A rule-based system for computer-aided learning in PC-MATLAB

Herb Gade

B.A.Sc., (University of Windsor), 1987

Department of Electrical Engineering
McGill University
Montréal
September, 1992

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Engineering

© Herb Gade, 1992
Abstract

This thesis presents the design and implementation of a rule-based system used for the design and presentation of problem-solving Computer Aided-Learning (CAL) programs. The system is called Tutorial Assistant, acronymed TA, and operates within PC-MATLAB. TA was developed at the McGill University / University of Zimbabwe Engineering CAL Project and was an attempt to create an Intelligent Computer-Aided Instruction (ICAI) system within PC-MATLAB. After an overview of CAL, the CAL project, and ICAI, several ICAI systems are examined. The implementation of TA and a description of its components follow. Two illustrative example sessions using TA, an evaluation of TA, and an outline for future extensions are presented.
Sommaire

Cette dissertation présente la conception et la réalisation d'un système à base-de-règles servant à conceptualiser et présenter les programmes résolution de problèmes d'Enseignement Assisté par Ordinateur (EAO). Ce système est appelé Tutorial Assistant, reconnu sous l'acronyme TA. TA a été développé dans le cadre d'un projet conjoint d'ingénierie de EAO impliquant l'université McGill et l'université de Zimbabwe. TA est une réalisation d'un système d'Enseignement Assisté par Ordinateur Intelligent (EAOI) qui fonctionne dans l'environnement de PC-MATLAB. Après une revue des projets EAO et EAOI, quelques systèmes EAOI sont étudiés. La mise en application du système TA et une description de ses composantes sont discutés. Enfin, deux sessions comme exemples illustrant l'utilisation de TA, une évaluation de TA, et un aperçu pour des extensions futures sont présentées.
Acknowledgements

At the conclusion of this enjoyable and exciting process, I would like to give special thanks to my thesis supervisor Alfred Malowany, and my supervisors at the UZ CAL Project, Robert Appiah and Gerald Ratzer. They not only provided me with the necessary guidance but also allowed me to take advantage of opportunities that made my studies an adventure in many respects. An acknowledgement is extended to the staff at both the UZCAL Project and McRCIM for maintaining the facilities that allowed me to complete my work. The financial support for the UZ CAL Project by both CIDA and UNDP is gratefully acknowledged. I would also like to thank my parents for their support during my graduate studies.
# Table of Contents

## Chapter 1 Introduction

1.1 Computer-Assisted Learning (CAL) ................................. 2
1.2 The University of Zimbabwe Engineering CAL Project ............. 4
1.3 Intelligent Computer-Aided Instruction (ICAI) ..................... 5
     1.3.1 Components of an ICAI System ............................ 7
1.4 Overview of Thesis .................................................. 13

## Chapter 2 ICAI Systems .................................................

2.1 MYCIN, GUIDON, NEO MYCIN, and GUIDON2 ....................... 15
2.2 The SOPHIE Systems ................................................ 19
2.3 The ACT* Systems .................................................. 22
2.4 Intelligent Curriculum Selection Systems .......................... 25

## Chapter 3 Tutorial Assistant (TA) ...................................

3.1 Worked Examples in Instruction .................................... 29
3.2 Rules in Problem-solving Activity ................................. 30
3.3 Implementation of TA ............................................... 32
     3.3.1 PC-MATLAB ................................................. 33
     3.3.2 The Components of TA .................................... 34
     3.3.3 The use of Petri Net Graphs ............................... 46

## Chapter 4 Examples, Evaluation, and Future Development ...........

4.1 Tutorial Session Examples ......................................... 50
     4.1.1 Example 1: Drawing a Root Locus Diagram ................. 51
4.1.2 Example 2: Calculating the Gain and Phase Margin .......... 58

4.2 Evaluation ......................................................... 61
  4.2.1 Expert Module ............................................. 65
  4.2.2 Tutoring Module .......................................... 67
  4.2.3 Diagnostic Module ........................................ 68
  4.2.4 Instructional Environment .............................. 69
  4.2.5 Human-Computer Interface .............................. 69

4.3 Future Development with MATLAB .......................... 69
  4.3.1 An Extension to TA ...................................... 70
  4.3.2 The Use of Static C Variables ......................... 74

Chapter 5 Conclusion .............................................. 76

References ............................................................. 77
List of Figures

1.1 Conceptual framework of an ITS ........................................... 7

3.1 Main screen of TA ................................................................. 35
3.2 DOS, PC-MATLAB, TA and memory requirements ......................... 36
3.3 Screen displays from a fact ..................................................... 39
3.4 Rule syntax .............................................................................. 40
3.5 Screen displays from a problem-solving rule ............................... 42
3.6 Flow chart of inference engine .................................................. 43
3.7 Display of step listing .............................................................. 45
3.8 Petri net components .............................................................. 46
3.9 Petri net's representation of AND and OR ................................. 48
3.10 Petri net's representation of variable relation ............................ 49

4.1 Petri net graph of solution space for the root-locus problem ........... 52
4.2 Rule listing for the root-locus problem ...................................... 53
4.3 Variable listing for the root-locus problem .................................. 54
4.4 Fact listing for the root-locus problem ....................................... 54
4.5 Screen displays for the root-locus problem .................................. 55
4.6 Screen displays for the root-locus problem (continued) ............... 56
4.7 Screen displays for the root-locus problem (continued) ............... 57
4.8 Petri net graph of solution space for the gain/phase margin problem . 59
4.9 Variable listing for the gain/phase margin problem ....................... 59
4.10 Fact listing for the gain/phase margin problem .......................... 59
4.11 Rule listing for the gain/phase margin problem .......................... 60
4.12 Screen displays for gain/phase margin problem ........................ 62
4.13 Screen displays for gain/phase margin problem (continued) .... 63
4.14 Variable listing with no values given to variables ............... 71
4.15 Screen following the selection of variable gainm .................. 72
4.16 Variable listing with some variables having values ............... 73
4.17 Screen showing selected variable and value ........................ 74
Chapter 1

Introduction

At the University of Zimbabwe there is a shortage of textbooks and other learning resources. This often makes the instructor of a class the only source of knowledge for a particular course. The McGill University / University of Zimbabwe Engineering Computer-Assisted Learning (CAL) Project was created to enhance the engineering education at the University of Zimbabwe (UZ) and is described in [Appiah and Daigle, 1988]. These enhancements were provided through the introduction of an additional knowledge source — the computer.

A section in this chapter will provide an overview of CAL and its use of the computer. The following section will discuss how CAL was implemented at the UZ Engineering Department, and integrated into the Control Systems course. CAL typically requires the use of software instructional programs, often referred to as Computer-Aided Instruction (CAI). CAI was developed for the CAL-based Control Systems course. The CAI provided good results, and was thus followed with an investigation into Intelligent Computer-Aided Instruction (ICAI). A section will provide an overview of ICAI. The development of an ICAI is the focus of this dissertation, and its overview is provided in the final section in this chapter.

1.1 Computer-Assisted Learning (CAL)

In [Burke, 1987], CAL is referred to the "direct use of the computer to provide primary instructional service". The terms "direct" and "primary" reflects a redirected function of the computer in education. Barker in [Barker, 1988] further makes this distinction of CAL as being either "implicit" or "explicit". Implicit CAL is what the computer has been typically used for in education, mainly as a resource tool
that is used in problem-solving activities. Explicit CAL, however, involves the production and presentation of instructional software that emulates some aspect of a conventional instructional process.

Explicit CAL programs, also referred to as Computer-Assisted Instruction (CAI), have been classified into four different methods of instruction in [Romiszowski, 1984]. They are listed below:

**Problem-solving CAI:** CAI can provide the drill and practice required for a student to learn a procedure. The classroom teacher provides the regular curriculum, and the computer supplements the activity with problem-solving exercises and review.

**Tutorial CAI:** CAI can also attempt to provide as much of the instruction as possible, rather than be a supplement to the regular curriculum. This CAI provides a complex network of instructional pathways through the material. The path followed by each student is unique to the individual and based on the individual's needs.

**Dialogue CAI:** This CAI is characterised by the type of interactions between the computer and the student. The interactions involve both the student and the computer in posing questions and giving answers in a problem situation. The objective of the CAI is to help the student learn through this interactive problem solving.

**Simulation CAI:** This CAI simulates the behaviour of a system and allows the student to interact with it. These simulations provide an environment that would be otherwise be unavailable. Its primarily up to the student to investigate and learn through discovery the behaviour of the system.
1.2 The University of Zimbabwe Engineering CAL Project

The UZ CAL Project established facilities that include:

- a student learning resource called the CAL Lab with approximately 25 IBM-PCs and two printers configured as workstations in a local area network using an IBM-AT fileserver running Novell Advanced Netware/286;
- a desktop publishing facility that includes an IBM-XT/286, a Macintosh Plus and two laser printers;
- a laboratory of three IBM-ATs and printers to encourage academic staff to develop CAI.

There is of course some question as to how much and what kind of the instruction can be effectively redirected from the conventional medium and presented via the computer as CAI. The strategy adopted at the UZ Engineering CAL Project utilised the facilities to define the role of three additional knowledge sources. These were subsequently incorporated into the Control Systems course, and are described below:

**Text Book** With the desktop publishing facility the project produced and made available for the first time a class text book [Appiah and Barker, 1989] for the Control Systems course. This required the examination and restructuring of the lecture material so that a coherent, structured relationship between the material in the text book and the items in the explicit CAL could be established.

**Explicit CAL** Two CAI philosophies were identified as an effective complement to the textbook: simulation and problem-solving CAI. A suite of CAI programs was developed with PC-MATLAB [Mat, 1989] and its Control Systems Toolbox [Laub and Little, 1986].
1. Introduction

Implicit CAL The use of PC-MATLAB in explicit CAL, compliments nicely with its use for implicit CAL. PC-MATLAB's designed use as a workbench for numeric computations make it ideal for use by students for assignments and projects.

The CAL-based Control Systems course at the UZ provided good results. An improvement in the students performance was recorded even though the number conventional lectures were reduced by 50%. With the success in the use of PC-MATLAB for CAI development, an investigation began into the development of an Intelligent Computer-Aided Instruction (ICAI) system within PC-MATLAB.

1.3 Intelligent Computer-Aided Instruction (ICAI)

The effective use of CAI usually requires the supplementing other sources of knowledge (e.g., lecturer and text book). These sources of knowledge provide the pedagogical elements that can not be provided through CAI. Researchers have been trying to expand the pedagogical role of CAI. Simulation CAI can often provide a high fidelity when simulating the interaction between a system and the examining student. However, it is simulating the activity of the teacher in an interaction with the student that provides CAI research with its greatest barrier for CAI's pedagogical expansion.

Noting this obstacle, CAI researches began to employ techniques found in Artificial Intelligence (AI). Carbonell was one of the first with his program SCHOLAR [Carbonell, 1970]. SCHOLAR used a semantic net of objects or concepts of knowledge of South American geography to diagnose the underlying misconceptions of the student, and then force the student to discover these errors by posing related problems (i.e., a Socratic method of tutoring). This representation of the knowledge as a semantic net provided a reactive mixed-initiative dialogue between the student and a virtual tutor. Instructional programs that use AI techniques have
1. Introduction

...since become known as Intelligent Computer-Aided Instruction (ICAI).

Because of the learning efficiency found in the studies of the one-to-one interaction between the student and tutor, such as that reported in [Bloom, 1984], the 'tutoring' paradigm became the dominant one for ICAI researchers to emulate. For this reason ICAI systems have also become known as Intelligent Tutoring Systems (ITS).

Wenger in [Wenger, 1987] admits that in order to take advantage of the capabilities a computer may provide, an ITS may not at all resemble the characteristics of human tutors. The analogy that he prefers instead, is that of an "active" book, able to intelligently communicate the required knowledge to the student. Wenger refers to the issues applied to ITS as the issues concerned in "knowledge communication"; where knowledge communication is defined as "the ability to cause and/or support the acquisition of one's knowledge by someone else, via a restricted set of communication operations". He contends that for a system to be able to intelligently communicate its knowledge, it requires that the knowledge (i.e., domain and pedagogical) be unassembled, explicitly represented and programmed into the ITS in a "raw" state. This is a shift from the CAI development methods where the knowledge is programmed at a level where it captures only the "decisions" of the domain and pedagogical knowledge. It is with this "raw" knowledge, Wenger explains, that an autonomous ITS can compose the interstructural dialogue dynamically. The decisions, instead of being programmed into the system, are constructed by referencing the knowledge, much in the same fashion as a human expert does. Wenger admits that ITS research is far from the ideal goal of an autonomous ITS.

In [Rickel, 1989] the author reviews current ICAI systems and states that ICAI is "still very much a research topic, with today's systems suffering the same basic limitations as more general AI systems (i.e., common sense reasoning)". These limitations have resulted in very few ICAI systems being available commercially.
1. Introduction

The participants in an ideal student-tutor interaction form the components in an ITS: the expert, the tutor, and the student. The expert has the expertise of the subject domain that is being taught, this knowledge is contained in an expert module. The pedagogical expertise of the tutor is contained in a tutoring module. In an ideal student-tutor interaction, the student makes explicit the knowledge he is missing and therefore requires. In the real world this is not always the case, and therefore in an ITS, a diagnostic module is used to find what knowledge is missing (or what knowledge is found) in the student. Other components reflect the fact that the ITS is defined within a computer: the instructional environment and the human-computer interface. Figure 1.1 provides a conceptual framework of an ITS as illustrated in [Burns and Capps, 1988].

The boundaries between the components of an ITS are not always sharp and
most researchers usually concentrate on some of these components at the expense of others [Wenger, 1987]. In [Burns and Capps, 1988], the authors concede that it is not easy to integrate all of these components into an ITS, as well as addressing the implementation and the evaluations issues. The next following sections will further examine each component.

Expert Module

The expert module contains the domain knowledge that is taught to the student. Anderson in [Anderson, 1988] considers the expert module as the “backbone” of any ITS. He categorises three approaches used to codify the domain knowledge:

Black Box Model The form of this domain knowledge does not resemble the form of human knowledge even though the result of applying this knowledge may be the same (e.g., the human symbol manipulation compared to the computer’s mathematical equation solving process).

Glass Box Model This approach uses methods and structures found in expert systems and their development process. The methodology of building an expert system typically involves a knowledge engineer and a domain expert. The knowledge engineer works with the domain expert in making explicit and then encoding the underlying knowledge in the domain expert’s reasoning.

Cognitive Model This approach goes beyond the glass box method of explicit knowledge representation, by making explicit and then simulating the processes that occur when humans compile and use the knowledge. Anderson argues that it is this form of the expert module that is necessary in producing an ITS that can most easily and effectively communicate the required knowledge.
1. Introduction

Diagnostic Module

The diagnostic module infers a model of the student’s current understanding of the domain knowledge. This model, known as the student model, is often provided as input to the other modules of the ITS. Three common uses for the student model as described in [VahnLehn, 1988] are listed below:

Advancement When the student model is used to represent the student’s level of mastery, the student model is examined to see if he has achieved the competency to move on to the next more advanced topic.

Offering unsolicited advice The ITS can determine if advice to the student is required by examining the state of student model.

Problem generation The ITS can provide dynamic problem generation by examining the student model and present a problem just beyond his present capabilities.

Adapting explanation The ITS can determine from the student model what concepts the student already knows and provides adequate explanation with these concepts.

In order to classify the possible types of student models, a classification has been made along three dimensions by [VahnLehn, 1988], and they are described below:

Bandwidth This dimension determines the quality of the input from the student. The highest bandwidth is possible if the ITS has access to the student’s mental states. The lowest bandwidth is available when the ITS only has access to the final solution of the problem.

Target knowledge type A student model can be interpreted and then applied to the problem of predicting a student’s answers. The interpretation process can either be for declarative knowledge or procedural knowledge.
**1. Introduction**

**Student and expert difference** The student model can sometimes be considered either a superset or a subset of the expert model. The missing concepts can be considered as the items that the expert model has but the student model does not. This student model is represented by a subset of the expert model and is known as an *overlay model*. Alternately, the student’s misconceptions and extra concepts are the items that the student model has and the expert model does not.

Diagnosis is the process of forming and updating the student model by analyzing the data made available to the system. This task sometimes resembles automated theory formation (where a theory is built to explain some data) or automatic programming (where a program is constructed in order to explain some behaviour); both of which are hard AI problems [Wenger, 1987]. Some of the diagnostic techniques that have been used in the past are described in [VahnLehn, 1988].

**Tutoring Module**

The tutoring module contains the pedagogical knowledge. The tutoring module selects and sequences the material to be presented to the student (i.e., the curriculum), and presents the actual material to the student (i.e., the instruction). Halff in [Halff, 1988] provides a review of the approaches typically used for curriculum and instruction in an ITS, and this section provides a summary.

Two types of tutors exist: the expository tutor and procedural tutor. The expository tutor attempts to teach factual knowledge and inferential skills; while the procedural tutor teaches procedures and the skills required to apply these procedures.

Curriculum selection for an expository tutor requires topic selection. The approach typically used is called *web teaching*, where the expository tutor teaches concepts that are closely related to existing knowledge, and then the generalities of
a topic are discussed before the specifics. Procedural tutors usually teach through example and exercise. They typically attempt to select and sequence the examples and exercises according to the following guidelines: every exercise should be solvable and every example should be comprehensible, the sequences of examples and exercises should reflect the structure of the procedure being taught, and the exercises should be selected to fit the needs of the individual.

The knowledge required for intelligent presentation requires that the system be able to decide: how the tutor should present the subject matter, how it should answer questions from the student, and when it should intervene and provide help. The presentation method used usually depends on the subject matter and the instructional objectives. Dialogues are often used to address factual objectives of the ITS. The use of worked examples and guided practice is often used to explain the procedures that the student must learn.

Instructional Environment

The instructional environment comprises the set of components in the ITS that specifies or supports the activities of the student to facilitate learning. Burton in [Burton, 1988] refers to these components as those that do not not require intelligence, and therefore do not belong with the other modules. The activities these components specify or support include those that: provide help, provide game situations, and use learning tools. The issues around the development of the instructional environment are reflective of the material being taught and the methods used. Burton in [Burton, 1988] provides several dimensions along which instructional environments can differ. Included in these dimensions are: what features of the real world are represented and at what level of abstraction, and what features of the real world are simulated and at what level of fidelity.

The instructional environment in present ITSs typically attempt to exploit the graphical and computation capabilities of the computer. As technology evolves, the
1. Introduction

Capabilities of these and other computer-based components (e.g. optical disks and speech processors) will provide a greater number possibilities in how to implement the activities into an ITS. This will then materialise into a wider selection in the levels of abstraction and levels of fidelity of the real world features to be included in the ITS. At present, instructional environments that attempt to combine the computer-based components that are available into an instructional environment are known as multimedia systems. A multimedia system such as the system described in [Midoro et al., 1991], provides many communication channels, and the storage capabilities for audio, video, text, graphics and computer data on optical disks. With this system, Midoro et al. propose to use AI techniques and create what they call a “multimedia database of learning material”. They envision a navigable instructional system that contains the entire structure and material of a subject domain.

Human-Computer Interface

A key component found in an ITS is the human-computer interface. Its importance lies in the fact that even an ITS with complete knowledge is ineffectual unless it is able to communicate that knowledge to the student. The interface becomes an AI problem when there is knowledge required to translate the interaction between the language of the computer and the language of the student. These issues are found in the study of natural-language processing (NLP). However as reported in [Anderson, 1988], NLP has become the “Achilles heel” of ITS. Such systems typically attempt to teach factual knowledge, and Anderson argues that this difficulty with NLP is the principle reason why there are not many declarative tutors. Singer in [Singer, 1990] however argues that the “graphical interfaces can offer users more effective means of communicating their intentions to the system than is possible with natural language.” He further explains that constructing a natural language interface is difficult and a graphical user interface provides the best alternative. To prove his argument he provides a demonstration of a graphical user interface that
is a substitute to the natural language interface found in an ITS called SOPHIE.

Miller in [Miller, 1988] categorises interface design in two types and then relates them to their use in ITSs. One of the types is the kind that use the graphical user interface. He terms these as “first-person” interfaces because they allow the user to acquire a feeling of controlling the domain by manipulating the representing objects (e.g., icons). The second type he calls the “second-person” interface because the commands are given to a “computerised intermediary” which then carries out the desired actions. In this category he includes command languages, menus, and natural language interfaces. Miller confesses that designing an interface in an ITS is especially critical because the student is already interacting with a subject domain that he may have difficulty with, and a confusing or poorly designed interface will compound that difficulty. A designer’s goal of an interface that is transparent to the user is especially relevant in ITS design.

1.4 Overview of Thesis

The author was fortunate to work two years at the McGill University / University of Zimbabwe Engineering CAL Project. One of the projects worked on, as well as the topic of this dissertation, was the attempt to develop an ICAI program within PC-MATLAB. This dissertation will discuss that work.

Chapter 2 will examine some ICAI systems that have been developed in the past. This section will provide a look at some of the issues that have faced developers of ICAI systems, and examine the various approaches they have used in their development.

Chapter 3 provides an overview of TA. The initial two sections discuss the use of worked examples in instruction and the use of rules in problem-solving activity. The following section examines the implementation of TA. Since TA operates within PC-MATLAB, an overview of the relevant components of PC-
MATLAB is provided. This is followed by a description of the various components in TA. Petri nets have been used to represent the knowledge base of TA, and this representation is described.

Chapter 4 provides two example sessions of TA. This is followed by an evaluation of TA that concentrates on the components of the ITS described in chapter 1. A discussion on a partly implemented extension of TA and a suggestion for a future implementation using MATLAB is presented before concluding in chapter 5.
This chapter will examine several ICAI systems. The first section examines the development of an ICAI system that was built around the expert knowledge found in an expert system. The ITS design went through various stages of development, which still continues today. With each stage in the design and development process, the developers became more educated in what knowledge is required and how to use that knowledge for their design of an effective ITS. Their representation of the domain knowledge developed from a glass box model to a cognitive model. The second section will examine a similar experience by developers that used a simulation package called SPICE; however, their progression to the cognitive model representation started with a black box model of expertise. The third section will examine two ITSs that were developed around a cognitive model of how humans acquire a skill. This model provided the specifications to follow for their design of the ITSs. The final section will focus on two ITSs that were designed so that they provide the intelligent curriculum selection of an ITS instead of centering around the expert module.

2.1 MYCIN, GUIDON, NEOMYCIN, and GUIDON2

A well-documented approach to the development of an ICAI program was made by Clancey; both [Clancey, 1986] and [Clancey, 1987] provide a general overview of his work. Clancey's initial task was to see if an expert system was a suitable knowledge source for a tutoring system. The expert system used by Clancey was MYCIN [Shortliffe, 1976]. MYCIN is a well-known and effective expert system of about 450 if-then rules with a backward chaining inference engine.
MYCIN’s rules encode the process used in the diagnosis of blood infections and its treatment. MYCIN initially selects the probable disease based on a list of symptoms. This disease forms the hypotheses (i.e., goal) and the inference engine evaluates this goal by matching it with a rule’s conclusion. The matching rule’s precedent is then examined. It contains a logical expression of subgoals. These subgoals either collect information or are themselves conclusions to other rules, which in turn are also evaluated in the same manner. Certainty factors, between -1 and 1, reflect the correctness of the rule and are propagated through the rules using fuzzy logic. This cyclic process continues until no more subgoals need to be evaluated. The resulting certainty factor forms the correctness of the prognosis.

The goals and subgoals decompose the problem into reasoning steps. The rules chain these reasoning steps together. MYCIN allows the user to view its line of reasoning (i.e., goals and subgoals) with a “Why” command. Repeated “Why” commands are answered by moving up the inference stack. This command shows the user how a request for data is related to a goal, how one goal leads to another goal, and how a goal is achieved.

GUIDON as described in [Clancey, 1983] is the tutoring module (containing the pedagogical knowledge) that uses MYCIN’s rules (containing the domain knowledge) to explore the suitability of an expert system’s rules for an ICAI system. A subset of MYCIN’s rules that referred to the diagnosis of infectious meningitis were used with GUIDON for the teaching purpose. GUIDON attempts to teach these rules to the student. GUIDON could itself be considered a expert system, of about 200 tutoring rules.

GUIDON requires that the student select a patient case from a case library. The student then plays the role of the consultant and is required to ask questions of the patient through GUIDON. GUIDON plays the role of expert, criticising the student’s responses by comparing the students actions with those defined by MYCIN’s rules. There is a mixed-initiative dialogue between the student and
GUIDON. The discussion of the dialogue is centered around the goals or subgoals that the student is currently trying to achieve. A model of what the expert knows for a particular case is constructed by using MYCIN prior to the student’s session with GUIDON. This model is an AND/GR tree of indices to the MYCIN’s rule conclusions and records of why certain rules did not apply. The student model forms an overlay of this expert model. This student model provides GUIDON with the information that allows it to infer whether it believes the student knows a MYCIN rule, whether the student can apply the MYCIN rule, and whether the student could provide this rule as supporting evidence to a student’s partial solution. Diagnostic rules update this model as the tutorial session proceeds.

GUIDON satisfied Clancey’s expectations as a tutoring module; however it was MYCIN’s role as the expert module that fell short. Students found the rules difficult to understand, remember, and incorporate into problem-solving activity. Those that could remember them, were unsure of when to apply them. The problem laid in the fact that GUIDON attempted to teach the students by rote the same procedural steps that MYCIN used. GUIDON as explained in [Clancey, 1983] could not “convey an approach or strategy for bringing those steps to mind — the plan that knowledge-base authors were following when they designed MYCIN’s rule set”. The rules were designed for optimal performance, and any strategy that the knowledge-base authors designed into the rules was implicit and compiled into the ordering of the rule’s clauses. This made it impossible for GUIDON to communicate to the student the strategy involved in the diagnostics.

Clancey proposed to make explicit the missing information and make it suitable for teaching diagnostics with a reconfiguration of MYCIN to NEOMYCIN [Clancey and Letsinger, 1984]. MYCIN’s rule set could not communicate the strategy for diagnostics because the knowledge used did not consider the competing hypothesis of other infections in the diagnostic process. Simply adding other diseases to the knowledge base would create competing hypothesis requiring a computationally expensive top down search one infection at a time. Diagnos-
tic experts do not operate in this manner, they typically work in two directions: first backward to find a list of probable diseases, and then forward to evaluate them. GUIDON is unable to communicate this strategy because the organisation and use of MYCIN’s rules was not the same as those of experts. Clancey in [Clancey and Letsinger, 1984] remarks that if GUIDON was to “articulate and recognise the hierarchical organisations of knowledge and search strategies that humans find useful, we need to reorganise MYCIN’s rule set and incorporate an explicit model of diagnostic thinking”.

NEOMYCIN as described in [Clancey and Letsinger, 1984] assumes that there is a fixed set of probable causes. These are kept in the differential which models the experts working (or short-term) memory. The meta-strategy for doing diagnosis consists of a hierarchy of domain independent meta-rules. These rules examine and focus on one hypotheses in differential and collect the datum to form this “hypothesis space” of the differential. There are four different kinds of domain rules that maintain the differential. One of these replaces the general causes in the differential with more specific descendants. NEOMYCIN backward chains through the strategic rules, while the domain rules, that manage the hypothesis space, are forward chained. A more detailed explanation is found in [Clancey and Letsinger, 1984]. Clancey claims that it is with this “psychological” model of the expert, GUIDON is able to teach a student when to consider a specific infectious disease and what competing hypothesis to consider in the same manner a diagnostic expert does.

With the construction of NEOMYCIN further work is being done on a tutoring module with the construction of GUIDON2 as described in [Clancey, 1988]. GUIDON2 will attempt to convey both the NEOMYCIN model of diagnosis as well as a model of learning to the student. The model of learning to be taught is the process of knowledge acquisition as demonstrated by knowledge engineers.

The knowledge engineer paradigm was chosen as the expert for learning because it is the knowledge engineer who infers (through interactions with the expert)
what knowledge is missing and requests it directly from the expert. The tutor on the other hand, interacts with the student so as to infer and then provide the student's missing knowledge from the domain expert. The knowledge engineer model provides a more active role for the student in the learning process. The tutor's role is now redirected to answering questions from the student, probe the student's understanding, and redirect his behaviour towards that as defined by the knowledge engineering model. Just as NEOMYCIN is considered as an expert model for diagnostic reasoning that the student should learn, it is hoped that this learning process known as knowledge engineering will be captured in an expert model for learning and conveyed to the student via the tutoring module of GUIDON2.

2.2 The SOPHIE Systems

SOPHIE (Sophisticated Instructional Environment) went through three phases and five-year period of development. The ICAI systems of SOPHIE use an electronic simulator called SPICE, to teach students how to troubleshoot faulty electronic circuits. The lengthy paper of [Brown et al., 1982] describes the knowledge-engineering techniques and pedagogical aspects of the SOPHIE systems. A chapter in [Wenger, 1987] describes the three phases of development in the SOPHIE systems.

SOPHIE-I was constructed following the analysis of protocol collected from experts troubleshooting the IP-28 regulated power supply. SOPHIE-I contains a natural language interface that provides good linguistic capabilities. The components of the expert module are the SPICE simulation model of the IP-28, a collection of rules referred to as procedural specialists, and a semantic network of factual information about the IP-28.

The student works with two components in SOPHIE-I: the tutor and the laboratory workbench. The workbench and its instruments allows the student to make
measurements on working and faulty models of the IP-28 device. The tutor com-
ments on the usefulness of these measurements toward finding the fault. The tutor
tells the student that the measurement taken is either useful, redundant, or has no
significance. The tutor's comments are based on the output from the procedural
specialists. These rule-based specialists transform a given problem into a set of
subproblems, each of which is then solved by simulating the appropriate model on
the simulator. These intermediate results are analysed and used to infer a solution
to the original problem. Three other tasks can be performed through the tutor.
One is to answer a hypothetical question about the consequence of an action or an
event. Another task is to check whether the student's hypothesis is a well-reasoned
conclusion (i.e., checks if the conclusion is consistent with the set of measurements
taken so far, and therefore removes the occurrence of a "wild guess"). The third
task generates a list of all the possible hypotheses.

SOPHIE-I allowed the student to be very active in the troubleshooting, it how-
ever contained very little pedagogical knowledge. SOPHIE-II was constructed to
overcome this deficiency. SOPHIE-II is an extension of SOPHIE-I that includes a
troubleshooting expert. SOPHIE-II requires that the student inserts an arbitrary
fault. The expert then goes through the processes required to find the fault, explain-
ing its tactics for choosing the measurements and its strategies for attacking the
problem. The expert includes the student in the troubleshooting activity by asking
relevant questions. The knowledge of the troubleshooting expert is contained in
a prestored decision tree. This fixed decision tree however removed much of the
student initiative that was found in SOPHIE-I. Both SOPHIE-I and SOPHIE-II have
been used in an instructional setting.

Brown et al. in [Brown et al., 1982] explain some of the strength and weaknesses
of using a simulation as the expert module. Its strengths lie in the fact that it
contains a large amount of expertise, but its weaknesses appear when attempting
to extract specific information (e.g., causality) in a new context (e.g., a faulted
circuit).
The designers of SOPHIE argue that students can better understand the circuit and remember their understanding "if an explanation reveals some of the underlying causality". Capturing this causality from a quantitative simulation model required the addition of the procedural specialist. These procedural specialists run a recursive series of simulations. Boundary conditions of the components define when a fault occurs, and each simulation propagates a fault and the effects of a fault to the next simulation. The simulations' results are recorded in a database. This data is then used by the tutor to infer the causal relationships. These relationships are then communicated to the student through the tutor's tasks.

SOPHIE-I allowed the student full control over what measurements were taken, and provided an inference facility that allowed the tutor to criticise the student's actions and answer questions. SOPHIE-I however could not provide the reasoning behind the tutor's actions. SOPHIE-II was able to communicate to the student this reasoning, but at the expense of removing much of the student's initiative. SOPHIE-III was an attempt to combine both full student initiative and a tutor that could communicate the knowledge required for trouble-shooting expertise.

The inference mechanisms and representational scheme based around the quantitative simulation model of SOPHIE-I, was redesigned for SOPHIE-III. SOPHIE-III used qualitative inference techniques based on those that students and experts used. With the qualitative reasoning, SOPHIE-III would be able to provide explanations for each deduction, and be able to hypothesis and reason about a student's partial knowledge.

SOPHIE-III contains three experts: an electronic expert, a coach, and a troubleshooter. The electronic expert contains knowledge at the local level and at the circuit level. At the local level, the electronic expert utilises general component knowledge and a local constraint propagator. At the circuit level the electronic expert handles circuit specific knowledge. The electronic expert propagates quantitative measurements at the local component level to qualitative assertions at the
circuit level. The coach's task is to examine the deductions made by the student and determine whether to interrupt and advise the student. The trouble-shooter uses the electronic expert to evaluate and manage a set of hypotheses. The design is complicated and requires the specifications of many interacting models of components. A more detailed description can be found in [Brown et al., 1982]. SOPHIE-III never made it to the instructional setting because the computational power required was too great for practical use. However the SOPHIE-III theme on qualitative models and the mental models that students possess inspired other projects [Wenger, 1987].

2.3 The ACT* Systems

Anderson's ACT* (Adaptive Control of Thought, Star) theory is an attempt to define the cognitive processes involved in the acquisition and development of a skill [Anderson, 1983]. For Anderson, constructing ITSs have allowed him to apply and test this theory.

The theory is quite complex, but there are only a few aspects of the theory that are relevant to ITS, and Anderson et al. provides a summary in [Anderson et al., 1987]. The theory assumes that all cognitive functions can be represented as productions (i.e., rules). The predicate of the rules are patterns that match the information in a working memory. The size of the working memory is bounded in capacity to reflect the size of short term memory of humans. This working memory is a space where rules are collected and compiled to produce new rules. The creation of these new rules is termed knowledge compilation. Two learning techniques are defined with this learning model: proceduralisation and rule composition. Proceduralisation is when a general piece of knowledge (i.e., an uninstantiated rule) is converted into a specific piece of knowledge (i.e., an instantiated rule). Rule composition is a process by which a sequence of rules are combined and collapsed into a single rule. A rule once created is never forgotten and stored in a long term memory of
unlimited size.

From the ACT* theory, Anderson et al. developed a set of principles to be followed in the design of ICAI systems. These principles are found in [Anderson et al., 1987] and are listed below.

1. Represent how a student should behave and not behave with rules.

2. Communicate to the student the hierarchical structure of the goals as defined by the rules.

3. Provide the instruction in the form of the problem-solving activity, so that the instruction will be learned more efficiently.

4. Promote an abstract understanding of the problem-solving knowledge.

5. Minimise the working load on the memory so that only the relevant information of the problem is held for further processing.

6. Provide immediate feedback on errors so that the incorrect rules will not occupy space in the working memory.

7. Adjust the grain size of the instruction so that it is consistent with the knowledge that has already been compiled.

8. Accept a partial solution and assist on the acquisition of the missing knowledge so that knowledge compilation can occur.

Anderson et al. concede that when applying these principles they were at a level of design he calls "human engineering". This deals with the issues of how to effectively communicate the domain knowledge. Anderson et al. in [Anderson et al., 1987] also describe which of these principles they were able to apply in their design of the Geometry Tutor and the Lisp Tutor.

The Geometry Tutor, as described in [Anderson et al., 1985], is a tutor for doing geometry proofs. It consists of a tutor, an interface, and a set of rules defining
the steps required for the construction of the proofs. The rules define the expert module.

The interface communicates to the student the logical structure of the proof and the process of the problem-solving activity. This is achieved by displaying an inference net of the steps the student takes. What is given in the problem is located on the bottom of the screen and the required proof is placed at the top of the screen.

The tutor examines each of the student's steps, matches it to a rule in the expert module, and provides feedback. Feedback is given by examining the rating of the matched rule. These ratings are predefined for each rule and measure the appropriateness of applying that rule given the state of the proof. This process of the student following the expert is referred to as model tracing.

The rules were designed so that they would generate proofs that were in a "natural, human-like way". In total there are 300 rules, both ideal (to capture the correct operations) and buggy (to capture the incorrect operations), for both forward and backward inferences. These were obtained following the analysis of the problem domain and the observations of the students and experts doing these proofs.

In [Anderson and Skwarecki, 1986], the authors advocates the use of "precise and realistic models of how students should program and how they actually program" in their design of ITSs to teach introductory programming. The Lisp Tutor is provided as a demonstration of this approach.

Development of these models requires the study of how students learn to program. With this empirical evidence, simulation models of how a student does program and how a student should program were constructed using rules. These rules, a total of 1200, were incorporated into the Lisp Tutor.

The Lisp Tutor's interaction with the student is also defined by the model tracing paradigm found in the Geometry Tutor. The rules consider the proper style, and
therefore also require the code by the student, to be in a top-down and left-to-right manner. Their motivation for this was to make the tutor efficient for practical use. The efficiency is obtained by compiling all of the rule sequences in advance and record them in a data structure, so that they can be used during the tutoring. Wenger states in [Wenger, 1987] that both the Geometry and Lisp tutor have found their way into the classroom and have provided encouraging results.

In [Anderson and Skwarecki, 1986], the authors also describe an ITS called the PUPS Tutoring Architecture (PTA), that will not require the student to enter the code in a restricted order. There two components in PTA: a tutoring engine and a problem solver. The tutoring engine refers to a data structure called a solution trace, to determine whether the student’s actions are correct and what feedback messages to apply. The problem solver generates this solution trace. During its construction all the information on how to respond to the student is attached to the solution trace. This solution trace is generated by rules off-line where real time is not critical. In this manner, the domain and pedagogical rules construct scripts of the many instances of the rules’ application and then enter them into the solution trace. The solution trace is large, however its construction is automatic and once generated, it can be reused with another student. The instruction it provides is efficient for real-time application. Anderson et al. however concedes that this method of instruction generation is a compromise between CAI and ICAI. It is hoped that PTA can be applied to several programming languages.

2.4 Intelligent Curriculum Selection Systems

In [Williams and Reynolds, 1991] the authors describe a ICAI system that uses a knowledge base of productions (i.e., rules) to provide instruction for the proper execution of a task. They argue that conventional instruction is often either incomplete, overwhelming, or incoherent, and because of this, learning by the student becomes inefficient. The student is often required to make sense out of the instruc-
tion, and therefore learn through discovery. They claim that in order to improve the efficiency of the instruction, a detailed cognitive analysis of the to-be-learned task is required. This analysis requires that a production model (i.e., set of rules and facts) be developed from the instructional material. The production model then provides the framework for the content of the instructional material.

The construction of this model requires that a statement describing the to-be-learned task become a conclusion of a rule. This task is then broken down into subtasks, which become the clauses in the precedent of the rule. These subtasks then become conclusions in their own rules and are further broken down to create the precedents of their rules. This cycle continues until all tasks are broken down to a declarative fact. This AND/OR graph of rules forms a hierarchical relationship of tasks required for the successful performance of a specified task.

Attached to each rule and fact in the model is: an explanation of the rule, an exercise that determines if the rule has been learned, and a diagnostic that determines which of the precondition rules have not been learned. An exercise selection heuristic selects an exercise based on past exercise results. It attempts to select an exercise that the student has the most knowledge about. This is achieved by selecting the rule with the most preconditioned tasks (i.e., precedent clauses) already learned. With this approach to curriculum selection the student is taught something that he almost already knows, and the learning is done with minimal difficulty.

The goal of the student is to work through the hierarchy of the exercises until all productions are acquired. This production model when learned by the student, becomes what they call a cognitive simulation model, which resides in the student's production memory.

The validity of the cognitive simulation model as the way humans acquire and organise knowledge is presented with quite a bit of support from cognitive psychology. Williams et al. claim that by designing instruction around this model,
it provides a method of diagnosis that can specify in detail what has and what has not been learned. The output from this diagnosis becomes the input to a teaching strategy which provides individualised instruction according to the individual's needs.

Antao et al. in [Antoa et al., 1992] describes a ICAI system that provides instruction on the use of a simulator. Their reasoning behind the selection of the instruction is similar to that described in [Williams and Reynolds, 1991]. Whereas they use a a network of rules, Antao et al. uses a hypertext network to model the domain knowledge.

The author of the instruction decomposes the process of using the simulator into a series of subprocesses. Descriptions of the subprocesses and tests that determine if a subprocess has been learned by the student are then developed. The descriptions and the tests are assigned to nodes in the hypertext network. Links between the nodes represent relationships between these subprocesses.

An interactive graphics-based browser allows the student to transverse the hypertext network. The nodes are represented by labelled icons, and their links are represented by directed arcs. The interface allows the student to view the entire network representation. The student is required to satisfy the test at each node.

The student is only given access to the nodes to which the system has deemed his current knowledge level as appropriate for the information contained in the node. Once a student has satisfactorily passed the test at a node, other nodes become accessible. A student record of test results is kept by the system from which these inferences are made. The author of the instruction is provided with tools to analyze the students records. This, plus the author's knowledge of the domain, allows the author to define the knowledge required (i.e., the prerequisite nodes accessed) for access to each node.
Chapter 3  

Given the enormous task involved for the development of a truly ‘intelligent’ ICAI system that was outlined in the previous chapters, our aims were more pragmatic. We required a ICAI program that would be workable in a classroom setting, developed within a limited time frame, and function well within the strategy adopted at the UZ CAL Project. With these constraints in mind, a program called Tutorial Assistant (TA) was developed. This chapter will examine TA. TA is also described in [Gade et al., 1992], [Appiah et al., 1991], and [Appiah and Gade, 1990].

TA is far from being a self-contained instructional system as those pursued by the developers of ITSs. It does however provide the framework for the construction and then presentation of individualised lessons on the computer. Because of this lack of ‘intelligence’, TA is simply referred to as a rule-based system that allows an instructor to encode and present a problem-solving tutoring session. These sessions typically involve the instructor stepping the class through problem-solving examples. Similar to these tutoring sessions are the worked examples found in text books, both of which present to the student the procedures required to solve a particular problem.

TA was developed around the expertise found in the numeric algorithms and simulation capabilities of PC-MATLAB. TA is very much reminiscent of systems such as the GUIDON and SOPHIE systems, where the ITS was developed around the expert module. TA can also therefore be considered as the initial exploration into PC-MATLAB’s capabilities as an expert module for an ITS.

The first section of this chapter will examine some issues in problem-solving instruction using worked examples that are relevant to TA. The following section will discuss the most common format used to represent cognitive procedural knowl-
edge: the production rule. The last section will examine the implementation of TA.

3.1 Worked Examples in Instruction

Worked examples have always performed an important function as a precursor to the actual exercise attempts. This section will examine the role of worked examples in instruction.

The practice of solving problems through exercises forms a large part of an engineering course. Prerequisites for the problem-solving exercises typically require the understanding of background information and the study of a few worked examples. Sweller in [Sweller, 1988] argues that this emphasis on problem-solving as a learning device is "based more on tradition than research findings". Sweller's argument is based on a reference to [Larkin et al., 1980], an article that examines the differences in how novices and experts solve physics problems. Novices used a means-end strategy to solve the problem while an expert moves forward immediately by choosing the appropriate operations. The difference in the two approaches is due to the fact that the experts were able to recall a cognitive structure called a schema. The schema allows the expert to recognise each problem and each problem state from previous experience, this in turn allows the expert to infer which moves are required. Sweller argues that the means-end strategy places a heavy cognitive load on the novice, which then interferes with schema acquisition. From these observations, Sweller suggests that a more effective manner in acquiring the problem-solving skills is to review and practice more worked examples of problems prior to attempting the problem exercises.

Sweller's arguments are an extreme case of applying Anderson's principle of minimising cognitive load, where the student is a novice with a level of skill that would justify the minimal cognitive load as found in a worked example. There
are others however, that argue that such extra cognitive activity, and hence the requirement to learn through discovery, is an important element in the learning process. These arguments are echoed by Greeno in [Greeno, 1991]. Greeno provides a summary of the two different roles, which he calls didactic and exploratory, that computers can play in education. In a didactic role, the computer is used to present instruction in a systematic, individualised, and effective manner. The computer used in an exploratory role requires that the student to learn through discovery, and thus the computer provides the student with the opportunity to investigate and interact with some phenomena. He concludes that didactic systems allow a student to follow and learn rules, and the exploratory systems teaches a student how to "think and construct meaning". Greeno believes that the both paradigms have their place in a learning environment.

In conventional education the greatest source of worked examples is found in textbooks. At the UZ textbooks are in short supply and therefore the only alternative source of examples comes from what the instructor provides to the class through tutorial lessons.

3.2 Rules in Problem-solving Activity

A rule allows the representation of one individual packet of information. Three related informations are contained in this packet: the operation the rule applies, the precondition or predefined state for applying the operation, and the goal of the rule as defined by the output from the rule operation. This section will attempt to show the importance of presenting an explicit representation, and that the rule provides a convenient format to represent and present that information.

As with all instruction, the objective is to make explicit and communicate the required knowledge to the student. This becomes more complicated in an ITS in that a representation must be specified that will allow the knowledge to be made
explicit and then communicated via the pedagogical knowledge found in the tutoring module. This representation must also allow comparisons with the student's knowledge via the diagnostic module. One of the most common representations used is the rule.

In the previous chapter, it was mentioned that in [Williams and Reynolds, 1991], the authors provide ample evidence from cognitive psychology literature of how humans process and formulate rules. One of the conclusions that they make is that an instructional system should make explicit the rules of the instruction so that they can be observed by the student. The authors also argued that conventional instruction does not always make explicit all the information required to solve a problem. Uncovering an implicit piece of information from an instruction further contributes to the load on cognition.

Chi et al. in [Chi et al., 1989] accept the theory that for skill acquisition to take place, it requires that the declarative knowledge (e.g., instruction) be compiled into domain-specific procedures via a knowledge compilation process such as that found in Anderson's ACT* theory. Chi et al. focus their research on the declarative encoding of the instruction that takes place prior to the knowledge compilation process. Specifically they examine the completeness of the representation of the knowledge that students acquire as they read from background text and worked examples, and relate this to how well they are able to do the exercises. They propose that a student learns and understands an example via the explanations that they give while studying the example. These self-explanations allow the student to infer and make explicit the conditions of each step in the example. These steps are often not made explicit by the authors of the examples. They found that all students were able to write down the declarative definitions and principles given in the background text, but only those students that had better success at solving the problems were able to produce the explanations while they studied the examples.

Chi et al. provide a possible explanation to why self-explanations help in un-
understanding and problem solving. They believe that the self-explanations consist of the creation of inference rules that are instantiations of the principles and definitions introduced in the background text. One purpose that these inference rules serve is to make explicit and accessible the components of an example that are not explicit and accessible. The inference rules make more clear the specific conditions in which specific actions are to take place. Chi et al. propose that "once these inference rules are constructed, then the process of compilation can take over and these declarative inference rules may be converted to procedures."

3.3 Implementation of TA

Problem-solving in an engineering course such as Control Systems is primarily driven by the variables defined in the problem. Typically, the problem consists of a set of variables whose values are given and a set of variables whose values are unknown, and the problem is to find values for the unknown. The process of solving the problem often requires that it be broken down by representing parts of the solution with other variables. These variables provide the meaningful subgoals that make the problem more comprehensible and its solution more accessible to the student. These variables form a dependency hierarchy, in the sense that the solution of one variable may be required prior to the solution of another variable.

Such a hierarchy defines a solution space. Stepping through the solution space from what is given, through the subgoal variables, to what is required, necessitates that a series of operations be applied. Each operation provides a value to a variable in the solution space. Applying an operation requires that preconditions, which define the dependency hierarchy, must be met (e.g., the existence of values for certain variables, or a specified value for a given variable).

When attempting to convey this solution space to a student via a worked example, meaningful subgoals must be defined that will reduce the complexity of the
solution. Also, information such as the operation to apply to obtain the subgoal, the precondition for operation application, and the resultant of the operation (i.e., subgoal) must be articulated to the student.

As was shown in the previous section, rules provide the format to make explicit and then articulate this information. Rules also allow a convenient method to map the entire solution space of a problem and therefore allow the creation of the many worked examples that, as was shown in 3.1, may be necessary for student acquisition of the solution space and the eventual skill acquisition.

TA requires that the steps required for solving a problem be made explicit with rules. These rules are then made visible and communicated to the student. TA operates much like an expert system, and therefore the presentation of the problem is considered a fact. It is assumed that these problems would be solved in a forward approach and therefore the rules are forward chained by an inference engine. The expertise of the system is contained in the algorithms and simulation capabilities of PC-MATLAB. A solution trace is presented following the completion of the problem to allow the student to review his solution steps.

The following sections will first provide an overview of PC-MATLAB and then discuss the components used in the implementation of TA. It will then further elaborate on the use of TA and describe the various components of TA. Finally we will examine the use of Petri nets in the representation of TA's knowledge base.

3.3.1 PC-MATLAB

MATLAB is considered by many as the de facto standard in computer-aided analysis and design of control systems. The PC version of MATLAB is called PC-MATLAB [Mat, 1989]. At the UZ CAL Lab, PC-MATLAB (v.3.21) is run on IBM-ATs and a Novell Network of approximately 25 IBM-PCs served from a IBM-AT fileserver.
MATLAB's popularity can be attributed to the large collection of functions and algorithms available to the user. These functions and algorithms are either found internally in MATLAB, or are interpreted by MATLAB from text files called M-files. These M-files use a simple programming syntax to call the internal functions or other M-files. The M-files provide a format in which modular and reusable pieces code can be written by the user. Collections of M-files, such as the Control Systems Toolbox [Laub and Little, 1986], can also be purchased.

The M-files and internal functions are executed by having the user enter them onto MATLAB's command-line. An executed M-file or function has access to, and enters variables into a global workspace. MATLAB's screen output is either in graphical plots or user-written text, both of which can be generated from an M-file.

MATLAB also provides a facility to link C or FORTRAN programs into code, called MEX-files, that will execute on the MATLAB command-line in the same manner as a M-file. It was with this facility that TA was developed. TA was written in C and could be successfully compiled and linked to create a MEX-file using either Borland Turbo C (version 2.0) and Borland C++ (version 2.0). Details on how to create a MEX-file is found in [Mat, 1989]. An executed MEX-file can also invoke the internal functions, M-files, or other MEX-files. This facility is used extensively by TA to call the M-files that present the problem-solving tutorial.

3.3.2 The Components of TA

There are two users of TA: the author/instructor and the student. The author defines problem and its solution, and encodes them into TA. The student then uses TA to learn the steps required to solve the problem.

TA requires that the author initially decompose the problem and its solution into meaningful subgoals. These subgoals are represented by MATLAB variables. Steps that acquire the subgoals are then defined, and the author writes a M-file for
3. Tutorial Assistant (TA)

PC-Matlab's Tutorial Assistant (TA) v3.00

Current Variable List is [ var.kb ]
Current Fact Knowledgebase is [ fct.kb ]
Current Rule Knowledgebase is [ rle.kb ]

0 = Load a Variable list
1 = Load a Fact knowledgebase
2 = Load a Rule knowledgebase
3 = Run
4 = Show facts
5 = Show problem-solving rules
6 = Show diagnostic rules for facts
7 = Show diagnostic rules for problem-solving rules
8 = Review Problem Steps
9 = Review Variable List
Q = Quit

Enter Selection:

Figure 3.1: Main screen of TA

each step. The M-files that present the problem to the student are considered the facts of the problem. The M-files that step the student through the problem solution are linked together with rules. Once the author has defined the subgoals, the facts and the rules of the problem, the author writes them into three text files: a variable listing, a fact listing, and a rule listing.

A student starts a problem-solving tutorial by executing PC-MATLAB on the DOS command-line. Once PC-MATLAB is invoked, TA is executed on the MATLAB command line to display the screen shown in figure 3.1. The menu items 0, 1 and 2 on the screen are used to read and load the variables, facts and rules from their respective text files. In figure 3.1 the top part of the screen indicates that TA has already read the files of 'var.kb', 'fct.kb', and 'rle.kb'. Menu selection number 3 of figure 3.1 starts the problem-solving tutorial session. Initially TA enters all the variables that were read from the variable listing into the MATLAB workspace. Once the variables have been entered, TA executes the M-files as defined by the facts. Following the execution of the facts, a forward chaining inference engine evaluates the rules and executes the necessary M-files to step the student through the
Figure 3.2 provides an illustration of the relationships between DOS, PC-MATLAB and TA, and their memory requirements. PC-MATLAB requires a minimum of 320K RAM. The size of PC-MATLAB's workspace is dependent on the amount of RAM installed; with 640K installed the workspace size is about 300K. A variable defined in the workspace will occupy 8 bytes every for every element the variable represents (e.g., a variable whose value is a vector with two characters or two numbers occupies 16 bytes of workspace memory). TA is executed on the PC-MATLAB command-line and occupies 35K of the workspace memory. TA will further occupy more workspace memory after it reads and loads the variable, fact and rule listings.

The next sections will further examine the variables, facts, rules and other components of TA. Samples of a problem solving session in the drawing of a root-locus diagram is used as an illustration.
3. Tutorial Assistant (TA)

Variables

A listing of variables is initially read by TA to define the variables that the TA problem-solving session will use. The variables are then entered and given values in the MATLAB workspace. A sample from the text file is shown below:

"The denominator for the transfer function"
den = [1 2 3]

The first line is a comment that explains the purpose of the variable. The second line is the MATLAB command used to give the variable a value. The value in this case is the vector [1 2 3]. The vector [1 2 3] is MATLAB’s representation of the polynomial $s^2 + 2s + 3$. Even though this variable is entered into the MATLAB workspace, TA’s rules do not recognise that the variable exists (i.e., it is still unknown) until a fact or rule’s conclusion gives the variable a value.

Facts

The facts listing contains a listing of the M-files that are evaluated only once at the onset of the tutorial, in a top to bottom order. These M-files present the problem on the screen and define what is given (i.e., the initial variables and values) of the problem. The author may design the M-files so that they query the student and allow him to define these variables or the author may encode the value of these variables within the M-file. It is these variables that will initially drive the rules of the problem-solving tutorial session.

Below is the fact in the form it is typed into the fact listing. This fact displays the problem statement for the drawing of a root-locus diagram:
"Input the numerator and denominator of the transfer function"
[num,den]=problem(num,den)

The first line is a sentence commenting on the purpose of the fact. Below the comment is the MATLAB command that will be executed. The num and den variables are the numerator and denominator of a transfer function. Default values of these variables, as were initially defined in the variable list, are passed to the M-file problem. Output from the M-file gives these variables their new value. Figure 3.3 illustrates the screen displays that appear when this fact is executed.

Problem-solving Rules

The rule’s precedent is a logical expression of MATLAB commands connected with the conjunction AND and/or the disjunction OR. The syntax for a rule is summarised in figure 3.4. A clause in the precedent is a MATLAB command that uses the variables found in the workspace to return a 0 (false) or 1 (true). The precedent determines if the operation associated with the rule can be applied. The precedent is evaluated in a left-to-right order. Evaluation of the precedent stops once the precedent is known to be false or known to be true. If the precedent is true then the rule is enabled. Following the rule’s activation, the rule is fired and the operation is applied by the execution of the rule’s conclusion. The rule’s conclusion is a MATLAB command that has access to the variables in the workspace and can enter other variables into the workspace. The rule’s conclusion is typically a M-file that should make visible and communicate to the student the prerequisite state for operation application, the operation being applied, and the resultant of the operation.

Typically in a problem-solving session, the operation requires a value be calculated. The prerequisite state in this case requires certain variables be in a proper state (i.e., have a certain value or simply have a value). These variables are then
The root-locus diagram comprises the paths of the poles of the closed-loop transfer function in the s-plane.

As the open-loop gain is varied, the closed-loop poles trace out loci.

Figure 3.3: Screen displays from a fact
usually the input to an equation. This equation defines the operation found in the M-file of the rule’s conclusion. The output from the M-file is the resultant of the operation. The author may write the conclusion’s M-file so that the student calculates and enters the correct value for the variables as part of a guided exercise, or the computations may be provided by MATLAB and displayed as part of a worked example.

An example of a rule as it appears in the text file is shown below:

"Rule 3"
"Since zeros or poles are on the real axis, find and plot loci on real axis"
IF
(AND ~exist('lr') exist('real.x') real.x==1)
THEN
lr=lreal(num,den,poles,zeros)
This rule, as the comment on the second line suggests, finds the loci on the real axis. A previous rule has already determined that there are poles or zeros on the real axis. This has been indicated by the variable \textit{real\_x} value of 1. The M-file \textit{lreal} uses the variables \textit{num}, \textit{den}, \textit{poles}, and \textit{zeros} to plot the loci and return a vector of the loci on the real axis to the variable \textit{lr}. The screen displays following the execution of this rule's conclusion is illustrated in figure 3.5.

As was already mentioned, TA does not consider a variable to exist (even though it is found in the MATLAB workspace) until it has been entered by a fact or rule's conclusion. In this case the conclusion outputs the variable \textit{lr}. The recognition of \textit{lr} by TA keeps the rule from firing again. This occurs because the first clause in the precedent, \texttt{- exist('lr')}, will now return false. To remove \textit{lr} from TA (but not from the MATLAB workspace), a rule's conclusion can execute the MATLAB command \texttt{clear lr}. Once this is done, the first clause in the rule will again return true. Both the \texttt{exist} and \texttt{clear} are MATLAB commands that are not executed within MATLAB, instead they are intercepted by TA where the commands are applied to the variable's data structure within TA.

\textbf{Inference Engine}

A C-coded forward chaining inference engine evaluates the facts and rules to present the problem and its solution. The inference engine initially evaluates the facts in the order they appear in the fact list. Once the facts are completed the inference engine evaluates the rules.

Figure 3.6 provides a flow chart of the algorithm of the inference engine. Initially the inference engine evaluates the facts. The inference engine then examines the precedent of each rule. All rules with a true precedent are said to have been \textit{enabled} and are put on the stack. The rule's conclusion on the top of the stack is popped and asserted (i.e., the MATLAB command is executed). A rule that is asserted is said to have \textit{fired}. This is followed by another cycle through the rules that are not
Since poles or zeros lie on the real axis, find and plot loci on the real axis.

The sections of the real axis where the sum of the number of poles and zeros to the right is odd lie on the root-locus diagram.

The real-valued zeros and poles were found to be at 
\[ s = -2 \]

The loci which are on the real axis are:
\[ s \text{ from } -2 \text{ to } -\infty \].

It is now possible to plot the loci on the real axis.

Figure 3.5: Screen displays from a problem-solving rule
Evaluate each fact.

Make one cycle through the rules not on the stack, and any rule that is enabled is put on the stack.

A new rule put on the stack?

- Yes: Pop stack.

- No:
  - Is the stack empty?
    - Yes: Stop
    - No: Pop stack.

- Is popped rule's precedent still true?
  - Yes
  - No

Figure 3.6: Flow chart of inference engine
on the stack. Any rules that are enabled are again put on the stack. If no rules become active then the rule on the top of the stack is popped. First, however, a check is done on the rule’s precedent to see if it is still true, and not turned false from the effects of the other rules that were previously asserted. If the precedent is false then the rule’s conclusion is not asserted and the next rule on the stack is popped. This continues until a rule is found that can be asserted. A cycle through the rules always follows after an assertion of a rule. This whole process continues until a cycle puts no new rules on the stack and the stack is empty.

The inference engine always asserts the latest rule that becomes active. This produces a depth-first path through the solution space, and therefore allows the student to follow a branch in the solution space to its end.

Diagnostic Rules

Associated with each fact and problem-solving rule is the option of connecting another set of rules. These rules are unique to the problem-solving rule or fact they are attached to and are evaluated following the evaluation of the specified fact or the assertion of the specified problem-solving rule’s conclusion. The diagnostic rules are used to provide feedback to the facts and problem-solving rules that require input from the student. Feedback may appear in the form of the reinforcement of concepts, the correction of input, or the assistance towards the correct input. Unlike the problem-solving rules, these rules are asserted as soon as they become active. Their proper use would require that control does not leave the diagnostic rule base until the correct input has been entered into the MATLAB workspace.

These rules are entered into the rule (or fact) listing file and are listed below the problem-solving rule (or fact) they are attached to. The rules have the same syntax as the problem-solving rules as was shown in figure 3.4, except that in order to identify it as a diagnostic rule the string Assist has to be found in the ‘rule id’ slot of the diagnostic rule.
Review of Problem Steps Traversed

0. Input the numerator and denominator of the transfer function
1. Given the transfer function, find and plot poles and zeros
2. Given the zeros, check if any are complex
3. Given the poles, check if any are complex
4. Since complex poles exist, determine the angles of departure
5. Given the poles and zeros, check if asymptotes exist
6. Since asymptotes exist, determine intersection points
7. Since asymptotes exist, find angles of asymptotes
8. Given the poles and zeros, check if they lie on the real axis
9. Since poles or zeros lie on the real axis, find and plot loci on real axis
10. Since poles or zeros lie on real axis, find points of departure
11. Given the number of poles, find the number of loci
12. Plot the full root-locus diagram

< ↑ ↓ > - Select Step < Enter > - Review Step < Esc > - Quit

Figure 3.7: Display of step listing

Solution Steps Listing

Once the problem has been solved, an ordered list of the facts and problem-solving rules that were asserted is displayed for review. Each fact and rule on the list displays the comment that describes its function as it was originally entered into the fact or rule listing's text file. A cursor lets the student select and assert a fact or a rule's conclusion (i.e., execute the associated M-file). The selected step is again executed to display the screen that appeared during the problem-solving session. The M-files should be written so that the screen displays, and provides as default, the value of the variables that were previously entered. In this way, the student can enter the default value and return to the list, or enter a new value. If a new value is entered and accepted, then all variables that were entered after the execution of this step are cleared so that TA considers these variables as unknown. The inference engine is then started again and the problem is solved with this new value. An illustration of this screen display with the steps required to solve the root-locus problem example is found in figure 3.7.
3. Tutorial Assistant (TA)

3.3.3 The use of Petri Net Graphs

Petri net graphs have been used to abstract the solution space of a given problem and make explicit the relationships between the MATLAB variables and the problem-solving rules. TA does not use Petri nets and therefore TA's use does not necessarily require the knowledge of Petri nets. The knowledge and use of Petri nets is meant to facilitate the developmental process in creating a problem-solving tutorial for TA by making explicit these relationships to the developer. Examples on the use of the Petri nets are given in the next chapter. This section will provide a brief overview of Petri nets and its implementation in TA.

Petri nets as described in [Peterson, 1981], are a tool for the study of systems. A Petri net consists of the following components: places, transitions, tokens, input functions and output functions. These components abstract features found within the system that a Petri net graph is modelling. Past applications of Petri nets include the modelling of expert system's rules. One such application is found in [Liu and Dillon, 1988], and features of this model have been borrowed and applied in a representation of the problem-solving rules in TA.

Figure 3.8a illustrates the components of a conventional Petri net, while figure 3.8b illustrates the representation used with TA. The circles are the place components of a Petri net, and they are used to represent the variables found within the MATLAB
workspace. Within each circle is the variable that the place represents. A transition in a Petri net is represented by a thick bar. The transition in TA's case represents a rule. An input function in a Petri net is represented by a directed arc from a place (i.e., circle) to a transition (i.e., bar), while an output function is represented by a directed arc from a transition to a place. TA's input arc is considered the rule's check as to whether the variable is known. This is accomplished through the MATLAB exist command that is typically found in the precedent of the rule. The variable, once existing, then may be further used in the precedent of the rule. The output arc represents the resulting output from the rule operation. Typically it provides the output variable with a value.

Tokens are represented by the small dots. Tokens are assigned to and reside in the places of the Petri net, and are used to define its execution. A transition fires by removing tokens from the input places and creating new tokens in the output places. A transition only fires if it is first enabled. A transition becomes enabled if each of the input places for a transition has at least one token. TA's interpretation of tokens in a place indicates that the relevant variable has acquired a value and therefore is now 'known' (i.e., exist returns true if token in place, otherwise it returns false). The variable that is known is available to the rule, and the rule determines whether the transition fires.

Figure 3.8b illustrates a rule which uses the existing variable A in the precedent of a rule. If the precedent is true then the rule's conclusion provides variable B with a value and the token is moved to its place.

A feedback output arc from the transition to the input place is assumed so that the input place, such as the previous example of variable A, does not lose its token after it is passed to the output place. Also assumed for TA's Petri net representation is that an input arc is found connecting from the output place to the transition. This arc typically represents the exist MATLAB command found in the precedent so that the rule will only become enabled if the variable contained in the output place.
a) Rule 2

Use variables C and D in predicate of Rule 2. IF predicate true then apply Rule 2's conclusion to give variable E a value.

b) Rule 3

Use variables F and G in predicate of Rule 3. IF either use is true then apply Rule 3's conclusion to give variable H a value.

Figure 3.9: Petri net's representation of AND and OR

is not known. Figure 3.8c shows these assumptions as they are implied in figure 3.8b.

Figure 3.9 illustrates the representation of logical constructs in the precedent of a rule. Below each of the Petri nets is the meaning attached to the Petri net configuration. Figure 3.9a provides an example where the logical construct AND is represented. In this case, if both variables C and D are known, they are then used in the precedent of 'Rule 2'. If the precedent is true then the conclusion of 'Rule 2' gives variable E a value. Figure 3.9b illustrates an example where the logical construct OR is used. In this example 'Rule 3' provides H with a value if either the variable F or the variable G is known, and any application to the precedent of 'Rule 3' is true.

As was already mentioned, rules provide a convenient method of mapping the solution space. Rules also provide a method to make explicit the operations that will take the problem from one state to the next. The relation between variables that defines the operation that takes the solution from one state to the next can often be traversed in both directions. Figure 3.10a illustrates this point. Here 'Rule 1' defines an operation that can be applied to find a value for B given the values for
where: \( E1 \) is \( A + B = C \)

then: "Rule 1"

"Given A and C, find B"

\[
\text{IF} \quad (\text{AND } \neg \text{exist('B')} \text{ exist('A')} \text{ exist('C')})
\text{THEN} \quad B = C - A
\]

"Rule 2"

"Given A and B, find C"

\[
\text{IF} \quad (\text{AND } \neg \text{exist('C')} \text{ exist('A')} \text{ exist('B')})
\text{THEN} \quad C = A + B
\]

Figure 3.10: Petri net's representation of variable relation

\( A \) and \( C \). Also 'Rule 2' has been defined that finds a value for \( C \) given the values for \( A \) and \( B \). Both rules share the same relation, \( A + B = C \), and use this relation in their conclusions to find the output variable's value. Figure 3.10b abstracts this relation and represents it with the transition bar. This relation, as defined by \( E1 \), is shown to be used by both 'Rule 1' and 'Rule 2'.

3. Tutorial Assistant (TA)
Chapter 4 Examples, Evaluation, and Future Development

As the previous chapter mentioned, TA is far from being a self-contained instructional system such as those pursued by the developers of ITSs. It does however provide a framework for the construction and presentation of individualised lessons, and can be considered a prototype for the design of a more 'intelligent' system. To explore its use and future design possibilities this chapter will provide two example sessions of using TA. Following the examples an evaluation as to how well the components of an ITS, as defined in chapter 1, were fulfilled in TA. This evaluation will use as reference the ITSs that were reviewed in chapter 2.

4.1 Tutorial Session Examples

How TA is used for the development of a tutorial session is very much dependent on the creativity of the author. The only requirement is that the problem solution be decomposed with variables and these variables form the input to the rules which define the problem’s solution. A guideline states that these variables and rules be communicated to the student. TA uses MATLAB’s display capabilities to communicate these rules, and its computational capabilities to provide the numerical results that the student is required to derive. This section will illustrate two examples in the use of TA.

Both instructions are not stand-alone, the sessions act to supplement existing material (e.g., text book and lecture notes). Both sessions use a suite of M-files that were designed to handle the student input and provide screen output. The sessions also require the installation of the ANSI.SYS driver to allow for the creation of the inverted screen characters. A default value is available for each variable that
the student is required specify. The default values in the variable list provide the student with correct values for one complete session.

4.1.1 Example 1: Drawing a Root Locus Diagram

This example takes the student through the steps involved in the drawing a root-locus diagram. Traditionally, instruction in text books on the drawing of root-locus diagram typically refer to the steps involved as 'rules'. Some probable reasons for using this representation could be the fact that the steps do not necessarily need to be applied in a particular order and rules provide a way of conveying this idea. Another possibility is the fact that each rule representation provides a convenient packet that conveys all the necessary information, independent of the actual number of open-loop poles and zeros that are involved. There could perhaps be other reasons.

The rules used were taken from the Control Systems course text book [Appiah and Barker, 1989] supplied to the students. The Petri net graph in figure 4.1 is used to illustrate the solution space of the root-locus problem to the reader. The rules referenced in this figure correspond to the rule listing found in figure 4.2. The variables that are referred to are described in variable listing of figure 4.3. Within the variables num and den places of the Petri net graph are tokens to indicate that these variables have values. These variables are given initial values at the onset by the facts that are listed in figure 4.4. These variable, fact, and rule listings are shown in the form that is required for entering into the relevant text file (single column instead of double column). The screen output from a session using these facts, rules, and variables is shown in figures 4.5, 4.6, and 4.7. The last screen illustrated in figure 4.7 shows the step listing that is displayed. The enumerated variables, facts, rules, and steps allow for cross-referencing between the previously mentioned figures.
4. Examples, Evaluation, and Future Development

Figure 4.1: Petri net graph of solution space for the root-locus problem
4. Examples, Evaluation, and Future Development

"Rule 1*"  
"Given the transfer function, find and plot poles and zeros"  
IF  
(AND -exist('poles') -exist('zeros')  
exist('num') exist('den'))  
THEN  
[poles,zeros]=find_zp(num,den)

"Rule 2"  
"Given number of poles, find number of loci"  
IF  
(AND -exist('nloc') exist('poles'))  
THEN  
nloc=get_nloc(poles)

"Rule 3*"  
"Given the poles and zeros, check if they lie on real axis"  
IF  
(AND -exist('real_x') exist('zeros') exist('poles'))  
THEN  
real_x=check_xx(zeros,poles,num,den)

"Rule 4*"  
"Since poles or zeros lie on the real axis, find and plot loci on real axis"  
IF  
(AND -exist('lr') exist('real_x') real_x==1)  
THEN  
lr=real(num,den,zeros,poles)

"Rule 5*"  
"Since poles or zeros lie on real axis, find points of departure"  
IF  
(AND -exist('sdep') exist('lr'))  
THEN  
sdep=get_sdep(num,den,lr,sdep)

"Assist 1*"  
"Help in determination of departure point"  
IF  
chksdep(num,den,lr,sdep)  
THEN  
sdep=sdep_helper(num,den,de,sdep,lr)

"Rule 6*"  
"Given poles and zeros, check if asymptotes exist"  
IF  
(AND -exist('asy') exist('poles') exist('zeros'))  
THEN  
asym=chk_asy(poles,zeros)

"Rule 7*"  
"Since asymptotes exist, find angles of asymptotes"  
IF  
(AND -exist('arg') exist('asy') asy==1)  
THEN  
arg=asym(poles,zeros)

"Rule 8*"  
"Since asymptotes exist, determine intersection point"  
IF  
(AND -exist('sint') exist('asy') asy==1)  
THEN  
sint=intsc(zeros,poles,sint)

"Assist 1*"  
"Help in determination of intersection points"  
IF  
chkint(poles,zeros,sint)  
THEN  
sint=intshlp(poles,zeros)

"Rule 9*"  
"Given the poles, check if any are complex"  
IF  
(AND -exist('cmplx') exist('poles'))  
THEN  
cmplxp=checkpp(num,den,poles)

"Rule 10*"  
"Since complex poles exist, determine the angles of departure"  
IF  
(AND -exist('theta_d') exist('cmplx') cmplx==1)  
THEN  
theta_d=get_angd(zeros,poles)

"Rule 11*"  
"Given the zeros, check if any are complex"  
IF  
(AND -exist('cmplxz') exist('zeros'))  
THEN  
cmplxz=checkzz(num,den,zeros)

"Rule 12*"  
"Since complex zeros exist, get angle of arrival"  
IF  
(AND -exist('theta_a') exist('cmplxz') cmplxz==1)  
THEN  
theta_a=get_anga(zeros,poles)

"Rule 13*"  
"Plot the full root-locus diagram"  
IF  
(AND -exist('rl') exist('nloc') exist('real_x')  
exisit('cmplx') exist('cmplxz') exist('asy')  
(OR real_x==0  
(AND exist('lr') exist('sdep'))))  
(OR cmplx==0  
exisit('theta_d'))  
(OR cmplxz==0  
exisit('theta_a'))  
(OR asy==0  
(AND exist('arg') exist('sint')))))  
THEN  
rl=rlplot(num,den,nloc)

Figure 4.2: Rule listing for the root-locus problem
Numerator of the transfer function.
\[ \text{num} = [12] \]

Denominator of the transfer function.
\[ \text{den} = [121] \]

Zeros of the transfer function.
\[ \text{zeros} = -2 \]

Poles of the transfer function.
\[ \text{poles} = [-1, -1] \]

Number of loci.
\[ \text{nloc} = 2 \]

Flag indicating whether poles or zeros on real axis.
\[ \text{real}_x = 1 \]

Loci on the real axis.
\[ \text{lr} = [-2, -1, -1] \]

Departure points from the real axis.
\[ \text{sdep} = [-3, -1] \]

Flag signifying whether asymptotes exist.
\[ \text{asy} = 0 \]

Angles of the asymptotes.
\[ \text{arg} = 180 \]

Intersection of the asymptotes.
\[ \text{sint} = 0 \]

Flag indicating whether complex poles exist.
\[ \text{cmplx}_p = 0 \]

Angles of departure from complex poles.
\[ \text{theta}_d = 0 \]

Flag indicating whether complex zeros exist.
\[ \text{cmplx}_z = 0 \]

Angles of arrival to complex zeros.
\[ \text{theta}_a = 0 \]

Flag indicating root-locus diagram is complete.
\[ \text{rl} = 0 \]

Figure 4.3: Variable listing for the root-locus problem

*Fact 1*
*Input the numerator and denominator of the transfer function*
\[ [\text{num}, \text{den}] = \text{problem}([\text{num}, \text{den}]) \]

Figure 4.4: Fact listing for the root-locus problem

In the second screen of figure 4.5, the student enters a non-default denominator value. This therefore makes the other default values incorrect. The rules 5 and 8 require input from the student, and the correct values have to be typed in. Both these rules also have a diagnostic rule that checks on the correctness of the value entered. Since the correct value is entered, the diagnostic rules do not fire.

Unlike the student who would not calculate any points on the root-locus problem, TA uses the M-files available within MATLAB to calculate the points and then to display the required root-locus plot.

Most of the variables defined are meaningful variables that a student drawing a root-locus plot would have to find values for. There are however five variables (\text{real}_x, \text{asy}, \text{cmplx}_p, \text{cmplx}_z, and \text{rl}) that act as logical flags. These flags make explicit the observations that the student is required to make. The results of these observations specify whether other operations should take place and other variables' values found.
4. Examples, Evaluation, and Future Developments

![Root Locus Diagram](image)

**Root Locus Diagram**

The root locus diagram comprises the paths of the poles of the closed loop transfer function in the s-plane.

\[
H(s) \rightarrow [W(s)] \rightarrow \rightarrow V(s)
\]

As the open loop gain is varied, the closed loop poles trace out loci.

**Step 0 / Fact 1**

Given the transfer function, find and plot poles and zeros.

There is one zero located at

\[ s = 2 \]

The poles of the given transfer function are at

\[ s = 1000, -1 \pm 1.5 \text{i} \]

**Step 1 / Rule 1**

There are no complex poles on the root locus diagram.

![Pole-Zero Pattern of Open-loop Transfer Function](image)

**Step 2 / Rule 11**

There are complex poles on the root locus diagram.

**Step 0 / Fact 1 (cont.)**

Enter the open loop transfer function, \( W(s) \)

Enter the numerator \( [1, 2] \)

Enter the denominator \( [1, 0] \)

**Step 1 / Rule 1 (cont.)**

Step 3 / Rule 9

![Pole-Zero Pattern of Open-loop Transfer Function](image)

**Figure 4.5:** Screen displays for the root-locus problem

55
4. Examples, Evaluation, and Future Developments

Figure 4.6: Screen displays for the root-locus problem (continued)
4. Examples, Evaluation, and Future Developments

Figure 4.7: Screen displays for the root-locus problem (continued)
On the Petri net graph in figure 4.1, 'Rule 13' has places that do not contain variables. These are places that use the logical construct OR to define an observation required by the student. These observations however were not made explicit by defining a rule and variable for each observation; instead they are made implicit by combining them with the logical construct AND to form the contents of Rule 13's precedent.

The root locus problem required the writing of 20 M-files for a total of 44K of text. These M-files' primary function were to display the figures on the screen and to call other M-files that handled the user's input and did the required computations. The user input M-files were developed at the CAL Project for previous applications, while most of the M-files that did the computations were supplied with PC-MATLAB.

4.1.2 Example 2: Calculating the Gain and Phase Margin

This session takes the student through the solution space of finding the gain and phase margin. Figure 4.8 illustrates the solution space representation using Petri nets. The Petri net graph uses the transition bar to represent either the rule or the relation between the variables. Figure 4.8 also lists these relations along with the rules that are defined from each relation. Figure 4.9, 4.10, and 4.11 provide the variable, fact, and rule listings.

The Petri net graph shows that two variables, the transfer function represented by the variable $G$ and the problem selection variable represented by the variable $prob$, are defined by the facts at the onset of the problem-solving session. The student determines the entry point into the solution space by providing a value for the variable $prob$. The student can either define a value for the gain margin, $gainm$, the controller gain, $K$, or the phase margin, $phasem$. The student then traverses the solution space by computing and entering the required values until
4. Examples, Evaluation, and Future Development

Figure 4.8: Petri net graph of solution space for the gain/phase margin problem

Problem selection variable.
prob = 1

The numerator and denominator of transfer function G(s).
G = [0 0 0 5; 1 6 5 0]

Frequency that nyquist plot G(jw) crosses the real axis.
w1 = 2.2361

Magnitude that nyquist plot G(jw) crosses the real axis.
modG = 0.1667

Magnitude that nyquist plot K*G(jw) crosses the real axis.
A = 0.2138

Phase margin.
phasem = 40
Gain margin.
gainm = 14.48

Controller gain.
K = 1.132

Flag to signal finished comparison.
com = 0

Figure 4.9: Variable listing for the gain/phase margin problem

*Fact 1*
Enter numerator and denominator of plant transfer function, G(s).
G=preamble(G)

*Fact 2*
Select problem to solve.
prob=sel_prob(prob)

Figure 4.10: Fact listing for the gain/phase margin problem
4. Examples, Evaluation, and Future Development

*Rule: 1*
"Given \( G(s) \), find freq., \( \nu_p \), that nyquist plot crosses over real axis."
   IF
   (AND ~exist('\( \nu_p \)') exist('\( G \)'))
   THEN
   \( \nu_p = \omega_{\nu p}(G, \nu_p) \)

*Rule: 2*
"Given \( \nu_p \) and \( G(s) \), find their magnitude, \( \text{mod}G \)."
   IF
   (AND ~exist('\text{mod}G') exist('\( \nu_p \)'))
   THEN
   \( \text{mod}G = \text{mod}G_{\nu p}(\nu_p, \text{mod}G) \)

*Rule: 3*
"Enter the given phase margin, \( \text{phasem} \)."
   IF
   (AND ~exist('\( \text{phasem} \)') exist('\( \text{prob} \)') prob==1)
   THEN
   \( \text{phasem} = \text{prob}1(\text{phasem}) \)

*Rule: 4*
"Enter the given controller gain, \( K \)."
   IF
   (AND ~exist('\( K \)') exist('\( \text{prob} \)') prob==2)
   THEN
   \( K = \text{prob}2(K) \)

*Rule: 5*
"Enter the given gain margin, \( \text{gainm} \)."
   IF
   (AND ~exist('\( \text{gainm} \)') exist('\( \text{prob} \)') prob==3)
   THEN
   \( \text{gainm} = \text{prob}3(\text{gainm}) \)

*Rule: 6*
"Given \( \text{phasem} \) and \( G(s) \), determine the cross-over frequency, \( \nu_1 \)."
   IF
   (AND ~exist('\( \nu_1 \)') exist('\( \text{phasem} \)'))
   THEN
   \( \nu_1 = \omega_{\nu_1}(G, \text{phasem}, \nu_1) \)

*Rule: 7*
"Given \( \nu_1 \) and \( G(s) \), determine the phase margin, \( \text{phasem} \)."
   IF
   (AND ~exist('\( \text{phasem} \)') exist('\( \nu_1 \)'))
   THEN
   \( \text{phasem} = \text{phase}_\text{in}(\nu_1, \text{phasem}) \)

*Rule: 8*
"Given the \( \nu_1 \) and \( G(s) \), determine the controller gain, \( K \)."
   IF
   (AND ~exist('\( K \)') exist('\( \nu_1 \)'))
   THEN
   \( K = \text{gain}(\nu_1, K) \)

*Rule: 9*
"Given \( K \) and \( G(s) \), determine the cross-over frequency, \( \nu_1 \)."
   IF
   (AND ~exist('\( \nu_1 \)') exist('\( K \)'))
   THEN
   \( \nu_1 = \text{mod}1(G, K, \nu_1) \)

*Rule: 10*
"Given \( \text{gainm} \), determine the magnitude, \( \text{A} \), of the nyquist plot on the real axis."
   IF
   (AND ~exist('\text{A}') exist('\text{gainm}'))
   THEN
   \( \text{A} = \text{realmag}(\text{gainm}) \)

*Rule: 11*
"Given \( \text{mod}G \) and \( K \), find the magnitude, \( \text{A} \), of the nyquist plot on the real axis."
   IF
   (AND ~exist('\text{A}') exist('\text{mod}G'))
   THEN
   \( \text{A} = \text{mag}_\text{real}(G, K, \text{mod}G) \)

*Rule: 12*
"Given \( \text{A} \) and \( \text{mod}G \), determine the controller gain, \( K \)."
   IF
   (AND ~exist('\text{K}') exist('\text{mod}G'))
   THEN
   \( K = \text{find}_\text{gn}(\text{A}, \text{mod}G, K) \)

*Rule: 13*
"Given \( \text{A} \), determine the gain margin, \( \text{gainm} \)."
   IF
   (AND ~exist('\text{gainm}') exist('\text{A}'))
   THEN
   \( \text{gainm} = \text{gain}_\text{m}(\text{A}) \)

*Rule: 14*
"Compare actual and user input values."
   IF
   (AND ~exist('\text{com}') exist('\text{gainm}') exist('\text{phasem}'))
   THEN
   \( \text{com} = \text{compute}(G, K, \text{phasem}, \text{gainm}, \nu_1, \nu_p) \)

Figure 4.11: Rule listing for the gain/phase margin problem
both the phase and gain margin have values. Once both of these variables have values then 'Rule 14' fires and provides a listing of the values the student has entered and the correct values that MATLAB has calculated. The student now can use the 'step listing' to review and correct any mistakes. Figures 4.12 and 4.13 show the screens of a session where the student enters the default value for each variable value requested. For this particular session the phase margin is given and the gain margin is required.

The gain/phase problem required the writing of 17 M-files for a total of 24K of text. As was the case in the root locus problem, these M-files' primary purpose were to display the figures on the screen and to call other M-files that handled the user's input and did the required computations. The user input M-files were the same ones used in the root locus problem, while most of the M-files that did the computations were supplied with PC-MATLAB.

4.2 Evaluation

In [Littman and Soloway, 1988], the authors state that the evaluation of their ITS has taken as much effort as the design of the ITS itself. They however acknowledge that evaluation is critical and that the information attained is well worth the effort. They explain that traditional evaluation of educational systems consists of two main categories: formative and summative. Formative evaluations are used by the developers during the design process to define and refine their goals and methods. Summative evaluation is used after the educational product has been built and used to determine its effectiveness. Littman et al. refers to the building of an ITS as "still somewhat of an art", and mentions that there are few ITSs that can be considered finished. Because of this they claim that the designers of ITSs are "currently more concerned with usefully guiding the development of their systems than with determining whether they are effective educational end products." Because of this they argue that a formative evaluation seems to be presently more of concern for
4. Examples, Evaluation, and Future Developments

![Step 0 / Fact 1 Diagram](image1)

**Step 0 / Fact 1**

A unity feedback control system has a plant transfer function \( G(s) = \frac{1}{s + 1} \).

Enter the numerator of \( G(s) \) \( \{\text{M}1\} 1 \times 1 \)

Enter the denominator of \( G(s) \) \( \{\text{M}2\} 1 \times 5 \times 0 \)

![Meyr Plot](image2)

**Meyr Plot**

The frequency at which the magnitude of \( G(j\omega) \) equals unity is known as the crossover frequency, \( \omega_c \).

**Select problem to solve**

1. Determine gain margin given a phase margin
2. Determine phase margin given a controller gain \( K \)
3. Determine phase margin given a gain margin

Input problem number (1, 2 or 3) \( \{\text{M}3\} 1 \)

Input the phase margin \( \{\text{M}4\} 40 \)

![Step 1 & 2 / Fact 2 & Rule 3 Diagram](image3)

**Step 1 & 2 / Fact 2 & Rule 3**

Given the phase margin, phase \( \phi \), set \( 45^\circ \) and solve the following equation for \( \omega_c \) in rad/s:

\[ \frac{\tan(\phi)}{\omega_c} = \frac{\omega_c}{\omega_r(1)} \]

Input the value for \( \omega_r \) \( \{\text{M}5\} 0.05 \)

![Step 3 / Rule 6 Diagram](image4)

**Step 3 / Rule 6**

Given that the crossover frequency, \( \omega_c \), was calculated as 0.05 rad/s, use the following formula to calculate the controller gain, \( K \):

\[ K = \frac{1}{\omega_c} \]

Input the value for \( K \) \( \{\text{M}6\} 152 \)

![Step 4 / Rule 8 Diagram](image5)

**Step 4 / Rule 8**

This plot is the Masyt diagram of \( G(s) \)

The frequency at which \( G(j\omega) \) crosses the real axis is labeled \( \omega_0 \)

![Step 5 / Rule 1 Diagram](image6)

**Step 5 / Rule 1**

(Figure 4.12: Screen displays for gain/phase margin problem)

---

---
Examples, Evaluation, and Future Developments

Step 5 / Rule 1 (cont.)

Gain Control

This diagram is the Nyquist plot of \( G(s) \) with \( N=1.2 \).

Given that \( |G(j\omega)| \) equals 0.067 and \( \phi = 1.12^\circ \), the magnitude at which the Nyquist plot crosses the real axis, can be calculated to equal 0.107 using the following formula:

\[ |G(j\omega)| = 0.107 \]

(Press any key to continue)

Step 6 / Rule 2

Gain Control

This diagram is the Nyquist plot of \( G(s) \) with \( N=1.2 \).

Given that \( |G(j\omega)| \) equals 0.067 and \( \phi = 1.12^\circ \), the magnitude at which the Nyquist plot crosses the real axis, can be calculated to equal 0.107 using the following formula:

\[ |G(j\omega)| = 0.107 \]

(Press any key to continue)

Step 7 / Rule 11

Gain Control

This is the Nyquist plot of \( G(s) \) with \( N=1.2 \).

The actual parameters are:
- phase margin = 39.9°
- gain margin = 14.5 dB
- frequency where the Nyquist plot crosses the real axis is 2.537 radians.

(Press any key to continue)

Step 8 / Rule 13

Step Listing

Figure 4.13: Screen displays for gain/phase margin problem (continued)
4. Examples, Evaluation, and Future Development

the developers of ITS. However because the field of ITSs "is too young" there is no standard set of methods for formative evaluation, and hence Littman et al. attempt to define evaluation methods that could be used. They base their methods on their experience in developing an ITS called PROUST.

In the development of TA, we were more concerned with the development of a system that could be developed and operational within a limited time frame, than with research into finding and encoding the intelligence that is required to make the system truly intelligent. Since TA is functional, a summative evaluation would seem appropriate. Unfortunately the author left the project before any sort of evaluation on the effectiveness of TA could be carried out. Through communications with colleagues still at the project, it was possible to find out about the current state on the use of TA. There were attempts to put TA onto the network so that it could be incorporated into the Control Systems course. Unfortunately they felt that TA ran to slowly on the networked PCs to be useful in a classroom setting and felt that response time is critical to keep the student attentive to the task. The slow performance of TA could be attributed to the fact that every time TA calls a MATLAB command, it requires access to the fileserver's hard disk. The access time for a MEX-file calling a M-file seems much longer than for a M-file calling a M-file. The 286-based, and especially the 386-based machines would provide a more suitable platform for the use of TA rather than the networked PCs. Also, parts of TA's C code may require rewriting for optimal performance. One such optimisation occurred through TA's interception of the clear and exist MATLAB commands.

Since we have also considered TA as the first step towards a more 'intelligent' CAL, a formative evaluation is also appropriate. The formative analysis provided in this section will reflect on the functional components of an ITS that were mentioned in chapter 1. It will examine these components and evaluate how well they fulfilled their role within TA and provide suggestions for improvements. Chapter 1 also mentioned that the boundaries of these components are not always that well defined, this section will attempt to define the boundaries of the components in TA.
4. Examples, Evaluation, and Future Development

and relate them to the ITSs that were examined in chapter 2.

4.2.1 Expert Module

MATLAB with its many numeric algorithms and simulations capabilities provided the expertise of TA, and TA was built around this expertise. MATLAB provided the solutions to variables which formed the intermediate steps to the final solution.

The SOPHIE system's expertise also revolved around its numeric algorithms and simulation capabilities. However it was SPICE that provided access to the knowledge that was contained in the algorithms. The goal of the designers of SOPHIE was to articulate to the student the causal relations of the circuit that they felt were necessary for trouble-shooting learning. Because of the 'black box' nature of the expertise found in SPICE the developers had to invent methods to make the required knowledge explicit. The MYCIN-based systems also went along a similar transformation. The diagnostic knowledge was compiled and found implicitly in the rules of MYCIN, and therefore GUIDON was unable to articulate this knowledge to the student. The reconfiguration of MYCIN to NEOMYCIN resulted in a diagnostic model that made explicit the hierarchical organisations of knowledge and the search strategies involved. Whereas the SOPHIE systems made the transition from a 'black box' model to a 'cognitive model' of expertise, the MYCIN-based systems evolved from the 'glass-box' model of expertise to a 'cognitive model' of expertise. In these transitions, a focus was made on the type of knowledge that was required by the student, and then how to make explicit and communicate this knowledge.

TA can only contain and attempt to teach by rote the steps required to perform a particular procedure. The forward chaining inference used by TA is similar to the approach that experts use when solving a problem. "Canned" text found in the M-files is used to provide some explanation as to why the particular rule is being
4. Examples, Evaluation, and Future Development

applied. Like the Lisp Tutor and the Geometry Tutor, rules are used to make explicit each step required to solve the problem. This knowledge is sufficient in situations where the student is required to perform a particular procedure. These situations are often found on a class test, where the problem has only one method or problem-solving procedure that is known by the student and the student is expected to apply this procedure to find the solution. Understanding is not necessary, performing the procedure is. Because it does not provide a greater understanding, TA can only supplement the existing course curriculum.

In [Steels, 1990], Steels examines the domain model of expertise, and refers to knowledge as being either deep or surface knowledge. Deep knowledge is referred to as the knowledge that "makes explicit the models of the domain and the inference calculus that operates over these models." Surface knowledge is referred to as the selected portions of the deep knowledge that are relevant to the particular problem. Steels refers to the fact that traditional expert systems only encode the surface knowledge, and that the deep knowledge is implicit within the coding. Because of this the surface expert systems are efficient but "brittle" and provide weak explanations. Deep expert systems however overcome these limitations by containing two components, one that implements deep knowledge of the domain and the other implements the surface knowledge. The surface level component provides the effective and efficient reasoning that is usually required, while the deep level component provides a back up if surface knowledge is not complete or when detailed explanation is required. Making the deeper knowledge explicit and deciding on what elements of the deeper knowledge are required as surface knowledge requires analysis of the domain. As well as the domain model, Steels also reviews other approaches for studying expertise: the task they perform, the problem-solving method they employ, and the pattern of inference they use. He then combines these and provides a framework for the analysis of expertise.

The procedures that TA is able to articulate does contain deeper knowledge (e.g., first principles), and such knowledge could be used to provide explanation.
4. Examples, Evaluation, and Future Development

In this case the largest grain size of knowledge would be the rule that defines the procedure. Going in the other direction, these rules could provide the smallest grain size of knowledge. In this case the knowledge contained in the expert module would be much broader in scope, and make explicit and encode the knowledge that is required by the student for the whole course. The pedagogical objectives of each system would differ, and both would require a different architecture for the encoding of the knowledge than that which TA is currently able to provide. The required architecture would be constructed following an analysis of the expertise involved such as that suggested in [Steels, 1990].

4.2.2 Tutoring Module

There is no clear distinct tutoring module in TA. The knowledge of how to solve the problem as well as its instruction, is found in the problem-solving rules, and therefore the tutoring module can be considered contained within the problem-solving rules. The tutoring module structure and function is similar to the decision tree of SOPHIE-II. There is very little initiative by the student, and therefore the tutoring module's primary function is to instruct how to solve the problem.

A guideline is provided namely that the problem solution be decomposed with meaningful variables and the rules that define the solution should be communicated to the student. It is believed that articulating the rule will facilitate the declarative encoding of the rule by the student, which can be considered a prerequisite for the knowledge compilation process. The communication of the rule is provided through various levels of abstraction. The step listing, the rule, and the reification of the rule through its instantiation represent these levels. The pedagogical objective of the tutoring module is that the student should learn the solution space of the problem. This objective is similar that of the system by [Williams and Reynolds, 1991] where the student is expected to obtain a cognitive simulation model of the task, which then resides in the student's production mem-
4. Examples, Evaluation, and Future Development

ory. The step listing is an attempt to communicate the solution space. The function of the step listing is similar to the proof graph of the Geometry Tutor. An improved communication of the solution space could involve the ability of displaying the whole solution space. This would require greater graphics capabilities to be developed for displaying something similar to the Petri net representation of the solution space.

Because there is only one procedure being taught, no selection within a curriculum is required. If many concepts and/or procedures were encoded in an ITS and required to be learnt, then exercise selection heuristics similar to the one found in the curriculum-based ITSs should be added.

4.2.3 Diagnostic Module

Student participation is limited to entering values for specified variables. Therefore any diagnostics must be centered around these values. The diagnostic rules provide these diagnostics. There is no defined student model, however if one was required within TA, then the student model could be considered as the values that he enters. This student model is then considered an overlay of the expert’s responses (i.e., TA’s MATLAB solutions). Such a student model is similar to the student models found in the Lisp and Geometry Tutor.

A diagnostic module’s task is to analyses the data found within the system, and then update the student model. The sample rules of for the root-locus problem, show the limited use that the diagnostic rules have been given so far: feedback as to the whether answer is correct or not. Deeper level diagnostics, so that a misconception can be identified and articulated, or other uses of the student model as outlined in chapter 1, require the analysis of the task and the student’s actions. The results from this analysis forms the knowledge contained in the diagnostic module. The present implementation of TA only captures the intermediate solutions to a
final solution. This information may be too narrow for an informative analysis of a student's actions.

4.2.4 Instructional Environment

TA operated in the environment that was provided by MATLAB. MATLAB is text based and provides up to four graphs on one graphics screen. ASCII characters provide the limited graphic capabilities on the text screen. An improved instructional environment would require greater graphic capabilities. Attempts were made to include the graphic routines found in Turbo C. Errors resulted when the object file provided by MATLAB and the Turbo C graphics object file were linked to make a MEX-file.

4.2.5 Human-Computer Interface

Input by the student is limited to command line input. Input such as that of a transfer function requires that the input be similar to that required by MATLAB and therefore the student requires some knowledge of the syntax of MATLAB. A M-file that provides help to explain this syntax is available and can be included in a problem-solving session.

4.3 Future Development with MATLAB

This section will provide information for possible future implementation of ITS within MATLAB. The first section will discuss an extension to TA that was partly implemented. An example of how it might be used is provided and its current implementation is illustrated with an example. The second section discusses the development of an ITS around a discovered attribute of the MEX-file.
4. Examples, Evaluation, and Future Development

4.3.1 An Extension to TA

This implementation attempted to provide the student with more initiative within TA. What was envisioned was that instead of TA instructing the student and only allowing him to enter in the values of the variables, he would instead be able to select the next variable that required a value, such as that found in the 'model tracing' paradigm, and then use MATLAB as a scratch pad to compute the value of that variable.

The student would initially be presented a problem to solve as is the case in the current version of TA. Following the problem presentation, a list of variables and their descriptions, as defined in the variable text file, is presented to the student. The student is then required to select from this list the variable which he feels is ready for a value at the particular stage of the problem.

In selecting a variable the student is actually selecting the rule whose output from the conclusion gives this variable a value. If the student selects an appropriate rule (i.e., variable) then the rule's precondition must be true (i.e., the variables that are used to calculate this variable must already have values). If the variable chosen was not ready for a value then help may be provided and the student is asked to try again. If the student was correct in his selection, then TA uses the MATLAB command `keyboard` to shell the student onto the MATLAB command-line. The student is then asked to compute and enter a value for the variable. The student now has access to the variables, and uses them and the MATLAB commands to calculate a value for the variable. MATLAB is now used as a scratch pad, in the sense that the student can fill the MATLAB workspace with many other variables, and only the required variable and its value will be passed onto the next variable calculation through the variable list. Pressing 'CTRL-z' returns the student back to the list. If the variable the student requested to find a value for does not have a value, then student is returned to the MATLAB command-line and is told the variable was not given a value. Once the variable has a value, then pressing 'CTRL-z'
4. Examples, Evaluation, and Future Developments

<table>
<thead>
<tr>
<th>Variable list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. prob : Problem selection variable.</td>
</tr>
<tr>
<td>2. G : The numerator and denominator of transfer function G(s).</td>
</tr>
<tr>
<td>3. wp1 : Frequency that nyquist plot G(jw) crosses the real axis</td>
</tr>
<tr>
<td>4. modG : Magnitude that nyquist plot G(jw) crosses the real axis.</td>
</tr>
<tr>
<td>5. A : Magnitude that nyquist plot K*G(s) crosses the real axis.</td>
</tr>
<tr>
<td>6. phasem : Phase margin.</td>
</tr>
<tr>
<td>7. gainm : Gain margin.</td>
</tr>
<tr>
<td>8. w1 : Cross-over frequency.</td>
</tr>
<tr>
<td>10. com : Flag to finish comparison.</td>
</tr>
</tbody>
</table>

Figure 4.14: Variable listing with no values given to variables

passes only the specified variable value from the scratch pad. The variable on the list now indicates that the variable has a value. TA could check on the correctness of the value at this point.

The current implementation of TA has some of the programming already completed. Two M-files are used by TA to load and save the variables from the workspace. These M-files use three MAT files called: gt.mat, ht.mat and it.mat. A MAT file is a MATLAB file that contains the saved workspace’s variables.

To demonstrate this partial implementation within TA, the following steps must be performed. First, clear the MATLAB workspace by executing a clear on the command-line. Then execute save gt, save ht, and save it, to make empty mat files. Execute TA on the MATLAB command-line, and then load only a variable list. Once loaded, selecting item 9 on the menu illustrated in figure 3.1 provides a listing of the variables with their descriptions. Figure 4.14 illustrates this listing using the gain/phase margin problem variable list that was shown in figure 4.9. The number on the left column indicates that the variable does not have a value.
4. Examples, Evaluation, and Future Developments

Moving the cursor to a numbered variable and then pressing enter, places the user on the MATLAB command-line and provides him with a display of the variables that have previously been entered and given values. If it is the first variable selection in the session, then no variables are listed. Figure 4.15 illustrates a screen in which three variables (G, phasem, and prob) have already been given a value, and the user has just selected the variable gainm.

These variables and their values are the only variables found in the workspace. The user can reload these to the workspace using the MAT files as is indicated in figure 4.15. The user can now use the workspace and the MATLAB commands to define as many variables as required to calculate a value for the specified variable gainm. However only the variable gainm and its value is passed back to the list. Pressing CTRL-z takes the user back to the list if a value had been given to gainm. If no value had been provided then the user is told so and returned to the MATLAB command-line. Once back on the list the variable gainm along with other listed variables with values are indicated with double asterisks (***) on the left column as shown in figure 4.16. Selecting one of these variables and pressing enter, now
4. Examples, Evaluation, and Future Development

---

**Variable List**

- **prob**: Problem selection variable.
- **G**: The numerator and denominator of transfer function $G(s)$.
- **$\omega_\pi$**: Frequency that Nyquist plot $G(j\omega)$ crosses the real axis.
- **$\text{mod}G$**: Magnitude that Nyquist plot $G(j\omega)$ crosses the real axis.
- **$A$**: Magnitude that Nyquist plot $K\cdot G(s)$ crosses the real axis.
- **phase**: Phase margin.
- **gain**: Gain margin.
- **$\text{m}$**: Cross-over frequency.
- **$K$**: Controller gain.
- **$\text{com}$**: Flag to finish comparison.

---

**Figure 4.16**: Variable listing with some variables having values displays the value of this variable. This is illustrated in figure 4.17 where the variable $G$ has just been selected.

In this extension the student has more autonomy in the problem-solving session since he must first select the required variable and then use MATLAB to calculate its value. This forces extra work on the student because he must first be knowledgeable of MATLAB and its commands to use the tutoring system. However, as explained in [Fuller, 1992], the MATLAB language is "closer to standard mathematical usage" than any other high-level language. This, he explains, provides the student with a quick understanding of the language and therefore a relatively short learning time. Fuller uses MATLAB to obtain and communicate the solutions to problems in a course that he teaches.
4.3.2 The Use of Static C Variables

Another implementation possibility that has not been fully explored is based on an observation that a variable defined as 'static' within a C MEX-file will retain its value after successive calls to the MEX-file from the MATLAB command-line. Unfortunately this observation came fairly late in the development of TA and therefore was not exploited in any way.

This facility provides the possibility of allowing MATLAB to have access to a C defined data structure and this data structure be updated and accessed from the command-line. Prior to this observation, the only alternate for the storage of data was either on disk or as a matrix within the MATLAB workspace.

An ITS could be developed around this facility where the student would remain on the MATLAB command-line while the ITS stayed in the background until called by the student whenever help is required. Through conversation with the student, the ITS would update a student model (i.e., a static C data structure) and provide
the level of help the ITS deemed appropriate. Once help has been administered, the student would return to the task at hand on the MATLAB command-line until the next call for help.
Chapter 5

Conclusion

This thesis has presented an attempt to develop an Intelligent Computer-Aided Instruction (ICAI) system within PC-MATLAB. The system developed is called Tutorial Assistant (TA). This thesis initially provided an overview of CAL, a description of the CAL project where TA was developed, and an overview of ICAI and its various components. Several ICAI systems were also examined which provided examples of the issues involved in ICAI design and the various approaches taken by their developers. PC-MATLAB, with its many algorithms and numerical computation abilities, provided the expertise in TA. The knowledge contained within PC-MATLAB is encoded in modular and reusable code called M-files. Through the use of the M-files, TA provides a method for the instructor to make explicit to the student, the knowledge required in solving 'tutorial' problems. An elaboration on the use and design of TA was presented. It was followed by two examples of its use, an evaluation, and suggestions for future implementation. TA is far from being an 'intelligent' system as pursued by some developers of ICAI systems. Like the development process of past ICAI systems, it does represent the initial step in the development of a more 'intelligent' CAL system within the MATLAB environment.
References


