c
b
More than 2 children can be re-ordered as desired
4)
a
b
c
d
.....
=
a
b
Proceed-n
c
d
The case for more than 3 children is similar
5)
a
b
c
d
=
a
c depends on b
d depends on b and c
Proceed-n
d
b
o
6)
a
o,d,e all depend on b
b
c
d
e
.....
a
b
Proceed-1
o
Proceed-2
d
e
7)
a
b
c
d
e
.....
a
b
Proceed-1
c
Proceed-2
d
e
c depends on b, d depends on c, etc.

Fig. 3.21

Looking ahead to figure 3.26, (which shows the result of applying the heuristics set out in the next section to the MS-DOS example), it can be seen that *rename* is a prerequisite for *rmdir*. This is not ideal, but is a consequence of putting all knowledge concerning files together as a whole, and then making all of it a prerequisite for knowledge concerning alteration of directories. Such groupings of material face human teachers every day when they make judgements involving the trade-offs inherent in various possible taxonomic approaches to the subject they are teaching. For LIY this is a potential complication only when the learner seeks to vary the pre-determined ordering of the topics, since this is the only time when the dependency arcs are examined.

3.5.2 Transformation to pedagogic ordering

The heuristics shown below have been developed as a result of experience with a number of example domains. These domains are the MS-DOS example described in this section, the DIALLER and ELICITOR (the development of which is described in appendices B and C respectively) and a financial application not reported here. These heuristics can be applied recursively at any level in the tree. Not all the heuristics and transformations are needed for the MS-DOS example.

(a) Identify common topics and group them together.

(b) Identify commonality between topics and extract as a new topic.

(c) Teach any "exit"-type topic first. Place it as the left child of the root, dependency-linked to the rest.

(d) Teach any "configuration"-type function last. Place it at the deepest point in the right sub-tree, non-dependency-linked.

(e) Arrange non-dependently-linked nodes at the same level within a sub-tree in order, with the most important on the left.

(f) Incorporate dependency knowledge not already present.

(g) Place destructive operations (e.g. "delete"-type functions) on any level to the right, dependently-linked. Such operations are invariably dependent upon a corresponding creative activity.

Consider how these rules could be applied to the MS-DOS subset of figure 3.19. Note that *park* (park hard disk heads before switching off) was explicitly inserted in this example to animate rule (c) about teaching "exit"-type functions first. This corresponds to the *quit* operation in many applications, including DIALLER.

Application of rules (a), (b), (c) and (g) to figure 3.19 yields figure 3.22.

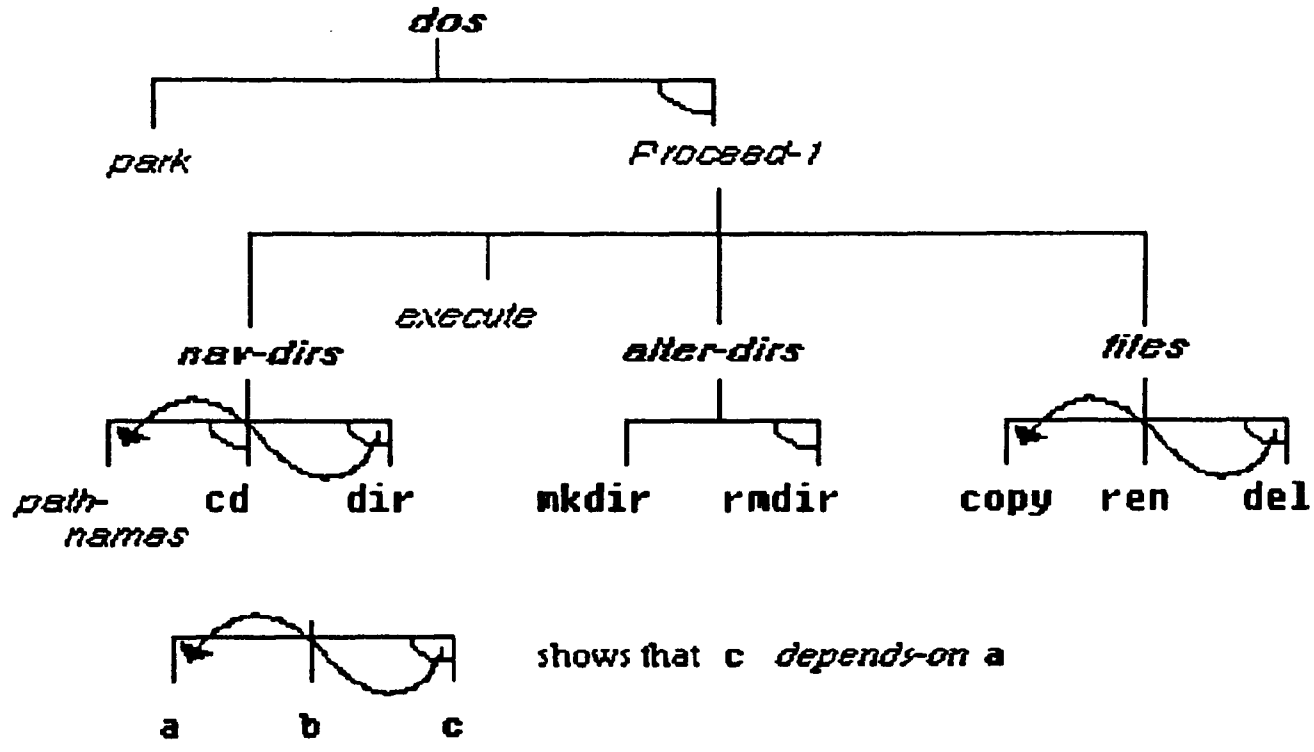


Fig. 3.22

nav-dirs, *alter-dirs* and *files* result from the groupings identified as a result of applying rule (a). Rule (b) yields *path-names* as a result of identifying the prerequisite knowledge common to the directory-handling commands *cd* and *dir*. Rule (c) causes the exit-type command *park* to appear top-left. Note that rule (g) is used to provide order and dependency knowledge within both *alter-dirs* and *files*.

Applying heuristics (e) and (f) to figure 3.22 yields a top level as shown in figure 3.23.

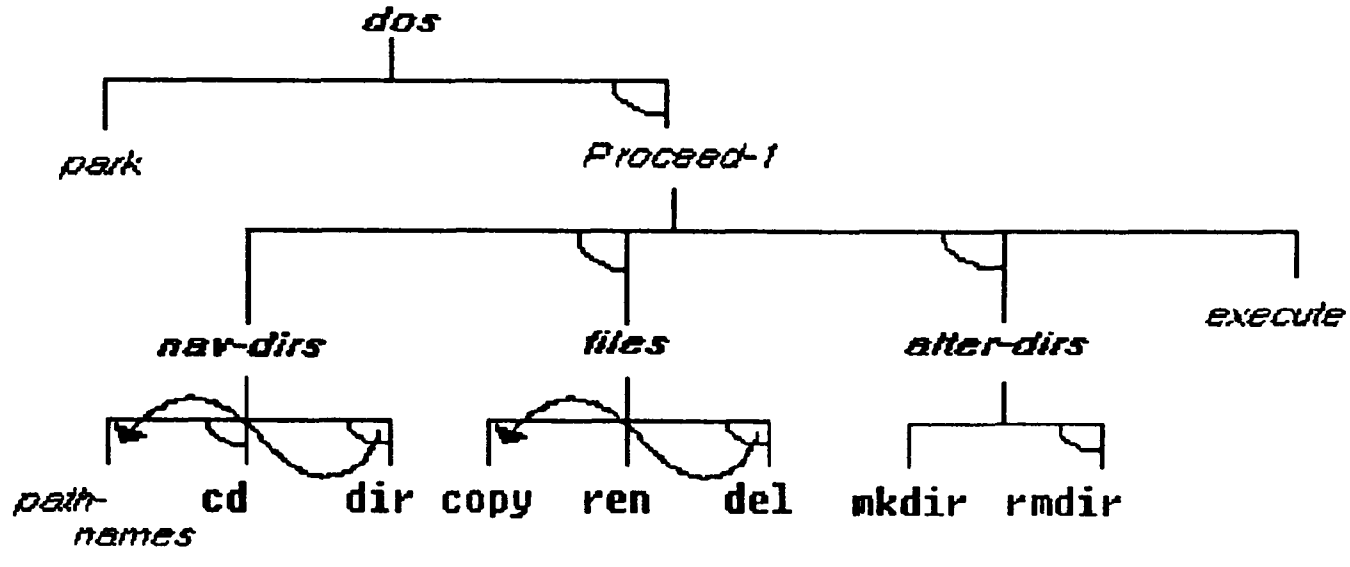


Fig. 3.23

The *nav-dirs* sub-tree from figure 3.22 can be transformed using transformation 6 to appear as shown in binary-tree form in figure 3.24. *files* can be transformed by applying a variation on transformation 3 (order equivalence for more than two children) and then transformation 2.

Applying transformations 2 and 7 to the task level shown in figure 3.23 gives the structure of figure 3.25.

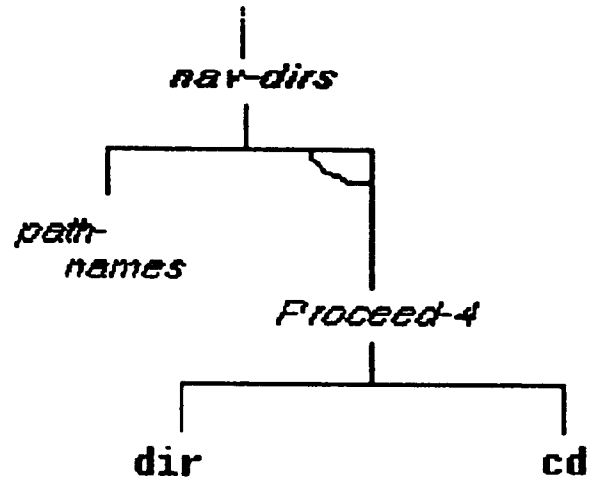


Fig. 3.24

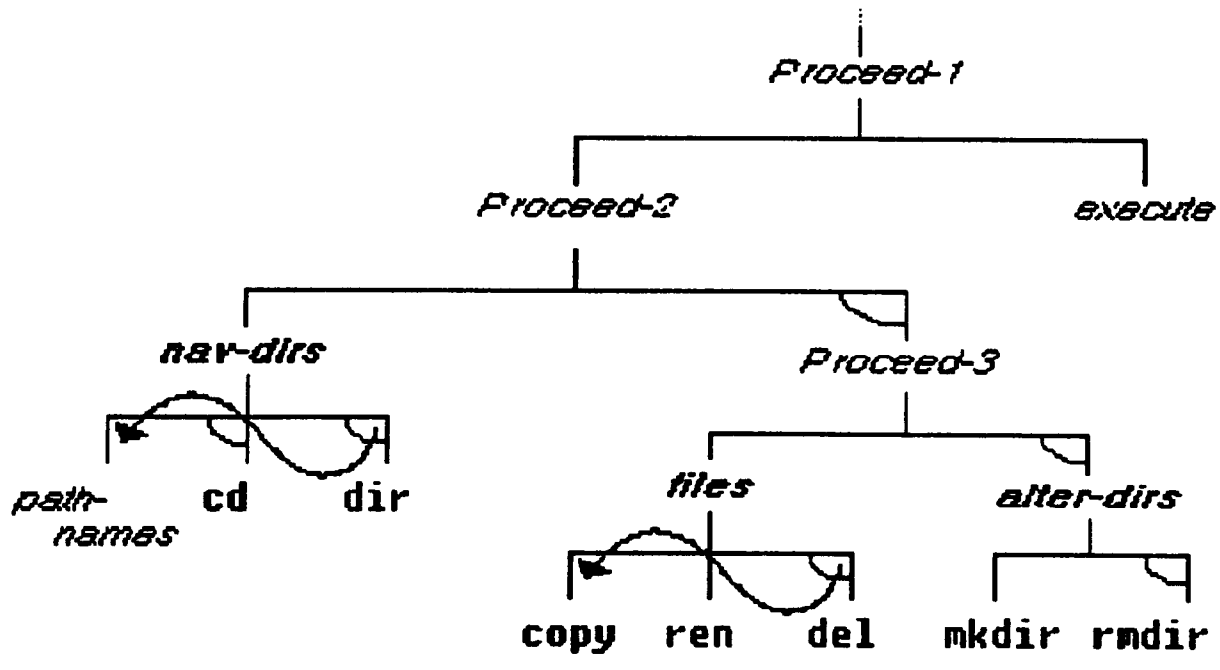


Fig. 3.25

Putting all this together gives us the completed pedagogic task classification structure shown in figure 3.26.

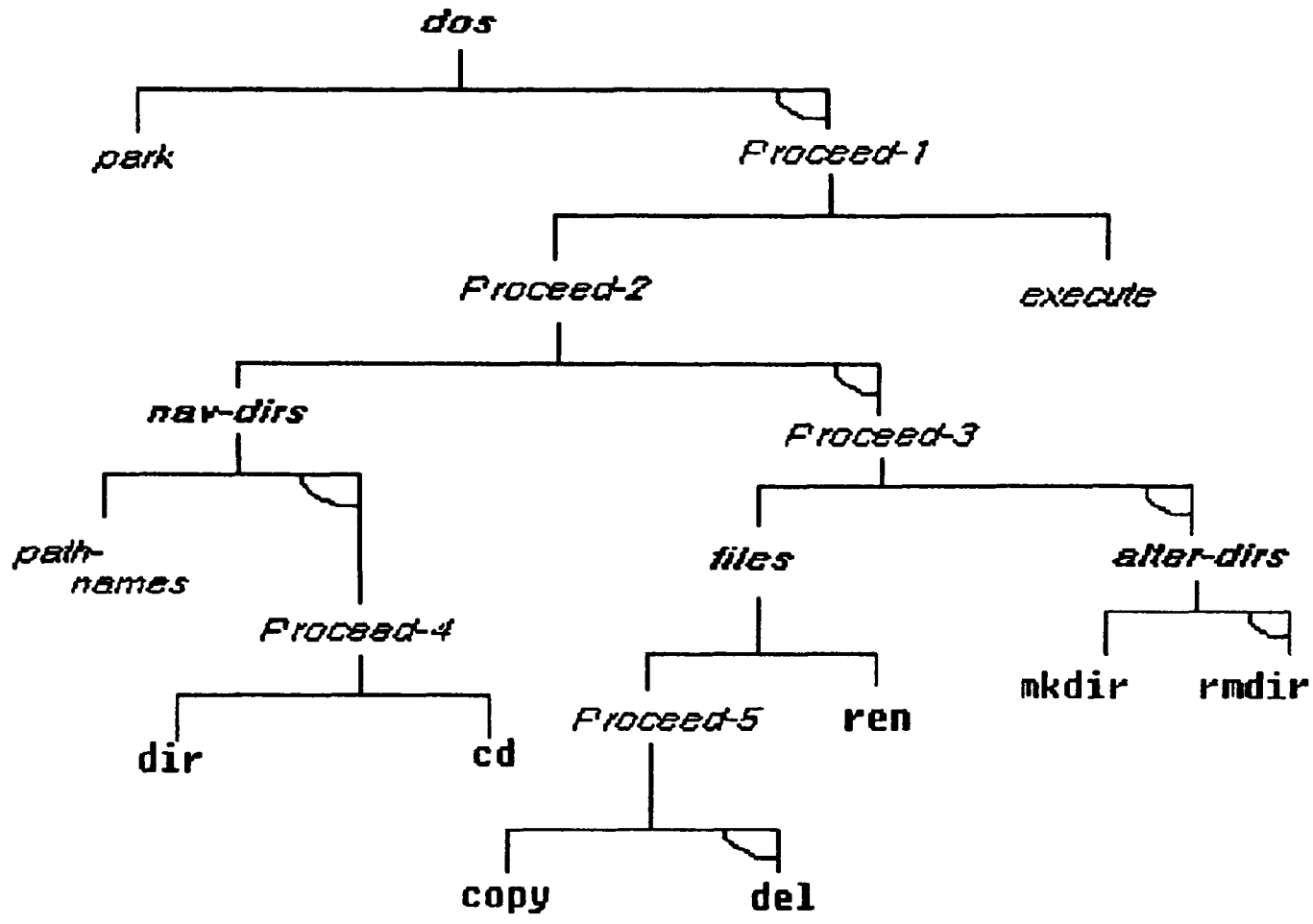


Fig. 3.26

It can be seen that all the nodes for which there is a corresponding operation within the application domain are in this case leaves of the tree. However, this need not always be the case, and is not so in the DIALLER application.

The commands in this domain are multi-letter. Although for simplicity the commands in the two LIY applications discussed earlier all happen to be single characters, LIY can also handle multi-letter commands such as those of the MS-DOS example. Syntax tokens are used by the tutorial designer to represent commands and were discussed in section 3.4.1. These are matched during diagnosis against the head of a recorded list of the key-strokes typed by the learner into the application. Accordingly they can be of any non-zero length.

3.5.3 Complete description of a pedagogic task classification tree

Every node in a task classification tree may exhibit the following attributes:

CONSISTS-OF

Indicates the names of up to two child nodes.

LINKS-BY

Indicates whether the children of this node link dependently or independently.

PARENT

For all children LIY generates the names of the parent node.

SYN-TOKEN

The value of a character-string for invoking the command associated with this node. Used in the domain model.

BARRIER

Used in the domain model to "hide" syntax tokens under certain circumstances.

STATE-CHANGING

Used during learner diagnosis. Described in chapter 4.

LEARNT

Part of the learner profile.

CURRENTLY-BEING-TAUGHT

Used by LIY internally when handling learner-control interventions.

X-CO-ORD, Y-CO-ORD

Used by LIY internally when handling a graphical display of the task structure.

3.6 Managing tutorial delivery

LIY's basic teaching strategy is to traverse the task classification tree looking for a topic to teach. The tree is traversed in pre-order, that is the parent node is examined first as a possible candidate, followed by a recursive search of its left sub-tree and then of its right sub-tree. Only untaught and non-dummy nodes (i.e. not named *proceed-n*) are candidates for this strategy.

The control behaviour of LIY's tutorial delivery mechanism is encapsulated as a rule-base. No claim is made that the rules represent a "knowledge base"; rather, they provide a distinction between the way in which, at a fairly high level, the rules declare *what* needs to be done, and the low-level procedural descriptions of *how* it should actually be carried out.

Appendix A contains a complete list of all the rules in LIY. Rules are in the format of Lisp lists, as follows:

```
(rule-name
        (zero or more antecedent conditions)
        (one or more consequent actions)
)
```

It can be seen from the appendix that both antecedents and consequents are in fact coded as Lisp. Where they would involve any complexity at the Lisp level they have been recoded as Lisp function calls. Functions which are called by the rules have not been turned into a "language" for general use. As pointed out earlier, in the discussion of teaching strategy in section 3.2.3, a rule consequent can include the invocation of the interpreter on another rule-set.

A rule-set is a list of rules assigned to some variable which is the name of that rule-set. In LIY the convention used is that rule-set names are descriptive, whereas rule-names take an arbitrary letter for each rule-set followed by an integer incrementing by the value ten. Thus, for example, rules in MAIN-CONTROL-RULES are named a10, a20 and so on.

The search for a topic to teach is initiated by the top-level rule-set MAIN-CONTROL-RULES (the "a" rule-set), which also handles detection of a learner-control request and proper termination when no topic to teach can be found. As explained in section 3.4.3, if there is a file of "designer" control rules then these are interpreted. In any case, the interpreter is then applied to the rules concerned with selecting a teaching operation. These rules constitute the set TEACHING-OP-RULES. Figure 3.27 illustrates the structure of the rule-base.

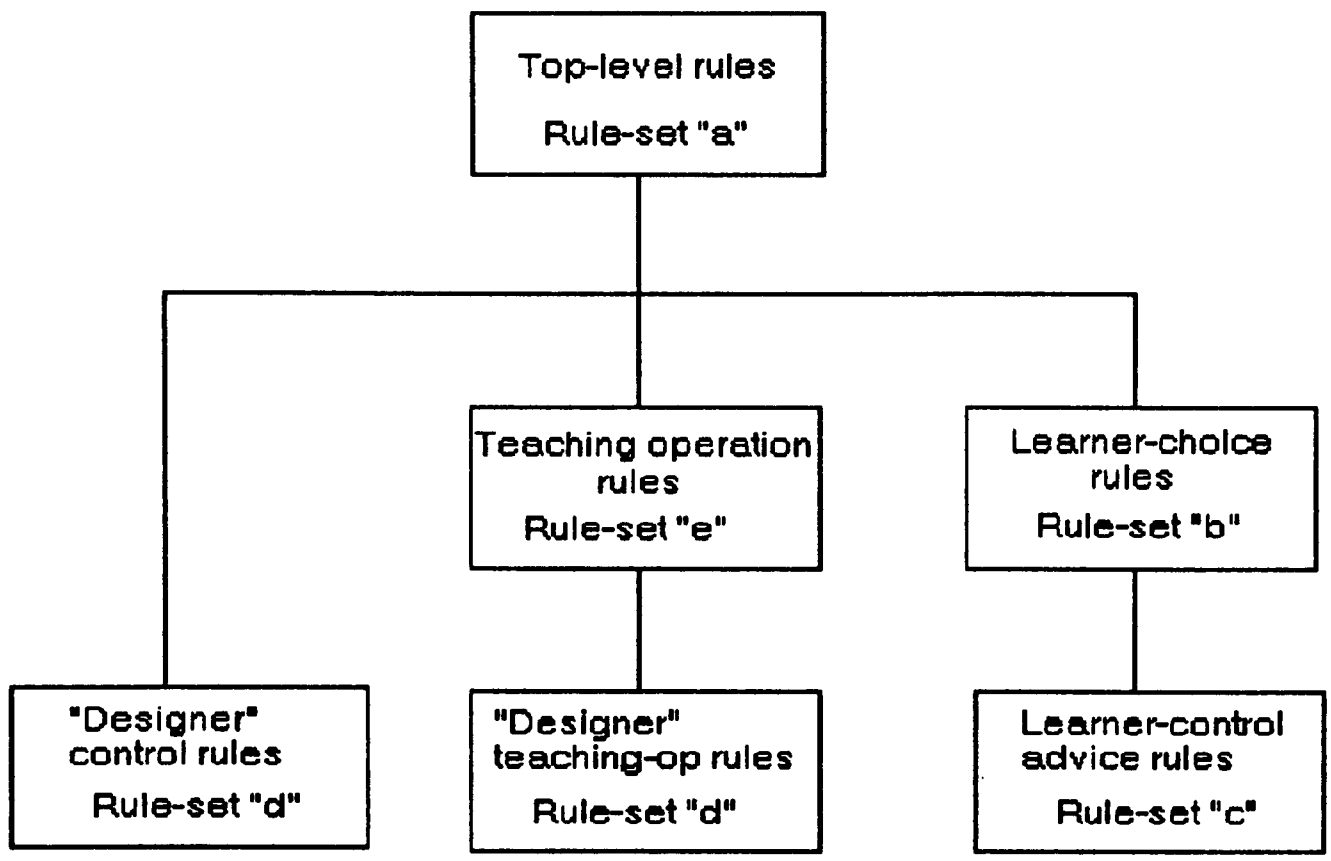


Fig. 3.27

Rules in TEACHING-OP-RULES are named starting with the letter "e". If any designer-supplied teaching operation rules exist - as opposed to designer *control* rules - these take over completely. (In principle such rules could be used to replace the current LIY teaching and assessment strategy, although to permit the tutorial designer to do so would require that a "rule language" be developed.

At present LIY tutorials do not use this facility.) Otherwise the remaining rules in TEACHING-OP-RULES are interpreted. These rules display tutorial text, handle assessment, and update the "competence" aspects of the learner profile. If, following diagnosis, the learner's input cannot be matched with expected results, rules in this set cause LIY to back up once for re-display and re-assessment on the current topic, and once again if necessary. In the latter case the learner is given the option of being shown the expected response and moving on to the next teaching fragment, which may or may not be within the current topic.

A typical rule from this set is:

```
(e40
        ((equal *LAST-INTERACTION* 'TEST)
         (equal *LAST-TEST-RESULT* 'PASS))
        ((INCR-COMPTNCE))
)
```

This increases the learner's competence score in the learner profile for the successful completion of an assessment. Note that the rule consequent invokes a Lisp function; this is typical of many LIY rules.

If the learner interrupts then control transfers immediately to the top-level rule-set MAIN-CONTROL-RULES, which then hands on to the "b" rule-set LEARNER-CHOICE. This offers the learner the menu of options shown in figure 3.5. If the learner has requested a move to a different topic then this is permitted provided that all prerequisites have been learnt. If the target topic has already been learnt then the fact that the learner is revising is reported. If there are unlearnt prerequisites then the learner-advice rule-set L-C-ADVICE, comprising the "c" rules, is invoked. This rule-set considers the learner's profile and the number of outstanding prerequisites and will either permit the move or offer graduated advice against it, as described earlier. If desired, the learner can always force a move against LIY's advice.

The letter "d" is reserved by convention for rules in the designer rule-set. As discussed previously, DESIGNER-CONTROL-RULES are used in ELICITUT for deleting any pre-existing sub-directory named LEARN.

If the learner quits without completing the tutorial then his or her environment is saved to allow continuation from this point on some future occasion.

3.7 Concluding remarks

This chapter has attempted to convey the flavour of the LIY software from the perspectives of both the learner and the tutorial designer.

The learner's viewpoint is illustrated by the DIALLER tutorial. The context of this research is that of software engineering rather than psychological evaluation and the major aim is to test the feasibility of the LIY approach rather than testing the efficacy of the two LIY tutorials themselves. Nonetheless the tutorials have been tried by a small number of knowledgable colleagues, friends and family. Their perspective has been that of learner rather than designer, and thus their input has been incorporated into the delivery, rather than authoring, aspects of LIY. Improvements made as a result of users' suggestions included error feedback to the learner following *all* inappropriate responses, rather than only on second and subsequent such responses; including the name of the package (e.g. "DIALLER") in messages; and enhancing the feedback to include a reference to the response-comparison heuristics described in the next chapter and illustrated by the messages in figure 3.11.

The designer's viewpoint has been illustrated with particular reference to ELICITUT, a tutorial for the ELICITOR software, and to MS-DOS, a well-known but (for LIY) hypothetical example. The latter was chosen in order to demonstrate the transformation heuristics for producing a pedagogically-oriented task classification structure.

Thus this chapter has, through examples, demonstrated the capabilities of LIY. Appendices B and C take the reader through the design, based on the specification elements discussed in chapter 2, of the tutorial structures for the DIALLER and ELICITUT respectively.

Chapter 4

An ITS perspective on LIY

This chapter considers the design and implementation of LIY within the context of *Intelligent Tutoring Systems* (ITSs). It also speculatively proposes a more general version of the so-called *five-ring model* (O'Shea *et al.* 1984). This is referred to here as the *figure-of-eight model* and would appear not only to support learner control but also to offer greater "intelligence" by being able to adapt or change its teaching strategy dynamically.

4.1 What is an ITS?

Near the beginning of his book, Wenger describes an intelligent tutoring system, or *knowledge communication system* to use his term, as consisting of four components (Wenger 1987). These components are:

domain expertise;
student model[1];
pedagogical expertise;
interface.

Pedagogical expertise in fact consists of "didactic process", which incorporates elements of Hartley and Sleeman's teaching strategy and teaching operations, and "degrees of control", also incorporating teaching strategy elements (Hartley and Sleeman 1973).

In a review paper Dede describes almost exactly the same architecture as Wenger's, although he uses the term *knowledge base* rather than *domain expertise* (Dede 1986).

The distinction between these modern views and the earlier perspective is not simply the addition of an interface to the learner - which was in any case implicit in Hartley and Sleeman's architecture - but more a refinement of the common components. To take just one example, the learner model is currently perceived as consisting of at least two sub-components. These perform the functions of predicting individual learner behaviour and diagnosing the causes of exhibited learner behaviour. Thus, considering the *predictive* aspect of the learner model, it can be used to try out a proposed teaching operation in order to establish that operation's suitability. The diagnostic function can be used particularly to identify and remediate recent non-optimal learner responses within some context, such as responding to an assessment teaching operation.

---

[1] The term *student model* has wide currency. LIY is concerned with learners rather than students, so the equivalent term *learner model* is used here in preference.

So modern ITSs have developed within the framework of the earlier systems. They exhibit better learner modelling, for example involving the use of more sophisticated cognitive models, bug catalogues for identifying learner misconceptions, and plan recognition strategies {*The Lisp Tutor* (Anderson and Reiser 1985)}. They employ better user interfaces with more powerful natural language capabilities based on a deeper understanding of the structure of discourse {*SOPHIE* (Brown *et al.* 1982)}, or based on simulation coupled with high-quality graphics {*STEAMER* (Hollan *et al.* 1987)}, and so on. As Hartley and Sleeman proposed, modern systems use ideas of search and inference from artificial intelligence to match learner behaviour against a model of that learner in order to optimise teaching and learning processes.

4.2 LIY: the ITS viewpoint

One of the principal aims of the LIY project is to demonstrate the feasibility of a portable tutoring shell. In order to demonstrate this portability it has been necessary to implement a complete tutoring system, rather than restricting research to one or two of the aspects of modern ITSs discussed above. LIY does not address *all* the aspects of ITS research, however. For example, the learner interface both to the delivery and authoring systems has not been developed very far. The following sub-sections describe LIY's domain modelling, learner modelling and teaching strategy from an ITS point of view. An alternative architecture for intelligent tutoring systems with learner control is proposed. In some cases where material has already been covered in earlier chapters the corresponding sections here are brief.

4.2.1 Modelling the domain

The principal requirements of domain modelling in LIY are to provide a pedagogic structure, which can be used to determine the next teaching process, and to support diagnosis. The basic knowledge structure of the domain model is the task classification tree. This structure, originally representing the operational ordering of the user command set, is transformed by the tutorial designer to a pedagogic ordering according to a set of heuristics. Chapter 3 contains a detailed description of the structure and the transformation process.

Domain knowledge is limited by the "rung" of Jacob's ladder selected for the interface specification (and by LIY's portability requirement) to the command structure of the domain and the syntax of the commands within it, forming a prerequisite knowledge hierarchy.

During the diagnosis phase the domain model is traversed under the control of a transition tree parser.

4.2.2 Modelling the learner

Profiles

In principle there are two *profiles* maintained for the learner: the *characterisation profile* and the *performance profile*. In fact they are not implemented as separate objects; they each comprise a number of separate components which are inspected and maintained under the control of the rule-base as described in chapter 3.

The characterisation profile is designed to reflect aspects of the learner which might be carried across from one tutorial to another. Consider as an example the tutoring system's perception of an individual learner's preferred learning style on the holist/serialist continuum. If the learner were to make considerable use of the learner control facility it could be inferred that he or she was trying to break away from LIY's serialist tutorial style.

The performance profile contains knowledge about the learner's current state both on a moment-to-moment basis and over time, particularly with respect to the tutorial's perception of the learner's achievements with assignments. It is used by the rule-base to attenuate the advice offered when a learner-control request to navigate to a topic with unlearnt prerequisites is encountered. It also comprises a measure of the learner's mastery of topics by marking the task classification tree.

Diagnosis

The diagnosis process in LIY attempts to determine the "correctness" of learner input to the software application being taught by comparing it with that provided by the tutorial designer. The comparison is more than just a character-for-character look-up, however. Differential modelling is used to compare "issues" in the learner's application input with those in the designer's input. This is

somewhat similar to the use of issues in WEST (Burton and Brown 1982). In the context of playing the game WEST, issue identification is used in particular to provide feedback concerning the player's moves. Issues in the game are skills which the player or the expert might utilise. In LIY the "issues" are four heuristic criteria, described below, concerned principally with the order and type of commands used by the learner and the designer. Interestingly, the authors of WEST suggest that issues, or rather knowledge about specific issues, can be used to provide context-sensitive help if it is requested. Developing the issue-based tutoring idea further than it has been taken in LIY would require much more domain-specific knowledge than that available from the task classification tree.

For comparison of designer and learner input at the diagnosis stage, the designer's input is assumed to be the minimum capable of achieving the result required for the particular assignment concerned. This will be referred to as the *minimally correct string*. The learner may provide input that achieves the same functional effect as the minimally correct string but with more key-strokes. To return to the MS-DOS example of the previous chapter, the learner might list a directory unnecessarily. (That is, unnecessarily with respect to the functional results required of MS-DOS for solving the problem posed to the learner by the tutoring system; it might well have been considered necessary by the learner.) The LIY diagnosis process would not deem such an action to be an error on the learner's part.

At the point at which diagnosis starts there are two sequences available: that of the learner's key-strokes and the minimally correct string. The two sequences are compared initially for perfect equality, in which case no further diagnosis is necessary: the learner is a "perfect subset of the expert". Otherwise diagnosis proceeds by referring to the domain model to build a *history list* for each of the sequences. Recall from chapter 3 (section 3.4) that the domain model reflects the structure of the command set in the application being taught. By parsing the key-stroke sequence against the domain model a history list can be constructed which records the nodes visited, in order, by the learner. The parse tree contains the commands associated with each topic so that when such commands are detected a transition can be made to a new node in the tree. Thus the model

tracks the *current node* as the learner navigates over the tree. Characters in the sequence which do not correspond to any transitions recognisable from the current node are considered to be application input. A history list can therefore contain not only the ordered sequence of nodes visited by the learner but also, at the correct points in the list, "packets" of application input.

There is one other important aspect to history lists: not all nodes in the task classification tree are equal. Certain topics in the tree are concerned with changing the state of the application whereas others are not. To return again to the MS-DOS example, deleting a file alters the state of the application whereas listing a directory does not. The tutorial designer, during the elicitation phase, is asked to declare those commands which are state-changing. In the DIALLER there is just one state-changing command: *save-setup*. A history list is therefore able to distinguish, among the topics "visited" by the learner (and by the tutorial designer by reference to the minimally correct string), between state-changing and non-state-changing commands.

Thus, unless the key-stroke sequences of the learner and the tutorial designer are identical, history lists are built from both the learner's key-stroke sequence and the tutorial designer's sequence. (To recap, the latter is referred to here as the *minimally correct string*.) The two history lists are then compared according to the following criteria:

(i) The learner and designer must end up at the same final node on exit.

(ii) The learner must visit all nodes visited by the minimally correct string in the same sequence.

(iii) If the correct string uses state-changing navigational commands, then the learner must also use such commands. In fact this requirement is covered by heuristic (ii) above. However there is a stronger requirement that the learner must not issue any state-changing commands *unless* they are in the correct string.

(iv) The learner must obey non-navigating key-stroke sequences character-for-character exactly, e.g. telephone numbers, file-names, etc.

Failure to match the history lists according to these criteria results in the learner receiving one of the following messages (a) to (d) below, which correspond with the heuristics (i) to (iv):

(a) ....

You appear to have quit ELICITUT in an abnormal way.

The exit command "ESCAPE" associated with the topic QUIT was expected.

....

(b) ....

Possibly you left out some of the commands,
or used them in the wrong order.

The command "ENTER" associated with the topic DIAL-DIGITS was expected.
....

(c) ....

Possibly you misused one or more of the commands which alter the state of DIALLER.

The command "S" associated with the topic SAVE should be avoided for this assignment.
....

(d) ....

Possibly one or more of the character strings which you typed into DIALLER was incorrect.

The input 123 4567 was expected.
....

As can be seen, the name of the application is incorporated into the messages. The fuller context of these messages is shown in figure 3.11 in the previous chapter, where the limitations of this diagnosis method were pointed out in relation to the "rung" on Jacob's ladder at which are fixed the user-interface specification components and thus this LIY implementation. Furthermore, LIY cannot explain its error messages. For example, it cannot explain *why*, in (c) above, *SAVE* is a state-changing operation; it can only say that it is.

4.2.3 Teaching strategy

The teaching strategy is represented as a set of rules. These rules implement a teaching process which is invoked by a search of the task classification tree to find a topic to teach. The rules are partitioned into distinct rule-sets to deal with particular aspects, such as determination of learner advice, as described in the previous chapter.

The five-ring model proposed by O'Shea and others is illustrated in figure 4.1 (O'Shea *et al.* 1984). The solid lines represent the flow of control between the components. The dotted line between the teaching administrator and teaching strategy nodes represents the possibility of an intervention by the learner such as a learner-control request.

The five-ring model is based upon the four-component model discussed earlier (Hartley and Sleeman 1973).

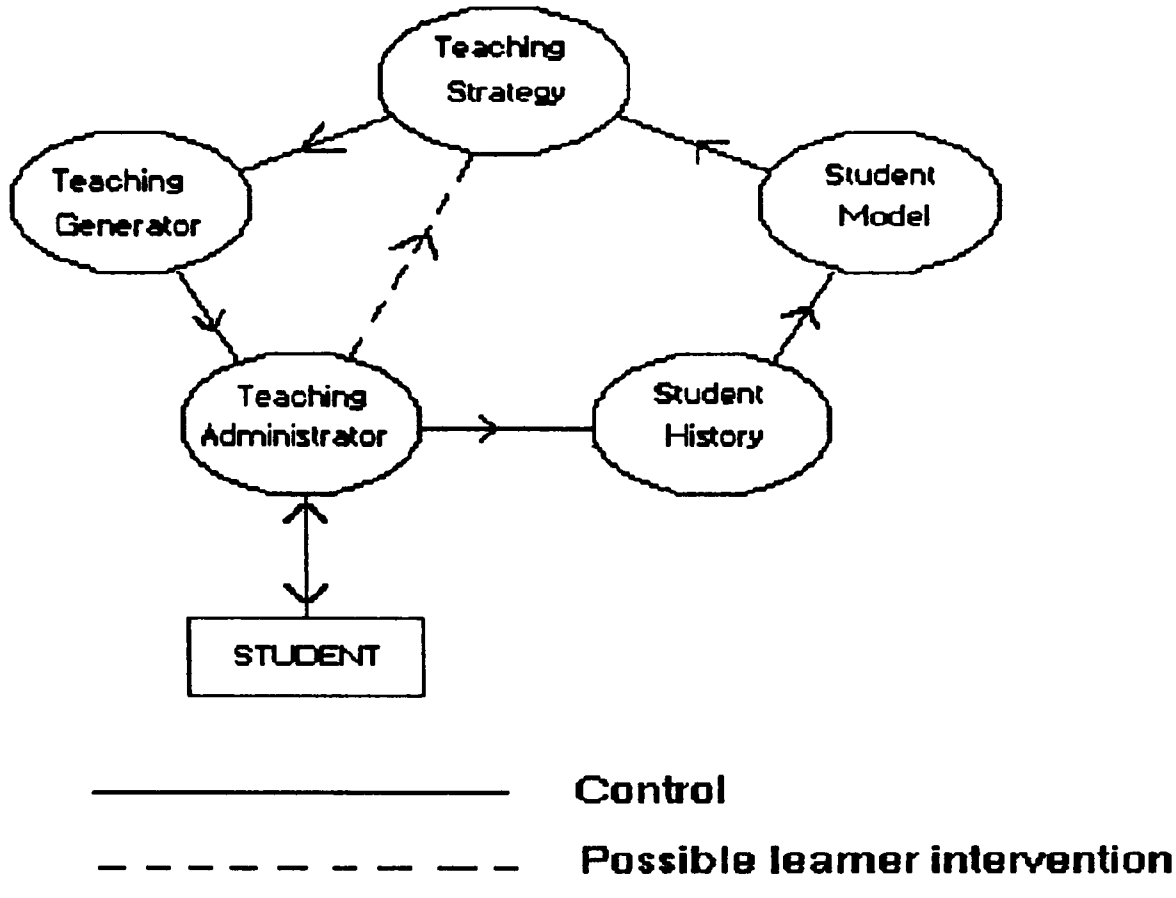


Fig. 4.1

The components of the five-ring model are described as follows:

Teaching Administrator	-	presents material to the learner and processes learner responses.
Student History	-	a record of material presented to the learner and his or her responses.
Student Model	-	makes predictions of the learner's future performance and current state of knowledge and ability.
Teaching Strategy	-	relates the systems view of the learner to the general types of teaching action that are possible, and decides the type of the next action.
Teaching Generator	-	a mechanism which yields a specific teaching action for use by the teaching administrator.

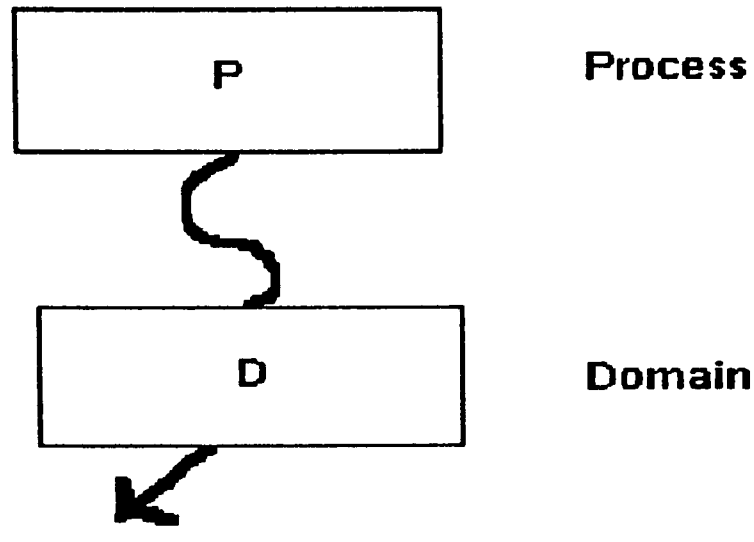
It can be seen that the teaching administrator represents an additional component compared with the Hartley and Sleeman criteria, but that the other four components are similar. It should be pointed out that the actual nature of each of the components varies depending upon the type of tutoring system being implemented. For example, the paper by O'Shea *et al.* describes two illustrative systems with widely differing student models, in the one case rudimentary and in the other incorporating a problem-solving expert system.

Consider the meaning of the word *strategy* (as in *teaching strategy*). It concerns planning, at a high level, the means to achieving some objective. In an ITS this objective would be achieving the demonstration of the learner's competence with the concept being taught. Lower-level preoccupations at the transaction level might be referred to as tactics. The semantic difficulty with strategy is how to describe a *change* of plan or strategy. As Self comments, it is a matter of opinion whether such a change represents part of an unfolding plan or the beginning of a new plan (Self 1987). Is a change of strategy part of the strategy? In this sense three levels of control can be considered: *tactics*, dealing with fine-grain interaction-level choices; *strategy*, which deals with a plan to achieve some pedagogic goal; and *meta-reasoning*, the ability to detect that a plan is failing and change it. Self observes that ITSs do not deal with meta-reasoning, but it is probable that this will change. In other areas of A.I., for example expert systems, research into the use of meta-level knowledge has become an important issue, e.g. the MOLGEN system (Stefik 1981).

Pask argues that a system such as that illustrated in figure 4.2, where process P operates upon some domain D, cannot be *adaptive* because P can have no knowledge of its effects upon D (Pask 1975).

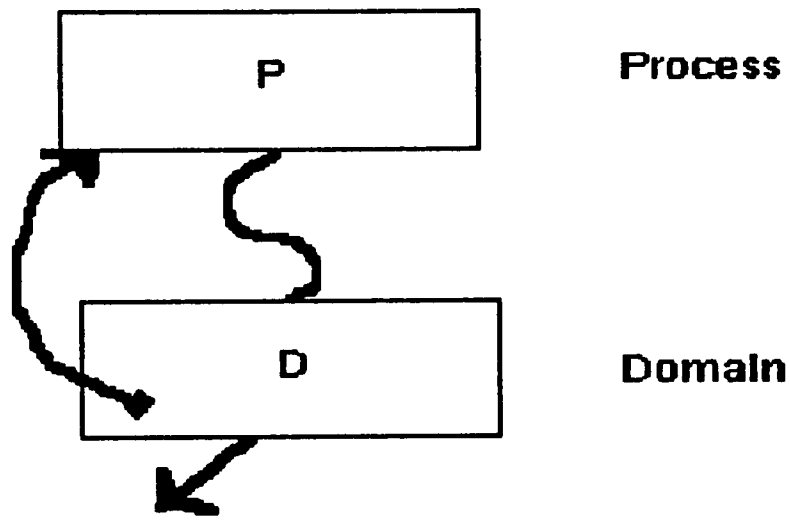
Figure 4.3 shows feedback information being provided from D to P.

An example might be a heating system fitted with a room thermostat. The problem with systems of the type shown in figure 4.3 is that they only operate



Non-adaptive system

Fig. 4.2

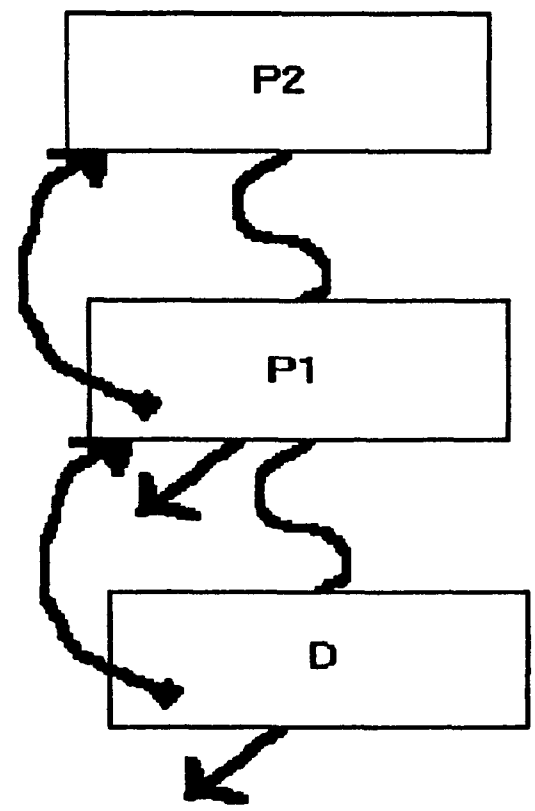


Adaptive system

Fig. 4.3

correctly in prescribed circumstances within preset limits. To continue the

example, such a system would fail if the thermostat had switched itself off but the room temperature continued to increase, perhaps due to a heat-wave. Such a controller cannot be said to be intelligent since it knows only one strategy. A more intelligent system would be able to examine the effects of P upon D and modify P, perhaps by a change in strategy, if necessary. Figure 4.4 illustrates this.



Self-modifying adaptive system

Fig. 4.4

Ogborn and Johnson give as an example of such a system a computer programmer developing a program (Ogborn and Johnson 1982). In a tutoring system P2 represents a meta-reasoning process which can modify the strategy applied by P1 to the learner D. Such a system is the minimum necessary to enable a tutor to reason about its performance. This represents a stronger definition of intelligence than that embodied in the Hartley and Sleeman-type architecture described earlier. Pask goes on to define figure 4.4 as being the

minimum architecture for the type of system which can learn, as described in his "conversation theory" (Ogborn and Johnson 1982).

What is therefore needed is a component in a top loop which can monitor the performance of the tutor operating in the bottom loop. Suppose that this component is called a *hypothesiser*. Its function is to suggest to the strategy component an appropriate teaching strategy, and to switch to an alternative strategy if the current one is inadequate. In other words the hypothesiser will implement the meta-reasoning described earlier. The strategy component on the other hand incorporates a range of plans, represented as sets of means-ends guidance rules, and will implement perhaps either a standard plan or one of those suggested by the hypothesiser. Such a tutoring system would have the architecture shown in figure 4.5, corresponding to the Pask representation of figure 4.4. The hypothesiser component acts directly on the strategy component, which has now been brought down to the central position. Further, this architecture enables attenuation of learner control to be handled, again by the central component as shown. Control flows around the complete figure-of-eight as two alternate cycles (bottom and top) starting with the courseware generation node in the lower left.

The architecture of this *figure-of-eight model* is speculative but would be general enough to encompass, with a non-functioning hypothesiser where appropriate, a range of CAL models. These include a *dumb* CAL system, a *help* system, an on-line manual, and systems with either total or constrainable learner control which are either dumb or intelligent. An earlier paper provides a fuller description of how these different types of tutoring system can be viewed as constrained subsets of the figure-of-eight architecture (Martin 1988). The hypothesiser is invoked on each cycle and acts directly on the strategy component, so that the desired meta-reasoning is not shut out by, for example, inappropriate unbroken cycles round the lower loop only. It would appear that O'Shea's self-improving quadratic tutor could, to an extent, be made to fit into this architecture (O'Shea 1982). In his system a teaching strategy, expressed as a set of production rules, can be altered experimentally by changes to the rule set, which corresponds to the hypothesiser acting upon the strategy component.

However, there is no direct feedback to the hypothesiser on a moment-to-moment basis: the feedback comes with the statistical evaluation of performance by the tutoring program at the end of a learner's session. (Note that O'Shea uses the term *hypothesis tester* in a totally different sense - as another name for the student model.)

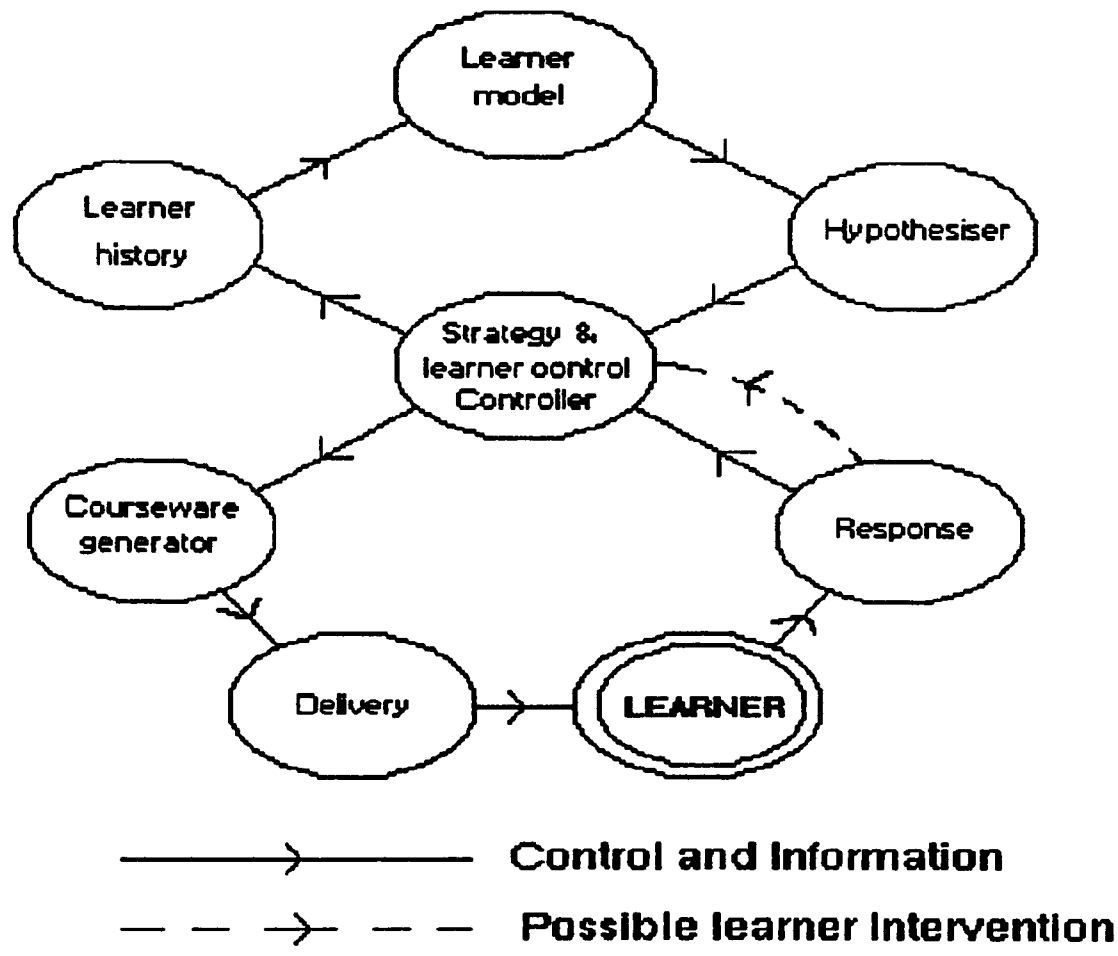


Fig. 4.5

The five-ring model appears to form the sound basis for representing the architecture of a class of tutors which do not use constrainable learner-control. At least one tutor has been constructed on the lines suggested by the five-ring model (Heines and O'Shea 1985). The above proposal demonstrates that with only a slight alteration the five-ring model can be adjusted to deal with constrainable learner control, the resulting model being called the *figure-of-eight model.* In addition to handling learner control, the *hypothesiser* component can

generate and monitor the choice of teaching strategy. Thus for LIY it would in principle be possible, for example, for a *hypothesiser* component to alter the teaching strategy used by substituting one set of teaching strategy rules with another. This might be an appropriate response by the tutoring system for supporting a learner's serialist or holist teaching preference, perceived in relation to the *FLITTER* concept described in chapter 3.

Chapter 5

Discussion and conclusion

This chapter falls into five parts. The first two are concerned with the way in which LIY matches up to the two principal aims set out in chapter 1. The third part considers the extent to which the subsidiary aims have been achieved and is followed by a section discussing the possibilities for further research and development of LIY. The final section summarises the achievements of this research.

5.1 "Dialogue specification can be used as the basis for courseware design".

This section heading is placed in quotation marks deliberately in order to emphasise the fact that it is the slogan representing one of the two major aims of the research. This aim has been achieved - but at a cost.

The possibility that the output from the systems analysis stage could produce some specifiable structure which could be used directly as the kernel of a tutoring system for that interface remains something of a chimera. The information needed by a tutoring system is deeper than that available from systems analysis. If this information be termed "domain information" then the issues pertinent to it are:

What to teach?
When to teach it?
How to teach it?

For software interfaces, it can be argued that the last of these questions - "How...?" - is answered by the design of the tutoring system itself. Chapters 3 and 4 have described how LIY addresses this problem. The first two questions, though, need careful consideration.

The question of *what* to teach is met at a rather basic level by the set of commands corresponding to user actions in the interface. Such a set of commands can be used as a skeleton to which tutorial material can be attached. This is satisfactory for hierarchically structured command sets such as menu systems where the command hierarchy itself groups together commands of a similar type. A word processor, for example, might offer PRINT as a "top-level" command. Sub-commands associated with printing, such as PAUSE PRINTER, would only be available once PRINT had been invoked at the higher level. Such a command structure naturally lends itself to a tutorial style since it is possible to associate successively focused levels of tutorial material with each command, and this material can be delivered as the learner moves down the

hierarchy. The structuring of the command set is reflected by the structuring needed for the teaching of the domain. Both the example applications developed for this research are in this style, which is the one to which LIY is most suited. It is also the style of pointer-based command systems, such as those which use a mouse, and the possibility of using LIY with such interfaces is discussed in section 5.4.

In contrast, a flat command structure requires that similar features, common to a range of commands, be identified and grouped together by the tutorial designer. Many operating systems offer just such a flat command structure. This is partly for historical reasons; operating system command sets have always been organised this way. Also it is due to the advantage, for skilled users, of having the entire command set available at any point. Commands can sometimes be issued using a reduced number of key-strokes compared with that needed for a hierarchy; certainly, commands can be issued with a reduced amount of slow interaction in comparison with a menu system. It is interesting that a number of proprietary shells are now available for MS-DOS which offer a menu-driven hierarchy to overcome its original flat structure. Such shells have much appeal both for the occasional user and the novice since they overcome the need to recall command syntax.

The question of *when* to teach is more thorny. Specifically the question must address the order of teaching topics, that is, the development of a suitable tutorial sequence. A flat command structure offers no support whatsoever in this respect. A hierarchical structure permits the observation that it is generally appropriate to teach topics at a higher level in the hierarchy before those at a lower level. (This is not always the case in an LIY hierarchy since there is also a left-right ordering as well as a vertical one.) Higher-level topics are concerned with more general issues which are prerequisite to the understanding of more specific, lower-level topics. The question then arises as to the ordering of topics at the same level in the hierarchy. This is exactly the same question as the ordering of topics in a flat command structure. No information is available from systems analysis to support such ordering. Consequently the rules described in

chapter 3 - "transformation from operational to pedagogic ordering" - have been used to provide a tutorial sequence.

These transformation heuristics are not proven to be sufficient for all domains. They were developed by trial-and-error but have been found satisfactory for the MS-DOS example of chapter 3, the two domains described here, and one other which has not been developed into a working and tutored application. This latter is a small financial application which represented the very first experiments with building a pedagogic task classification hierarchy. In a rather ad-hoc manner such a structure emerged from the operational hierarchy. Subsequently, building the DIALLER system, an attempt was made to formalise the transformation heuristics needed. As a test, they were then applied retrospectively to the financial application's operational hierarchy to yield a further pedagogic version. The two pedagogic versions were compared and found to be equivalent.

It would be pleasing to be able to report that the application of the transformation heuristics has been automated, but alas this is not at present the case. The problem is the difficulty a program has in interpreting commands such as "teach SETUP functions last, linked by INDEPENDent". (The linkage referred to is concerned with representing left/right dependency in the structure. It is used by the learner-control advice system.) If significant nodes in the classification structure were to be tagged by the designer as being of type SETUP, EXIT, and so on, then automatic transformation might be possible. The programming effort required should not be underestimated. The operational classification is represented - on paper at present - as a general tree. Application of the heuristics and other transformations yields a binary tree structure for the pedagogic classification. The resulting tree is more straightforward to navigate during the tutorial delivery phase. The reasons for this are concerned with the DEPEND/INDEPENDent left-right linkage and have been more fully described in chapter 3.

In summary, the task command hierarchy element of the dialogue specification from the systems analysis and design stage *can* be used as the domain

information required by a tutoring system. However, such information must first be transformed from its original operational ordering into a pedagogic ordering.

The questions about teaching - *what* and *when* - have been addressed in terms of the curriculum being taught. They are not addressed here in relation to meta-level reasoning about knowledge of teaching, such as teaching operations to be invoked following diagnosis of particular learner errors. Dealing with such issues requires that a tutor incorporate far more knowledge about teaching and a much finer-grained model of the learner than the rudimentary model used in LIY. Both would be necessary - teaching knowledge alone would not provide a significant advantage over the current situation in LIY without there being knowledge of the learner's current state to which the teaching knowledge could be applied.

5.2 *"LIY is a portable tool for producing and delivering tutoring systems".*

This aim has been achieved, although the LIY approach is most suited to hierarchical command systems, as discussed in the previous section. The applications which are tutored by LIY differ from each other in that the DIALLER program does not alter its environment, although it appears to. On the other hand, the ELICITOR program does useful work as a general-purpose LIY tutorial elicitor and saves the elicited information; it thus alters its environment and updates files on disk.

To build a tutor for a new application the designer runs the ELICITOR program. This will create a subdirectory for that application and then interact with the designer to create within it a Lisp-based representation of the task classification structure, TA.LSP. Consistency checking is performed on the nodes in this structure as it is elicited from the designer. When complete the structure is displayed on the screen and the designer can use the mouse to point to particular nodes. They can then be enhanced with such information as path-names for slide shows, sequences of teaching operations on the node, and so on. This enhancement must currently be performed by adding the appropriate Lisp code.

For tutorial delivery there are only two files which contain application-dependent code: TA.LSP, of which there is a copy to hold the task classification structure for each particular application domain, and LIY.LSP. This latter file comprises the top-level routine which invokes everything else; its domain dependence is due to its requirement that it must know the path-name of the TA.LSP file for the application being taught.

The desire for portability has shut out from this research many topical issues germane to ITSs. To take one example, several workers have built into their tutors libraries of common misconceptions or "bugs" (Anderson and Reiser 1985). Such an approach would not be appropriate in LIY since it doesn't employ a set of learner modelling rules to perform diagnosis. (Diagnosis in LIY is issue-based, as described in section 4.2.) If the diagnosis module were to be replaced

by a rule-tracing approach then rules - and mal-rules (Sleeman and Brown 1982) - would need to be built for each application by hand, thus losing the goal of portability. Sleeman approaches the domain-independence issue from the other end, in that he is proposes a scheme for the automatic generation of mal-rules for a given domain (Sleeman 1987). The price paid for the robustness needed by the portability requirement of LIY is a lack of sophistication in its approach to diagnosis in comparison with rule-based approaches.

5.3 Meeting the subsidiary aims

The first of these aims was that

"the learner should be allowed to interact directly with the software interface being taught".

This aim has been met, the learner being able to run the application program being tutored directly, or being placed under tutorial control at some pre-determined point in the application. This means that the learner can be required by the tutoring system to manipulate the application program from some "internal" position, for example by making a selection from a menu at other than the top level. Furthermore the tutorial designer can set up the application in any desired way by use of the teaching operation "create environment". This feeds into the application a sequence of commands which would normally be keyed in directly by a user.

This facility is a most significant feature of LIY and appears to work well. More programming effort could usefully be expended on making the application more learner-proof, for example inhibiting file-creation activities which had not been requested by the tutorial designer. It is not at present clear the extent to which this would be possible while maintaining the strict separation between the application and the tutorial; at present the application software is implemented without making any concessions to the existence of the tutorial.

One difficulty with this type of approach is that LIY operates in more than one mode, with the learner interacting at different times with the tutorial and with the application. There can be possible confusion on the learner's part as to the current state. This is a problem which has been pointed out in connection with earlier systems (du Boulay *et al.* 1981). They discuss the difficulties faced by novices in using the BIP system which was designed to aid the learning of the programming language BASIC (Barr *et al.* 1976). It incorporated not only the BASIC interpreter and editor but also the tutor and a commentator on BASIC programs. Novices were unclear at certain points as to whether they were

interacting with the BASIC editor or the interpreter, or indeed the tutoring system. This type of confusion is currently quite possible in LIY, which has three modes. During a tutorial slide show the learner must press the space bar to advance; pressing anything else does no harm but results in a warning beep. This overcomes the difficulty which can arise when the learner, having been instructed to manipulate the application, is still at a control-point in the tutorial delivery phase rather than in the application proper. Intended application input simply elicits the warning beep. Alternatively the learner can be asked by the tutorial for "direct" input; the learner is being asked a question but must respond directly, rather than interact with the application. Here, the prompt displayed is very different from the slide show. The more it differs from the application the less likely it is to confuse the learner but of course this is dependent on the screen design of the application program. Finally, the learner can interact with the application directly while perhaps being under the impression that a slide show is being displayed. LIY does not really address this difficulty, although with a more powerful windowing facility it would in principle be possible to run each of the modes consistently in different windows on the screen so as to minimise the possibility of learner confusion. In different situations, however, users have reported confusion in relation to multiple window systems; such a problem has been seen with students meeting Turbo Prolog for the first time.

"The learner should be able to interrupt at any time."

This aim has been fully met. The learner can interrupt in a uniform way whether the tutorial is in the delivery phase or an assessment phase, including the case in which the application program itself is running. Following an interruption the learner can select a course of action from the menu shown in figure 3.5 of chapter 3. Switching to a different topic, either by browsing using the mouse or by keying in the name of the topic, results in "advice rules" being applied with reference to the projected move and the state of the current learner profile. This was described in detail in chapters 3 and 4. The effect of these rules is either to enable the requested move or to offer one of a graded sequence of messages advising against the move. A factor considered in the decision

process resulting in this advice is the skill level of the learner; this is determined from the history of recent previous assessments. However, the advice the learner receives is expressed in terms of the missing prerequisite topics not so far taught. Whatever the strength of advice against the move that LIY offers, the learner can choose to ignore it and force the move if desired.

This ability to interrupt is intuitively appealing. The need for an evaluation of its usefulness is proposed in section 5.4.1 below.

"LIY should comprise not only a delivery system but also an authoring system."

This aim was proposed in order to support the portability of LIY and as such it was discussed in the previous section. A limited authoring environment now exists, mainly for eliciting the task classification. It would be useful to expand the capability of the authoring system if LIY were to be used in a development environment, although to do so would contribute nothing to this research.

"LIY should incorporate intelligent tutoring technology where possible."

Current research with LIY has focused on producing a complete tutoring system in order to experiment with a particular approach based upon dialogue specification. Many of the issues dealt with in this undertaking, such as replacing input-output routines for the purpose of logging user interaction with the application software, clearly bear no relation to the issues central to ITSs. However, it was felt to be highly desirable to base the LIY software upon an existing ITS architecture, which in this case was that proposed by Hartley and Sleeman (Hartley and Sleeman 1973). Without going over all the ground covered in detail earlier, the LIY tutorial delivery program is structured to incorporate domain and learner representations, with a separate diagnosis module. There is in addition a representation of the teaching strategy as a set of rules, together with a rule interpreter. Finally, there is a set of teaching operations, including diagnosis, which can be invoked by the rules. The declarative nature

of the teaching strategy was found to be very helpful in the development phase; changes to the control behaviour of the tutor were straightforward. The omission of a bug catalogue - an enhancement to the original Hartley and Sleeman architecture proposed in more recent research - has been discussed earlier in this chapter.

No natural language interface has been built into LIY because, unlike for example the SOPHIE system (Brown *et al.* 1982), it cannot solve problems set by the learner in the domain. Software interfaces, as an application domain, are restricted in terms of their syntax and semantics, so that posing problems in natural language about such interfaces can arguably be regarded as unnecessary. More specifically a distinction can be made between questions concerning objects in the application domain, which the interface permits through its syntax and semantics, and questions concerning the manipulation of the interface itself, which might well be posed in natural language. To enable LIY to understand relevant natural language would require that it contain much more domain knowledge than the present task classification structure. It could be done - as for example in SOPHIE - but it would remove the portability which LIY currently offers.

Diagnosis in the LIY system uses a form of issue-based differential modelling, as exemplified by the diagnostic method used in the WEST game (Burton and Brown 1982). As discussed in chapter 4, in LIY there are just four issues. This is sufficient to provide accurate diagnosis in terms of evaluating the correctness of the learner's interaction with the application software in most cases. It permits the learner's interaction to be *overloaded*, in the sense that more interface operations are permissible than those in the *correct minimal string* provided by the tutorial designer. This gives a much more "intelligent" feel to the system. In a small proportion of cases there can be problems: the learner can carry out state-changing commands unnoticed by the tutoring system. In the DIALLER, saving the result of a setup is a state-changing command; manipulating setup parameters is not. If the learner is requested to alter and save one such parameter but alters two before the state-changing setup, LIY will not be aware of this in its present configuration.

The feedback provided to the learner by LIY's diagnosis is rather limited. Since the issues with which it is concerned are based on key-stroke sequences - the lowest level of abstraction - it can really only provide feedback at this level. An interesting way to make the best use of this feedback might be to replay the "correct" key-stroke sequence through the application in slow motion in "teach me" mode, making the learner follow on the keyboard. Other feedback provided by LIY is implicit in terms of mastery of topics from the task classification. This provides rather a coarse representation of procedural skills at a level of abstraction above the task level.

5.4 Further work

The sections below describe ways in which the LIY work might be taken forward in terms of both research and development. In the latter case the discussion concentrates on measures which would improve performance and appearance.

5.4.1 Research

Jacob's ladder

Section 2.3 described a series of points on a graduated range of syntactic and semantic possibilities, referred to (in this thesis only) as *Jacob's ladder* (Jacob 1983).

The selection of a rung for LIY was motivated by a desire for parsimony; only the minimum elements of an interface specification have been used in order that the LIY approach should be applicable across a range of domains. The consequences of using alternative rungs would be interesting to explore.

Moving up to rung (iv) would add specification of application input syntax to the pre-existing command syntax. Even this addition would greatly inhibit portability unless a universal parser could be developed. Clearly, however, the quality of diagnostic messages to the learner would be improved.

Adding semantics would obviously complicate matters further. The objective of portability would be swept away under a wave of domain-specific information. However, the tutor would appear to the learner to be much more intelligent since it would be able to detect equivalences at a higher level than currently. With a suitable knowledge representation technique, such as a semantic net, a tutoring system could, to take an earlier example, detect the equivalence between

renaming a file on the one hand and copying and deleting the original on the other.

As a next step one might therefore elect to augment the current LIY specification elements with the command semantics - rung (ii) without the application input syntax. An implementation at this level might then draw considerably on research in planning - especially the idea of the *procedural network* (Sacerdoti 1977) - as incorporated, for example, into the TOTS system (Rickel 1988).

Bound up with the question of semantics is the requirement that the LIY tutoring system be able to navigate over the task command hierarchy for diagnostic purposes. Restrictions on the types of user interface for which LIY is suited have been stated earlier and exclude, amongst other things, the use of *hot keys*. However, it should be possible to incorporate such devices into LIY so long as the control-flow model were more elaborate. Currently, the specification must provide semantic information concerning control-flow following leaf-processing. Hot-key processing differs slightly from leaf-processing in that there is a variable return-point - the point of invocation - so that it might be appropriate to augment the domain model so as to process dynamically a stack of "return addresses" along the lines of the familiar software subroutine or interrupt-processing mechanisms.

Evaluation

Although LIY has been developed into a complete system, no evaluation of its effectiveness has been performed. *Because* it is a complete system dealing with a domain - software interfaces - which has not been greatly researched at present, there are several aspects of LIY which deserve to be evaluated separately. Consider some examples:

(i) Do learners like the way the tutor invokes the application software?

(ii) Do they find this confusing?

(iii) How often should a tutorial use this teaching method - on every topic, or just now and again?

(iv) Feedback on errors has been identified earlier as being rather limited: is this critical in this kind of domain?

(v) If so, to what extent?

(vi) Is the question of feedback so important that issue-based diagnosis should be abandoned, or is it a matter of incorporating further issues into the diagnosis heuristics?

(vii) Is the domain structure based upon task classification sufficiently supportive to the learner, or would a task structure based upon learners' views of their tasks, perhaps elicited from protocols, be better?

If the answers to some of these questions indicate that changes should be made to LIY then it would be useful to consider the *cost* of introducing techniques to improve the tutor. LIY is attempting at present to exploit to maximum effect a rather minimal amount of information - the task classification structure of a domain. It would thus be useful to know how the economics of tutorial production would alter if any improvements introduced to LIY significantly increased its cost. Clearly there is a trade-off between the cost of producing a tutor on the one hand and its effectiveness at teaching on the other; the question which needs to be resolved is how to determine, for a given market, some optimum point on the cost-performance curve.

One of the subsidiary aims of LIY is that the learner should be able to interrupt. Hartley has twice studied groups of learners who were offered learner control (Hartley J.R. 1981, Hartley, J.R. and Tait, K. 1986). It was found that, although learners apparently liked learner control, the effectiveness it offered to their learning behaviour could not be identified from evaluative studies. Given that many software packages are quite similar to each other, it is believed that LIY would support learners transferring their skill acquired with one package to some other product. In this case it is quite probable that such learners would not want to study a complete tutorial; learner control provides an effective way for them to adapt the tutorial to their own needs. In the software interface domain it is intuitively appealing that the learner should be able to interrupt in order to exert some control over what is being taught; this intuition needs to be tested.

The notion of categorising learners as "holists" and "serialists" is not really germane to LIY and has not been fully developed. Nevertheless it would appear to be possible to make such a distinction by observing how a learner uses learner control and thus adapting the teaching strategy accordingly. Continuing with this theme of identifying features of the learner, it would be interesting both to develop and to test the idea of maintaining separate characterisation and performance profiles, as proposed earlier and implemented in skeleton form. The characterisation profile contains information inferred about the learner's interaction style, particularly holist/serialist, which could carry *across* tutorials. Currently, this characterisation profile information is based on the qualitative variables *DUCKER* and *FLITTER* (see section 3.2.2).

Littman and Soloway write:

> *"There can be no doubt that evaluating Intelligent Tutoring Systems (ITSs) is costly, frustrating and time-consuming. In fact, in our own work to build PROUST ... evaluation has consumed nearly as much effort as the design of PROUST itself."*
>
> (Littman and Soloway 1988)

This is depressing. They ask the question as to whether evaluation is really worthwhile. Would it not perhaps be better to let the marketplace decide on the value - or otherwise - of a particular ITS? However, they go on to assert that evaluation *is* very important in order to further our understanding of cognitive science, artificial intelligence and education. This point is fundamental to all science; there is little to be gained from any experiment which involves the construction of something unless the constructive phase is followed by an evaluative phase. The issues that LIY poses as a portable tutoring shell for software interfaces need to be evaluated, yet to do so is beyond the present phase of LIY research.

Any evaluation would need to concentrate upon those issues which are central to LIY rather than those which are peripheral, i.e. it should focus on learners' views of the following:

(a) interaction style, in which the tutor causes the learner to interact with the application software;

(b) feedback concerning learners' errors. This topic is intimately associated with the effectiveness of issue-based diagnosis;

(c) the pedagogic task structure;

(d) learner control.

Scaling up

LIY has been developed and tried with two sample applications which are both rather small. It would be very interesting - and important if LIY were to be considered for tutoring a real application - to discover the limits or weaknesses of the LIY approach when applied to a large software interface. There would be implications for both the authoring and delivery environments. Common to both would be the management of a large task space. For the learner this would require the ability to handle a substantial task structure. The browsing facility, with a display of parts of the task structure on the screen, could well present the learner with a severe test of short-term memory. Equally, research would be needed to investigate the effects of large domains upon a learner's long-term memory; it would be necessary to examine the learner's ability to remember the correct task structure, as a result both from inferring it from the tutorial directly and from manipulating a display of it when browsing. To complicate matters, interface users do not usually have to remember *the whole* of an interface; so long as the user can recall that there exists a particular operation which will achieve the *user's task*, menu systems can provide prompts which will support navigation to and execution of the appropriate interface command. Thus research into LIY's effectiveness for learners of large software interfaces would need to separate effects of the LIY tutorial from effects of the software interface design.

For the design of a tutorial for a large software interface it would be instructive to test the effectiveness of the transformation heuristics. These are used for conversion of a task classification structure from an operational ordering to a pedagogic ordering. Because they are applied recursively to a tree structure there is no reason to suppose that the size of the tree imposes any limitation on their applicability.

Making LIY more "intelligent"

In the light of the major LIY aim of portability, section 5.2 above discussed the difficulties of enhancing an LIY tutorial by including a "bug catalogue" and using model-tracing for diagnosis, either with or without a plan recognition strategy. Such techniques appear to provide the most promising approach to making LIY more intelligent. Hoppe describes the use of a task-oriented parser applied to the user's input stream to identify higher-level tasks (Hoppe 1988). Taking Unix as an example, the higher-level external task *replace* can be broken down into the internal (i.e. command-level) tasks *delete* and *move*. Hoppe's Prolog task parser can then recognise the user's intention of replacement by identifying delete and move commands in the user's key-stroke sequence. This is somewhat similar to the plan recognition strategy of the MACSYMA Advisor (Genesereth 1982).

Rule-based model tracing in the Lisp Tutor (Anderson and Reiser 1985) is based upon the ACT* theory of learning (Anderson 1983), whereas LIY incorporates a pragmatic knowledge of instruction based on a prerequisite knowledge hierarchy, similar to that proposed by Gagne (Gagne *et al.* 1988). Ohlsson points out that despite the paucity of theories of learning, people actually *do* learn, and he emphasises the importance of looking at the way in which *teaching* is carried out (Ohlsson 1986). Much is known about teaching: not all tutors need to be based upon a theory of learning. Other A.I. techniques might be considered for LIY, such as incorporating a module for understanding natural language. It is unclear how any of these techniques would be compatible with the portability requirement or with a domain based solely on the task classification structure.

With the objective of retaining LIY's portability, it would be useful to examine whether the set of issues tested during student diagnosis could be enlarged, permitting a greater degree of feedback to be provided to the learner.

The teaching strategy incorporated in LIY's rules could be improved. If the architecture were to be enhanced so as to reflect the structure of the figure-of-eight model proposed in chapter 4, then there would be a component - the "hypothesiser" - which could suggest a change of teaching strategy as a remedy for poor learner performance in specific instances. This could substitute one set of teaching strategy rules for another, for example navigating over the task tree in a different sequence in the light of inferred holist/serialist learner preferences. Alternatively, various teaching strategies - expressed as alternative sets of rules - might perhaps be drawn from those used in DOMINIE (Spensley and Elsom-Cook 1988), for example "cognitive apprenticeship" and "discovery learning".

Continuing with the figure-of-eight model, the central ring is responsible for the current teaching strategy and for learner-control. Although in LIY the learner can *always* interrupt, in principle the degree to which such interruption is permitted could be changed dynamically. This might be useful when teaching - for the first time - some critical operation for which interruption could be seen as undesirable, such as use and recovery of back-up files. If learner modelling in LIY were to use a more fine-grained representation, there might be a case for considering altering, on a moment-to-moment basis, the degree of permitted learner-control in the light of the learner's current state.

Implementation issues : direct-manipulation devices

In LIY the key-stroke sequence is captured in "watching mode" to watch and record the learner's actions for subsequent diagnosis. LIY could not at present interpret the learner's use of the mouse in this context. LIY applications must be menu- and text-based. Use of the mouse can only occur at delivery-time when the user has interrupted, whereupon any "watching" is abandoned.

Suppose that using a mouse or some other pointing device were part of the application. Use of the mouse in a menu-based system is functionally equivalent to the use of the arrow cursor-control keys. However, because of their discrete

nature the arrow keys permit the retention of the state-transition paradigm. This is not the case with the mouse because of its inherently continuous nature.

What does the mouse do? In cases such as menu systems it causes a value to be returned which is associated with the "clicked" object. In other situations, for example where painting or drawing programs are in use, it merely causes a re-arrangement of the screen. In this latter case it is not possible for the LIY approach to make any contribution; indeed, it is hard to see how any tutoring system using current technology could suitably assess the result of the user carrying out the task required by, for example, "Using the mouse, draw an apple on the screen". In the former case, in which a functional value is returned as the result of a clicked object, this functional value can be considered to be the answer to the implicit question "What would you like to select?". As such, the mouse routine would be in a position to insert the returned value into the learner's input stream as though the learner had typed the answer to the implicit question. Thus the task model of the application would behave as if all I/O were text-based. In summary, LIY could address the difficulties inherent in interpreting user responses in mouse-driven systems in which mouse clicks return functional results, but cannot deal with systems in which the mouse is used to produce some behaviour which cannot be represented as an I/O stream, such as altering the state of the screen. An example of the latter would be the request, in an ARK-like system, to "bounce the ball three times" (Smith 1986). Further, in an application without a mouse but in which the cursor keys were used for pointing, although as argued above it would in principle be possible to incorporate cursor movements into the model of the application as state transitions, it would be better to regard them as being like mouse movements, finally returning a functional value to be incorporated into the learner's input stream. What is needed therefore is a *user-interface management system* to separate details of the mechanics of manipulation of the interface from the representation of objects being manipulated. The application would need still to be text- or menu-based whether or not a pointing device were to be used, although the menus could be of the "pull-down" variety. Such a system could then allow a tutor to access objects at a higher level of abstraction than the key-stroke-sequence level currently used by LIY.

Implementation issues : OOPS

With the natural separation in LIY between the tutoring system and the application being taught, it would appear promising to consider implementation using an *object-oriented programming system*, or OOPS. Since LIY and its sample tutored applications are presently implemented in Common Lisp, an obvious system for re-implementation would be the Common Lisp Object System, CLOS. Object-oriented systems provide several advantages to the program designer and implementor, not the least of which is powerful support for abstraction in terms of module interfaces. It would be instructive to discover how OOPS might help, not only in designing and implementing the architecture of the obvious tutor/application interface, but also in the exploitation of the *user-interface management system* described in the previous section. Whether or not OOPS techniques were used, it would be interesting to examine the effects of redesigning the LIY architecture so that the tutoring system and the application run as separate processes under a multiprocessing system such as UNIX or OS/2.

5.4.2 Development

A considerable improvement could be made to the authoring system by the use of a "show me" mode. This would permit the authoring system to record the tutorial designer's interaction with the target application, typically so that it could be set up in some desired state for the learner. LIY forces the application into such a state by transparently substituting a character string, stored with a lesson, for an equivalent key-stroke sequence which the application expects from the keyboard. Because it is performed by substituting Lisp's normal input-output routines the application need not be aware of this. The desired key-stroke sequence to achieve the effect is presently built into LIY by the designer, simply as a string. With a "show me" mode this string could be inferred while the designer *used* the application to reach the state desired for the learner. Such a

device would be useful for setting the application to a state which displayed a particular sub-menu, possibly combined with data entry to build some desired application environment. There is an analogy here between this "show me" mode and the "teach me" mode proposed in section 5.3 for providing the learner with error feedback.

For the student profile it was intended to incorporate some measure of "literacy" associated with the application package being tutored. This could be elicited from the learner statically at the start of the tutorial in answer to a question such as "Have you ever used a *XXXX* before?". *XXXX* might stand for "word processor" or "spreadsheet" and would need to be a key-word provided by the designer. Such knowledge could be used by the tutoring system in a similar way to that suggested earlier when the need for a change of strategy was detected: an alternative teaching strategy could be used by interpreting a different rule-set. A possible strategy, having detected a package-literate learner, would be to teach all the non-leaf nodes of the task classification structure first. This would provide the learner with an overview without communicating a vast amount of detail. It might prove sufficient to enable the package-literate learner to grasp the entire domain so long as the use of the actual commands, given their context, were straightforward.

At present there is a certain inconsistency between the way LIY uses menus and the mouse. Figure 3.5 illustrated the menu which appears when the learner interrupts. If option "B" (browse) is selected, then a diagram of the task classification structure appears from which the learner can select an item with the mouse. Ideally the mouse could also be used to select from the menu of figure 3.5. The browser could be improved if it enabled summary text information to appear in a window. Thus the user might click with the left mouse button to see summary information on the function of a command in the task structure, and click with the right button to select that topic for the full tutorial.

5.5 Conclusion

The work described above situates a technique for the specification of a user interface within a spectrum of such techniques. It demonstrates how an interface specification can be used as the basis for constructing a tutorial for teaching the use of that interface. Furthermore, the specification itself would, in an ideal world, form part of the systems development effort for implementing the application. In principle, therefore, an economy could be made as a result of sharing the specification between the application development and tutorial development stages. In fact the major part of the user-interface specification which is utilised, the task classification structure, needs to be transformed from an operational to a pedagogic ordering. Heuristics are proposed to achieve this, although human expertise is required to apply them.

A portable rule-based shell has been developed which supports the delivery of tutorials for a range of software application package interfaces. The use of the shell with two such interfaces is reported. This has additionally required the construction of the applications and their tutorials, although it is important to note that the applications themselves do not depend in any way on features in the shell; they can be run together with the tutoring shell or in stand-alone mode. A computer-based authoring environment provides support for the development of tutorials.

The shell allows the learner of a software interface to interact directly with the application software being learnt while remaining under tutorial control. The learner can always interrupt in order to request a tutorial on any topic, although advice may be offered against this in the light of the tutor's current knowledge of the learner. This advice can always be over-ridden, however.

Learner diagnosis is effected by recording the learner's key-stroke sequence from interaction with the package and then parsing it against a model of the application which is based on the task classification structure. The same operation is carried out on a sequence provided by the tutorial designer.

Heuristics applied to the differences between the two outcomes provide the basis for feedback to the learner.

Currently, applications which could use the approach described are restricted to being text- and menu-based. They cannot make use of a mouse, for example. Furthermore they need to be written in the language used to implement the shell, which is Common Lisp, although another language could be used so long as there was a suitable interface medium, such as C, to the shell.

The approach reported here is suitable for an unsupported software interface learner and is named LIY ("Learn It Yourself"). With further development, it would appear to provide a promising method for augmenting a software engineering tool-kit with a new technique for application tutorial production.

References

Alty, J.L. 1984. Path algebras: a useful CAI/CAL analysis technique. In Smith, P.R. (ed.) *Proc. CAL '83*, 5-13, Pergamon Press.

Anderson, J.R. 1989. Psychology and intelligent tutoring. In Bierman, D., Breuker, J. and Sandberg, J. (eds.) *Artificial Intelligence and Education, Proceedings of the 4th International Conference on AI and Education*, May 1989, IOS, Amsterdam.

Anderson, J.R. 1983. *The architecture of cognition*, Harvard University Press, Cambridge, MA.

Anderson, J.R. and Reiser, B.J. 1985. The LISP Tutor. *BYTE magazine*, **10**, 4, 159-175.

Anderson, S. 1986. Proving properties of interactive systems. In Harrison, M.D. and Monk, A.F. (eds.) *People and Computers: Designing for Usability, Proc. HCI '86*, 402-416, Cambridge University Press.

Barr, A., Beard, M. and Atkinson, R.C. 1976. The computer as a tutorial laboratory: the Stanford BIP project. *International Journal of Man-Machine Studies*, **8**, 567-595.

Breuker, J. 1988. Coaching in help systems. In Self, J. (ed.) *Artificial intelligence and human learning*, 310-337, Chapman and Hall.

Brown, J.S., Burton, R.R. and de Kleer, J. 1982. Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II, and III. In Sleeman, D. and Brown, J.S. (eds.) *Intelligent Tutoring Systems*, 227-282, Academic Press.

Burton, R.R. and Brown, J.S. 1982. An investigation of computer coaching for informal learning activities. In Sleeman, D. and Brown, J.S. (eds.) *Intelligent Tutoring Systems*, 79-98, Academic Press.

Clancey, W.J. 1987. Methodology for building an intelligent tutoring system. In Kearsley, G.P. (ed.) *Artificial Intelligence and Instruction*, 193-227, Addison Wesley.

De Marco, T. 1978. *Structured analysis and system specification*, Yourdon Press.

Dede, C. 1986. A review and synthesis of recent research in intelligent computer-assisted instruction. *International Journal of Man-Machine Studies*, **24**, 329-353.

du Boulay, B., O'Shea, T. and Monk, J. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, **14**, 237-249.

Elsom-Cook, M. 1983. A user interface - a Lisp training system. *Proc. Ergonomics Soc. MMI Conference*, Leicester Polytechnic.

Ferraris, M., Midoro, V. and Olimpo, G. 1984. Petri nets as a modelling tool in the development of CAL courseware. In Smith, P.R. (ed.) *Proc. CAL '83*, 41-49, Pergamon Press.

Foley, J., Gibbs, C., Won Chul Kim and Kovacevic, S. 1988. A knowledge-based user-interface management system. In Soloway, E., Frye, D. and Sheppard, S.B. (eds.) *Proc. CHI '88*, 67-72, Addison Wesley.

Foley, J.D., Won Chul Kim and Gibbs, C.A. 1987. Algorithms to transform the formal specification of a user-computer interface. In Bullinger, H.-J. and Shackel, B. (eds.) *Proc. Interact '87*, 1001-1006, Elsevier.

Gagne, R.M., Briggs, L.J. and Wager, W.W. 1988. *Principles of instructional design, 3rd. edition.* Holt, Rinehart and Winston.

Gane, C. and Sarson, T. 1979. *Structured systems analysis: tools and techniques.* Prentice Hall.

Genesereth, M.R. 1982. The role of plans in intelligent teaching systems. In Sleeman, D. and Brown, J.S. (eds.) *Intelligent Tutoring Systems*, 137-155, Academic Press.

Greenfield, P.G. 1988. An investigation into the applicability of definite clause grammars for use in intelligent tutoring systems. *Proc. ITS-88*, 415-422, Montreal.

Hartley, J.R. 1981. Learner initiatives in computer assisted learning. In Howe, J.A.M. and Ross, P.M. (eds.) *Microcomputers in secondary education - issues and techniques*, 102-117, Kogan Page.

Hartley, J.R. 1973. The design and evaluation of an adaptive teaching system. *International Journal of Man-Machine Studies*, **5**, 421-436.

Hartley, J.R. and Sleeman, D.H. 1973. Towards more intelligent teaching systems. *International Journal of Man-Machine Studies*, **5**, 215-236.

Hartley, J.R. and Tait, K. 1986. Learner control and educational advice in computer based learning: the study-station concept. *Computers and Education*, **10**, (2), 259-265.

Heines, J.M. & O'Shea, T.M. 1985. The design of a rule-based CAI tutorial. *International Journal of Man-Machine Studies*, **23**, 1-25.

Hoare, C.A.R. 1972. Proof of Correctness of Data Representations. *Acta Informatica* **1**, (3), 271-281.

Hollan, J.D., Hutchins, E.L. and Weitzman, L.M. 1987. STEAMER: an interactive, inspectable, simulation-based training system. In Kearsley, G.P. (ed.) *Artificial Intelligence and Instruction*, 113-134, Addison Wesley.

Hoppe, H.U. 1988. Task-oriented parsing - a diagnostic method to be used by adaptive systems. In Soloway, E., Frye, D. and Sheppard, S.B. (eds.) *Proc. CHI '88*, 241-247, Addison Wesley.

Jackson, M.A. 1983. *System development.* Prentice Hall.

Jackson, P. and Lefrere, P. 1984. On the application of rule-based techniques to the design of advice-giving systems. *International Journal of Man-Machine Studies*, **20**, 63-86.

Jacob, R.J.K. 1983. Using formal specifications in the design of a human-computer interface. *Communications of the A.C.M.*, **26**, (4), 259-264.

Johnson, P., Diaper, D. and Long, J. 1984. Tasks, skills and knowledge: task analysis for knowledge based descriptions. *Proc. Interact '84*, 23-27.

Johnson, W.L. and Soloway, E. 1987. PROUST: an automatic debugger for Pascal programs. In Kearsley, G.P. (ed.) *Artificial Intelligence and Instruction*, 49-67, Addison Wesley.

Jones, C.B. 1980. *Software development: a rigorous approach.* Prentice-Hall.

Kearsley, G. 1982. Authoring systems in computer based education. *Communications of the A.C.M.*, **25**, (7), 429-437.

Kemke. C. 1987. Representation of domain knowledge in an intelligent help system. In Bullinger, H.-J. and Shackel, B. (eds.) *Proc. INTERACT '87*, 215-220, Elsevier.

Kernighan, B.W. and Lesk, M.E. 1979. *LEARN - computer-aided instruction on Unix. (2nd. edn.).* Bell Laboratories.

Kieras, D. and Polson, P.G. 1985. An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*, **22**, 365-394.

Leveson, N.G. 1980. *Applying behavioural abstractions to information system design and integrity.* Technical Report No. 47, Laboratory of Medical Information Science, University of California, San Francisco.

Leveson, N.G., Wasserman, A.I. and Berry, D.M. 1983. BASIS: a behavioural approach to the specification of information systems. *Information Systems*, **8**, (1), 15-23.

Lewis, M.W., Milson, R. and Anderson, J.R. 1987. The TEACHER'S APPRENTICE: designing an intelligent authoring system for high school mathematics. In Kearsley, G.P. (ed.) *Artificial Intelligence and Instruction*, 269-301, Addison Wesley.

Littman, D. and Soloway, E. 1988. Evaluating ITSs: the cognitive science perspective. In Polson, M.C. and Richardson, J.J. (eds.) *Foundations of Intelligent Tutoring Systems*, 209-242, Lawrence Erlbaum.

Martin, F.A. 1988. Control models in computer-assisted learning. *Expert Systems*, **5**, (4), 316-326.

Martin, F.A. 1987. LIY: learn-it-yourself software interfaces. *Computational Intelligence*, **3**, 28-34.

Martin, F.A. 1983. DYCAL: a tutorial system for the guided exploration of software. In *Module IIA, Proc. Informatics '83*, Singapore.

Mayer, S.R. 1967. Computer-based subsystems for training the users of computer systems. *IEEE Transactions on Human Factors in Electronics*, **HFE-8**, (2), 70-75.

Merrill, D. 1980. Learner control in computer based learning. *Computers and Education*, **4**, 77-95.

Moran, T.P. 1981. The Command Language Grammar: a representation for the user interface of interactive computer systems. *International Journal of Man-Machine Studies*, **15**, (1), 3-50.

Morgan, C.C. 1985. *The Schema Language*. Programming Research Group, Oxford.

Nicolson, R.I. and Scott, P.J. 1986. *Towards an intelligent authoring system.* Internal report, Department of Psychology, University of Sheffield.

Ogborn, J.M. and Johnson, L. 1982. *Conversation Theory*. Brunel University internal report MCSG/TR30.

Ohlsson, S. 1986. Some principles of intelligent tutoring. *Instructional Science*, **14**, 393-326.

O'Shea, T. 1989. Magnets, Martians and microworlds: learning with and learning by OOPS. In Bierman, D., Breuker, J. and Sandberg, J. (eds.) *Artificial Intelligence and Education, Proceedings of the 4th International Conference on AI and Education*, May 1989, IOS, Amsterdam.

O'Shea, T. 1982. A self-improving quadratic tutor. In Sleeman, D. and Brown, J.S. (eds.) *Intelligent Tutoring Systems*, 309-336, Academic Press.

O'Shea, T., Bornat, R., du Boulay, B., Eisenstadt, M. and Page, I. 1984. Tools for creating intelligent computer tutors. In Elithorn, A. and Banerjii, R. (eds.) *Artificial and Human Intelligence*, 181-199, Elsevier.

Papert, S. 1980. *Mindstorms - children, computers and Powerful ideas*. Basic Books, New York.

Parnas, D.L. 1969. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 24th. National A.C.M Conference*, 379-385.

Pask, A.G.S. 1975. *The cybernetics of human learning and performance*. Hutchinson.

Payne, S.J. 1984. Task-action grammars. In *Proceedings of Interact '84*, 139-144.

Reisner, P. 1981. Formal Grammar and human factors design of an interactive graphics system. *IEEE Transactions Software Engineering*, **SE-7**, (2), 229-240.

Rickel, J. 1988. An intelligent tutoring framework for task-oriented domains. In *Proceedings of ITS-88*, 109-115, Montreal.

Rumelhart, D.E. and Norman, D.A. 1978. Accretion, tuning and restructuring: three modes of learning. In Cotton, J.W. and Klatzky, R.L. (eds.) *Semantic factors in cognition*, 3-36, Hillsdale, New Jersey: Erlbaum.

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

Self, J. 1987. Student models: what use are they? In *IFIP/TC3 conference: AI tools in education*, Frascati, May 1987.

Sharratt, B.D. 1987. Top-down interactive systems design: some lessons learnt from using Command Language Grammar. In Bullinger, H.-J. and Shackel, B. (eds.) *Proceedings of Interact '87*, 395-399, Elsevier.

Sleeman, D. 1987. PIXIE: a shell for developing intelligent tutoring systems. In Lawler, R. W. and Yazdani, M. (eds.) *Artificial intelligence and education, volume 1*, 239-265, Ablex Publishing Corporation.

Sleeman, D. and Brown, J.S. (eds.). 1982. *Intelligent Tutoring Systems*, Academic Press.

Smith, R.B. 1986. The alternate reality kit. In *Proceedings of the 1986 IEEE Workshop on Visual Languages*, 99-106.

Spensley, F., Elsom-Cook, M., Byerley, P., Brooks, P., Federici, M., Scaroni, C. 1990. Using multiple teaching strategies in an ITS. In Frasson, C. and Gauthier, G. (eds.) *Intelligent tutoring systems: at the crossroads of artificial intelligence and education*, Ablex Publishing Co.

Spensley, F. and Elsom-Cook, M. 1988. *DOMINIE: teaching and assessment strategies.* Open University CAL Research Group Technical Report. No. 74.

Spivey, J.M. 1989. *The Z notation: a reference manual.* Prentice-Hall.

Stallman, R.M. 1979. *EMACS - the extensible, customizable, self-documenting display editor.* Memo No. 519, Artificial Intelligence Laboratory, MIT, Cambridge, Mass.

Stefik, M. 1981. Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, **16**, (2), 141-169.

Sufrin, B. 1986. Formal Methods and the Design of Effective User Interfaces. In Harrison, M.D. and Monk, A.F. (eds.) *People and Computers: Designing for Usability, Proc. HCI '86*, Cambridge University Press, England.

Sufrin, B. 1982. Formal specification of a display-oriented text editor. *Science of Computer Programming*, **1**, (3), 157-202.

Tang, H., Major, N. and Rivers, R. 1989. From users to dialogues: enabling authors to build an adaptive, intelligent system. In Sutcliffe, A. and Macaulay, L. (eds.) *People and computers V*, 121-135, Cambridge University Press, England.

Waddington, R. and Johnson, P. 1989. A family of task models for interface design. In Sutcliffe, A. and Macaulay, L. (eds.) *People and computers V*, 137-148, Cambridge University Press, England.

Wasserman, A.I. 1984. Developing interactive information systems with the User Software Engineering methodology. In *Proceedings of Interact '84*, **1**, 471-477.

Wenger, E. 1987. *Artificial intelligence and tutoring systems*, Morgan Kaufmann, Inc.

Whitefield, A. 1987. Models in human-computer interaction: a classification with special reference to their uses in design. In Bullinger, H.-J. and Shackel, B. (eds.) *Proceedings of Interact '87*, 57-63, Elsevier.

Woodroffe, M.R. 1988. Plan recognition and intelligent tutoring systems. In Self, J. (ed.) *Artificial intelligence and human learning*, 212-225, Chapman and Hall.

Woods, W.A. 1970. Transition network grammars for natural language analysis. *Communications of the A.C.M.*, **13**, (10), 591-606.

Woolf, B.P. 1987. Theoretical frontiers in building a machine tutor. In Kearsley, G.P. (ed.) *Artificial Intelligence and Instruction*, 229-267, Addison Wesley.

Yourdon, E. and Constantine, L.L. 1979. *Structured design*. Prentice Hall.

Appendix A - Teaching strategy rules

```
; Top-level strategy rules.  Rulesets here are
; concerned both with
; default control (including default handling of
; learner-control requests)
; and a default choice of teaching operation.

(setf MAIN-CONTROL-RULES '(

(a10
 (*BREAK-PRESSED* (not *L-C-ENABLED*))
 ((setf *BREAK-PRESSED* nil)
  (princ "Learner-control is not available in
          this implementation.")
  (MSG-GET-ANY-KEY))
 )
  (a20
    (*BREAK-PRESSED*)
    ((setf *BREAK-PRESSED* nil)
     (setf *LOOPLIMIT* 1)
     (INTERP LEARNER-CHOICE)
     (setf *LOOPLIMIT* nil))
)
(a30
  ((not (NODES-TO-LEARN)) *L-C-ENABLED*)
  ((setf *LOOPLIMIT* 1)
   (INTERP LEARNER-CHOICE)
   (setf *LOOPLIMIT* nil))
)
(a40
  ((not *CURRENT-GOAL*) (not (NODES-TO-LEARN)))
  ((RESET-WHOLE-TA)
   (HALT "All nodes learnt & no further goal
  established."))
)
(a50
  ((not *CURRENT-GOAL*) (not *CURRENT-NODE*))
  ((setf *CURRENT-NODE* (ROOT *TA*)))
)
(a60
  ((not *CURRENT-GOAL*))
  ((setf *CURRENT-GOAL* (NEXT-NODE *CURRENT-NODE*))
   (setf *CURRENT-NODE* *CURRENT-GOAL*))
)
```

```
  (a70
    (*CURRENT-GOAL*
    (NEW-NODE-P *CURRENT-GOAL*)
    (DESIGNER-CONTROL-RULES-EXIST *CURRENT-GOAL*))
    ((INTERP (eval (DESIGNER-CONTROL-RULES-EXIST
                    *CURRENT-GOAL*))))
  )
  (a80
    (*CURRENT-GOAL*)
    ((SELECT-TEACHING-OP)) ; = (INTERP TEACHING-OP-RULES)
  )
      ))

;LEARNER-CHOICE rules. *L-C-ENABLED* is definitely true.
;Either BREAK has been
;pressed (there may or may not be a current goal), or
;there are no more nodes
;to learn so revision is being offered.

(setf LEARNER-CHOICE '(
  (b10
    ;Unconditionally:-
    ()
    ((terpri)
     (princ "OK - What would you like to learn? ")              (terpri) (terpri)
     (princ "Press RETURN to continue with your original
             topic.") (terpri)
    (princ "Type  Q to quit LIY") (terpri)
    (princ "      B to browse") (terpri)
    (princ "      E to explore ")
    (princ *APPLICATION*)
    (princ " freely") (terpri)
    (princ "      or the topic's name.") (terpri)
    (princ "All end with RETURN") (terpri)
    (VIDEO HI) (princ ">") (VIDEO)
    (setf *RESPONSE* (READ-RESPONSE))
    (terpri))
)
(b20
  ((equal *RESPONSE* 'Q))
  ((princ "Saving your current environment..")
   (terpri)
   (UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
   (SAVE-TA)
   (HALT "You will automatically restart from here next
          time. "))
)
(b30
  ((equal *RESPONSE* 'B))
  ((setf *RESPONSE* (DIS-TA)))
)
```

```
(b40
  ((equal *RESPONSE* 'E))
  ((FREE-EXPLORATION)
   (EXIT-RULESET))
)
(b50        ; For debugging.
  ((equal *RESPONSE* '!))
  ((setf sys:*break-event* 'break)
   (break))
)
(b60
  ((not *RESPONSE*)
   (equal *INTERACTION-BEFORE-BRK* 'TEST)
   *L-C-NODE-LIMIT*); i.e. already navigating...
  ((setf *L-C-NODE-LIMIT* nil)
   (UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
   (setf *CURRENT-GOAL* nil);Go back to
   (setf *CURRENT-NODE* nil); "natural" next node.
   (setf *DUCK-CNT* (+ *DUCK-CNT* 1))
   (setf *INTERACTION-BEFORE-BRK* 'UNKNOWN)
   (EXIT-RULESET))
)
(b70
  ((not *RESPONSE*)
   *L-C-NODE-LIMIT*);i.e. already navigating...
  ((setf *L-C-NODE-LIMIT* nil)
   (UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
   (setf *CURRENT-GOAL* nil);Go back to
   (setf *CURRENT-NODE* nil); "natural" next node.
   (EXIT-RULESET))
)
(b80
  ((not *RESPONSE*))
  ((EXIT-RULESET))
)
(b90
  ((not (boundp *RESPONSE*)))
  ((princ "is not a correct name.") (terpri)
   (princ "Press Ctrl-Break again and type ")
   (princ "the correct name when requested.")
   (MSG-GET-ANY-KEY)
   (EXIT-RULESET))
)
(b100
  ((ALREADY-LEARNT *RESPONSE*))
  ((UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
   (setf *CURRENT-GOAL* *RESPONSE*)
   (setf *CURRENT-NODE* *CURRENT-GOAL*)
   (setf *L-C-NODE-LIMIT* *CURRENT-GOAL*)
   (princ "You are revising ") (princ *RESPONSE*)
   (terpri)
```

```
   (MSG-GET-ANY-KEY)
   (EXIT-RULESET))
)
(b110
  ((PREREQUISITES-LEARNT *RESPONSE*)
              ; PREREQUISITES-LEARNT is a
            ; boolean function on TA.
   (equal *INTERACTION-BEFORE-BRK* 'TEST))
  ((UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
   (setf *CURRENT-GOAL* *RESPONSE*)
   (setf *CURRENT-NODE* *CURRENT-GOAL*)
   (setf *L-C-NODE-LIMIT* *CURRENT-GOAL*)
   (setf *DUCK-CNT* (+ *DUCK-CNT* 1))
   (setf *INTERACTION-BEFORE-BRK* 'UNKNOWN)
   (L-C-OUTCOME-A)
   (EXIT-RULESET))
)
(b120
  ((PREREQUISITES-LEARNT *RESPONSE*))
  ((UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
   (setf *CURRENT-GOAL* *RESPONSE*)
   (setf *CURRENT-NODE* *CURRENT-GOAL*)
   (setf *L-C-NODE-LIMIT* *CURRENT-GOAL*)
   (setf *FLIT-CNT* (+ *FLIT-CNT* 1))
   (L-C-OUTCOME-A)
   (EXIT-RULESET))
  )
  (b130
     ;Otherwise:-
    ()        ; Unconditionally...
    ((setf *LOOPLIMIT* 1) (INTERP L-C-ADVICE))
  )
     ))

;Learner-Control-ADVICE rules. At this point
; there's a distinct
; (i.e. non-null and valid) student *RESPONSE*
; concerning a routing
; request, possibly (but not necessarily) following
; BREAK, but there are
; unlearnt prerequisites in TA.
;          The learner-control outcomes are :-
;
;   L-C-OUTCOME-A : Message that the learner's
;                   move is about to take place.
;
;   L-C-OUTCOME-B : Mild advice against moving.
;
;   L-C-OUTCOME-C : Stronger advice against moving.
;
;   L-C-OUTCOME-D : Very strong advice against moving.
```

```
(setf L-C-ADVICE '(

  (c10
    ((equal (NO-OF-PREREQ *RESPONSE*) 1)
     (not (WEAK)) (not (DUCKER)) (not (FLITTER))
     (equal *INTERACTION-BEFORE-BRK* 'TEST))
    ((L-C-OUTCOME-A)
     (UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
     (setf *CURRENT-GOAL* *RESPONSE*)
     (setf *CURRENT-NODE* *CURRENT-GOAL*)
     (setf *L-C-NODE-LIMIT* *CURRENT-GOAL*)
     (setf *DUCK-CNT* (+ *DUCK-CNT* 1))
     (setf *INTERACTION-BEFORE-BRK* 'UNKNOWN)
     (EXIT-RULESET))
  )
  (c20
    ((equal (NO-OF-PREREQ *RESPONSE*) 1)
     (not (WEAK)) (not (DUCKER)) (not (FLITTER)))
    ((L-C-OUTCOME-A)
     (UNMARK-AS-BEING-TAUGHT *CURRENT-GOAL*)
     (setf *CURRENT-GOAL* *RESPONSE*)
     (setf *CURRENT-NODE* *CURRENT-GOAL*)
     (setf *L-C-NODE-LIMIT* *CURRENT-GOAL*)
     (setf *FLIT-CNT* (+ *FLIT-CNT* 1))
     (EXIT-RULESET))
  )
  (c30
    ((equal (NO-OF-PREREQ *RESPONSE*) 1)
     (not (WEAK)) (not (DUCKER)) (FLITTER))
    ((L-C-OUTCOME-B))
  )
  (c40
    ((equal (NO-OF-PREREQ *RESPONSE*) 1)
     (not (WEAK)) (DUCKER) (not (FLITTER)))
    ((L-C-OUTCOME-B))
  )
  (c50
    ((equal (NO-OF-PREREQ *RESPONSE*) 1)
     (not (WEAK)) (DUCKER) (FLITTER))
    ((L-C-OUTCOME-B))
  )
  (c60
    ((equal (NO-OF-PREREQ *RESPONSE*) 1)
     (WEAK) (not (DUCKER)) (not (FLITTER)))
    ((L-C-OUTCOME-B))
  )
  (c70
    ((equal (NO-OF-PREREQ *RESPONSE*) 1)
     (WEAK) (not (DUCKER)) (FLITTER))
    ((L-C-OUTCOME-B))
  )
```

```
(c80
  ((equal (NO-OF-PREREQ *RESPONSE*) 1)
   (WEAK) (DUCKER) (not (FLITTER)))
  ((L-C-OUTCOME-B))
)
(c90
  ((equal (NO-OF-PREREQ *RESPONSE*) 1)
   (WEAK) (DUCKER) (FLITTER))
  ((L-C-OUTCOME-C))
)
(c100
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (not (WEAK)) (not (DUCKER)) (not (FLITTER)))
  ((L-C-OUTCOME-B))
)
(c110
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (not (WEAK)) (not (DUCKER)) (FLITTER))
  ((L-C-OUTCOME-C))
)
(c120
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (not (WEAK)) (DUCKER) (not (FLITTER)))
  ((L-C-OUTCOME-C))
)
(c130
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (not (WEAK)) (DUCKER) (FLITTER))
  ((L-C-OUTCOME-C))
)
(c140
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (WEAK) (not (DUCKER)) (not (FLITTER)))
  ((L-C-OUTCOME-C))
)
(c150
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (WEAK) (not (DUCKER)) (FLITTER))
  ((L-C-OUTCOME-C))
)
(c160
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (WEAK) (DUCKER) (not (FLITTER)))
  ((L-C-OUTCOME-C))
)
(c170
  ((> (NO-OF-PREREQ *RESPONSE*) 1)
   (WEAK) (DUCKER) (FLITTER))
  ((L-C-OUTCOME-D))
)
```

```
(c180
  ((equal *RESP* 'F);Used in the L-C-OUTCOMEs.
   (equal *INTERACTION-BEFORE-BRK* 'TEST))
  ((UNMARK-AS-BEING-TAUGHT *CURRENT-NODE*)
   (setf *CURRENT-GOAL* *RESPONSE*)
   (setf *CURRENT-NODE* *CURRENT-GOAL*)
   (setf *L-C-NODE-LIMIT* *CURRENT-GOAL*)
   (setf *INTERACTION-BEFORE-BRK* 'UNKNOWN)
   (setf *DUCK-CNT* (+ *DUCK-CNT* 1))
   (EXIT-RULESET))
)
(c190
  ((equal *RESP* 'F));Used in the L-C-OUTCOMEs.
  ((UNMARK-AS-BEING-TAUGHT *CURRENT-NODE*)
   (setf *CURRENT-GOAL* *RESPONSE*)
   (setf *CURRENT-NODE* *CURRENT-GOAL*)
   (setf *L-C-NODE-LIMIT* *CURRENT-GOAL*)
   (setf *FLIT-CNT* (+ *FLIT-CNT* 1)))
  )
    ))
```

```
;Having selected a node to teach, select an appropriate
;teaching operation.

;The logic is as follows:
; If there are DES-control-rules, run them
;- they must take over completely.
;Select a teaching operation by INTERP TEACHING-OP-RULES.
;This looks to see if there are DES-teaching-op-rules,
;in which
;case they're interpreted and they must take over
;from this point.
;If not, then the next default teaching operation
;is chosen which is
;stored in the COMMAND-STR attached to each node.

(setf TEACHING-OP-RULES '(

  (e10
    ((DESIGNER-TEACHING-OP-RULES-EXIST *CURRENT-GOAL*))
    ((INTERP (eval (DESIGNER-TEACHING-OP-RULES-EXIST
                    *CURRENT-GOAL*)))
     (EXIT-RULESET))
  )
  (e20
    ((NEW-NODE-P *CURRENT-GOAL*))
    ((GET-NEW-NODE-READY *CURRENT-GOAL*))
  )
```

```
(e30
  ((equal *LAST-INTERACTION* 'TEST)
   (equal *LAST-TEST-RESULT* 'UNKNOWN))
  ((EVALUATE-LAST-TEST))
)
(e40
  ((equal *LAST-INTERACTION* 'TEST)
   (equal *LAST-TEST-RESULT* 'PASS))
  ((INCR-COMPTNCE))
)
(e50
  ((equal *LAST-INTERACTION* 'TEST)
   (equal *LAST-TEST-RESULT* 'FAIL))
  ((DECR-COMPTNCE))
)
(e60
  ((equal *LAST-INTERACTION* 'TEST)
   (equal *LAST-TEST-RESULT* 'FAIL)
   (equal *SAME-TEST-PREVIOUS-TIME* 'FAIL))
  ((terpri)
   (princ "Type NEXT if you would like to move on, ")
   (princ "or press RETURN to try again: ")
   (setf *SKIPPING* (READ-RESPONSE)))
)
(e70
  ((equal *LAST-INTERACTION* 'TEST)
   (equal *LAST-TEST-RESULT* 'FAIL)
   (equal *SAME-TEST-PREVIOUS-TIME* 'FAIL)
   (not *SKIPPING*))
  ((SET-REVISING-NODE-COM-STR *CURRENT-GOAL*))
)
(e80
  ((equal *LAST-INTERACTION* 'TEST)
   (equal *LAST-TEST-RESULT* 'FAIL)
   (equal *SAME-TEST-PREVIOUS-TIME* 'FAIL)
   (equal *SKIPPING* 'NEXT))
  ((terpri) (princ "The correct response should have
                          been:")
   (terpri) (princ *CORRECT-STR*)
   (CLEAR-CURRENT-TEST-RESULT)
   (CLEAR-PREVIOUS-TEST-RESULT)
   (MSG-GET-ANY-KEY)
   (GET-NEW-FRAGMENT-READY))
)
(e90
  ((equal *LAST-INTERACTION* 'TEST)
   (equal *LAST-TEST-RESULT* 'FAIL)
   (not (equal *SAME-TEST-PREVIOUS-TIME* 'FAIL)))
  ((SAVE-CURRENT-TEST-RESULT)
   (SET-REVISING-NODE-COM-STR *CURRENT-GOAL*))
)
```

```
(e100
  ((equal *LAST-INTERACTION* 'DEL-BRK))
  ((SET-REVISING-NODE-COM-STR *CURRENT-GOAL*))
)
(e110
  ((equal *LAST-INTERACTION* 'APP-BRK))
  ((SET-REVISING-NODE-COM-STR *CURRENT-GOAL*))
)
(e120
  ((null *NODE-COM-STR*))
  ((UPDATE-ENVIRONMENT)
   (setf *CURRENT-GOAL* nil)
   (EXIT-RULESET))
)
(e130
  (*NODE-COM-STR*)
  ((DO-NEXT-TEACHING-OP))
)
    ))
```

Appendix B

Development of the DIALLER tutorial

The objective of this section is to demonstrate the evolution of the full task classification structure for the DIALLER tutorial - essential for modelling the domain during learner diagnosis - from the necessary elements of the user interface specification. Appendix C demonstrates the techniques described here in the development of the ELICITUT tutorial.

The elements of the user interface were discussed in chapter 2 and comprise:

(i) task command hierarchy;

(ii) syntax of input *commands*;

(iii) terminators for application (non-command) input;

(iv) semantics indicating flow of control following leaf command processing.

The tutorial development process will demonstrate, in sequence, the following stages:

(1) The task hierarchy is shown - unstructured - from the systems analysis stage. This will not be in pedagogic sequence and will be very similar to figure 2.3 showing the DIALLER operational ordering and to figure 3.19 showing a similar structure for the MS-DOS example.

(2) Next, the commands which are associated with the tasks in the hierarchy from the previous stage are indicated. For the two LIY tutorials these commands are very simple, but they represent the *command syntax* or "syntax tokens" for each task.

(3) The terminators for non-command (i.e. "application") input are defined in relation to the hierarchy in stage 1.

(4) Semantic information concerned with the application control flow following the processing of a leaf node is indicated.

(5) Disregarding the added information from stages 2 to 4 for the moment, diagrams show the results of applying the heuristic and binary tree transformations described in section 3.5 to the structure from stage 1.

(6) Finally, the Lisp representation is shown which is actually used in the DIALLER tutorial and which corresponds to the structure from stage 5, augmented with the extra information from stages 2 to 4.

(1) The development of the task hierarchy

This stage illustrates the design process in action following the systems analysis stage. A "first cut" design (along the lines of fig. 2.3) yields figure B.1.

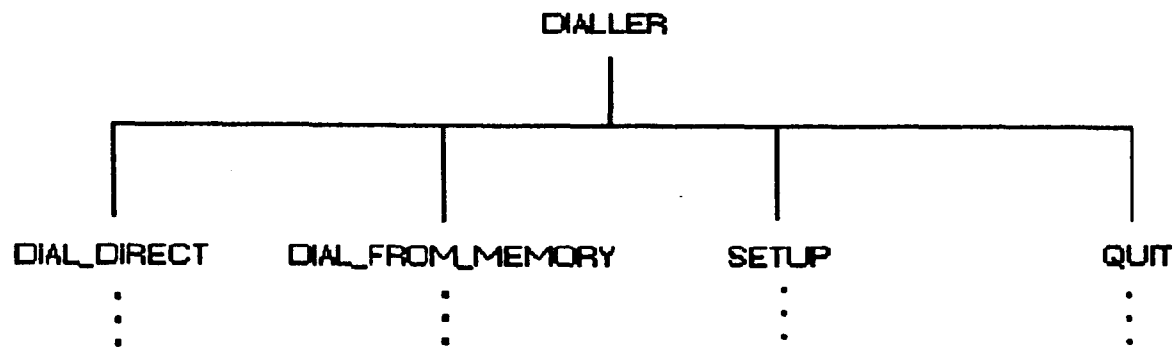


Fig. B.1

This shows that, at the top level, the DIALLER operations can be thought of as consisting of the following:

DIAL_DIRECT	dialling from the keyboard;
DIAL_FROM_MEMORY	using a file of stored numbers;
SETUP	set up various parameters, including a file of stored numbers;
QUIT	leave the DIALLER program.

The leaves from figure B.1 can each be further decomposed. As an example, consider SETUP. This breaks down into the further operations as shown in figure B.2.

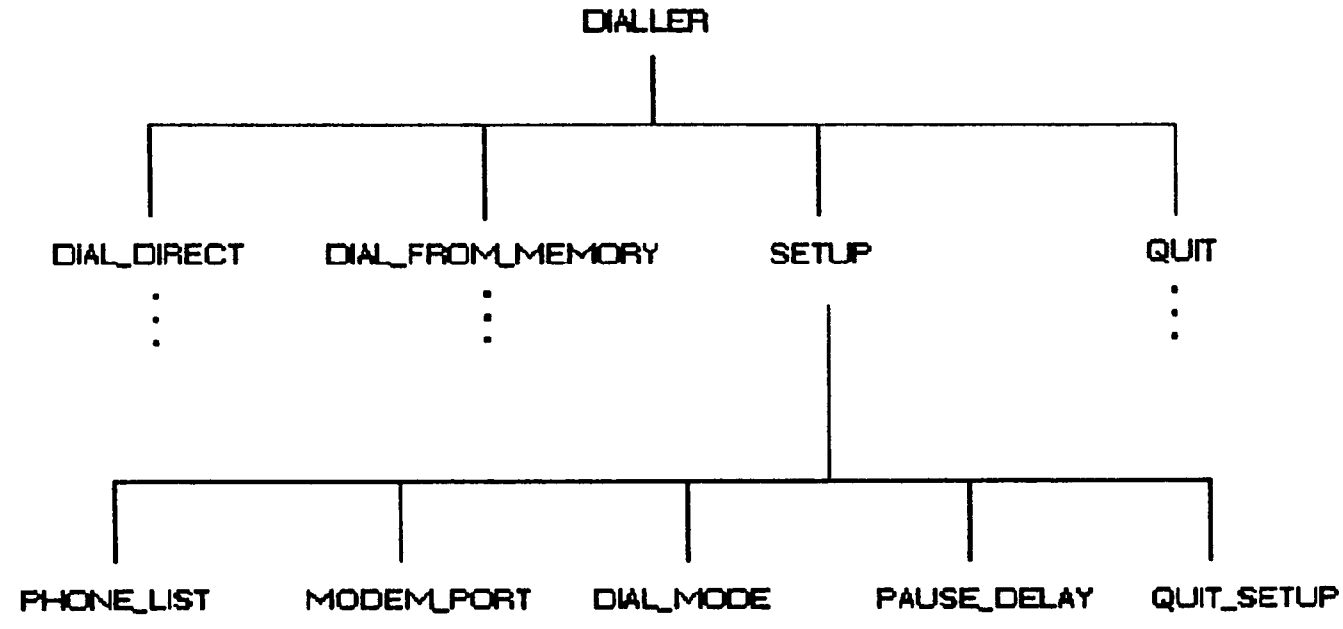


Fig. B.2

PHONE_LIST	is the location of a file of stored numbers.
MODEM_PORT	is the port address for the modem (1 or 2).
DIAL_MODE	is either *pulse* or *tone* dialling (P or T).
PAUSE_DELAY	built-in dialling pause - a default value which can be changed within SETUP.
QUIT_SETUP	will enable the user to quit the SETUP phase. (The option to quit without saving the altered SETUP must be allowed for, but is not shown in this figure.)

Again, further operational sub-division is possible. Figure B.3 illustrates PHONE-LIST decomposed into separate operations connected with the file-name and the directory. There is a *quit* operation for this sub-operation.

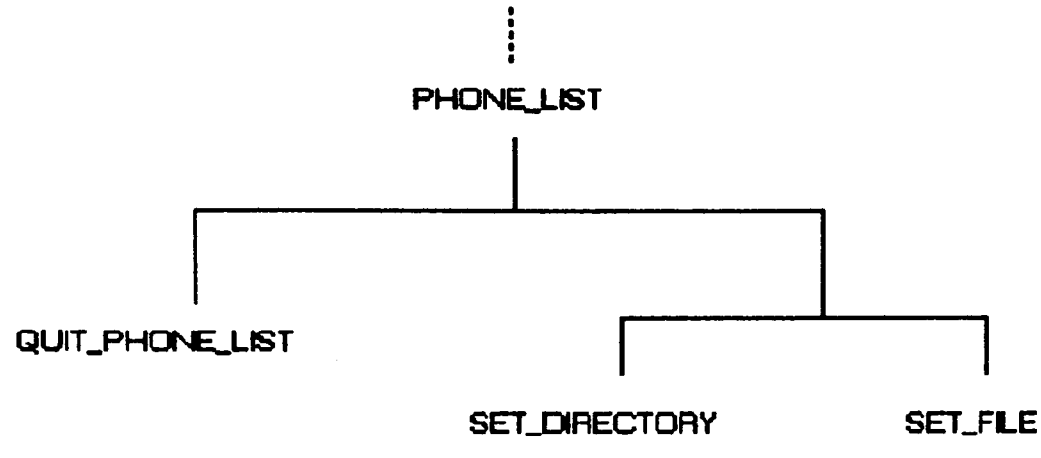


Fig. B.3

In fact the directory- and file-name operations decompose still further, as shown in figure B.4. Not only will there be an operation to invoke the setting up of the directory; there will also be the action of entering the directory-name. A similar consideration applies to the file-name.

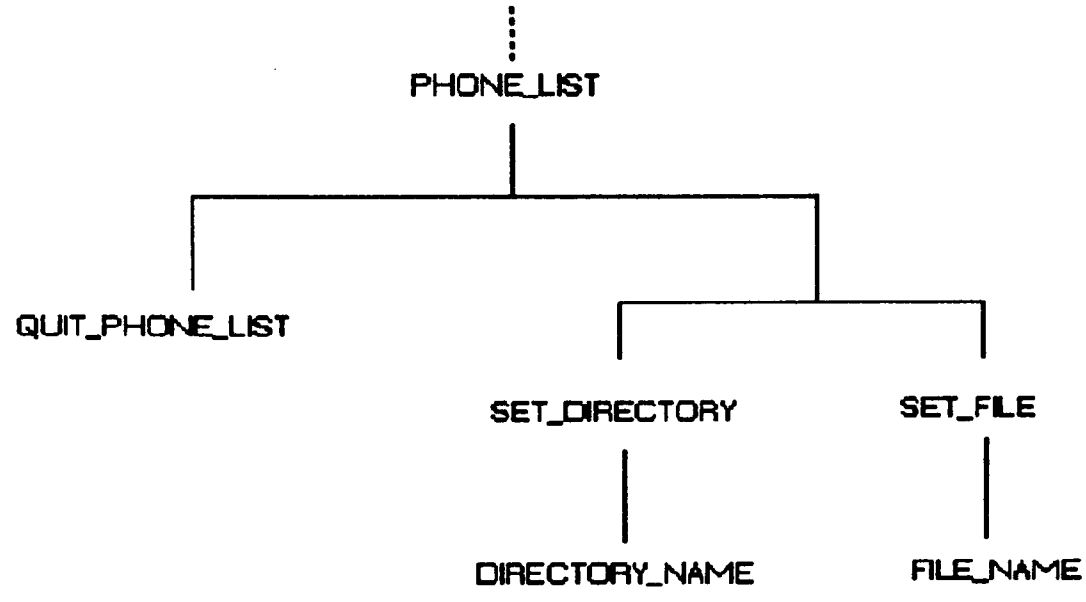


Fig. B.4

The full set of DIALLER operations is shown in figure B.5.

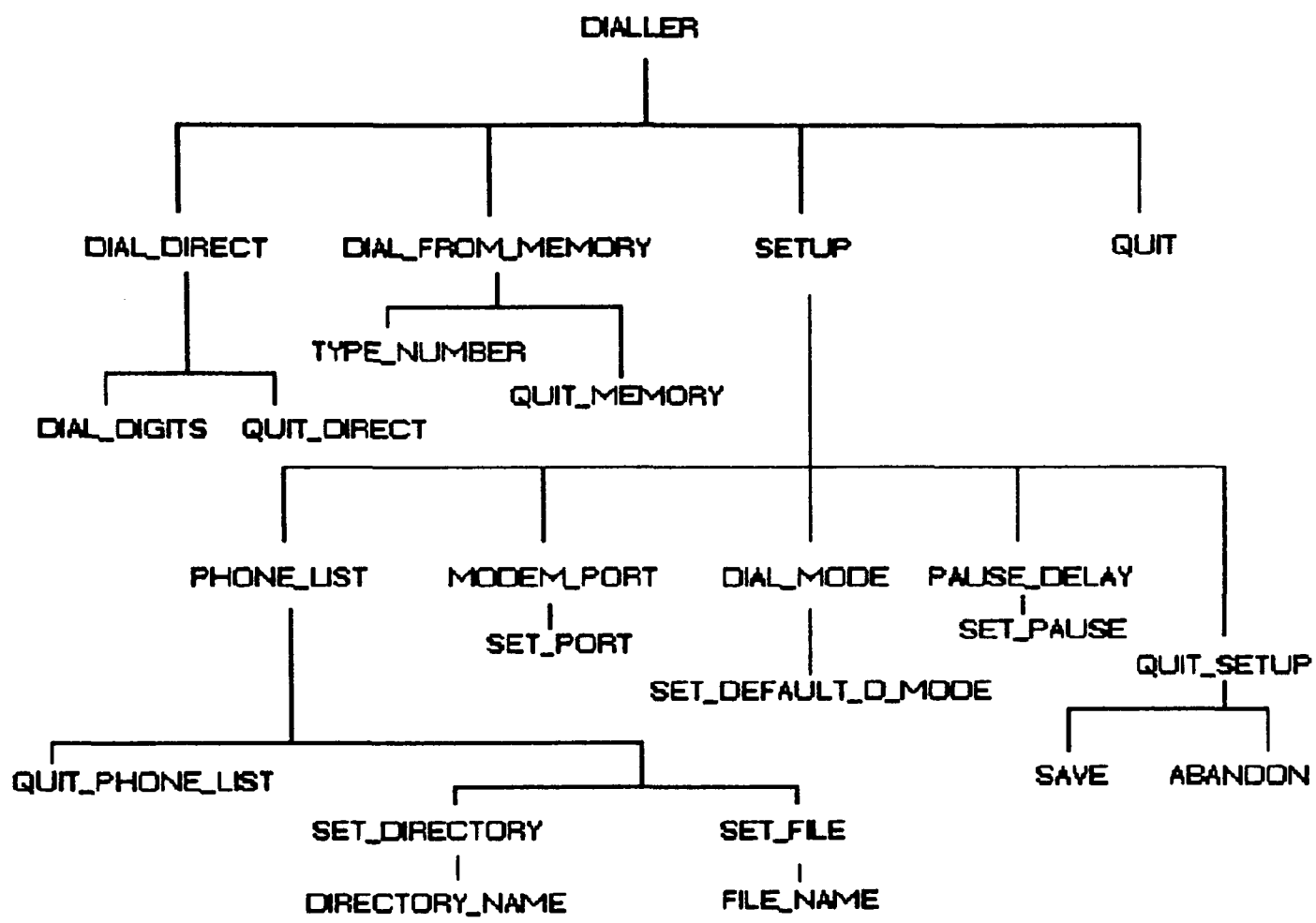


Fig. B.5

(2) Command syntax

Commands are designed for those tasks in the hierarchy which can be invoked by the user, as follows. All the tasks are listed, but only the command operations are assigned syntax at this point. Non-command (i.e. "application") input is dealt with in the next stage.

DIAL_DIRECT	dlD
DIAL_FROM_MEMORY	mlM
SETUP	slS
QUIT	esc
DIAL_DIGITS	
QUIT_DIRECT	esc
TYPE_NUMBER	
QUIT_MEMORY	esc
PHONE_LIST	flF
MODEM_PORT	mlM
DIAL_MODE	dlD
PAUSE_DELAY	plP
QUIT_SETUP	esc
QUIT_PHONE_LIST	esc
SET_DIRECTORY	dlD
SET_FILE	flF
SET_PORT	
SET_DEFAULT_D_MODE	
SET_PAUSE	

SAVE	s|S
ABANDON	esc

DIRECTORY_NAME

FILE_NAME

(3) Non-command input

The terminators for non-command input for the following operations were all designated as being ENTER:

DIAL_DIGITS
TYPE_NUMBER

SET_PORT *
SET_DEFAULT_D_MODE *
SET_PAUSE *

DIRECTORY_NAME
FILE_NAME

* A variation was implemented in which fixed-length input was accepted (just one character in the first two of these cases) rather than requiring termination with ENTER. See section 3.4.1.

(4) Flow of control

The semantic flow-of-control information is required. The flow over the task tree as a whole follows the tree when moving from the top down. Having processed a leaf command, though, the control flow to a new task needs to be specified. It is shown in figure B.6 as dotted lines:

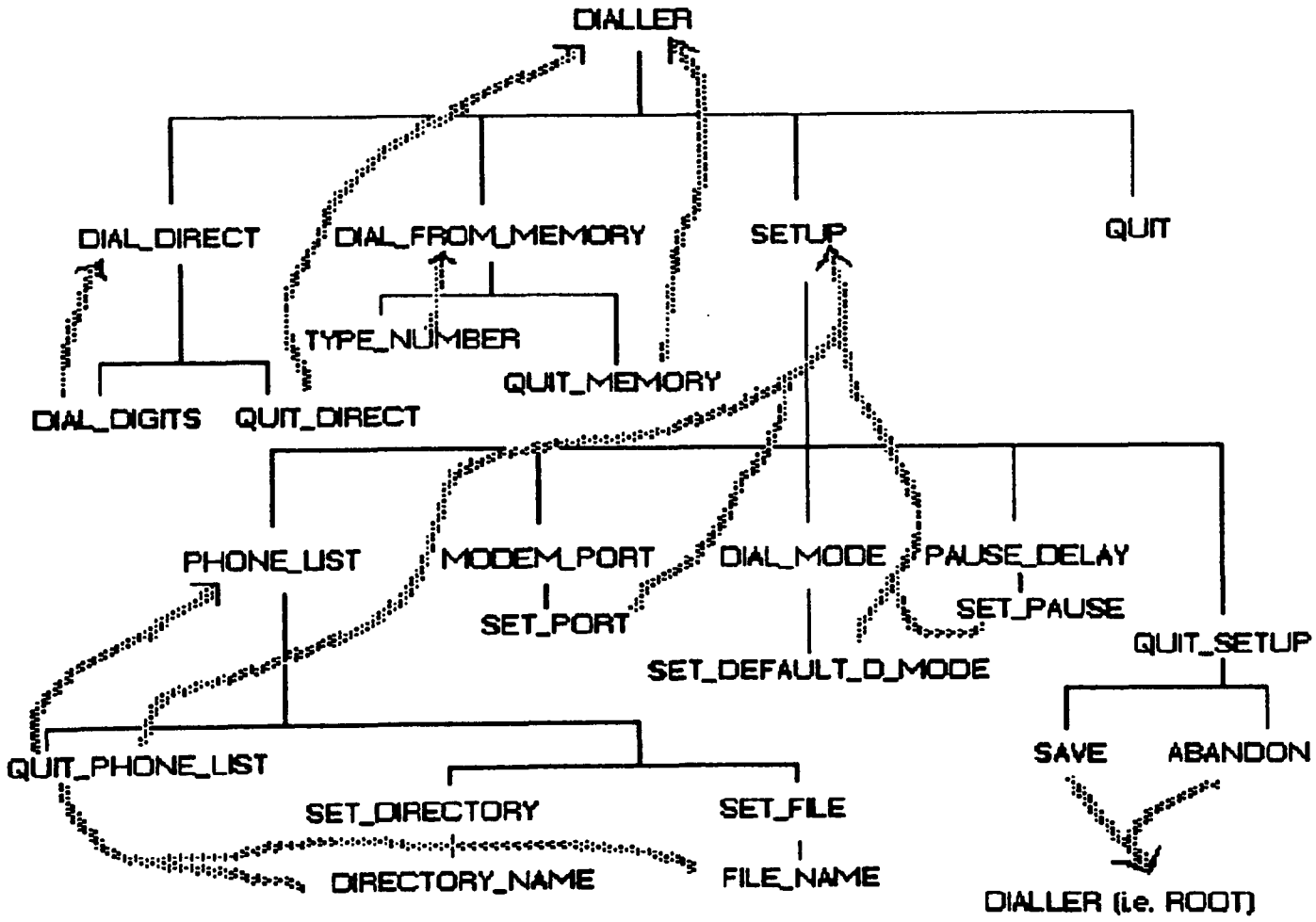


Fig. B.6

(5) Application of heuristic and binary tree transformations

The objective of the transformations carried out at this stage is to turn the tree from an ordering which is *operational* to an ordering which is *pedagogic* (and to turn it from a *general* tree, in which a node can have an arbitrary number of children, to a *binary* tree, in which the number of children cannot exceed two. In this latter case, dummy nodes, labelled *proceed-n*, are introduced into the structure although they do not represent teachable topics in the tutorial domain). The heuristics and transformations applied here are set out in section 3.5. Dependency information between siblings is indicated my marking the right-hand arc in the binary tree as described in chapter 3.

First, apply heuristic (c) ("teach exit-type operations first") to figure B.5 yielding figure B.7.

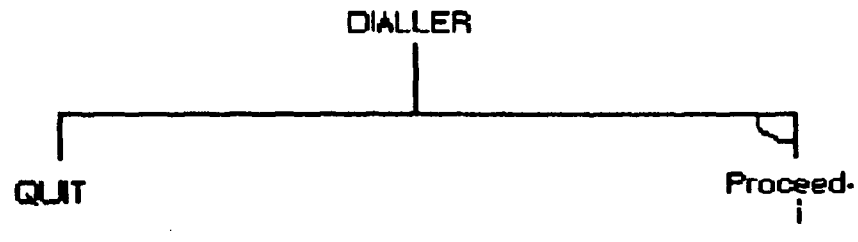


Fig. B.7

Now apply heuristic (d) ("teach configuration-type operations last") to give figure B.8.

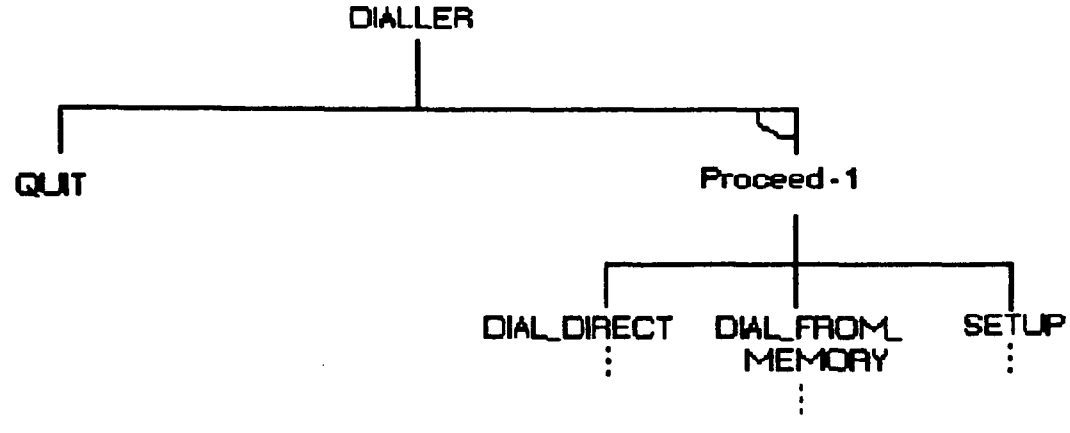


Fig. B.8

Apply heuristic (c) again to the component operations of DIRECT and MEMORY giving figure B.9.

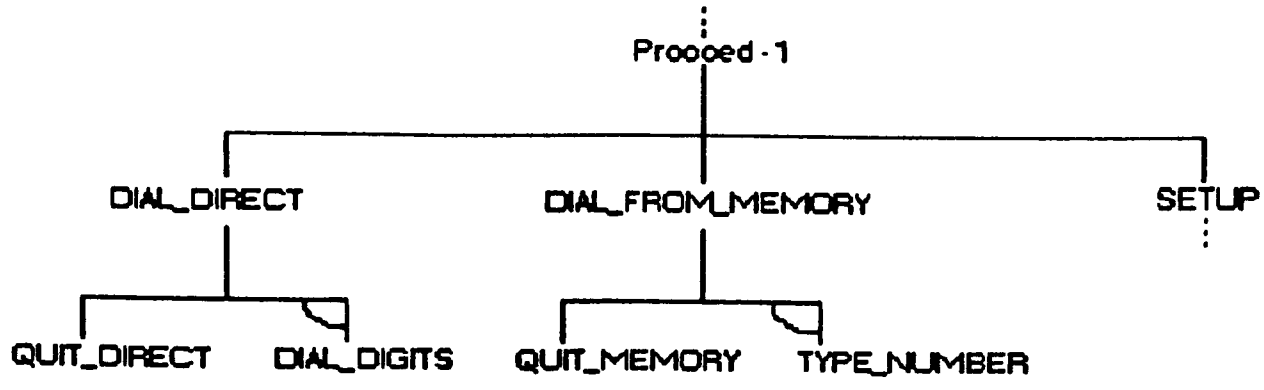


Fig. B.9

Dealing with the components of SETUP separately in due course, figure B.9 can be transformed to binary-tree form as shown in figure B.10 by applying transformation 4.

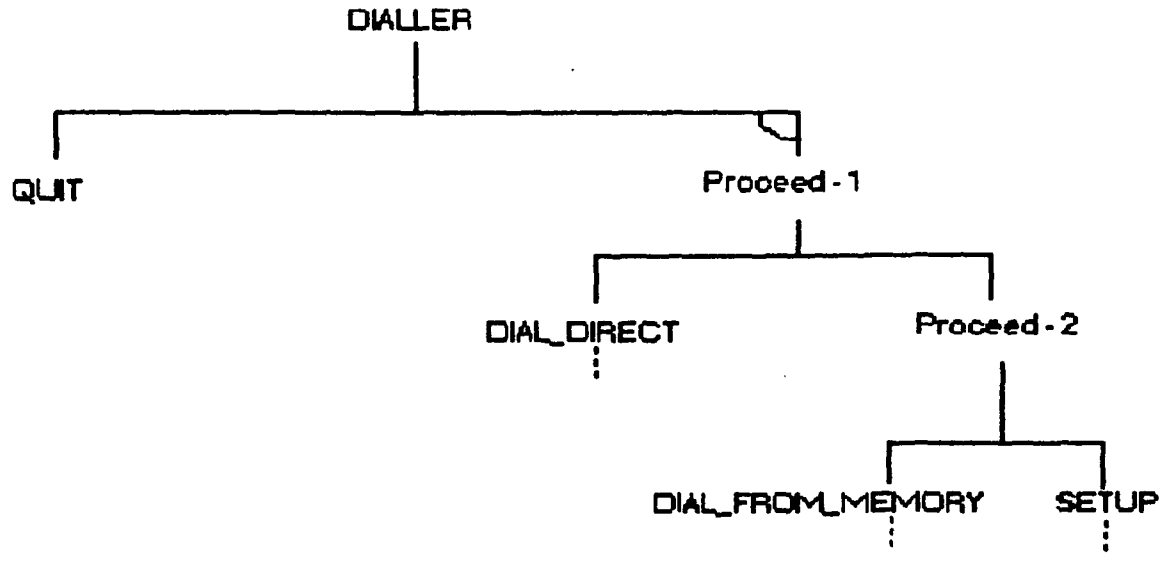


Fig. B.10

Figure B.11 shows the application of heuristic (c) again to the components of SETUP.

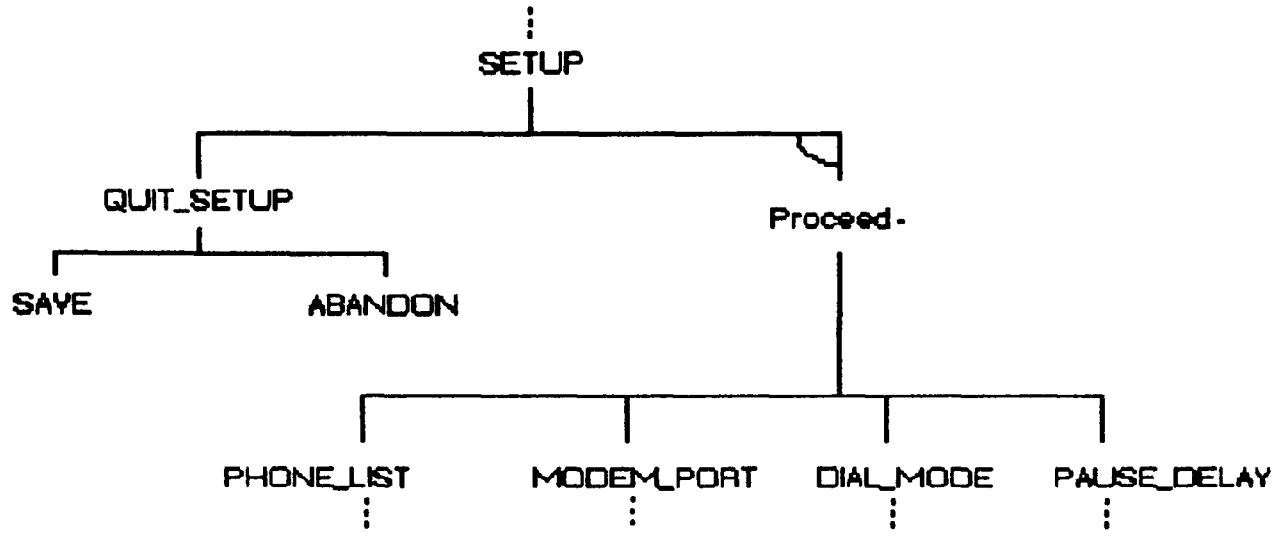


Fig. B.11

Applying heuristic (c) again to PHONE_LIST from figure B.11 yields figure B.12.

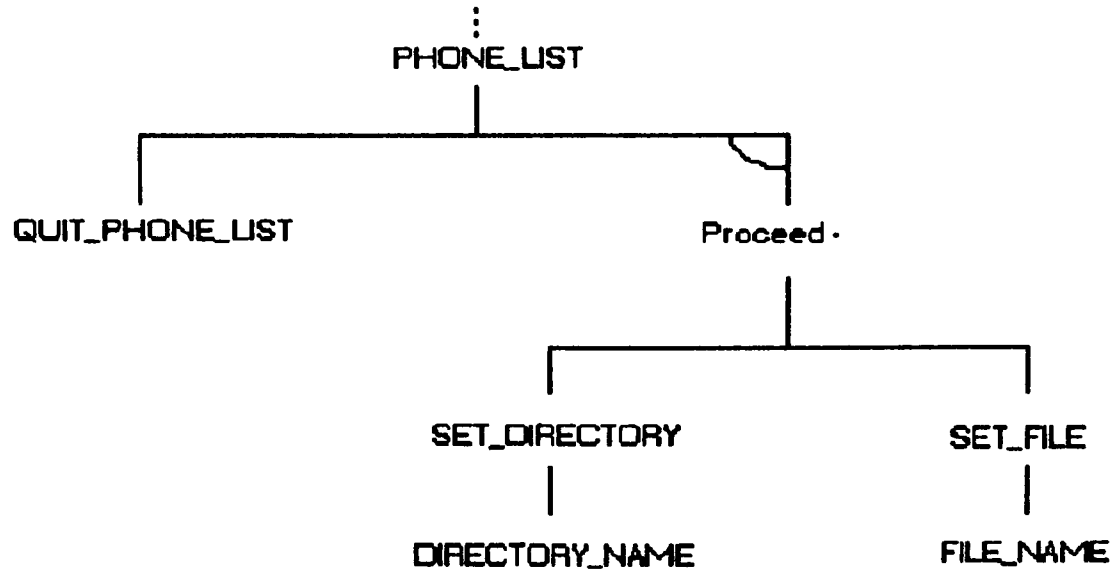


Fig. B.12

The remaining children of SETUP can be arranged in binary tree-form by using transformation 4, giving figure B.13.

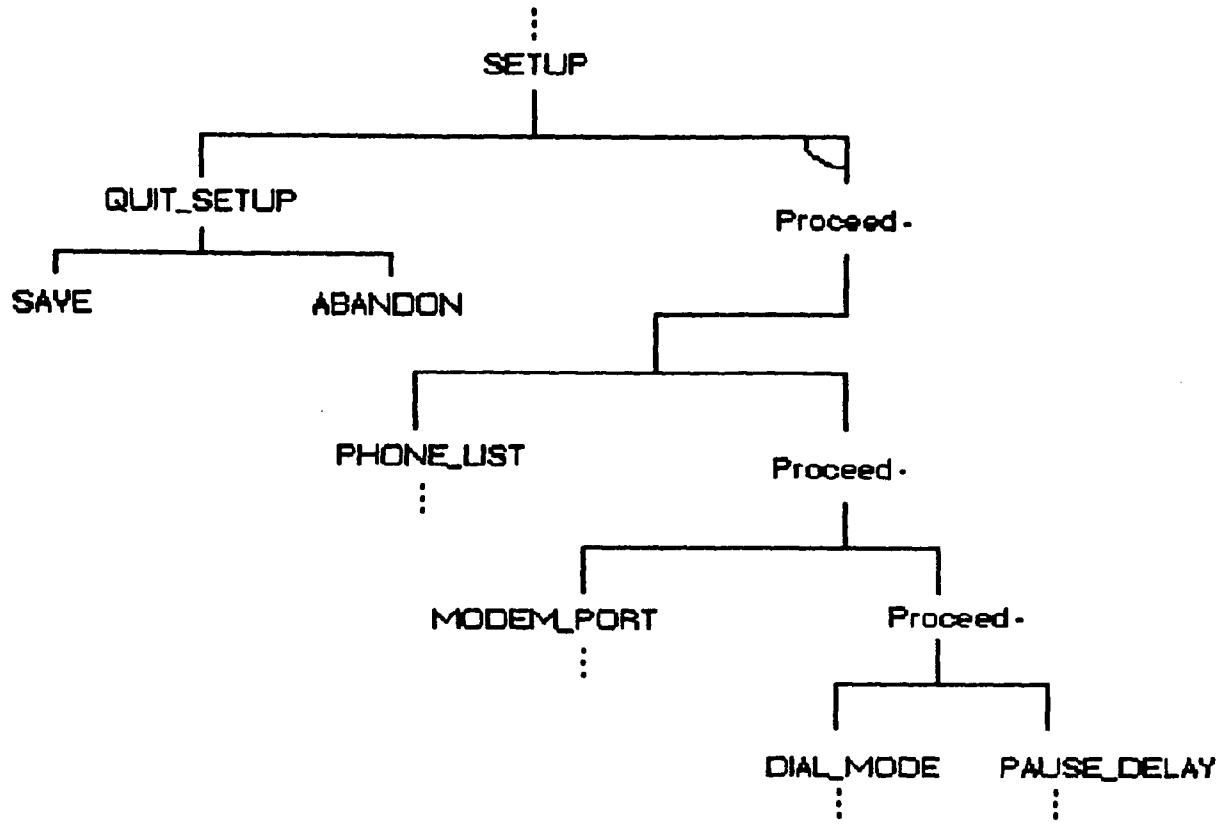


Fig. B.13

The complete structural representation of DIALLER, in pedagogic ordering, is brought together in figures B.14(a) and B.14(b). To preserve the clarity of the diagrams the information developed in stages (2) to (4) above has not been shown, although it constitutes parts of the Lisp representation given in the next section.

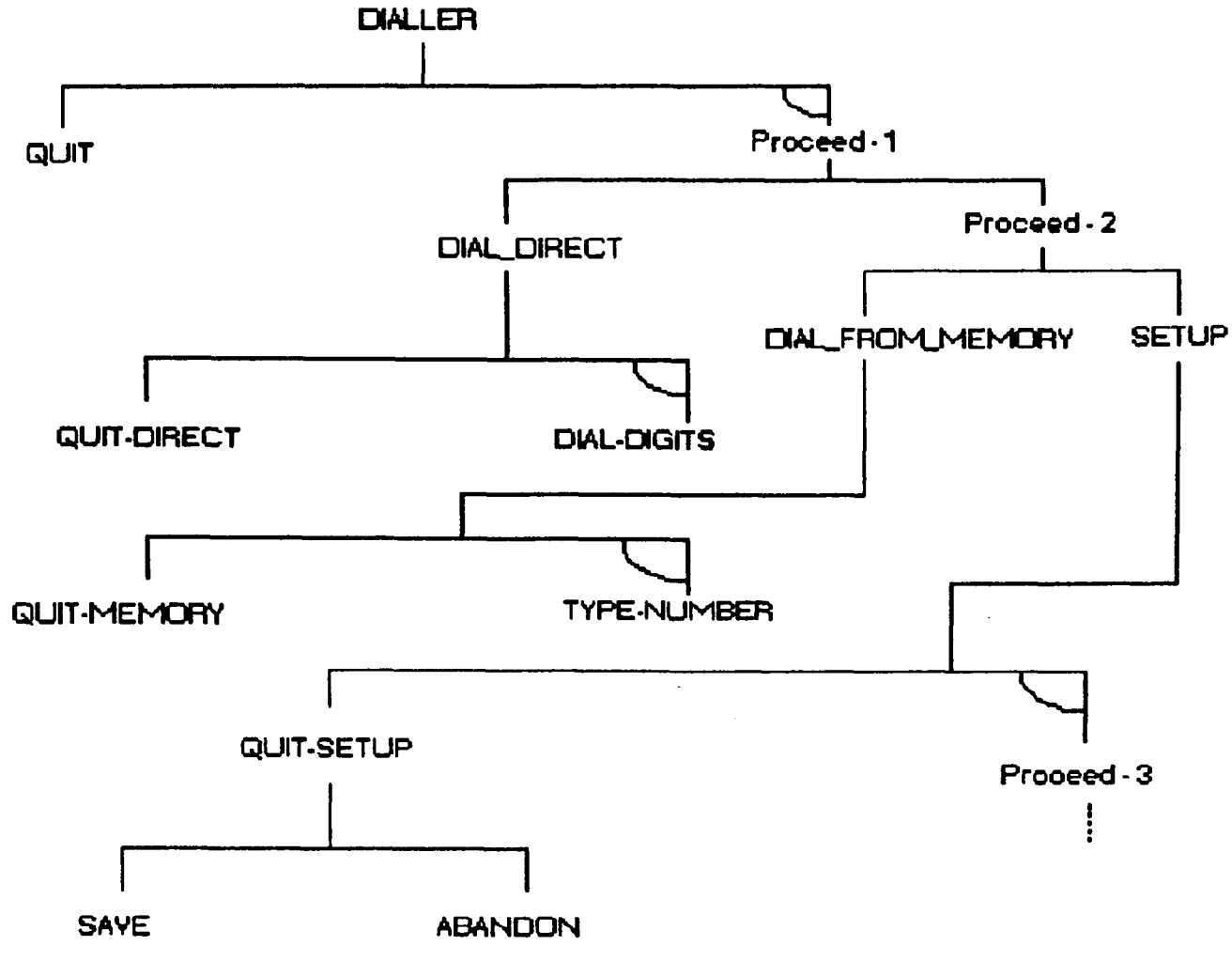


Fig. B.14(a)

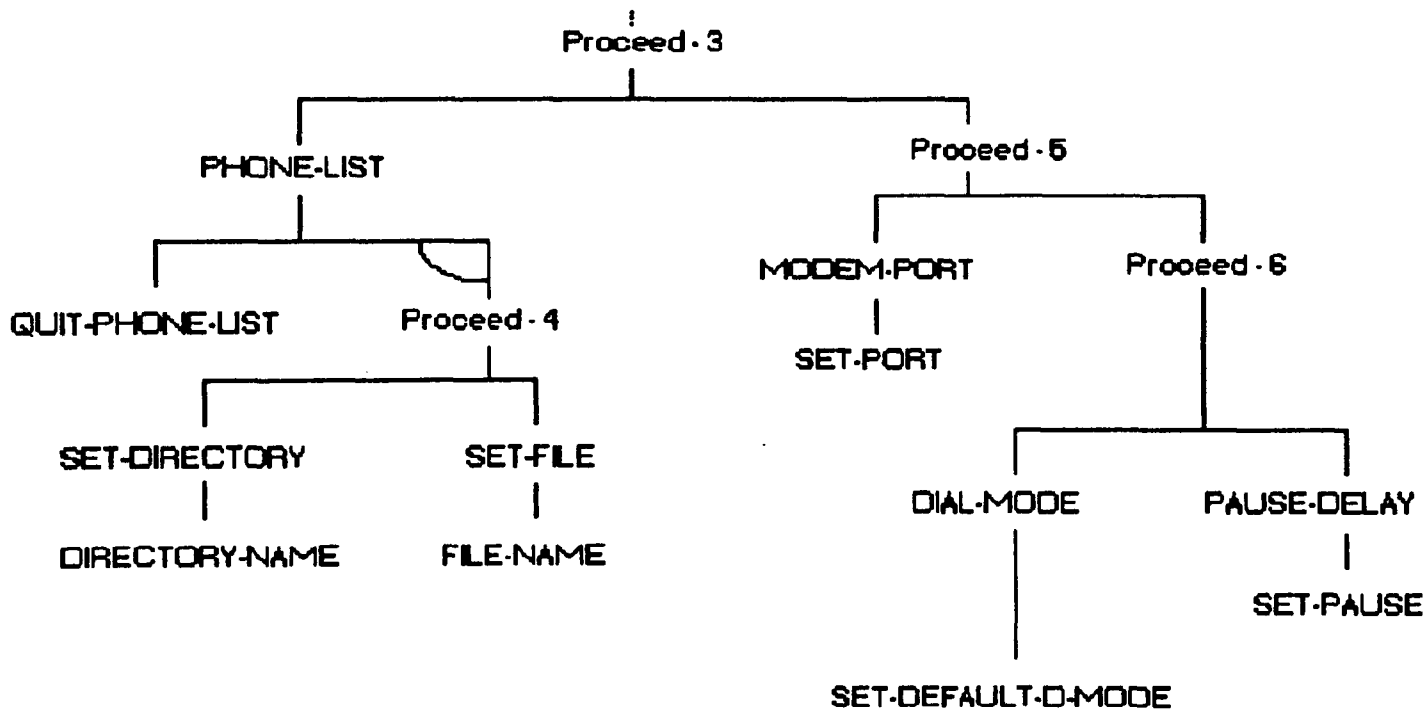


Fig. B.14(b)

(6) Lisp representation of the DIALLER structure

Where appropriate, the slots in the node structure refer to stages (1) to (5) above.

```
(defstruct (NODE)

  (CONSISTS-OF nil)        ;Children. THE TASK HIERARCHY FROM STAGES 1 AND 5.
  (LINKS-BY nil)                      ;IN/DEPEND children linkage.
  (PARENT nil)                        ;Parent. Computed for efficiency.



  ;;; Where the learner will navigate to on a leaf node if the corresponding
  ;;; syntax token is invoked.  Only of interest for leaf nodes, i.e.
  ;;; (NODE-CONSISTS-OF nil).  If LEADS-TO is nil for a leaf node this
  ;;; signifies quitting the application :-
  (LEADS-TO nil)           ; THE "CONTROL FLOW" INFORMATION FROM STAGE 4.


  ;;; DUMMY node - typically PROCEED-n - which is not taught;
  ;;; Could well be a node generated by splitting up a general
  ;;; subtree into component binary and/or trees. Default nil :-
  (DUMMY nil)


  ;;; SYNTAX-CONSTRUCT required to invoke each node :-
  (SYN-TOKEN nil)          ; THE COMMAND SYNTAX FROM STAGES 2 AND 3.


  ;;; BARRIER - to prevent a parent node from "seeing" the SYN-TOKENs of
  ;;; subordinate nodes in the domain model.  Useful where e.g. a filename
  ;;; has to be input, some of the characters of which could be "seeable"
  ;;; SYN-TOKENs.
  (BARRIER nil)


  ;;; STATE-CHANGING syntax construct?
  (STATE-CHANGING nil)
```

```
;;; SM slots :-
(LEARNT nil)


;;; Teaching operations for this node :-
(COMMAND-STR nil)


;;; Need to know if this is a "new" node for teaching operations :-
(CURRENTLY-BEING-TAUGHT nil)


;;; Co-ordinates when tree represented on 2-D array :-
(X-CO-ORD nil)
(Y-CO-ORD nil)



    )       ; end of defstruct NODE.
```

```
;      The Task Analysis structure :


(setf DIALLER-TA
      (make-NODE :CONSISTS-OF '(QUIT PROCEED-1) :LINKS-BY 'DEPEND
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\DIALLER-.SS"))))


(setf PROCEED-1
      (make-NODE :CONSISTS-OF '(DIRECT PROCEED-2)
                 :LINKS-BY 'INDEPEND :DUMMY t))


(setf DIRECT
      (make-NODE :CONSISTS-OF '(QUIT-DIRECT DIAL-DIGITS) :LINKS-BY 'DEPEND
                 :SYN-TOKEN "D"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\DIRECT.SS"))))


(setf PROCEED-2
      (make-NODE :CONSISTS-OF '(MEMORY SETUP) :LINKS-BY 'INDEPEND :DUMMY t))


(setf MEMORY
      (make-NODE :CONSISTS-OF '(QUIT-MEMORY TYPE-NUMBER) :LINKS-BY 'DEPEND
                 :SYN-TOKEN "M"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\MEMORY.SS"))))


(setf SETUP
      (make-NODE :CONSISTS-OF '(QUIT-SETUP PROCEED-3) :LINKS-BY 'DEPEND
                 :SYN-TOKEN "S"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\SETUP.SS"))))


(setf QUIT-SETUP
      (make-NODE :CONSISTS-OF '(SAVE ABANDON) :LINKS-BY 'INDEPEND
                 :SYN-TOKEN ESCAPE
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\QUIT-SET.SS"))))
```

```
(setf PROCEED-3
      (make-NODE :CONSISTS-OF '(TELL-PHONE-LIST PROCEED-5)
                 :LINKS-BY 'INDEPEND
                 :DUMMY t))


(setf TELL-PHONE-LIST
      (make-NODE :CONSISTS-OF '(QUIT-PHONE-LIST PROCEED-4) :LINKS-BY 'DEPEND
                 :SYN-TOKEN "F" :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\TELL-PHO.SS"))))


(setf PROCEED-4
      (make-NODE :CONSISTS-OF '(SET-DIRECTORY SET-FILE) :LINKS-BY 'INDEPEND
                 :DUMMY t))


(setf SET-DIRECTORY
      (make-NODE :CONSISTS-OF '(DIRECTORY-NAME) :SYN-TOKEN "D"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\SET-DIRE.SS"))))



(setf SET-FILE
      (make-NODE :CONSISTS-OF '(FILE-NAME) :SYN-TOKEN "F"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\SET-FILE.SS"))))


(setf PROCEED-5
      (make-NODE :CONSISTS-OF '(MODEM-PORT PROCEED-6) :LINKS-BY 'INDEPEND
                 :DUMMY t))
(setf MODEM-PORT
      (make-NODE :CONSISTS-OF '(SET-PORT) :SYN-TOKEN "M"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\MODEM-PO.SS"))))


(setf PROCEED-6
      (make-NODE :CONSISTS-OF '(DIAL-MODE PAUSE-DELAY) :LINKS-BY 'INDEPEND
                 :DUMMY t))


(setf DIAL-MODE
      (make-NODE :CONSISTS-OF '(SET-DEFAULT-D-MODE) :SYN-TOKEN "D"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\DIAL-MOD.SS"))))
```

```
(setf PAUSE-DELAY
      (make-NODE :CONSISTS-OF '(SET-PAUSE) :SYN-TOKEN "P"
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\PAUSE-DE.SS"))))
```

```
;;; Leaves :-
```

```
(setf QUIT (make-NODE
             :LEADS-TO nil    ;which with :CONSISTS-OF also nil means
                              ; it quits the application.
             :SYN-TOKEN ESCAPE
             :COMMAND-STR
             '(("S" "\\LISP\\DIALLER\\SLIDES\\QUIT.SS"))))
```

```
(setf QUIT-DIRECT (make-NODE
                    :LEADS-TO 'DIALLER-TA
                    :SYN-TOKEN ESCAPE
                    :COMMAND-STR
                    '(("S" "\\LISP\\DIALLER\\SLIDES\\QUIT-DIR.SS")
                      ("W" (string-append "D" ESCAPE))
                      ("P" '(DIRECT)))))
```

```
(setf DIAL-DIGITS (make-NODE
                    :LEADS-TO 'DIRECT
                    :SYN-TOKEN #\Newline
                    :COMMAND-STR
                    '(("S" "\\LISP\\DIALLER\\SLIDES\\DIAL-DI1.SS")
                      ("W" (string-append "123 4567"
                                          (string #\Newline)
                                          ESCAPE
                                          ESCAPE))
                      ("P")
```

```
                    ("S" "\\LISP\\DIALLER\\SLIDES\\DIAL-DI2.SS")
                    ("W" (string-append "9@123"
                                        (string #\Newline)
                                        ESCAPE
                                        ESCAPE))
                    ("P")
                    ("S" "\\LISP\\DIALLER\\SLIDES\\DIAL-DI3.SS")
                    ("W" (string-append "9t123"
                                        (string #\Newline)
                                        ESCAPE
                                        ESCAPE))
                    ("P"))))


(setf QUIT-MEMORY (make-NODE
                    :LEADS-TO 'DIALLER-TA
                    :SYN-TOKEN ESCAPE
                    :COMMAND-STR
                    '(("S" "\\LISP\\DIALLER\\SLIDES\\QUIT-MEM.SS"))))


(setf TYPE-NUMBER (make-NODE
                    :LEADS-TO 'MEMORY
                    :SYN-TOKEN #\Newline
                    :COMMAND-STR
                    '(("S" "\\LISP\\DIALLER\\SLIDES\\TYPE-NUM.SS")
                      ("W" (string-append "m3"
                                          (string #\Newline)
                                          ESCAPE
                                          ESCAPE))
                      ("P" '(MEMORY)))))


(setf SAVE (make-NODE
             :LEADS-TO 'DIALLER-TA
             :SYN-TOKEN "S"
             :STATE-CHANGING t        ;;;N.B. The only STATE-CHANGING token.
             :COMMAND-STR
             '(("S" "\\LISP\\DIALLER\\SLIDES\\SAVE.SS")
               ("W" (string-append "S"
                                   ESCAPE
                                   "S"
                                   ESCAPE))
               ("P" '(SETUP)))))

(setf ABANDON (make-NODE
                :LEADS-TO 'DIALLER-TA
                :SYN-TOKEN ESCAPE
                :COMMAND-STR
                '(("S" "\\LISP\\DIALLER\\SLIDES\\ABANDON.SS")
                  ("F"))))            ;N.B. free exploration!
```

```
(setf QUIT-PHONE-LIST
      (make-NODE :LEADS-TO 'SETUP
                 :SYN-TOKEN ESCAPE
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\QUIT-PHO.SS")
                   ("W" (string-append "F"
                                       ESCAPE
                                       ESCAPE
                                       ESCAPE
                                       ESCAPE))
                   ("P" '(TELL-PHONE-LIST)))))

(setf DIRECTORY-NAME (make-NODE
                       :LEADS-TO 'TELL-PHONE-LIST
                       :SYN-TOKEN #\Newline
                       :COMMAND-STR
                       '(("S" "\\LISP\\DIALLER\\SLIDES\\DIRECTO1.SS")
                         ("W" (string-append "S" "F" "D" "C:\\DIALLER"
                                             (string #\Newline)
                                             ESCAPE
                                             ESCAPE
                                             "S"
                                             ESCAPE))
                         ("P" '(SETUP))
                         ("S" "\\LISP\\DIALLER\\SLIDES\\DIRECTO2.SS")
                         ("G" "A:\\TOP\\NEXT"))))

(setf FILE-NAME (make-NODE
                  :LEADS-TO 'TELL-PHONE-LIST
                  :SYN-TOKEN #\Newline
                  :COMMAND-STR
                  '(("S" "\\LISP\\DIALLER\\SLIDES\\FILE-NAM.SS")
                    ("W" (string-append "F" "MEMDIAL1.FIL"
                                        (string #\Newline)
                                        ESCAPE
                                        ESCAPE
                                        "S"
                                        ESCAPE))
                    ("P" '(SET-FILE)))))

(setf SET-PORT (make-NODE
                 :LEADS-TO 'SETUP
                 :SYN-TOKEN '(WILD)   ;Any single character terminates.
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\SET-PORT.SS")
                   ("W" (string-append "M" "1"
                                       ESCAPE
                                       "S"
                                       ESCAPE))
                   ("P" '(MODEM-PORT)))))
```

```
(setf SET-DEFAULT-D-MODE (make-NODE
                          :LEADS-TO 'SETUP
                          :SYN-TOKEN '(WILD) ;Any single character
                                             ; terminates.
                          :COMMAND-STR
                          '(("S" "\\LISP\\DIALLER\\SLIDES\\SET-DEFA.SS")
                            ("W" (string-append "D" "T"
                                                ESCAPE
                                                "S"
                                                ESCAPE))
                            ("P" '(DIAL-MODE)))))
(setf SET-PAUSE (make-NODE
                 :LEADS-TO 'SETUP
                 :SYN-TOKEN '(WILD)     ;2 chars max incl CR.
                 :COMMAND-STR
                 '(("S" "\\LISP\\DIALLER\\SLIDES\\SET-PAUS.SS")
                   ("W" (string-append "P" "10"
                                       ESCAPE
                                       "S"
                                       ESCAPE))
                   ("P" '(PAUSE-DELAY)))))
```

Appendix C

Development of the

ELICITOR tutorial

("ELICITUT")

The objective of this section is to demonstrate the evolution of the full task classification structure for the ELICITOR tutorial. It follows the same stages as were adopted in appendix B for the DIALLER tutorial and there is thus less step-by-step explanation here.

(1) The development of the task hierarchy

A first attempt at a design yields figure C.1.

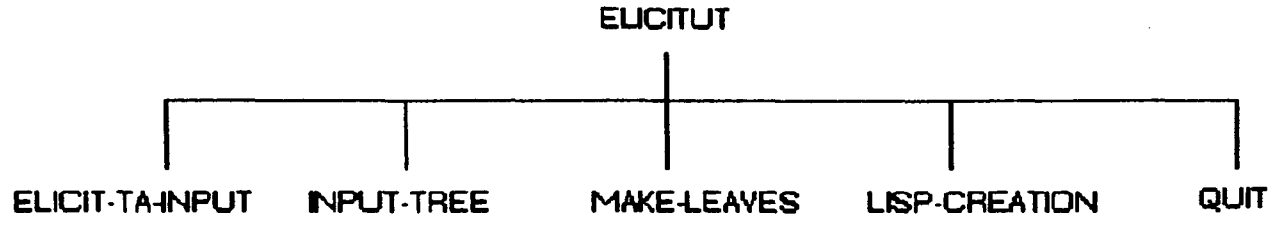


Fig. C.1

This shows that, at the top level, the ELICITOR operations can be thought of as consisting of the following:

ELICIT-TA-INPUT	Obtain application name from user.
INPUT-TREE	User input of task tree.
MAKE-LEAVES	Generate implied leaves of tree where not already specified.
LISP-CREATION	Turn the tree structure into a Lisp structure.
QUIT	

There is little further decomposition in this small application although INPUT-TREE consists of the operation to obtain the tree data (INPUT-TREE-DATA) as shown in figure C.2.

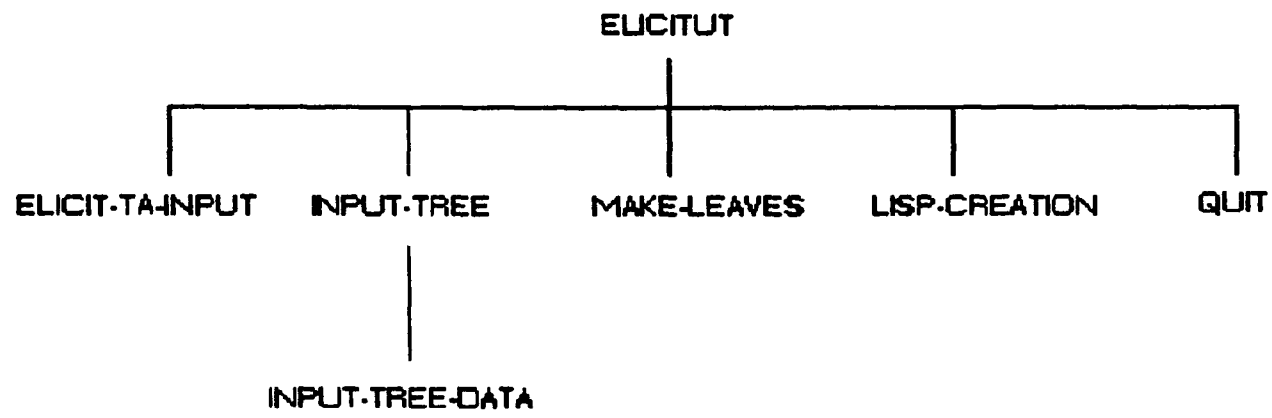


Fig. C.2

(2) Command syntax

Commands are designed for those tasks in the hierarchy which can be invoked by the user, as follows. All the tasks are listed, but only the command operations are assigned syntax at this point. Non-command (i.e. "application") input is dealt with in the next stage.

ELICIT-TA-INPUT	
INPUT-TREE	t|T
INPUT-TREE-DATA	
MAKE-LEAVES	l|L
LISP-CREATION	p|P
QUIT	esc

(3) Non-command input

The terminators for non-command input for the following operations were designated as follows:

ELICIT-TA-INPUT	enter
INPUT-TREE-DATA	enter

(4) Flow of control

The semantic flow-of-control information is straightforward and is shown in figure C.3 as dotted lines. Note, though, that in stage (5) it will have to be slightly altered to point to the top-most *proceed* node.

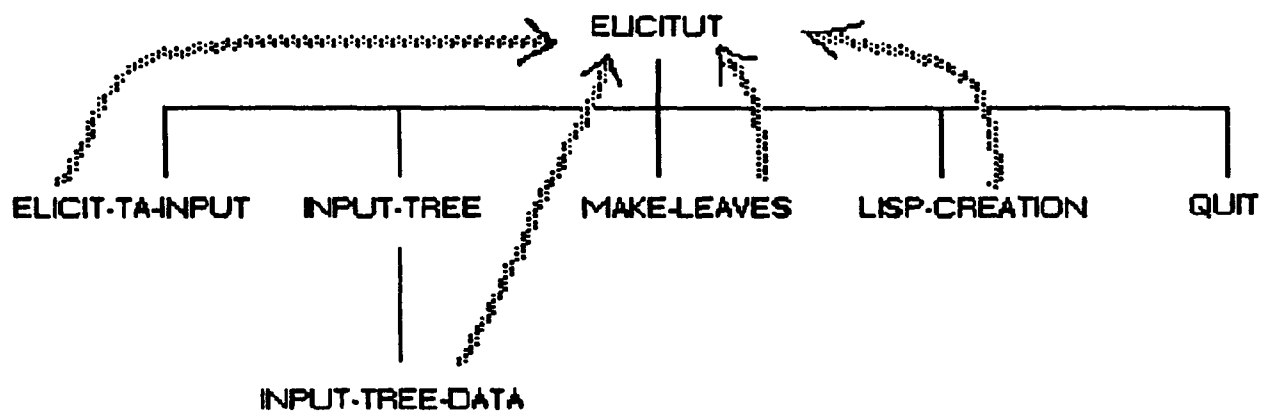


Fig. C.3

(5) Application of heuristic and binary tree transformations

ELICIT-TA-INPUT is not a user-requestable operation but is a component of the root and must therefore be placed at the highest level below it, separate from everything else. Heuristic (a) can be applied here, grouping all the remaining topics together to yield figure C.4. Note that the incorporated linkage is dependent (heuristic f).

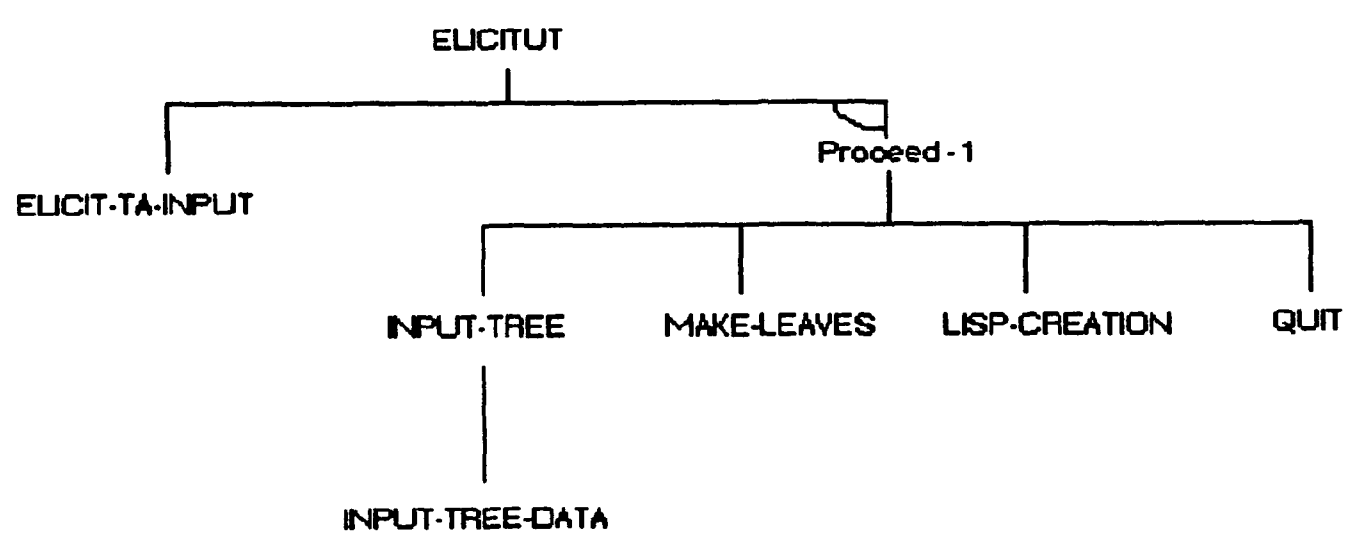


Fig. C.4

Next, apply heuristic (c) ("teach exit-type operations first") to figure C.4 yielding figure C.5.

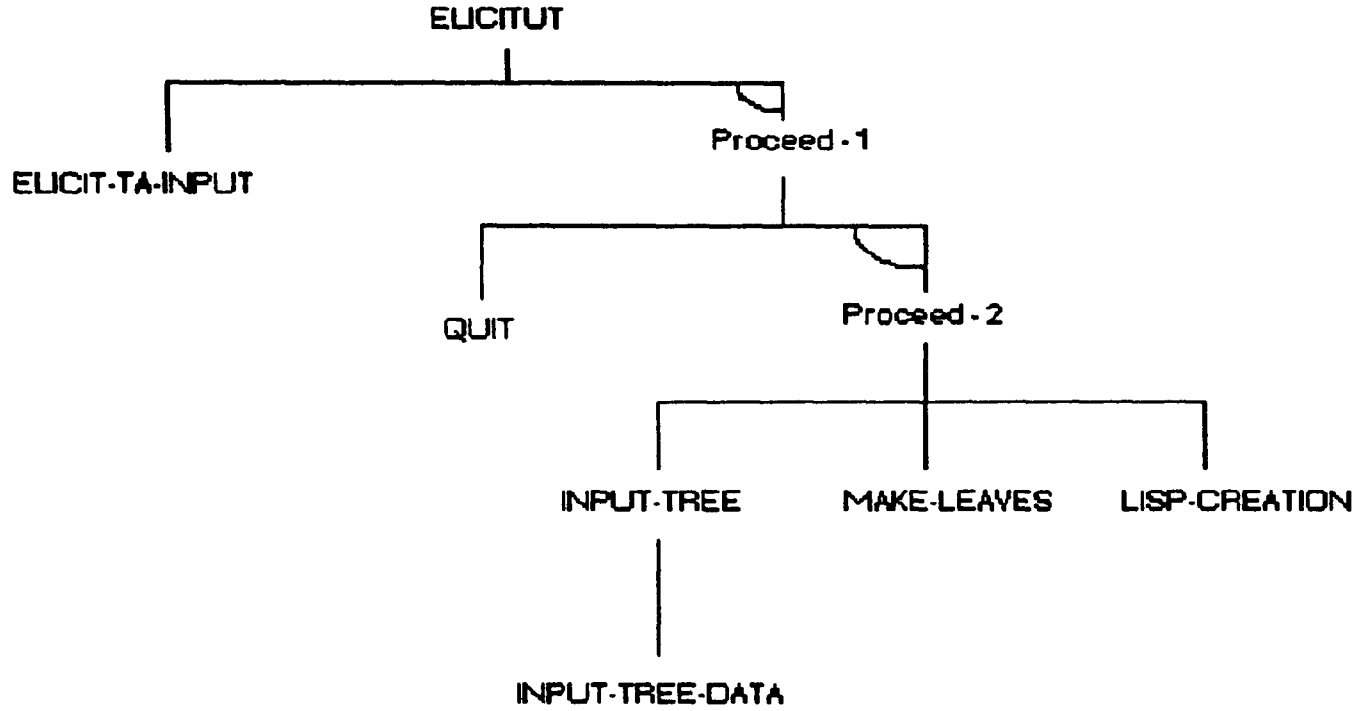


Fig. C.5

Figure C.6 shows the effect of adding dependency information to figure C.5: *LISP-CREATION* depends on *MAKE-LEAVES*, *MAKE-LEAVES* depends on *INPUT-TREE*.

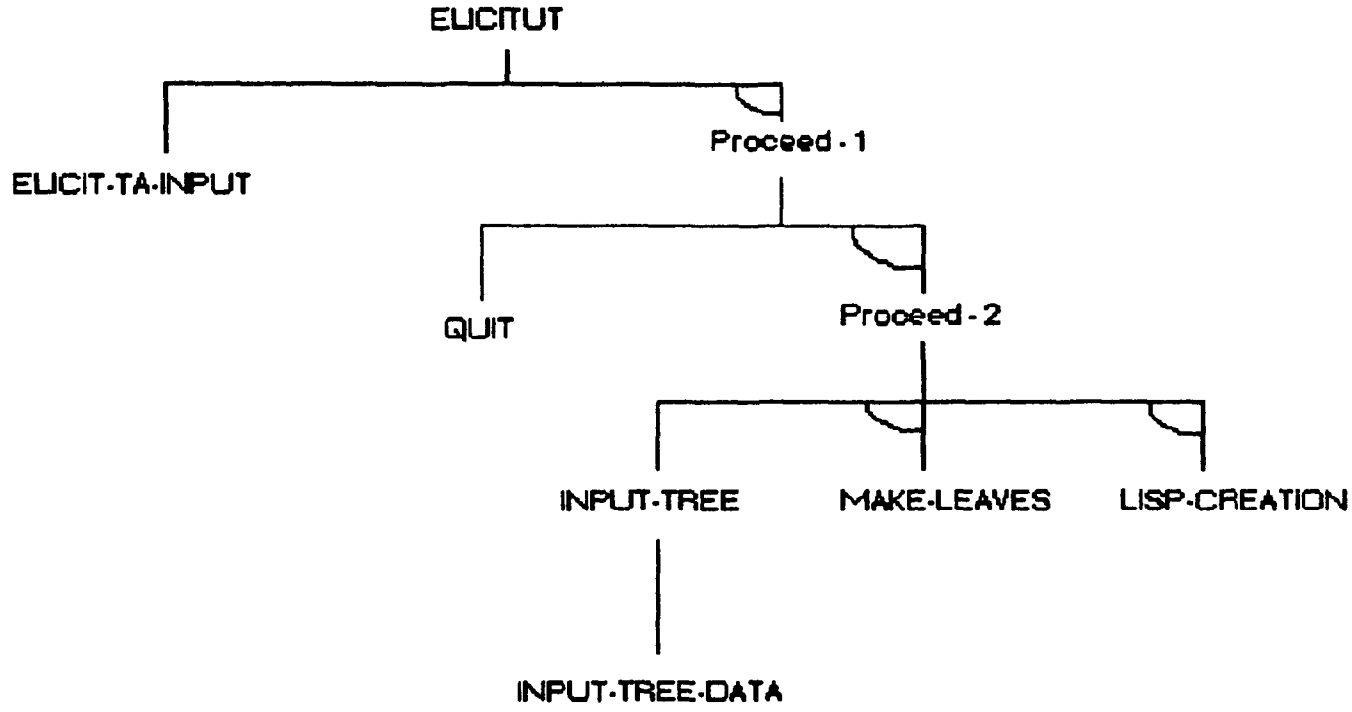


Fig. C.6

Finally, use transformation (7) to represent the children of *Proceed-2* in binary-tree form, as shown in figure C.7.

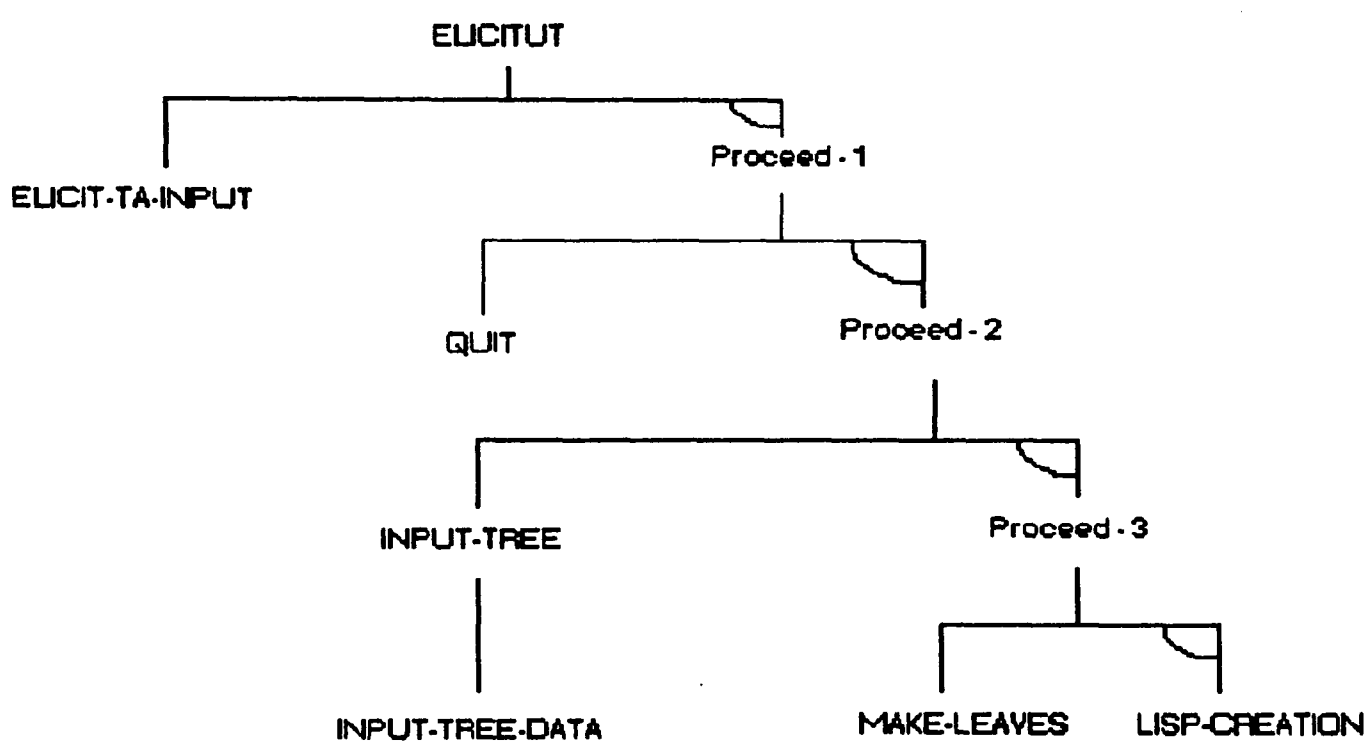


Fig. C.7

(6) Lisp representation of the ELICITUT structure

The Lisp structure of the nodes uses the same representation as that detailed in appendix B.

```
(SETF ELICITUT-TA
      (make-NODE :CONSISTS-OF '(ELICIT-TA-INPUT PROCEED-1) :LINKS-BY 'DEPEND
                 :COMMAND-STR
                 '(("S" "\\LISP\\ELICITUT\\SLIDES\\ELICITU1.SS"))))


(SETF PROCEED-1
      (make-NODE :CONSISTS-OF '(QUIT PROCEED-2) :LINKS-BY 'DEPEND
                 :DUMMY t
                 :BARRIER t))


(SETF PROCEED-2
      (make-NODE :CONSISTS-OF '(INPUT-TREE PROCEED-3) :LINKS-BY 'DEPEND
                 :DUMMY t))
```

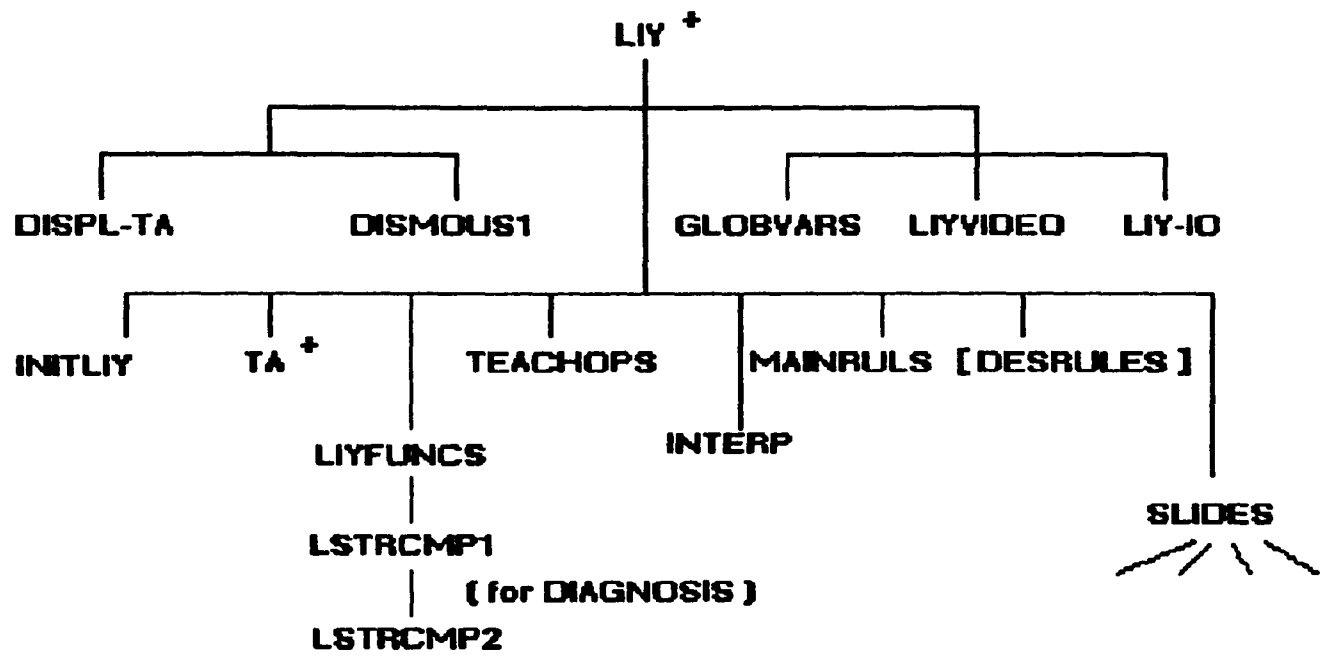
```
(SETF INPUT-TREE
      (make-NODE :CONSISTS-OF '(INPUT-TREE-DATA) :SYN-TOKEN "T"
                 :COMMAND-STR
                 '(("S" "\\LISP\\ELICITUT\\SLIDES\\INPUT-TR.SS")
                   ("D")
                   ("W" (string-append "T" ESCAPE "Y" ESCAPE))
                   ("P" (string-append "LEARN" (string #\NewLine)))
                   )))


(SETF PROCEED-3
      (make-NODE :CONSISTS-OF '(MAKE-LEAVES LISP-CREATION)
                 :LINKS-BY 'DEPEND
                 :DUMMY t))
```

```
;;;Leaves :-

(SETF ELICIT-TA-INPUT
      (make-NODE :LEADS-TO 'PROCEED-1
                 :SYN-TOKEN #\NewLine
                 :COMMAND-STR
                 '(("S" "\\LISP\\ELICITUT\\SLIDES\\ELICITU2.SS")
                   ("D")
                   ("W" (string-append "LEARN"
                                       (string #\Newline)
                                       ESCAPE))
                   ("P")

                   ("S" "\\LISP\\ELICITUT\\SLIDES\\ELICITU3.SS")
                   ("D")
                   ("R")
                   ("W" (string-append "LEARN"
                                       (string #\Newline)
                                       "YR" ESCAPE))
                   ("P")
                   )))

 (SETF QUIT (make-NODE :LEADS-TO nil
                       :SYN-TOKEN ESCAPE
                       :COMMAND-STR
                       '(("S" "\\LISP\\ELICITUT\\SLIDES\\QUIT.SS"))))

(SETF INPUT-TREE-DATA
      (make-NODE :LEADS-TO 'PROCEED-1
                 :SYN-TOKEN ESCAPE
                 :COMMAND-STR
                 '(("S" "\\LISP\\ELICITUT\\SLIDES\\INP-T-D1.SS")
                   ("D")
                   ("W" (string-append "PART1" (string #\NewLine)
                                       "PART2" (string #\NewLine)
                                       "Y" ESCAPE))
                   ("P" (string-append "LEARN" (string #\NewLine) "T"))

                   ("S" "\\LISP\\ELICITUT\\SLIDES\\INP-T-D2.SS")
                   ("D")
                   ("R")
                   ("W" (string-append "CT" ESCAPE))
                   ("P" '((string-append
                            "LEARN" (string #\NewLine) "Y"
                            )
                          PROCEED-1))

                   ("S" "\\LISP\\ELICITUT\\SLIDES\\INP-T-D3.SS")
                   ("D")
                   ("R")
                   ("P" (string-append "LEARN" (string #\NewLine) "YCT"))
                   ("S" "\\LISP\\ELICITUT\\SLIDES\\INP-T-D4.SS")   )))
```

```
(SETF MAKE-LEAVES
      (make-NODE :LEADS-TO 'PROCEED-1
                 :SYN-TOKEN "L"
                 :STATE-CHANGING t
                 :COMMAND-STR
                 '(("S" "\\LISP\\ELICITUT\\SLIDES\\MAKE-LEA.SS")
                   ("D")
                   ("W" (string-append ESCAPE
                                       "LY "
                                       ESCAPE))
                   ("P" '((string-append
                             "LEARN" (string #\NewLine)
                             "T"
                             "LEARN-TA-INPUT" (string #\NewLine)
                             "PROCEED-1" (string #\NewLine) "Y"
                             "PROCEED-1" (string #\NewLine)
                             "QUIT" (string #\NewLine)
                             "MAKE-LEAVES" (string #\NewLine) "Y"
                             )
                           INPUT-TREE-DATA))   )))

(SETF LISP-CREATION
      (make-NODE :LEADS-TO 'PROCEED-1
                 :SYN-TOKEN "P"
                 :STATE-CHANGING t
                 :COMMAND-STR
                 '(("S" "\\LISP\\ELICITUT\\SLIDES\\LISP-CRE.SS")
                   ("D")
                   ("E" (string-append
                           "LEARN" (string #\NewLine)
                           "T"
                           "LEARN-TA-INPUT" (string #\NewLine)
                           "PROCEED-1" (string #\NewLine) "Y"
                           "PROCEED-1" (string #\NewLine)
                           "QUIT" (string #\NewLine)
                           "MAKE-LEAVES" (string #\NewLine) "Y"
                           ESCAPE
                           "LY "
                           ESCAPE
                           ))
                   ("W" (string-append "PY " ESCAPE))
                   ("P" (string-append
                          "LEARN" (string #\NewLine)
                          "YC"
                          ))
                   ("S" "\\LISP\\ELICITUT\\SLIDES\\MAKE-TA2.SS")
                   ("D")
                   ("R")
                   ("X" '(progn
                           (cd "D:\\LISP\\LEARN")
                           (MAKE2TUT)   ))      )))
```

Appendix D - Program file dependencies

Tutorial delivery system :-



+

LIY	CONTAINS APPLICATION-DEPENDENT PATHNAMES.
TA	CONTAINS ALL OTHER APPLICATION-DEPENDENT INFO.

For the delivery system :-

GLOBVARS	is the file of global variables.
INITLIY	initialises vars. in rules, SM etc.
MAINRULS	contains the LIY control rules.
TEACHOPS	is the set of teaching operations.
LSTRCMP1	is part of the diagnosis routine to compare the learner's and the correct response string.
LSTRCMP2	is the rest of it.
INTERP	is the rule interpreter.
LIYVIDEO	is code to control the video for LIY.
LIY-IO	is replacement I/O routines so that the learner can be planted inside an application.
LIYFUNCS	is the remaining LIY function definitions.
DISPL-TA	is code to display the TA for mouseing.
DISMOUS1	is the mouseing code.

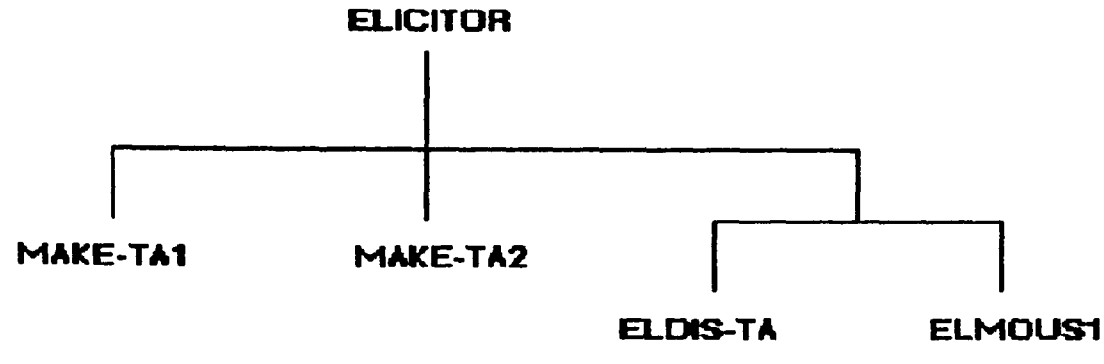
For PARTICULAR applications :-

LIY	is the opening routine which loads the other files and starts the rule interpreter on the top-level rules.
TA	is the task classification structure for a particular application, and its associated functions. IT CONTAINS ALL REMAINING APPLICATION-DEPENDENT INFO.
DESRULES	is the "designer ruleset" for (this) application. It contains application-dependent pathnames.

Not shown :-

LOADLIY	is code simply to load up a tutorial and its application.
TA-SKEL	is a skeleton TA, copied into an application directory, and enhanced by MAKE-TA1.
SENTINEL	is copied into a directory made for an application by MAKE-TA1 so Lisp can check for the directory's existence.
SLIDES	is a sub-directory comprising text files of tutorial display material.

Tutorial authoring system :-



For the ELICITOR authoring system :-

MAKE-TA1 is code to allow the designer to make an "empty" version of TA - *"consists-of"* only.

MAKE-TA2 is code to permit the designer to enhance nodes in the output TA from MAKE-TA1.

ELDIS-TA and **ELMOUS1** are adapted versions of the "mouse" files to allow the designer to mouse with MAKE-TA2.

Appendix E - Program listings

There are two microfiches: