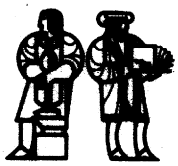
LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-176

A DIGITALIS THERAPY ADVISOR WITH EXPLANATIONS

William R. Swartout

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

MIT/LCS/TR-176

A Digitalis Therapy Advisor with Explanations

William R. Swartout

February, 1977

Massachusetts Institute of Technology

Laboratory for Computer Science
(formerly Project MAC)

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

A Digitalis Therapy Advisor with Explanations

by

William Roy Swartout

Submitted to the Department of Electrical Engineering and Computer Science on January 16, 1977 in partial fulfillment of the requirements for the Degree of Master of Science.

Abstract

This thesis describes the English explanation facility of the OWL Digitalis Advisor, a program designed to advise physicians regarding digitalis therapy. The program is written in OWL, an English-based computer language being developed at MIT. The system can explain, in English, both the methods it uses and how those methods were applied during a particular session. In addition, the program can explain how it acquires information and tell the user how it deals with that information either in general or during a particular session.

Most explanations are produced directly from the code used in prescribing digitalis and from information which is generated by the OWL interpreter as it runs. The ability of the program to translate its internal structure to an English explanation is provided by structuring the program using Semantic Model Programming. Each OWL procedure attempts to represent a single concept or idea that should be meaningful to the physician using the system. By organizing the program in this way, the explanations produced by the system tend to relate well to ideas with which the physician is already acquainted.

In many current systems which ask the user a series of questions, a problem occurs if the user wishes to change his answer to a previous question. These systems accept the change, but must recompute all the results computed subsequent to that question to insure that none of them are affected. Clearly, this may involve a considerable amount of unnecessary recomputation. By using OWL, we obtain the data structures necessary to avoid this problem. An algorithm is described that allows the system to accept a changed answer without recomputing all prior results. This process is called updating. The updating algorithm presented here also allows the system to provide concise explanations of the effects of the changed answer.

Thesis Supervisor: William A. Martin
Title: Associate Professor of Electrical Engineering and Management

Thesis Supervisor: Peter Szolovits
Title: Assistant Professor of Electrical Engineering

This empty page was substituted for a blank page in the original document.

Table of Contents

Chapter 1: Introduction 7
1.1 Background 8
1.1.1 Some Aspects of Digitalis Therapy 8
1.1.2 Previous Digitalis Advisors 10
1.1.3 Other Work in Explanation 12
1.2 An Overview of the OWL Digitalis Advisor 13

Chapter 2: A Sample Session 15
2.1 The Initial Session 15
2.2 The Follow-up Session 21
2.3 Explanations 25
2.3.1 Explaining Methods 25
2.3.2 Explaining Events 28
2.3.3 Explaining How a Variable is Used in General 32
2.3.4 Explaining How a Variable is Set in General 32
2.3.5 Explaining How a Variable was Used in Particular 33
2.3.6 Explaining How a Variable was Set in Particular 34
2.3.7 Explaining How a Method may be Called 35
2.3.8 Explaining Why a Method was Called 36

Chapter 3: Explanation -- How It's Done 37
3.1 Introduction 37
3.2 The OWL Knowledge Base and Interpreter 37
3.2.1 The Knowledge Base 38
3.2.2 The OWL I Interpreter 39
3.2.3 OWL I and Explanation 41
3.3 The English Generator -- Turning OWL I into English 42
3.4 Semantic Model Programming: Programming for Explanation 45
3.4.1 Introduction 45
3.4.2 Semantic Model Programming and OWL I 47
3.4.3 Semantic Model Programming and Structured Programming 50
3.5 The Explanation Routines -- How They Work 51
3.5.1 Introduction 51
3.5.2 Describing Methods 51
3.5.3 Describing Events 53
3.5.4 Describing the Use and Setting of Variables 56
3.5.5 Describing When an Event or Plan is Called 57
3.6 Summaries and Alternate Models 58
3.6.1 Summaries 58
3.6.2 Alternate Models 60
3.7 Extensions for Iteration 62

Chapter 4: Updating 64

4.1 Introduction 64
4.2 A Sample Session 66
4.3 An Outline of the Issues in Updating 70
4.3.1 Restrictions 70
4.3.2 Types of Time 71
4.3.3 Special Data Structures for Updating 72
4.4 Updating: the Algorithm 74
4.4.1 An English Description 74
4.4.2 The Program 76
4.5 The Nitty-Gritty 78
4.5.1 Determining Precedence Time-Order 78
4.5.2 Editing Environment Lists 79
4.5.3 A Proof of Correctness 80
4.6 Comparison of Different Updating Strategies 85
4.7 Current Performance and Possible Improvements 90
4.8 Explaining Updates 91
4.9 Procedures, Rules and Updating 92

Chapter 5: Conclusions and Suggestions for Further Research 93
5.1 Further Research 93

References 95

Table of Illustrations

3.1 An OWL Plan 40
3.2 Types of Specialization 43
3.3 The OWL Code to Check for Sensitivity Due to Myxedema 52
3.4 An English Explanation of the Code to Check Sensitivity Due to Myxedema 53
3.5 An Explanation of the Event of Checking for Sensitivity Due to Myxedema 56
3.6 A Procedure Using Iteration 62
4.1 The Program 74
4.2 The Update List 74
4.3 The Environment List 75
4.4 The Environment List and Temporary Environment List 75
4.5 The New Update List 76
4.6 The New Environment List 76

Acknowledgements

I would like to thank all those who made this possible. Howard Silverman and Dr. Stephen Pauker were very helpful with medical assistance. Alex Sunguroff helped with the OWL Interpreter. Various discussions with members of the Automatic Programming Group and the Clinical Decision Making Group provided many valuable insights. I would especially like to thank my supervisors, Professors William A. Martin and Peter Szolovits for their patience and advice.

This research was supported in part by the Health Resources Administration, U. S. Public Health Service, under grant 1 R01 MB 00107-01 from the Bureau of Health Manpower and under grant HS 00911-01 from the National Center for Health Services Research.

Chapter 1: Introduction

The documentation of programs (or the lack of it) is a problem that continues to be troublesome. Existing documentation is frequently outdated or inaccurate, can be difficult to obtain, and often can only be comprehended by other programmers.

This problem exists for a number of reasons. Documentation is often written only as an after-thought, after a system has been completed. Frequently, the programmer is the only link between the system and its documentation. Thus, changes in the system are not reflected in the documentation unless the programmer remembers to make them. The documentation is also frequently physically separated from the system, so that a user may not have documentation available when he wishes to use the system. Some programmers try to document the code they produce by using mnemonic names for variables and procedures, yet such documentation remains unavailable to non-programmers.

If a program can explain its reasoning processes, user acceptance can be more easily obtained, since the user can assure himself that the program makes reasonable deductions which result in reasonable conclusions. Additionally, an explanation facility may serve a valuable pedagogical function. A student or practitioner may use the system and improve his understanding of the material that he is studying by comparing his own reasoning with that of the system. Finally, the ability to provide explanations serves as a valuable tool for debugging the system.

In this thesis, a system is described which can explain itself. This system, called the OWL Digitalis Advisor, is designed to advise physicians concerning digitalis therapy. It is written in OWL I, which is a prototype of the OWL language currently under development at MIT[13,14,20,21]. The system is "self-documenting" in the sense that it can produce English explanations of the procedures that it uses and the actions it takes directly from the code it

executes. Most of the explanations provided are produced in this manner, although a few types of explanation are produced by displaying canned phrases. The physician may request explanations during a consultation session. The explanations are designed to be understood by a physician with no programming experience.

In the remainder of the introduction, some of the medical aspects of digitalis therapy will be outlined, followed by a review of previous digitalis advisors and work in explanation. Finally, a very brief overview of the CMU Digitalis Advisor is presented.

1.1 Background

1.1.1 Some Aspects of Digitalis Therapy

The digitalis glycosides are a group of drugs that were originally derived from the foxglove, a common flowering plant. This group includes digoxin, digitoxin, cubain, cedelanid and digitalis leaf. Among these, digoxin is currently by far the most commonly used drug. The use of digitalis was first documented by William Withering in an article written in 1785. He noticed that the drug caused increased urine flow, and used the drug to treat abnormal accumulations of fluid, a condition known as dropsy, which is often the result of a failing heart. Later, it was discovered that this diuretic effect is only secondary to the principal effect of digitalis, which is to strengthen and stabilize the heartbeat.

In current practice, digitalis is prescribed chiefly to patients who show signs of congestive heart failure and/or conduction disturbances of the heart. Congestive heart failure refers to the inability of the heart to provide the body with an adequate blood flow. This condition causes fluid to accumulate in the lungs and outer extremities and it is this aspect that gives rise to the term "congestive". Digitalis is useful in treating this condition, because it

increases the contractility of the heart, making it a more effective pump. A conduction disturbance appears as an arrhythmia, which is an unsteady or abnormally paced heartbeat. Digitalis tends to slow the conduction of electrical impulses through the conduction system of the heart, and thus steady certain types of arrhythmias. Due to the positive effect that digitalis has on the heart, it is one of the most commonly used drugs in the United States. In 1971, it was fifth on the list of drugs most frequently prescribed by doctors through pharmacies in the US [4,5].

There is, however, a darker side to digitalis. Like many other drugs, digitalis can also be a poison if too much is administered. In the case of digitalis, the ratio between a dose which will cause a therapeutic effect and one which will cause a toxic reaction is only about 1 to 2. This "therapeutic window" is particularly small when compared with other drugs. The window for aspirin, for example, is about 1 to 20. In addition, there are a number of factors such as age, weight, electrolyte balance, and history of heart damage (to name a few) that may cause the patient to be more sensitive to digitalis and more likely to develop a toxic reaction. These factors must be taken into account in prescribing digitalis.

Digitalis toxicity may assume many different forms. It may manifest itself as blurred or colored vision. Certain gastro-intestinal symptoms such as anorexia (loss of appetite), nausea or vomiting may appear. Toxicity may also appear as certain types of abnormal heart rhythms.

The clinician must be particularly careful in interpreting toxic signs, since they may have other causes unrelated to digitalis, or in the case of some arrhythmias, they may be mistaken for a lack of therapeutic effect. Thus, it is possible that a doctor may give a greater dose of digitalis, mistakenly thinking that the patient is not showing adequate therapeutic effects, when in fact he should withhold digitalis until the patient's toxic symptoms disappear.

In the body, digitalis tends to accumulate and dissipate in an exponential fashion like the

charge on a capacitor in an RC circuit [5,6,7]. Digitalis leaves the body through two routes. Much of the drug is excreted in the urine, and the rest leaves via the liver. The exact proportions depend on the preparation used, and how well the patient's kidneys are functioning (renal function). A doctor must consider these elements in assessing a patient's response to the drug.

Because it is so difficult to predict *a priori* how much digitalis a patient should receive, cardiologists generally use feedback to determine the correct dose. A certain amount of digitalis is given to a patient, the therapeutic and/or toxic effects that appear are evaluated, and the dose the patient receives is adjusted appropriately. Once it is felt that the patient is receiving the correct amount, the patient is placed on a maintenance program so that the amount of digitalis he receives each day is equal to the amount lost through excretion.

Since there are a large number of factors to consider, and the exponential model is somewhat inconvenient, many patients are treated incorrectly. Studies indicate that as many as 20% of all patients receiving digitalis show toxic symptoms, and that the mortality rate among these patients may be as high as 30% [4,8].

1.1.2 Previous Digitalis Advisors

Several computer programs have been constructed to provide physicians with advice about digitalis therapy. One of the first such programs is described by Jelliffe [9,10]. This program was written shortly after the pharmacokinetics of the digitalis glycosides became understood, and was designed to compute initial dosage regimens, based on the patient's weight, renal function, history of digitalis therapy, and route of administration. The program is only applicable for use with patients having normal thyroid and liver function and normal electrolyte balance. It is capable of calculating a reasonable initial dosage regimen subject to

the restrictions stated above. However, the program is deficient in two important ways. First, the program does not take into account all the factors influencing digitalis administration. The effects of digitalis are very much affected by electrolyte balance. This limitation makes the program useless for those patients with altered electrolyte balances. Second, the program only provides the initial dosage regimen. It is up to the doctor to moniter the patient's progress and make adjustments as toxic effects appear, or initial conditions (such as renal function) vary.

Shainer [8,11] produced an improved system by using feedback control techniques. The doctor specifies a desired blood level of digitalis. The program computes an initial dosage regimen, and after the patient is given the drug, the level of serum digitalis is determined. This data tells the program whether the digitalis is being used by the patient at the same level that was anticipated in computing the initial regimen. The program uses this new information to determine a new regimen, and the feedback loop is repeated until a stable condition is reached.

Shainer's program solves one of the problems in Jelliffe's program, but it has some other flaws. The objective of the program is the achievement of some level of serum digitalis. In a clinical setting, it may not be easy to specify what this level should be, since the proper level is affected by what condition the patient is receiving digitalis for, as well as certain medical conditions the patient may suffer from, such as potassium depletion, that would make him sensitive to digitalis. More importantly, the serum level of digitalis is not a good indicator of clinical effect[3]. In addition, Shainer's complex statistical methodology would make it difficult for his system to provide clear explanations to the user.

Recently, a program has been developed by Pauker, Silverman and Gorry which differs from earlier ones in two important respects[3]. First, it constructs a patient-specific model, reflecting the program's knowledge of pharmacokinetics and special features of the patient's

condition which may alter his response to therapy. This model is used to construct the initial dosage recommendations. Second, the program makes assessments of the toxic and therapeutic effects which actually occur in the particular patient (after he has received the initial dose) to formulate subsequent dosage recommendations, rather than using the blood level of digitalis.

A limited clinical trial was performed in which the program "followed" a series of patients managed by clinicians on a cardiology service. That trial demonstrated the program's ability to recommend appropriate therapy in acutely ill patients. Each of the few patients who developed toxicity had received more digitalis than would have been recommended by the program. The program anticipated each episode of toxicity before it was recognized clinically. Thus, although the trial was limited, it was very encouraging. The program was used as a basis for the GWL Digitalis Advisor.

1.1.3 Other Work in Explanation

Explanation capabilities have been implemented for systems operating in domains other than digitalis therapy. Winograd's SHRDLU[1] is a good example of a system able to provide the user with some sort of explanation for its actions. The system can explain to the user why certain actions were taken and provide the user with an English "translation" of its goal stack. One of the problems Winograd encountered was the conversion of MICRO-PLANNER expressions to English.

Shortliffe[2] and Davis[12] describe the explanation system that has been implemented for MYCIN, a system designed to help doctors in prescribing antibiotics. MYCIN functions in an interactive manner, and is capable of explaining why certain questions were asked, as well as the reasoning chain that it employs. The explanation systems of MYCIN and the GWL Digitalis Advisor are compared in chapters 3 and 4.

Mikelsons has been working on the problem of trying to explain programs written in BDL (Business Definition Language) to a user unfamiliar with programming[16]. His system uses two models: one to model the program's understanding of the problem and the other the user's. It uses a PLANNER-like mechanism to draw inferences between the models. Mikelsons' system is still under development, hence it is impossible to compare the performance of his system with that of the OWL Digitalis Advisor. However, it does seem safe to say that his system is fundamentally different from the Digitalis Advisor. For one thing, the Digitalis Advisor does not employ any PLANNER-like inference schemes. Another difference is that when the Digitalis Advisor was written, an effort was made to combine the user model and program model into one structure as much as possible. We will see that in most cases this single model is sufficient to give the Digitalis Advisor a good explanatory capability. In those cases where a single model is not adequate, the Digitalis Advisor employs Alternate Models (described in section 3.7.2). Thus, while Mikelsons' system will use the more sophisticated (but also more complex) two model approach exclusively, the OWL Digitalis Advisor relies on a simpler single model whenever possible, resorting to multiple models only when necessary.

1.2 An Overview of the OWL Digitalis Advisor

The OWL Digitalis Advisor consults with a physician in an interactive manner. The Advisor asks the clinician a number of questions about the patient and then produces a set of recommendations. After the patient has received an initial dose, the program can produce a new dosage regimen based on the reaction of the patient.

While a session is taking place, the system can explain why it is asking a question. At the end of any session, the system can provide a number of different types of explanation. It can explain the procedures it uses and the actions it takes either in general or for the patient

at hand. It can explain how variables are set or used either in general or for a particular patient. The system can offer the above explanations for previous sessions as well as the current one. These explanations are described in Chapters 2 and 3. The system also allows the user to change his answers (called "updating") to determine the effect of different inputs on the system's recommendations. When an answer is changed, the system recomputes the steps that are affected--it does not recompute everything. Once the affected steps have been recomputed, the system can provide the user with a concise explanation of the effects of the change. Updating is described in Chapter 4.

Originally, I had hoped that the Digitalis Advisor would be able to accept requests in English for explanations. Unfortunately, work on the OWL Parser[1] was delayed. Until it is completed, requests for explanations must be made in the following formal notation.

1. A part of the OWL system which converts natural English sentences into OWL forms.

Chapter 2: A Sample Session

In this chapter two sample sessions with the OWL Digitalis Advisor are outlined. The first session prescribes an initial dose for a patient with an arrhythmia. The second session is a follow-up session which occurs a few hours after the patient received his first dose. A number of features of the explanation facility are illustrated along the way. Output typed by the computer appears in upper case, while entries made by the user are in lower case. Comments about the session are in italics.

2.1 The Initial Session

```
ENTER AN OWL FORM TO BE CARRIED OUT
<>[(begin therapy)]
```

The user indicates to the system that he wishes to start therapy for a patient by typing the OWL form [(begin therapy)]. Eventually, the system will be able to accept this command and others in English.

```
WHAT IS THE AGE OF THE PATIENT?
ENTER NUMERICAL VALUE -----> 56

WHAT IS THE WEIGHT OF THE PATIENT?
UNITS: KG
ENTER NUMERICAL VALUE -----> 72

WHAT IS THE SEX OF THE PATIENT?
THE POSSIBILITIES ARE:
        1. MALE
        2. FEMALE
ENTER SINGLE VALUE -----> male
```

Note that for answers that are single words, the user may either enter the number of the answer or type it out.

WHAT IS THE CARDIAC RHYTHM?
THE POSSIBILITIES ARE:
1. NORMAL SINUS RHYTHM
2. ATRIAL FIBRILLATION
3. ATRIAL FLUTTER
4. PAT
5. PAT WITH BLOCK
6. NON-PAROXYSMAL JUNCTIONAL TACHYCARDIA
ENTER SINGLE VALUE ----> 2

IS THERE REGULARIZATION OF THE VENTRICULAR RHYTHM
THE POSSIBILITIES ARE:
1. PRESENT
2. ABSENT
ENTER SINGLE VALUE ----> why?

MY[2] TOP GOAL IS TO BEGIN THERAPY. I AM NOW TRYING TO CHECK THE CARDIAC RHYTHM. IF THE CARDIAC RHYTHM IS ATRIAL FIBRILLATION AND THERE IS VENTRICULAR-RHYTHM REGULARIZATION THIS SUGGESTS THAT THE PACEMAKING IMPULSES FOR THE VENTRICLES ARE NO LONGER COMING FROM THE ATRIA. IF THE STATUS OF VENTRICULAR-RHYTHM REGULARIZATION IS PRESENT IT IS POSSIBLE THERE MAY BE NON-PAROXYSMAL JUNCTIONAL TACHYCARDIA WHICH IS A SIGN OF DIGITALIS TOXICITY.

The system now re-asks the user for a value.
ENTER SINGLE VALUE ----> 2

Here, the user has asked the system to explain why it is asking this question. The system responds by converting the goals on its goal stack to English to produce the first two sentences. The system then finds and displays a canned explanation which is associated with the concept that the question is being asked about. The canned explanation usually provides information which could not be deduced from the code of the Advisor. Occasionally, the explanation provides a brief summary of information that the explanation routines can determine from the code.

---

2. The Digitalis Advisor refers to itself in the first person. Although some may object that this tends to anthropomorphize the computer, I feel that the explanations produced in this manner are much clearer than they would have been if they were stated in the third person or passive voice.

WHAT IS THE VENTRICULAR RATE?
ENTER NUMERICAL VALUE ----> 123

WHAT IS THE STATUS OF PVCS?
THE POSSIBILITIES ARE:
1. PRESENT
2. ABSENT
ENTER SINGLE VALUE ----> 2
PVCs are premature ventricular contractions. They can be an indication of digitalis toxicity.

WHAT IS THE REASON OF DIGITALIZATION?
THE POSSIBILITIES ARE:
1. CHF
2. ARRHYTHMIA
3. SURGICAL-PROPHYLACTIC
ENTER VALUE(S) ----> 2

The system needs to know the reason that the patient is receiving digitalis so that it can set an initial body stores goal (which may later be modified due to sensitivities or the patient's weight).

WHAT IS THE STATUS OF DIGITALIS USE?
THE POSSIBILITIES ARE:
1. PRESENT
2. ABSENT
ENTER SINGLE VALUE ----> absent

If the patient is already receiving digitalis, the system will take that into account when prescribing a new dosage regimen.

The next few questions ask about the patient's renal function in some detail. It is necessary to determine his renal function since a primary path for digitalis excretion is through the kidneys.
IS THE URINE OUTPUT OF THE PATIENT GREATER THAN 400CC PER DAY?
THE POSSIBILITIES ARE:
1. GREATER THAN 400
2. LESS THAN 400
ENTER SINGLE VALUE ----> 1

IS THE RENAL FUNCTION STABLE?
THE POSSIBILITIES ARE:
1. STABLE
2. UNSTABLE
ENTER SINGLE VALUE ----> 1

WHAT SORT OF RENAL FUNCTION MEASURE IS AVAILABLE?
THE POSSIBILITIES ARE:
1. SERUM-CREATININE
2. CREATININE-CLEARANCE
3. NONE
ENTER SINGLE VALUE ----> 2

ENTER RECENT CREATININE-CLEARANCE VALUES WITH DATE AND TIME. TO DELETE AN ITEM ENTER ITS DATE AND TIME, TYPE A RETURN WHEN TEST VALUE IS REQUESTED.

DATE	TIME	TEST VAL	RENAL F
9/21/76	10:00	100	0.00

DO YOU WANT TO ADD OR CHANGE ITEMS? (YES OR NO) n

The system now asks the user about several conditions that can cause the patient to be more sensitive to digitalis (that is, more likely to develop digitalis toxicity).
WHAT IS THE LEVEL OF SERUM CALCIUM?
ENTER NUMERICAL VALUE ----> 7

HAS IV CALCIUM BEEN ADMINISTERED WITHIN THE LAST TWO HOURS?
THE POSSIBILITIES ARE:
1. GIVEN
2. NOT GIVEN
ENTER SINGLE VALUE ----> 2

WHAT IS THE LEVEL OF SERUM POTASSIUM?
ENTER NUMERICAL VALUE ----> why?

MY TOP GOAL IS TO BEGIN THERAPY. ONE STEP IN DOING THAT IS TO CHECK SENSITIVITIES. I AM NOW TRYING TO CHECK SENSITIVITY DUE TO POTASSIUM. IF THE LEVEL OF SERUM POTASSIUM IS UNDER 3.70 IT WILL ENABLE THE BODY-STORES GOAL TO BE REDUCED SINCE A LOW POTASSIUM CONDITION WILL INCREASE DIGITALIS SENSITIVITY.

ENTER NUMERICAL VALUE ----> 4

IS THE PATIENT BEING GIVEN OR ABOUT TO BE GIVEN DIURETICS?
THE POSSIBILITIES ARE:
1. GIVEN
2. NOT GIVEN
ENTER SINGLE VALUE ----> 2

DOES THE PATIENT HAVE CARDIOMYOPATHY OR AN ACUTE MI?
THE POSSIBILITIES ARE:
1. PRESENT
2. ABSENT
3. SUSPECTED
ENTER SINGLE VALUE ----> 2

```
WHAT IS THE STATUS OF HYPOXEMIA?
THE POSSIBILITIES ARE:
        1. SUSPECTED
        2. NOT SUSPECTED
ENTER SINGLE VALUE ----> 2

WHAT IS THE STATUS OF MYXEDEMA?
THE POSSIBILITIES ARE:
        1. PRESENT
        2. ABSENT
        3. UNKNOWN
ENTER SINGLE VALUE ----> 1

HOW LONG DO YOU WISH TO TAKE TO REACH FULL DIGITALIZATION?
THE POSSIBILITIES ARE:
        1. INSTANTANEOUS RATE  (1-2 HOURS)
        2. RAPID RATE  (1 DAY)
        3. MODERATE RATE  (2-3 DAYS)
        4. SLOW RATE  (4-7 DAYS)
ENTER SINGLE VALUE ----> 3
```

Normally, the patient is loaded with digitalis gradually so that if any unforeseen toxic reactions develop, the amount of the overdose will be relatively small. If the patient is given enough digitalis to achieve the body stores goal all at once (the "instantaneous rate") and an overdose occurs, the amount of the overdose will be larger, with a corresponding increase in the patient's risk.

Finally, the system produces its recommendations.

```
PARAMETERS USED TO MAKE RECOMMENDATIONS:
BODY STORE GOAL ---->   8.564 MG
PROJECTED AMOUNT ON BOARD ---->   0.000 MG
BODY STORES SHIFT ---->   8.564 MG
REMAINING TIME TO REACH GOAL ---->     48.0 HRS.
HALF LIFE ---->    43.7 HOURS         1.8 DAYS
DAILY LOSS AT EQUILIBRIUM ---->   0.175 MG
ABSORPTION FACTOR ---->0.75


HOW OFTEN DO YOU WISH TO ADMINISTER DIGITALIS?
THE POSSIBILITIES ARE:
        1. QD
        2. BID
        3. TID
        4. QID
        5. Q4H
        6. Q8H
ENTER SINGLE VALUE ----> 3
```

The system asks the user how often he wishes to give digitalis. The expression "tid" means three times per day.

THE DOSAGE RECOMMENDATIONS ARE:

HOURS FROM NOW	ORAL	IV
NOW	.25 MG	.125 + .0625 MG

REPORT BACK AFTER THE FIRST DOSE.

8	.125 + .0625 MG	.125 MG
16	.125 MG	.125 MG
24	.125 MG	.125 MG
32	.125 MG	.0625 MG
40	.125 MG	.0625 MG

ORAL MAINTENANCE SCHEDULE:

.25 MG

IV MAINTENANCE SCHEDULE:

ALTERNATE .25 AND .125 MG

The system produces recommendations in amounts that represent actual pill sizes by finding the pill or combination of pills that comes closest to the ideal amount of digitalis the patient should receive. This feature, first incorporated in Silverman's Advisor, represents an improvement over earlier digitalis advisors.

2.2 The Follow-up Session

The follow up session starts here, approximately 4 hours later. As before, the user types an OWL form to indicate to the OWL interpreter what he wants to do.

```
ENTER AN OWL FORM TO BE CARRIED OUT
<>[(obtain follow-up-info)]
```

The system asks several questions about the patient's heartbeat.

```
WHAT IS THE CARDIAC RHYTHM?
THE POSSIBILITIES ARE:
        1. NORMAL SINUS RHYTHM
        2. ATRIAL FIBRILLATION
        3. ATRIAL FLUTTER
        4. PAT
        5. PAT WITH BLOCK
        6. NON-PAROXYSMAL JUNCTIONAL TACHYCARDIA
ENTER SINGLE VALUE ----> 2

IS THERE REGULARIZATION OF THE VENTRICULAR RHYTHM?
THE POSSIBILITIES ARE:
        1. PRESENT
        2. ABSENT
ENTER SINGLE VALUE ----> 2

WHAT IS THE VENTRICULAR RATE?
ENTER NUMERICAL VALUE ----> 105
```

The heart rate has decreased. This is a sign of therapeutic effect.

```
WHAT IS THE STATUS OF PVCS?
THE POSSIBILITIES ARE:
        1. PRESENT
        2. ABSENT
ENTER SINGLE VALUE ----> 2

THE DIGITALIS HISTORY OF THE PATIENT IS NEEDED.
TO DELETE AN ENTRY, RE-ENTER THE DATE AND TIME AND TYPE RETURN ALONE
WHEN THE DOSE IS REQUESTED.  WHEN DONE TYPE RETURN ALONE WHEN DATE IS
REQUESTED.  ENTER ALL DOSES SINCE THE LAST SESSION
DATE            TIME            DOSE            TYPE
9/21/76         11:30           0.25            PO
DO YOU WANT TO CHANGE OR ADD ITEMS?  (YES OR NO) n
```

The user informs the system of the time and amount of the single digitalis dose the patient received.

ARE ANY OF THE FOLLOWING THE CONDITIONS PRESENT OR LIKELY TO APPEAR?
THE POSSIBILITIES ARE:

1. HYPOKALEMIA
2. HYPOKEMIA
3. CARDIOMYOPATHIES-MI
4. POTENTIAL POTASSIUM LOSS DUE TO DIURETICS
5. NONE

ENTER VALUE(S) ====> 1

The Advisor recalls those conditions from the initial session which can degrade, and asks the user if any of them have appeared or become worse. Since the user responded that hypokalemia might become worse the system will ask about it in detail later. The next question asks about any conditions that the patient was showing during the previous session that might have improved.

HAVE ANY OF THE FOLLOWING THE CONDITIONS IMPROVED?
THE POSSIBILITIES ARE:

1. NONE
2. MYXEDEMA

ENTER VALUE(S) ====> 1

Since the user indicated that the patient might be showing signs of hypokalemia, the system now asks him about serum potassium.
WHAT IS THE LEVEL OF SERUM POTASSIUM?
ENTER NUMERICAL VALUE ====> why?

MY TOP GOAL IS TO OBTAIN THE FOLLOW-UP INFORMATION. ONE STEP IN DOING THAT IS TO ADJUST FOR CHANGE IN SENSITIVITIES. I AM NOW TRYING TO CHECK SENSITIVITY DUE TO POTASSIUM. IF THE LEVEL OF SERUM POTASSIUM IS UNDER 3.70 IT WILL CAUSE THE BODY-STORES GOAL TO BE REDUCED SINCE A LOW POTASSIUM CONDITION WILL INCREASE DIGITALIS SENSITIVITY.

Note that the answer to the "why" question is different from the answer given during the first session when the user asked the system why it was asking about serum potassium. The difference is due to the different goal structures that lead to the question.
ENTER NUMERICAL VALUE ====> 3

IS THE PATIENT BEING GIVEN OR ABOUT TO BE GIVEN DIURETICS?
THE POSSIBILITIES ARE:

1. GIVEN
2. NOT GIVEN

ENTER SINGLE VALUE ====> 2
IT IS GENERALLY AGREED THAT PATIENTS WITH LOW SERUM POTASSIUM LEVELS ARE MORE PRONE TO DEVELOP DIG TOXICITY. PLEASE WATCH THIS PATIENT CAREFULLY AND ADMINISTER POTASSIUM SUPPLEMENTS.
The system warns the user to try to correct the patient's hypokalemia.

HAS THERE BEEN A CHANGE IN RENAL FUNCTION?
THE POSSIBILITIES ARE:
1. PRESENT
2. ABSENT
ENTER SINGLE VALUE ----> 2
The system asks about any changes in renal function.

IS A SERUM DIG LEVEL AVAILABLE?
THE POSSIBILITIES ARE:
1. AVAILABLE
2. NOT AVAILABLE
ENTER SINGLE VALUE ----> 2

A serum digitalis level is a measurement of the amount of digitalis in the patient.

ARE THERE SIGNS OF EITHER NAUSEA, ANOREXIA, OR VISUAL
DISTURBANCE PRESENT?
THE POSSIBILITIES ARE:
1. PRESENT
2. ABSENT
ENTER SINGLE VALUE ----> 2

THIS PATIENT IS SHOWING NO TOXIC EFFECTS. THE PATIENT IS
SHOWING ONLY PARTIAL THERAPEUTIC EFFECT. SINCE THE PATIENT IS IN
THE LOADING STAGE IT IS BEST TO CONTINUE THE LOADING AND MAINTENANCE
REGIMEN BELOW.

The system observes that the patient is showing a partial therapeutic effect, and no toxic effects.

PARAMETERS USED TO MAKE RECOMMENDATIONS:
BODY STORE GOAL ---> 8.371 MG
Note that the body store goal has dropped due to the patient's hypokalemia.
PROJECTED AMOUNT ON BOARD ---> 8.177 MG
BODY STORES SHIFT ---> 8.194 MG
REMAINING TIME TO REACH GOAL ---> 44.2 HRS.
HALF LIFE ---> 43.7 HOURS 1.8 DAYS
DAILY LOSS AT EQUILIBRIUM ---> 8.117 MG
ABSORPTION FACTOR --->8.75

DO YOU STILL WISH TO GIVE DIGITALIS TIDY [illegible]

THE DOSAGE RECOMMENDATIONS ARE:

HOURS FROM NOW	ORAL	IV
5	.125 MG	.0625 MG
13	.0625 MG	.0625 MG
21	.125 MG	.0625 MG
29	.0625 MG	.0625 MG
37	.0625 MG	.0625 MG

ORAL MAINTENANCE SCHEDULE:

ALTERNATE .25 AND .125 MG

IV MAINTENANCE SCHEDULE:

.125 MG

2.3 Explanations

Some of the explanation capabilities are shown below. Since the system cannot yet accept English input, the English questions asked by the user are listed in italics, followed by the LISP form actually used to produce the explanation. All the explanations in this section are produced by examining the actual OWL I code and the event structure created by the interpreter. They are not canned explanations.

2.3.1 Explaining Methods

"How do you check sensitivities?"

```
(describe-method [(check sensitivities)])
```

```
TO CHECK SENSITIVITIES I DO THE FOLLOWING STEPS:
1. I CHECK SENSITIVITY DUE TO CALCIUM.
2. I CHECK SENSITIVITY DUE TO POTASSIUM.
3. I CHECK SENSITIVITY DUE TO CARDIOMYOPATHY-MI.
4. I CHECK SENSITIVITY DUE TO HYPOXEMIA.
5. I CHECK SENSITIVITY DUE TO THYROID-FUNCTION.
6. I CHECK SENSITIVITY DUE TO ADVANCED AGE.
7. I COMPUTE THE FACTOR OF ALTERATION.
```

This is a good example of the way the Digitalis Advisor is structured to control the amount of information given the user. When the user asks how the program checks for sensitivities, the program lists several more specific routines that check for special types of sensitivities. While explaining the general method, the system does not indicate how the more specialized routines work, but the user is free to ask about those routines that interest him (as he does below).

"How do you check sensitivity due to thyroid-function?"

```
(describe-method [(check (sensitivity (due (to thyroid-function))))])
```

```
TO CHECK SENSITIVITY DUE TO THYROID-FUNCTION I DO THE FOLLOWING STEPS:

1. IF THE CURRENT VALUE OF THE STATUS OF MYXEDEMA IS UNKNOWN THEN I ASK
THE USER THE LEVEL OF T4.
```

2. I DO ONE OF THE FOLLOWING:

2.1 IF EITHER THE STATUS OF MYXEDEMA IS PRESENT OR THE STATUS OF MYXEDEMA IS UNKNOWN AND THE LEVEL OF T4 IS LESS THAN 2.60 THEN I DO THE FOLLOWING SUBGOALS:

2.1.1 I ADD MYXEDEMA TO THE PRESENT AND CORRECTABLE CONDITIONS.

The present and correctable conditions is a set of conditions that the patient is exhibiting, but that may become better.

2.1.2 I REMOVE MYXEDEMA FROM THE DEGRADEABLE CONDITIONS.

The degradeable conditions represent those conditions that may become worse.

2.1.3 I SET THE FACTOR OF REDUCTION DUE TO MYXEDEMA TO 0.67.

2.1.4 I ADD MYXEDEMA TO THE REASONS OF REDUCTION.

2.2 OTHERWISE, I ADD MYXEDEMA TO THE DEGRADEABLE CONDITIONS, REMOVE MYXEDEMA FROM THE PRESENT AND CORRECTABLE CONDITIONS, SET THE FACTOR OF REDUCTION DUE TO MYXEDEMA TO 1.00 AND REMOVE MYXEDEMA FROM THE REASONS OF REDUCTION.

"How do you check sensitivity due to potassium?"

```
(describe-method [(check (sensitivity (due (to potassium))))])
```

This is the longest single explanation of a plan.

TO CHECK SENSITIVITY DUE TO POTASSIUM I DO THE FOLLOWING STEPS:

1. I ASK THE USER THE LEVEL OF SERUM POTASSIUM.

2. I ASK THE USER THE STATUS OF DIURETIC USE.

3. IF THE PATIENT IS RECEIVING DIURETICS THEN I ASK THE USER THE TYPE OF

DIURETIC USE AND ASK THE USER THE STATUS OF POTASSIUM SUPPLEMENT USE.

4. I DO ONE OF THE FOLLOWING:

4.1 IF THE LEVEL OF SERUM POTASSIUM IS LESS THAN 3.70 THEN I DO THE FOLLOWING SUBSTEPS:

4.1.1 I SET THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA TO 0.67.

4.1.2 I ADD HYPOKALEMIA TO THE PRESENT AND CORRECTABLE CONDITIONS.

4.1.3 I REMOVE HYPOKALEMIA FROM THE DEGRADEABLE CONDITIONS.

4.1.4 I ADD HYPOKALEMIA TO THE REASONS OF REDUCTION.

4.1.5 I SUGGEST WATCHING FOR TOXICITY DUE TO HYPOKALEMIA.

4.2 OTHERWISE, I ADD HYPOKALEMIA TO THE DEGRADEABLE CONDITIONS, REMOVE HYPOKALEMIA FROM THE PRESENT AND CORRECTABLE CONDITIONS, REMOVE HYPOKALEMIA FROM THE REASONS OF REDUCTION AND SET THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA TO 1.00.

5. IF THE LEVEL OF SERUM POTASSIUM IS LESS THAN 3.70, THE PATIENT IS RECEIVING DIURETICS, AND THE PATIENT IS NOT RECEIVING POTASSIUM SUPPLEMENTS THEN I SUGGEST POTASSIUM SUPPLEMENT USE.

6. I DO ONE OF THE FOLLOWING:

6.1 IF THE PATIENT IS RECEIVING DIURETICS, THE PATIENT IS NOT RECEIVING POTASSIUM SUPPLEMENTS, AND THE TYPE OF DIURETIC USE IS ACUTE THEN I DO THE FOLLOWING SUBSTEPS:

6.1.1 I SET THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS TO 0.67.

6.1.2 I ADD POTENTIAL POTASSIUM LOSS DUE TO DIURETICS TO THE REASONS OF REDUCTION.

6.1.3 I ADD POTENTIAL POTASSIUM LOSS DUE TO DIURETICS TO THE PRESENT AND CORRECTABLE CONDITIONS.

6.1.4 I REMOVE POTENTIAL POTASSIUM LOSS DUE TO DIURETICS FROM THE DEGRADEABLE CONDITIONS.

6.1.5 I SUGGEST WATCHING FOR TOXICITY DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS.

6.2 OTHERWISE, I ADD POTENTIAL POTASSIUM LOSS DUE TO DIURETICS TO THE DEGRADEABLE CONDITIONS, REMOVE POTENTIAL POTASSIUM LOSS DUE TO DIURETICS FROM THE PRESENT AND CORRECTABLE CONDITIONS, SET THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS TO 1.00 AND REMOVE POTENTIAL POTASSIUM LOSS DUE TO DIURETICS FROM THE REASONS OF REDUCTION.

7. IF THE LEVEL OF SERUM POTASSIUM IS LESS THAN 3.70, THE PATIENT IS RECEIVING DIURETICS, AND THE PATIENT IS RECEIVING POTASSIUM SUPPLEMENTS THEN I SUGGEST INCREASING POTASSIUM SUPPLEMENT.

2.3.2 Explaining Events

The system can also describe events. That is, it can tell the user what happened during a particular session. In addition to its intended use by physicians, this feature has been very useful in developing the GWL Digitalis Advisor, because it may be used like an "ex post facto" trace facility. Several examples are given below.

"How did you check sensitivity due to thyroid function for this patient?"

(describe-event [(check (sensitivity (due (to thyroid-function))))])

DO YOU ONLY WANT TO SEE EVENTS FROM THE CURRENT SESSION? (YES OR NO) n

The system can explain events from previous sessions as well as the current session, since it remembers all earlier sessions. In this case, only one event from a previous session is shown, since the method for checking sensitivity due to thyroid-function is executed only once.

DURING THE SESSION ON 9/21/76 AT 11:18, I CHECKED SENSITIVITY DUE TO THYROID-FUNCTION BY EXECUTING THE FOLLOWING STEPS:

1. I ASKED THE USER THE STATUS OF MYXEDEMA. THE USER RESPONDED THAT THE STATUS OF MYXEDEMA WAS PRESENT.

2. SINCE THE STATUS OF MYXEDEMA WAS PRESENT I DID THE FOLLOWING:

2.1 I ADDED MYXEDEMA TO THE PRESENT AND CORRECTABLE CONDITIONS. THE PRESENT AND CORRECTABLE CONDITIONS THEN BECAME MYXEDEMA.

2.2 I REMOVED MYXEDEMA FROM THE DEGRADEABLE CONDITIONS. THE DEGRADEABLE CONDITIONS THEN BECAME HYPOKALEMIA, HYPOXEMIA, CARDIOMYOPATHIES-MI, AND POTENTIAL POTASSIUM LOSS DUE TO DIURETICS.

2.3 I SET THE FACTOR OF REDUCTION DUE TO MYXEDEMA TO 0.67. THE FACTOR OF REDUCTION DUE TO MYXEDEMA WAS PREVIOUSLY UNDETERMINED.

2.4 I ADDED MYXEDEMA TO THE REASONS OF REDUCTION. THE REASONS OF REDUCTION THEN BECAME MYXEDEMA.

"How did you check sensitivity due to potassium for this patient?"

```
(describe-event '(check (sensitivity (due (to potassium)))))
```

DO YOU ONLY WANT TO SEE EVENTS FROM THE CURRENT SESSION? (YES OR NO) n

DURING THE SESSION ON 9/21/76 AT 11:10, I CHECKED SENSITIVITY DUE TO POTASSIUM BY EXECUTING THE FOLLOWING STEPS:

1. I ASKED THE USER THE LEVEL OF SERUM POTASSIUM. THE USER RESPONDED THAT THE LEVEL OF SERUM POTASSIUM WAS 4.

2. I ASKED THE USER THE STATUS OF DIURETIC USE. THE USER RESPONDED THAT THE PATIENT WAS NOT RECEIVING DIURETICS.

3. SINCE THE LEVEL OF SERUM POTASSIUM WAS NOT LESS THAN 3.70, I DID THE FOLLOWING:

3.1 I SET THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA TO 1.00. THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA WAS PREVIOUSLY UNDETERMINED.

4. SINCE THE PATIENT WAS NOT RECEIVING DIURETICS, I DID THE FOLLOWING:

4.1 I SET THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS TO 1.00. THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS WAS PREVIOUSLY UNDETERMINED.

DURING THE CURRENT SESSION, I CHECKED SENSITIVITY DUE TO POTASSIUM BY EXECUTING THE FOLLOWING STEPS:

1. I ASKED THE USER THE LEVEL OF SERUM POTASSIUM. THE USER RESPONDED THAT THE LEVEL OF SERUM POTASSIUM WAS 3.

2. I ASKED THE USER THE STATUS OF DIURETIC USE. THE USER RESPONDED THAT THE PATIENT WAS NOT RECEIVING DIURETICS.

3. SINCE THE LEVEL OF SERUM POTASSIUM WAS LESS THAN 3.70 I DID THE FOLLOWING:

3.1 I SET THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA TO 0.67. THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA WAS PREVIOUSLY 1.00.

3.2 I ADDED HYPOKALEMIA TO THE PRESENT AND CORRECTABLE CONDITIONS. THE PRESENT AND CORRECTABLE CONDITIONS THEN BECAME MYXEDEMA AND HYPOKALEMIA.

3.3 I REMOVED HYPOKALEMIA FROM THE DEGRADEABLE CONDITIONS. THE DEGRADEABLE CONDITIONS THEN BECAME HYPONEMIA, CARDIOMYOPATHIES-MI, AND POTENTIAL POTASSIUM LOSS DUE TO DIURETICS.

3.4 I ADDED HYPOKALEMIA TO THE REASONS OF REDUCTION. THE REASONS OF REDUCTION THEN BECAME MYXEDEMA AND HYPOKALEMIA.

3.5 I SUGGESTED WATCHING FOR TOXICITY DUE TO HYPOKALEMIA.

4. SINCE THE PATIENT WAS NOT RECEIVING DIURETICS, I DID THE FOLLOWING:

4.1 I SET THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS TO 1.00. THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS DID NOT CHANGE.

"How did you compute the factor of alteration for this case?"

(describe-event [(compute (factor alteration))])

DO YOU ONLY WANT TO SEE EVENTS FROM THE CURRENT SESSION? (YES OR NO) n

DURING THE SESSION ON 9/21/76 AT 11:10, I COMPUTED THE FACTOR OF ALTERATION BY EXECUTING THE FOLLOWING STEPS:

1. I SET THE FACTOR OF ALTERATION DUE TO SENSITIVITIES TO THE PRODUCT OF THE FACTOR OF REDUCTION DUE TO ADVANCED AGE (1.00), THE FACTOR OF REDUCTION DUE TO HYPERCALCEMIA (1.00), THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA (1.00), THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS (1.00), THE FACTOR OF REDUCTION DUE TO HYPOXEMIA (1.00), THE FACTOR OF REDUCTION DUE TO MYXEDEMA (0.67), AND THE FACTOR OF REDUCTION DUE TO CARDIOMYOPATHY-MI (1.00). THE FACTOR OF ALTERATION DUE TO SENSITIVITIES CHANGED FROM UNDETERMINED TO 0.67.

2. SINCE THE IDEAL WEIGHT OF THE PATIENT WAS UNDETERMINED I SET THE FACTOR OF ALTERATION TO THE PRODUCT OF THE FACTOR OF ALTERATION DUE TO SENSITIVITIES (0.67) AND THE QUOTIENT OF THE WEIGHT OF THE PATIENT (72) AND 70.00. THE FACTOR OF ALTERATION CHANGED FROM UNDETERMINED TO 0.69.

Note that when a numerical variable is used in a computation, the value of the variable is printed in parentheses following the variable.

DURING THE CURRENT SESSION, I COMPUTED THE FACTOR OF ALTERATION BY EXECUTING THE FOLLOWING STEPS:

1. I SET THE FACTOR OF ALTERATION DUE TO SENSITIVITIES TO THE PRODUCT OF THE FACTOR OF REDUCTION DUE TO ADVANCED AGE (1.00), THE FACTOR OF REDUCTION DUE TO HYPERCALCEMIA (1.00), THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA (0.67), THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS (1.00), THE FACTOR OF REDUCTION DUE TO HYPOXEMIA (1.00), THE FACTOR OF REDUCTION DUE TO MYXEDEMA (0.67), AND THE FACTOR OF REDUCTION DUE TO CARDIOMYOPATHY-MI (1.00). THE FACTOR OF ALTERATION DUE TO SENSITIVITIES CHANGED FROM 0.67 TO 0.45.

2. SINCE THE IDEAL WEIGHT OF THE PATIENT WAS UNDETERMINED I SET THE FACTOR OF ALTERATION TO THE PRODUCT OF THE FACTOR OF ALTERATION DUE TO SENSITIVITIES (0.45) AND THE QUOTIENT OF THE WEIGHT OF THE PATIENT (72) AND 70.00. THE FACTOR OF ALTERATION CHANGED FROM 0.69 TO 0.46.

2.3.3 Explaining How a Variable is Used in General

The system can also explain how a particular variable is used by the system either in plans or events. The system distinguishes between the setting of a variable and the evaluation of the variable, and different explanation routines are used to describe each process. The example below describes all the ways the variable [(REASONS REDUCTION)] is used (i.e. evaluated) in the ONC Digitalis Advisor.

"In general, how do you use the reasons of reduction?"

(describe-use-in-method [(reasons reduction)])

I USE THE REASONS OF REDUCTION IN THE FOLLOWING WAYS:

WHILE TREATING DEFINITE TOXICITY I DO THE FOLLOWING STEP:

1. IF EITHER ONE OF THE REASONS OF REDUCTION IS HYPOXEMIA, ONE OF THE REASONS OF REDUCTION IS HYPOKALEMIA, OR ONE OF THE REASONS OF REDUCTION IS POTENTIAL POTASSIUM LOSS DUE TO DIURETICS THEN I SAY THE SENTENCE "SINCE THE PATIENT HAS A CORRECTABLE CONDITION WHICH MAY BE CONTRIBUTING TO THIS TOXIC RESPONSE TRY TO CORRECT THE CONDITION AS SOON AS POSSIBLE".

WHILE SUGGESTING DIGITALIS THERAPY IS NOT APPROPRIATE I DO THE FOLLOWING STEP:

1. IF HYPOXEMIA IS NOT ONE OF THE REASONS OF REDUCTION AND HYPOKALEMIA IS NOT ONE OF THE REASONS OF REDUCTION THEN I SAY THE SENTENCE "SINCE RESPONSE OF THE PATIENT IS TOXIC AND NO THERAPEUTIC EFFECTS ARE OBSERVED IT IS LIKELY THAT DIGITALIS IS NOT AN APPROPRIATE AGENT FOR USE IN THIS INSTANCE".

2.3.4 Explaining How a Variable is Set in General

This question asks the system to explain all the ways that the body stores goal can be set.

"How do you set the body stores goal?"

(describe-set-in-method [(quanta body-stores-goal)])

I SET THE BODY-STORES GOAL IN THE FOLLOWING WAYS:

WHILE COMPUTING THE BODY-STORES GOAL I DO THE FOLLOWING STEP:

1. I SET THE BODY-STORES GOAL TO THE PRODUCT OF THE FACTOR OF ALTERATION AND THE BASIC BODY-STORES GOAL.

WHILE TREATING NO TOXICITY ACCOMPANIED-BY DEFINITE-THERAPEUTIC-EFFECT I DO THE FOLLOWING STEP:

1. I DO ONE OF THE FOLLOWING:

    1.1 IF THE PHASE OF TREATMENT IS LOADING-STAGE THEN I DO THE FOLLOWING SUBSTEPS:

        1.1.1 I SWITCH TO MAINTENANCE.

        1.1.2 I SAY THE SENTENCE "DISCONTINUE THE LOADING PROGRAM AND PLACE THE PATIENT ON THE MAINTENANCE PROGRAM OUTLINED BELOW".

    1.2 OTHERWISE, I SAY THE SENTENCE "CONTINUE THE MAINTENANCE PROGRAM AND REPORT ANY CHANGES" AND SET THE BODY-STORES GOAL TO THE QUOTIENT OF THE LEVEL OF THE PROJECTED AMOUNT OF DIGITALIS IN THE PATIENT AND THE FACTOR OF ALTERATION.

2.3.5 Explaining How a Variable was Used in Particular

This is a question asking how the factor of alteration was used for this particular patient. Note that events from the previous session are found and displayed as well as those from the current session.

"How did you use the factor of alteration in this case?"

(describe-use-in-event [(factor alteration)])

DO YOU ONLY WANT TO SEE EVENTS FROM THE CURRENT SESSION? (YES OR NO) n

DURING THE SESSION ON 9/21/76 AT 11:10, I USED THE FACTOR OF ALTERATION IN THE FOLLOWING WAY WHILE COMPUTING THE BODY-STORES GOAL:

1. I SET THE BODY-STORES GOAL TO THE PRODUCT OF THE FACTOR OF ALTERATION (0.69) AND THE BASIC BODY-STORES GOAL (0.80). THE BODY-STORES GOAL CHANGED FROM UNDETERMINED TO 0.55.

DURING THE CURRENT SESSION, I USED THE FACTOR OF ALTERATION IN THE FOLLOWING WAY WHILE COMPUTING THE BODY-STORES GOAL:

1. I SET THE BODY-STORES GOAL TO THE PRODUCT OF THE FACTOR OF ALTERATION(0.46) AND THE BASIC BODY-STORES GOAL (0.80). THE BODY-STORES GOAL CHANGED FROM 0.55 TO 0.37.

I USED THE FACTOR OF ALTERATION IN THE FOLLOWING WAY WHILE COMPUTING THE BODY-STORES GOAL:

1. I SET THE BODY-STORES GOAL TO THE PRODUCT OF THE FACTOR OF ALTERATION (0.46) AND THE BASIC BODY-STORES GOAL (0.80). THE BODY-STORES GOAL DID NOT CHANGE FROM 0.37.

2.3.6 Explaining How a Variable was Set in Particular

This is the corresponding question asking how the factor of alteration was set.

"How did you set the factor of alteration in this case?"

```
(describe-set-in-event [(factor alteration)])
```

DO YOU ONLY WANT TO SEE EVENTS FROM THE CURRENT SESSION? (YES OR NO) n

DURING THE SESSION ON 9/21/76 AT 11:10, I USED THE FACTOR OF ALTERATION IN THE FOLLOWING WAY WHILE COMPUTING THE FACTOR OF ALTERATION:

1. SINCE THE IDEAL WEIGHT OF THE PATIENT WAS UNDETERMINED I SET THE FACTOR OF ALTERATION TO THE PRODUCT OF THE FACTOR OF ALTERATION DUE TO SENSITIVITIES (0.67) AND THE QUOTIENT OF THE WEIGHT OF THE PATIENT (72) AND 70.00. THE FACTOR OF ALTERATION CHANGED FROM UNDETERMINED TO 0.69.

DURING THE CURRENT SESSION, I USED THE FACTOR OF ALTERATION IN THE FOLLOWING WAY WHILE COMPUTING THE FACTOR OF ALTERATION:

1. SINCE THE IDEAL WEIGHT OF THE PATIENT WAS UNDETERMINED I SET THE FACTOR OF ALTERATION TO THE PRODUCT OF THE FACTOR OF ALTERATION DUE TO SENSITIVITIES (0.45) AND THE QUOTIENT OF THE WEIGHT OF THE PATIENT (72) AND 70.00. THE FACTOR OF ALTERATION CHANGED FROM 0.69 TO 0.46.

2.3.7 Explaining How a Method may be Called

The system can also inform the user of the way that a plan or event was called. In the example below, we first see all the possible ways that a plan may be called, followed by an explanation of the way that it was called for this patient.

"When do you check sensitivity due to potassium?"

```
(find-why-method [(check (sensitivity (due (to potassium))))])
```

I CALL CHECK SENSITIVITY DUE TO POTASSIUM IN THE FOLLOWING WAYS:

WHILE CHECKING SENSITIVITIES I DO THE FOLLOWING STEP:

1. I CHECK SENSITIVITY DUE TO POTASSIUM.

WHILE ADJUSTING FOR CHANGE IN SENSITIVITIES I DO THE FOLLOWING STEPS:

1. IF ONE OF THE IMPROVED CONDITIONS IS HYPOKALEMIA THEN I CHECK SENSITIVITY DUE TO POTASSIUM.

2. IF ONE OF THE WORSENED CONDITIONS IS HYPOKALEMIA THEN I CHECK SENSITIVITY DUE TO POTASSIUM.

2.3.8 Explaining Why a Method was Called

"When did you check sensitivity due to potassium for this patient?"

(find-why-event [(check (sensitivity (due (to potassium))))])

DO YOU ONLY WANT TO SEE EVENTS FROM THE CURRENT SESSION? (YES OR NO) n

DURING THE SESSION ON 9/21/76 AT 11:10, I CALLED CHECK SENSITIVITY DUE TO POTASSIUM IN THE FOLLOWING WAY WHILE CHECKING SENSITIVITIES:

1. I CHECKED SENSITIVITY DUE TO POTASSIUM. THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA WAS 1.00 AND THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS WAS 1.00.

DURING THE CURRENT SESSION, I CALLED CHECK SENSITIVITY DUE TO POTASSIUM IN THE FOLLOWING WAY WHILE ADJUSTING FOR CHANGE IN SENSITIVITIES:

1. SINCE THE WORSENED CONDITIONS WAS HYPOKALEMIA I CHECKED SENSITIVITY DUE TO POTASSIUM. THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA WAS 0.67 AND THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS WAS 1.00.

Chapter 3: Explanation -- How It's Done

3.1 Introduction

Several features of the OWL I language and the structure of the Digitalis Advisor itself make it possible to produce explanations conveniently. First, the OWL I interpreter and data base provide a number of data structures that are helpful in constructing explanations. Second, since the form of OWL I expressions is close to English, a relatively simple program can be used to generate English from OWL I. Third, the program structure of the Advisor attempts to model the problem solving techniques used by expert cardiologists. Fourth, the use of alternate models allows the system to provide the user with different perspectives. These features will be described in detail in this chapter.

This chapter will also describe the functions that explain events and plans, those that describe how the plans are called and why the events were created, and those that describe the use and setting of variables in events and plans. Updating will be described in Chapter 4.

3.2 The OWL Knowledge Base and Interpreter

OWL I has a number of features which facilitate producing English explanations of OWL I code. The entire OWL I system is too complex to be described here, however, I will attempt to outline the basics of OWL I, and describe in some detail those features that are particularly important in making explanations. If the reader desires a deeper understanding of the linguistic theory on which the OWL project is based he can consult Martin[20] and Hawkinson[14].

3.2.1 The Knowledge Base

Almost all the information that an OWL I system possesses resides in the knowledge base. Here one may find programs, traces of programs, hierarchical structures of English, and so forth. All the modules that make up the OWL I system have access to the knowledge base and they communicate through it. In this way, any module may determine the state of the world at any time. A conscious effort has been made to avoid "hiding" information in LISP recursive push-down stacks and similar internal structures.

All the information in the knowledge base is represented as *concepts*. A concept has three parts: a *generalizer*, a *specializer*, and a *reference list*. The concepts are organized in a hierarchy. The generalizer of a concept is a link to a concept which is higher in the hierarchy. It corresponds to the "a-kind-of" link in other very high level languages. The specializer of a concept is also a link to a concept[3]. The specializer of a concept is the chief feature or notion that makes that concept different from others of its class. The reference list of a concept is, as the name implies, a list of all the references to the concept. All the concepts which use some other concept as generalizer or specializer will appear on the reference list of that concept. In addition, the value (if any) of a concept will be on the reference list as well as all those concepts which use that concept as a value. The OWL Knowledge Base Handler causes all the items above to appear on reference lists automatically. In addition, the user may place a concept on the reference list of another concept explicitly.

When printed by the OWL printer, concepts appear enclosed in brackets as two-tuples followed by their reference lists:

---

3. There is one exception. The specializer of a concept may also be a link to a *symbol*. Symbols are character strings which roughly correspond to English words. Symbols are used to place English words in the knowledge base.

```
[(generalizer specializer)
    reference-item-1
    reference-item-2
    reference-item-3
        .
        .
        .
    reference-item-n]
```

Perhaps a few examples taken from the Digitalis Advisor will help to clear all this up. As the Advisor computes the body stores goal[4], it adjusts the amount based on the factor of alteration. In OWL I, the factor of alteration appears as [(FACTOR ALTERATION)]. FACTOR is the generalizer, while ALTERATION is the specializer. [(FACTOR ALTERATION)] is automatically placed on the reference lists of both FACTOR and ALTERATION by the OWL Knowledge Base Handler:

```
[ALTERATION
    (FACTOR ALTERATION)]

[FACTOR
    (FACTOR ALTERATION)]
```

It is possible to find all the places that a concept C_i is used by examining the concepts on the reference list of C_i and then recursively examining their reference lists. This feature of the knowledge base makes it quite easy to provide explanations that tell how a particular variable is used.

3.2.2 The OWL I Interpreter

Programs may be written in OWL I. These programs are stored in the OWL knowledge base and they are run by the OWL I interpreter. The representation of OWL I programs follows the same conventions outlined above. The OWL I interpreter has not been extensively

---

4. The body stores goal is the amount of digitalis that should be "in" the patient.

documented; however, some information may be found in unpublished papers by Long[13] and Sunguroff[21].

A *plan* in OWL I corresponds to a procedure in other programming languages. A plan is a kind of predicate[5]. A series of steps is linked to the plan. These steps may be basic OWL I primitives or calls to other plans. A (somewhat simplified) plan from the Digitalis Advisor is shown below:

```
[(CHECK (SENSITIVITY (DUE (TO ADVANCED-AGE))))
 METHOD: <--- (OR
               (IF-THEN (GREATER-THAN 70 (AGE PATIENT))
                        (BECOME (FACTOR REDUCTION-ADVANCED-AGE 0.75)))
               (BECOME (FACTOR REDUCTION-ADVANCED-AGE 1.0))))]
```

3.1 An OWL Plan

This is a plan for checking for digitalis sensitivity due to advanced age. It says that if the age of the patient is greater than 70 then the factor of reduction due to advanced age is set to 0.75, otherwise, it is set to 1.0.

It is not necessary that the reader understand all the details of the representation, however, I will outline a few major points. The name of the plan appears immediately to the right of the left bracket, (i.e. [(CHECK (SENSITIVITY (DUE (TO ADVANCED-AGE))))]), while the steps of the plan follow the concept METHOD. An OR in OWL I works much like a COND in LISP, that is, if the predicate of the first clause is not true the next clause is examined and so on, until the first clause which either has no predicate or a true one is found. A BECOME statement is an OWL I primitive used for making assertions.

Plans are invoked by *calls* in OWL I. As in PLANNER[22], calls are matched to plans by a pattern matching mechanism. Depending on the state of the world, the same call may invoke two different plans at two different times.

---

5. Predicates in OWL I may be thought of as verbs in English. See Martin[20] for a more complete discussion.

As a plan executes, an event is created. The event may be thought of as a trace of the execution of the plan. Each event is unique to a particular execution of a plan. The event structure contains information concerning when the plan started and stopped executing, which plan was used, which call invoked it, and what events started execution during the event. Events are also created by the execution of many of the OWL system primitives, such as IF-THEN, OR, and AND. The calls for these events are the OWL steps that created them. The plans for these events are internally defined within the OWL system.

3.2.3 OWL I and Explanation

It should be clear that OWL I provides the user with a number of features useful in producing explanations. First, the fact that OWL I is an English-based language makes it relatively easy to translate programs into English. Second, since all the system's knowledge resides in one place and in one representation, it is easier to find objects and determine the relationships between them. Third, the events created by the interpreter make it possible to describe what happened. The sections below will describe in more detail how these features are put to use.

3.B The English Generator — Turning OWL I into English

The English generator is a module in the OWL I system that converts simple OWL I expressions into English. The generator is a simple program. Although it knows about the tenses of verbs, it does not try to achieve subject-verb agreement, or perform any relatively sophisticated operations like pronoun substitution. In the Digitalis Advisor, the explanation routines break apart fairly complex programming constructs into simple phrases and expressions which are output by the generator.

The generator is passed some OWL I expression. If the specializer of the expression is a symbol, which means that the expression is just a simple English word, the system prints it. On the other hand, if the expression is more complex, the generator must determine a number of things. One of the first is to determine whether to print the generalizer or specializer of the concept first.

In OWL I, the concept [(A B)][6] may appear in English as *A* B, *B* A, *A* of B, or just *A* or *B*. The English form of a concept is indicated by placing a flag (or descriptor, to use the proper OWL terminology) on the reference list of the concept or one of its generalizers. For example, since the descriptor OF-SPECIALIZER is found on the reference list of the concept TYPE, the English form of [(TYPE CARDIAC-RHYTHM)] is "type of cardiac-rhythm". When trying to determine the English form of a concept, the generator examines the reference list of the concept. If there is no descriptor there, it examines the reference lists of the successive generalizers of the concept until it finds a descriptor that indicates the English form of the concept. The various types of concepts and their output forms are listed below:

---

6. *A* and *B* are variables here.

Concept Type	English Form of the Concept [(A B)]
Internal Specializer	A
Naming Specializer	B
Classifying Specializer	B A
Of Specializer	A of B
Object Specializer	A B

3.2 Types of Specialization

After determining the proper order for output, the generator calls itself recursively to output the specializer and generalizer part of the concept. It continues to break concepts apart until the pieces are just OWL symbols (corresponding to English words), which are then printed.

There are a few considerations which make this scheme a little more complex. For one thing, there is not a one-to-one mapping between OWL concepts and English. A "run" in a lady's stocking is very different from the "run" in "run around the block", and we would want to have separate concepts in OWL I to represent each idea, yet the same English word is used to express both ideas. To get around this problem, OWL I allows the user to specify the English "name" of a concept. Thus the notion of a run in a stocking might appear as:

```
[(TEAR STOCKING)
 NAME: RUN]
```

This notation says that the concept (TEAR STOCKING) is expressed in English as "run". While outputing concepts, the generator checks to see if an expression is "named" by some other expression. If it is, the generator outputs the name of the concept instead of the concept.

The generator is a rather simple program, yet it is quite flexible, and it is adequate for producing explanations. As the OWL I system becomes more sophisticated, it is likely that a more complex generator will be required. The current version of the generator is controlled largely by syntax, and that makes it difficult to output an expression properly when semantic considerations are important. As an example, the generator has a great deal of difficulty

deciding whether or not to place a "the" in front of a noun group, because that decision is based on semantics as well as syntax. Thus, although the current generator is not by any means the ultimate generator, it is adequate for the Digitalis Advisor.

3.4 Semantic Model Programming: Programming for Explanation

3.4.1 Introduction

The designer of a system that can explain itself faces a number of problems. One is to provide the user with an explanation that answers his question, yet does not swamp him with irrelevant details. To accomplish this, the information contained in the system needs to be structured in some way. Different methods for structuring the information have been proposed.

In the MYCIN system, Davis[12] uses the certainty factor of a rule as an indication of its "informational content". Those rules that have a higher certainty factor are said to contain less information, because the designers of MYCIN feel that they are more like definitions. Rules with lower certainty factors (hence less certain conclusions) supposedly contain more information. This information is used by the system in conjunction with a number supplied by the user to determine how many goals to display when a "WHY" question is asked. If the goals all have high certainty factors then many of them will be displayed at once, while if low certainty factors predominate, few will be displayed. If too many or too few goals are displayed, the user may adjust the number he supplies. In this way, MYCIN attempts to provide the user with a summary. This approach rests on the rather weak assumption that the importance of a rule is reflected by how certain one may be about its conclusion. Yet in fact, in many applications, the importance of a rule is completely independent of the certainty of its conclusion. In a MYCIN-based system for auto repair [12, page 26] conclusions could be reached with little inexactness: thus, if the scheme outlined above were applied to the rules of the auto repair system, it would indicate that they were all about equally important. It is not likely that that is correct. It seems that for many applications a different method for providing summaries is required.

In a similar vein, designers of rule-based systems have had trouble expressing some knowledge in the rule format. Davis [12, page 29] notes:

> "A ... problem is the limit on the amount of knowledge which can conveniently be expressed in a single rule. Actions which are "larger" than this limit are often achieved by the combined effects of several rules. For many reasons, this is difficult to do, and often produces opaque results."

Davis [12, page 261] also observes:

> "Rules are a reasonably natural and convenient form of knowledge encoding for what may be termed "single level" phenomena - it is easy to think of single decisions or actions in terms of a rule.
>
> Experience with MYCIN has demonstrated, however, that even experts acquainted with the program tend to think of a sequence of operations in procedural terms, and find flowcharts the most convenient medium of expression. While the flowcharts can always be converted to an equivalent set of rules, the conversion is non-trivial, and sometimes requires reconsidering the knowledge being expressed, since the two methodologies offer different perspectives on knowledge organization and use.

In designing the OWL Digitalis Advisor, it was decided to use a procedural system so that knowledge could be placed in a hierarchical structure of procedures. Since it is possible to group knowledge conveniently, we will see that the explanations produced by the OWL Digitalis Advisor are well-structured.

Another problem that confronts the system designer is the problem of reconciling the user's model of the problem with the program's model of the problem. That is, when explaining the program to the user, it is necessary to take into account the possibility that the user's model of the problem is very different from the program. Wilkinson[16] has proposed the use of two models, one to represent the program and the other the user's model of the problem. A problem with this approach is that when the system is modified, changes must be made not only to the program, but to the structures linking it to the user's model as well. It

seems that to avoid the dangers of discrepancies between the models it would be a good idea to incorporate the user's model into the actual program as much as possible.

Still another problem that others[12] have noticed is the problem of indicating the "intent" of a piece of code. Programmers attempt to indicate intent by chosing mnemonic names for variables and procedures and placing comments in their code. Yet common programming languages throw this information away. To the LISP interpreter it makes no difference whether a variable is called FACTOR-OF-REDUCTION or PATIENTS-WEIGHT (or G00001, for that matter) yet there is a tremendous difference in the intended meaning of the variable. Being able to understanding the intention of a variable or procedure is vital if one is to understand the intention of a system. Even experienced programmers find it very difficult to understand a program if the names used in the program are misleading or meaningless. If a system is to explain itself, the designer must be able to indicate the intent of the code, and this information should be maintained in a structured manner.

3.4.2 Semantic Model Programming and OWL I

In this section, I will attempt to show how the problems outlined above may be ameliorated through the use of OWL I and Semantic Model Programming (SMP). SMP is actually a synthesis of several separate ideas. One key point is that the name of a procedure should be a conceptual summary of the actions that the procedure performs. Likewise, the role of a variable should be indicated by its name. Another notion is that each procedure should be a conceptual unit that models some action that an expert takes in solving a problem. By using the principles of structured programming, it is possible to produce a hierarchy of procedures analogous to the hierarchies produced by the OWL I notions of specialization and decomposition[7].

7. Decompositon is described by Martin [20].

In the Digitalis Advisor, the procedure used to start treating a patient with digitalis is called [(BEGIN THERAPY)] in OWL I. Its English translation should be clear. One of the functions that [(BEGIN THERAPY)] calls is a function that checks for any sensitivities the patient may have. It is called [(CHECK SENSITIVITIES)]. [(CHECK SENSITIVITIES)], in turn, calls a number of functions. One of these is [(CHECK (SENSITIVITY (DUE (TO POTASSIUM))))] which checks for digitalis sensitivity that the patient may have due to a potassium imbalance. When the plan or event for beginning therapy is described, [(CHECK SENSITIVITIES)] is displayed without any of the structure beneath it. It summarizes the calls below it, so that they do not have to be displayed. If the user is curious about how sensitivities are checked, he may ask, and he will see that one of the steps is to check sensitivity due to potassium. If he is still curious, he may inquire about that step as well. Notice that if he is not interested, the entire process of checking sensitivities will be summarized as one step, so that he does not have to examine reams of output that he does not care about. It should also be pointed out that the user need not ask questions in a "top-down" fashion as described above, but rather he may directly ask how the system checks sensitivity due to potassium at any time if he desires.

In the current implementation of the Digitalis Advisor, when a plan is explained, it is assumed that when a call is made to another plan, the call is to be taken as a summary of the actions performed by that plan. Thus, only the call is displayed. The plan referred to in the call is not examined unless the user specifically asks about it. In the future, it might turn out that it would be desirable if the call could be flagged to indicate that the plan it refers to should be displayed. In the current implementation it has been adequate to treat all calls to plans as summarizations.

Notice that this method of summarizing output contrasts with the certainty factor approach adopted in MYCIN. Rather than attempting to make conclusions about the

information content of a rule based on its certainty factor, we are attempting to structure the procedures of the Advisor so that they model the structure of the problem. This methodology places a burden on the system designer since he is no longer free to structure the program in any manner, but instead he must attempt to model the problem with it. The fact that OWL I is an English-based language may be used to advantage in constructing an appropriate program structure.

Linguists generally believe that the language used by a group of people reflects the world around them[23]. Since snow is important to them, Eskimos have several different words for it to reflect different textures and types, yet English-speaking people only have one word for snow, since it is much less critical to their lives. Like the Eskimos, physicians have developed special vocabularies and procedures to deal with the problems that commonly confront them. When it is useful to think of a series of steps as an aggregate, they are grouped together and given a name which is an English word or phrase. Since it is easy for an OWL programmer to name his procedures after English words or phrases, he may use the English terminology used by physicians to structure his program. In that way, the structure of the program will reflect the structures used by the best problem-solvers in the domain -- the human physicians.

3.4.3 Semantic Model Programming and Structured Programming

The idea of Semantic Model Programming parallels ideas developed in structured programming. In structured programming, one decomposes a problem into smaller and smaller pieces until the code to solve a part of the problem can be written directly. Dijkstra is clearly aware of the relationship between explanation and structured programming. In his "Notes on Structured Programming" [18, page 44] he states:

> If I judge a program by itself, my central theme, I think, is that I want the program written down as I can understand it, I want it written down as I would like to explain it to someone.

Semantic Model Programming can be viewed as an extension of structured programming. As in structured programming, the person using Semantic Model Programming decomposes a problem into its components. The chief addition of Semantic Model Programming is that it advocates the use of English as a guide in chosing the most appropriate decomposition of a problem from the many possible decompositions.

3.5 The Explanation Routines -- How They Work

3.5.1 Introduction

In this section, the various explanation routines provided by the Digitalis Advisor are outlined. The explanation routine that deals with hypothetical situations is not described here, it is discussed in Chapter 4. The explanation routines produce explanations directly from the OWL I code that the interpreter runs, and from the event structure that the interpreter creates. The explanations are not canned -- a change in the procedures used by the Advisor will be reflected in changed explanations. Even though they can convert the OWL I methods to English, the explanation routines are quite simple. Simplicity is possible because the OWL I code itself is close to English and the procedures are written in the style of Semantic Model Programming, which facilitates explanation.

3.5.2 Describing Methods

One of the simplest explanation routines is DESCRIBE-METHOD which describes OWL I methods (or plans, as they are also called). This procedure is designed to answer the question "In general, how do you _____?". DESCRIBE-METHOD describes how an OWL I procedure works in general, not how it applies to a particular patient. The routine is called with a single argument which is the OWL I plan to be described. DESCRIBE-METHOD traces out the links which connect the steps of the OWL plan and converts steps to English as it encounters them. Special routines are called recursively to explain certain OWL primitives such as BECOME, IF-THEN, and OR. Note that only OWL primitives are "taken apart". If the system encounters a call to another OWL I plan, it only displays that call, it does not examine or describe the called plan, since it takes the call to be a summary of the actions performed by that plan (because

the system is programmed using SMP). As it produces an explanation, the system indents the output to indicate the structure of the OWL method. As an example, an OWL plan is listed below, followed by the English explanation of it listed in Chapter 2.

```
[(CHECK (SENSITIVITY (DUE (TO THYROID-FUNCTION))))
 PLAN
 SUMMARY: (FACTOR REDUCTION-MYXEDEMA)
 METHOD: ((IF-THEN
           (CURRENT-VAL (STATUS MYXEDEMA) UNKNOWN)
           (ASK-USER (QUANTA T4)):1,
          (OR
           (IF-THEN
            (OR:15
             (STATUS MYXEDEMA PRESENT)
             (AND:10
              (STATUS MYXEDEMA UNKNOWN)
              (LESS-THAN 2.5 (QUANTA T4))))
            (BECOME-ALSO
             (CONDITIONS CORRECTABLE-AND-PRESENT MYXEDEMA)):1,
            (UNBECOME (CONDITIONS DEGRADEABLE-MYXEDEMA)):1,
            (BECOME (FACTOR REDUCTION-MYXEDEMA 0.67)):1,
            (BECOME-ALSO (REASONS REDUCTION MYXEDEMA)):1)
           (AND
            (BECOME-ALSO
             (CONDITIONS DEGRADEABLE MYXEDEMA)):2
            (UNBECOME
             (CONDITIONS CORRECTABLE-AND-PRESENT MYXEDEMA)):2
            (BECOME (FACTOR REDUCTION-MYXEDEMA 1.0)):2
            (UNBECOME (REASONS REDUCTION MYXEDEMA)):1)))]
```

3.3 The OWL Code to Check for Sensitivity Due to Myxedema

```
(describe-method [(check (sensitivity (due (to thyroid-function)))))])
```

TO CHECK SENSITIVITY DUE TO THYROID-FUNCTION I DO THE FOLLOWING STEPS:

1. IF THE CURRENT VALUE OF THE STATUS OF MYXEDEMA IS UNKNOWN THEN I ASK THE USER THE LEVEL OF T4.

2. I DO ONE OF THE FOLLOWING:

    2.1 IF EITHER THE STATUS OF MYXEDEMA IS PRESENT OR THE STATUS OF MYXEDEMA IS UNKNOWN AND THE LEVEL OF T4 IS LESS THAN 2.68 THEN I DO THE FOLLOWING SUBSTEPS:

        2.1.1 I ADD MYXEDEMA TO THE PRESENT AND CORRECTABLE CONDITIONS.

        2.1.2 I REMOVE MYXEDEMA FROM THE DEGRADEABLE CONDITIONS.

        2.1.3 I SET THE FACTOR OF REDUCTION DUE TO MYXEDEMA TO 0.67.

        2.1.4 I ADD MYXEDEMA TO THE REASONS OF REDUCTION.

    2.2 OTHERWISE, I ADD MYXEDEMA TO THE DEGRADEABLE CONDITIONS, REMOVE MYXEDEMA FROM THE PRESENT AND CORRECTABLE CONDITIONS, SET THE FACTOR OF REDUCTION DUE TO MYXEDEMA TO 1.00 AND REMOVE MYXEDEMA FROM THE REASONS OF REDUCTION.

3.4 An English Explanation of the Code to Check Sensitivity Due to Myxedema

3.5.3 Describing Events

The explanation routine which describes events is called, oddly enough, DESCRIBE-EVENT. It is designed to answer the question "For this patient, how did you _____?". This routine is a little more sophisticated since a certain amount of editing must be done to avoid making nonsensical explanations. The principal difference between explaining events and explaining plans is that when plans are explained all possible paths through the plan are outlined, but when events are explained, only the specific path taken during the event is

displayed. Thus, as one would expect, the chief differences between DESCRIBE-METHOD and DESCRIBE-EVENT are to be found in the routines that explain conditional statements.

When a simple conditional statement is encountered while explaining an event, a check is made to see if the predicate of the conditional succeeded or failed. The event structure contains this information. If the predicate failed, the statement is normally not described[8]. If the predicate succeeded, the predicate is given as the reason for the actions taken by the statement. It is not always easy to determine the reason of success or failure. If the predicate is a disjunction of clauses that succeeds or is dual, a conjunction of clauses that fails, it is not possible to tell which sub-clause was responsible for the success or failure without recomputing the entire expression unless that information is stored in the event structure. For that reason, when such a situation is detected during execution, the interpreter also lists that sub-clause responsible for the outcome in the event structure associated with the step. In this way, it is always possible to give the correct reason for a particular action.

The OR statement, which may contain several IF-THEN statements, is a more complex case. Recall that the OR statement corresponds to the COND statement in LISP. As such, its purpose is somewhat ambiguous. On the one hand, it can be used like a CASE statement in ALGOL. That is, each of the clauses of the OR may involve the same variable, and all of the clauses together cover a set of disjoint possibilities. In that case, the order of the clauses usually does not matter, and the most appropriate explanation is merely to give the predicate that succeeded as the reason for the action taken. On the other hand, each of the clauses of the OR may involve a different variable. In that case, the ordering of the clauses is often important, and it seems that in explaining the OR, the predicates that failed as well as the one that succeeded should be given as reasons for the actions taken by the statement. To determine the type of the OR statement, the explanation routine examines the variables used

8. There are exceptions described in Chapter 4.

in the predicates before explaining the statement. It seems that in future versions of the OWL I interpreter in might be well to use two different types of OR statements to resolve the ambiguity.

There are a few additional considerations. Since a method can be executed several times, there may be several events to explain. If so, they are explained in order. If some of the events occurred during previous sessions, the user is asked if he wishes to see them. If he does, the time and date of the session is given as the events are explained.

To make explanations of numerical computations clearer, the value of a numeric variable is printed in parentheses following the variable whenever it is displayed. The values of non-numeric variables are usually clear from the context of the explanation and are not specifically displayed, unless an assertion about the variable is being described. Whenever a new assertion is made, the new value and the old value of the variable are both given.

A final issue is that events should be explained in the past tense. When events are explained, a flag is set so that the generator converts all verbs to past tense. A sample explanation from chapter 2 is reproduced below.

```
(describe-event [(check (sensitivity (due (to thyroid-function))))])
DO YOU ONLY WANT TO SEE EVENTS FROM THE CURRENT SESSION? (YES OR NO) n
DURING THE SESSION ON 9/21/76 AT 11:18, I CHECKED SENSITIVITY DUE TO
THYROID-FUNCTION BY EXECUTING THE FOLLOWING STEPS:
1. I ASKED THE USER THE STATUS OF MYXEDEMA. THE USER RESPONDED THAT THE
STATUS OF MYXEDEMA WAS PRESENT.
2. SINCE THE STATUS OF MYXEDEMA WAS PRESENT I DID THE FOLLOWING:
     2.1 I ADDED MYXEDEMA TO THE PRESENT AND CORRECTABLE CONDITIONS. THE
     PRESENT AND CORRECTABLE CONDITIONS THEN BECAME MYXEDEMA.
     2.2 I REMOVED MYXEDEMA FROM THE DEGRADEABLE CONDITIONS. THE
     DEGRADEABLE CONDITIONS THEN BECAME HYPERCALCEMIA, HYPOXEMIA,
     CARDIOMYOPATHIES-MI, AND POTENTIAL POTASSIUM LOSS DUE TO DIURETICS.
     2.3 I SET THE FACTOR OF REDUCTION DUE TO MYXEDEMA TO 0.67. THE FACTOR
     OF REDUCTION DUE TO MYXEDEMA WAS PREVIOUSLY UNDETERMINED.
     2.4 I ADDED MYXEDEMA TO THE REASONS OF REDUCTION. THE REASONS OF
     REDUCTION THEN BECAME MYXEDEMA.
```

3.5 An Explanation of the Event of Checking for Sensitivity Due to Myxedema

3.5.4 Describing the Use and Setting of Variables

The Advisor can also explain how variables are set and used in both methods and events. This is done by finding the relevant steps or events and using the routines described above to explain them. This sort of explanation is particularly useful in determining the interdependencies between plans and events. The function DESCRIBE-USE-IN-EVENT finds all the uses of a variable by examining the function-evaluation-use link[9] of the variable. The

9. Function-evaluation-use links are described in chapter 4.

function-evaluation-use link lists all uses of the variable. The events that are found are then explained by the routines to describe events outlined above. The function DESCRIBE-SET-IN-EVENT finds all the places where a variable was set by examining the reference list of the variable, where all assertions about the variable are listed. These events are also described by the routines discussed above. The functions DESCRIBE-USE-IN-METHOD and DESCRIBE-SET-IN-METHOD work in a similar manner. Examples of the use of the four functions may be found at the end of chapter 2.

3.5.5 Describing When an Event or Plan is Called

The procedures DESCRIBE-METHOD and DESCRIBE-EVENT both go down the event and program structures, that is, they tell the user what subevents or subgoals are made by an event or method. It is also possible to go up.

If a user wishes to know when a particular plan is invoked, he may use the routine called FIND-WHY-METHOD. This function finds all the places where a plan is called (using a mechanism similar to the one outlined in the previous section) and displays them to the user. Similarly, the user may find out why an event was created by using the function FIND-WHY-EVENT. Examples of the use of both functions are at the end of chapter 2.

3.6 Summaries and Alternate Models

This section describes some other aids to explanation. Summaries and Alternate Models were developed to deal with certain limitations of the explanation facilities.

3.6.1 Summaries

Several procedures in the Digitalis Advisor are designed to determine the value of some clinical parameter, check some problem, or compute some value. Some examples are [(DETERMINE RENAL-FUNCTION)], [(CHECK SENSITIVITIES)], and [(COMPUTE BODY-STORES-GOAL)]. When the system is describing the methods it uses, the names of these procedures adequately summarize the goals they accomplish. However, when the system describes the events they create their names are not good summaries. It is not sufficient to say "I computed the body stores goal.", because the user is left wondering what the body stores goal is. Thus, there is a SUMMARY associated with certain plans. The SUMMARY is a variable or group of variables that reflect the values that the plan is designed to determine, or in more complex cases, the SUMMARY may be a LISP procedure which determines and displays the relevant information when executed. The OWL variable [(QUANTA BODY-STORES-GOAL)] is a SUMMARY of the plan [(COMPUTE BODY-STORES-GOAL)]. When an event is explained, if there is a SUMMARY associated with the plan that produced the event then the variables associated with that SUMMARY are displayed. If the SUMMARY is a LISP procedure rather than a list of variables, then it is executed. In addition, whenever an ASK-USER event is explained, the answer given by the user is listed. Below, a description of the event [(BEGIN THERAPY)] is given. Summaries are listed in italics.

DURING THE CURRENT SESSION, I BEGAN THERAPY BY EXECUTING THE FOLLOWING STEPS:

1. I INITIALIZED THE SYSTEM VARIABLES.

2. I SET THE TYPE OF THE SESSION TO INITIAL. THE TYPE OF THE SESSION WAS PREVIOUSLY UNDETERMINED.

3. I ASKED THE USER THE AGE OF THE PATIENT. *THE USER RESPONDED THAT THE AGE OF THE PATIENT WAS 60.*

4. I ASKED THE USER THE WEIGHT OF THE PATIENT. *THE USER RESPONDED THAT THE WEIGHT OF THE PATIENT WAS 72.*

5. I ASKED THE USER THE SEX OF THE PATIENT. *THE USER RESPONDED THAT THE SEX OF THE PATIENT WAS MALE.*

6. I CHECKED THE CARDIAC RHYTHM. *THE CARDIAC RHYTHM WAS ATRIAL FIBRILLATION.*

7. I DETERMINED THE REASON OF DIGITALIZATION. *THE REASON OF DIGITALIZATION WAS ARRHYTHMIA.*

8. I ASKED THE USER THE STATUS OF DIGITALIS USE. *THE USER RESPONDED THAT DIGITALIS WAS NOT GIVEN.*

9. I SELECTED THE TYPE OF PREPARATION. *THE TYPE OF PRESENT PREPARATION WAS DIGOXIN.*

10. I DETERMINED THE RENAL FUNCTION. *THE MOST RECENT RENAL FUNCTION MEASURE SHOWED THAT THE RENAL FUNCTION WAS 80%.*

11. I CHECKED SENSITIVITIES. *THE REASONS OF REDUCTION WERE MYXEDEMA AND THE FACTOR OF ALTERATION WAS 0.09.*

12. I COMPUTED THE BODY-STORES GOAL. *THE BODY-STORES GOAL WAS 0.65.*

13. I DETERMINED THE PHASE OF TREATMENT. *THE PHASE OF TREATMENT WAS LOADING-STAGE.*

14. I SET THE STATUS OF DIGITALIS USE TO PRESENT. THE STATUS OF DIGITALIS USE WAS PREVIOUSLY ABSENT.

15. I GAVE RECOMMENDATIONS.

3.8.2 Alternate Models

When writing a computer program, it is occasionally necessary to use methods that are totally foreign to the users of the system. This may be brought about by pragmatic considerations, a desire to improve the system's performance, or possibly because the methods used by humans are not suitable for computers and vice versa. Whenever this situation occurs, it will not be possible to give explanations using the ideas of Semantic Model Programming alone. To solve this problem, it is instructive to reflect on the techniques used by human teachers in similar circumstances.

When a teacher is trying to explain a new concept to his students, he will often try to draw an analogy between what the students already know and the new concept. For example, a teacher trying to explain the fundamental notions of electrical potential, current, and resistance may use the familiar model of a water tank with an outlet at the bottom. The depth of water in the tank is analogous to the potential, the flow of water through the outlet may be taken as current, and the notion of resistance is analogous to the diameter of the outlet.

In the Digitalis Advisor, a weighted sum is computed to indicate whether or not the condition of a patient suffering from congestive heart failure is improving. It seems likely that many doctors are not acquainted with the idea of using a weighted sum to evaluate the condition of a patient. For that reason, the routines that assess the condition of the patient are linked with an alternate model. The alternate model describes in canned English text what the routine is trying to accomplish. In addition, some of the steps of the routine are linked to the text descriptions that describe what they do. The reason for linking specific steps to the alternate model is that that way, when events created by the routine are described, only those parts of the alternate model linked to steps which actually executed will be displayed. A procedure to check weight gain in patients with congestive heart failure is shown below. The parts of the alternate model are printed in italics.

(describe-method [(check weight-gain)])

TO CHECK THE WEIGHT GAIN *I USE A WEIGHTED SUM SCHEME. THAT IS, THE CONDITION OF THE PATIENT IS REFLECTED BY THE VALUE OF THE MEASURE OF THERAPEUTIC IMPROVEMENT. A POSITIVE VALUE INDICATES IMPROVEMENT WHILE A NEGATIVE VALUE INDICATES A WORSENING. THE MAGNITUDE OF THE MEASURE OF THERAPEUTIC IMPROVEMENT INDICATES THE DEGREE OF IMPROVEMENT OR WORSENING.*
I DO THE FOLLOWING STEPS:

1. I DO ONE OF THE FOLLOWING:

    1.1 IF THE CURRENT VALUE OF THE WEIGHT OF THE PATIENT IS NOT GREATER THAN THE IDEAL WEIGHT OF THE PATIENT AND THE BASE-LINE VALUE OF THE WEIGHT OF THE PATIENT IS GREATER THAN THE IDEAL WEIGHT OF THE PATIENT THEN I DO THE FOLLOWING SUBSTEPS:

        1.1.1 I ADD ACTUAL-WEIGHT-LESS-THAN-IDEAL-WEIGHT TO THE SIGNS OF THERAPEUTIC EFFECT.

        1.1.2 I SET THE MEASURE OF THERAPEUTIC IMPROVEMENT TO 15. *IN OTHER WORDS, I NOTE THAT THERE HAS BEEN A SIGNIFICANT IMPROVEMENT.*

    1.2 OTHERWISE, IF THE PREVIOUS VALUE OF THE WEIGHT OF THE PATIENT IS GREATER THAN THE WEIGHT OF THE PATIENT THEN I DO THE FOLLOWING SUBSTEPS:

        1.2.1 I ADD WEIGHT-LOSSAGE TO THE SIGNS OF THERAPEUTIC EFFECT.

        1.2.2 I SET THE MEASURE OF THERAPEUTIC IMPROVEMENT TO 6. *IN OTHER WORDS, I NOTE THAT THERE HAS BEEN A REASONABLE IMPROVEMENT.*

    1.3 OTHERWISE, IF THE PREVIOUS VALUE OF THE WEIGHT OF THE PATIENT IS LESS THAN THE WEIGHT OF THE PATIENT THEN I SET THE MEASURE OF THERAPEUTIC IMPROVEMENT TO -15. *IN OTHER WORDS, I NOTE THAT THE CONDITION OF THE PATIENT HAS BECOME CONSIDERABLY WORSE.*

3.7 Extensions for Iteration

In this section, we will describe what might need to be done to explain programs that use iteration extensively. As a sample problem, we will try to explain a simple program that determines whether or not a number is prime and returns a message. The algorithm appears in ALGOL below:[10]

```
PROCEDURE PRIME? (X)
   BEGIN
   INTEGER J;
   J := 1;
   WHILE (J < SQRT(X)) AND (J = J * TRUNCATE(X/J)) DO
      J := J + 2;
   IF (J ≥ SQRT(X)) THEN
      RETURN("IT'S PRIME")
   ELSE
      RETURN("IT'S NOT PRIME");
   END;
```

3.6 A Procedure Using Iteration

This simple example differs from most of the code found in the Digitalis Adviser. Most of that code does not use iteration. However, it is clearly necessary to be able to explain iteration. The OWL I interpreter has no higher level constructs for expressing loops other than the simple goto-conditional construction. Programmers generally find the more explicit constructs such as FOR loops and WHILE loops useful. It seems that it would be desirable to add some similar statements to OWL.

The addition of some new statements would not only make programming easier, but would be an aid to producing explanations as well. Normally, the OWL I interpreter remembers every computation it makes. In the example above, it is rather unlikely that it would ever be

10. I make no claim that this is the best way to determine if a number is prime. This example is used for illustration only.

desirable to remember all the computations made during the WHILE loop. The WHILE statement could be a signal to the OWL interpreter that it was to summarize the actions taken during the loop.

When people explain a loop in a program, they often do it in the following way: They explain all the actions taken during the first iteration of the loop, and possibly the second, then they do not explain subsequent iterations until the terminating conditions are reached. Usually this is a sufficient explanation, because the actions taken on each iteration are so similar that they can be understood in general by merely examining a few specific cases. The OWL I interpreter could adopt a similar strategy. When executing a WHILE statement, the interpreter could save the results of the first couple of iterations and the terminating conditions of the loop. When asked to explain the loop, the system would use this summary. Using this method, a great deal of storage could be saved in programs that use iteration extensively, yet, clear explanations could still be given.

Chapter 4: Updating

4.1 Introduction

This chapter deals with a different type of explanation. Updating refers to the process of changing a previously given answer to a question, and explaining the effect of that change on the recommendations.

Since the digitalis advisor receives information from the user by asking the user a number of multiple choice questions, there are only a limited number of possible answers. A user may sometimes feel that the correct answer is "in between" two of the those answers. Though forced to give one answer, he may wish to know how much the answer he chooses affects the outcome. Additionally, the user may wish to see the effects of a different answer to become better acquainted with the program. Updating provides a solution to these problems. The user may give one answer during the course of a session, and then change that answer at the end of the session to determine how sensitive the final recommendations are to that answer.

Ideally, updating would not require any data structures not normally created by the interpreter during execution. It would not re-execute any steps not effected by an update. Finally, it would perform an update in such a way that it could easily give the user a concise explanation of the effects of that update.

There are many different ways to do updating. Each of them approximates to some degree the ideal outlined above, though none of them achieves it. The end of this chapter will detail a number of them with their advantages and disadvantages; however, a brief overview of various approaches will be given here. MYCIN uses what might be called the recomputation approach[12]. When a user wishes to change an answer, the system accepts the new answer

from the user, saves all other answers that the user has given, then re-initializes itself and starts the session over from the very beginning. The system does not inquire about anything it has already asked the user, since it has stored all previous answers and since the questions in MYCIN are generally assumed to be independent[11] (as they are in the OWL Digitalis Advisor). This approach does not require any extra storage (other than the storage for answers to previous questions), but it takes a considerable amount of computation time to change an answer.

Another approach might be called the "support" approach. Many deductive systems have used this technique. I refer to it as the "support" approach because the key idea is that the system links the facts that support a conclusion to that conclusion. The expectation is that when a fact is changed, the system will, in most cases, be able to change its conclusions accordingly without recomputing everything, thereby saving computation time. One potential problem with this approach is that the overhead required to do updating based on the supports may be so great that there is little actual saving of computer time.

The updating mechanism that the OWL Digitalis Advisor uses is essentially a support approach. This technique was chosen not only to try to save computation time, but more importantly, because it makes it easier to produce concise explanations of the update. The goal is to recompute only affected steps. Concise explanation is then easy, because only the affected steps should be explained to the user.

---

11. That is, a change in the answer to one question does not, in general, imply that other answers must be changed.

4.2 A Sample Session

This section shows a sample session with a user. The user's inputs are in lower case, while the advisor's output is in upper case. Comments about the session appear in italics. This sample is a continuation of the sample initial session in chapter 2. It begins just after the system has presented dosage recommendations during the initial session. The user wants to vary a few parameters to see what effect they have on the outcome of the session.

```
ENTER AN OWL FORM TO BE CARRIED OUT.
<>[(hypothesize (status hypoxemia))]
```

The user indicates to the system that he wishes to make a new hypothesis about the status of hypoxemia.

```
YOU STATED THAT THE STATUS OF HYPOXEMIA WAS NOT SUSPECTED. IF YOU WISH TO
CHANGE THAT ANSWER, PLEASE GIVE YOUR NEW RESPONSE BELOW.
WHAT IS THE STATUS OF HYPOXEMIA?

THE POSSIBILITIES ARE:

        1. SUSPECTED
        2. NOT SUSPECTED

ENTER SINGLE VALUE ----> suspected
```

The system reminds the user of his previous response, and asks him for a new one. The user indicates that he suspects hypoxemia. This answer is different than the one he gave during the original session.

```
WHAT IS THE LEVEL OF PO2?

ENTER NUMERICAL VALUE ----> 45

PLEASE TRY TO CORRECT THE HYPOXEMIA AS SOON AS POSSIBLE.
```

The user indicates that the patient is quite hypoxemic, and the advisor suggests correcting the condition.

Since the change in the status of hypoxemia has resulted in a change in the recommendations, the advisor prints the new parameters and makes a new prescription.

```
PARAMETERS USED TO MAKE RECOMMENDATIONS:

BODY STORE GOAL --->   8.371 MG
PROJECTED AMOUNT ON BOARD --->   0.000 MG
BODY STORES SHIFT --->   8.371 MG
REMAINING TIME TO REACH GOAL --->     48.0 HRS.
HALF LIFE --->    43.7 HOURS        1.8 DAYS
DAILY LOSS AT EQUILIBRIUM --->   0.117 MG
ABSORPTION FACTOR --->0.75
```

Recall that in Chapter 2 the recommended body stores goal was 0.55 mg.

```
THE DOSAGE RECOMMENDATIONS ARE:
HOURS FROM NOW          ORAL                    IV
NOW                     125 + .0625 MG          .125 MG

REPORT BACK AFTER THE FIRST DOSE.


8                       .125 MG                 .0625 MG
16                      .0625 MG                .125 MG
24                      .125 MG                 .0625 MG
32                      .0625 MG                .0625 MG
40                      .0625 MG                .0625 MG

ORAL MAINTENANCE SCHEDULE:

ALTERNATE .25 MG AND .125 MG

IV MAINTENANCE SCHEDULE:

.125 MG
UPDATE COMPLETED.
```

The system now returns, indicating that all necessary changes have been made. The user may now ask the system to describe the update.

(describe-update)

THE PRINCIPAL RESULTS OF THE UPDATE WERE:

1. WHILE CHECKING SENSITIVITY DUE TO HYPOXEMIA, I ASKED THE USER THE STATUS OF HYPOXEMIA. I MADE A DIFFERENT DECISION THAN BEFORE THE UPDATE. SINCE THE CURRENT VALUE OF THE STATUS OF HYPOXEMIA WAS SUSPECTED I DID THE FOLLOWING:

    1.1 I ASKED THE USER THE LEVEL OF PO2. THE USER RESPONDED THAT THE LEVEL OF PO2 WAS 45.

    1.2 SINCE THE LEVEL OF PO2 (45) WAS LESS THAN 50.00 I DID THE FOLLOWING:

        1.2.1 I SET THE FACTOR OF REDUCTION DUE TO HYPOXEMIA TO 0.67. (BEFORE UPDATING, THE VALUE WAS 1.00.)

        1.2.2 I ADDED HYPOXEMIA TO THE PRESENT AND CORRECTIABLE CONDITIONS. THE PRESENT AND CORRECTIABLE CONDITIONS BECAME HYPOXEMIA AND MYXEDEMA.

        1.2.3 I REMOVED HYPOXEMIA FROM THE DEGRADEABLE CONDITIONS. THE DEGRADEABLE CONDITIONS BECAME HYPOKALEMIA, CARDIOMYOPATHIES-MI, AND POTENTIAL POTASSIUM LOSS DUE TO DIURETICS.

        1.2.4 I ADDED HYPOXEMIA TO THE REASONS OF REDUCTION. THE REASONS OF REDUCTION BECAME HYPOXEMIA AND MYXEDEMA.

        Recall from chapter 2 that these are sets which indicate why a reduction was made, and which conditions can degrade or improve.

2. I MADE A DIFFERENT DECISION THAN BEFORE THE UPDATE. SINCE THE STATUS OF HYPOXEMIA WAS SUSPECTED AND THE LEVEL OF PO2 (45, FORMERLY UNDETERMINED) WAS LESS THAN 65.00 I SUGGESTED CORRECTING HYPOXEMIA.

3. WHILE COMPUTING THE FACTOR OF ALTERATION, I SET THE FACTOR OF ALTERATION DUE TO SENSITIVITIES TO THE PRODUCT OF THE FACTOR OF REDUCTION DUE TO ADVANCED AGE (1.00), THE FACTOR OF REDUCTION DUE TO HYPERCALCEMIA (1.00), THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA (1.00), THE FACTOR OF REDUCTION DUE TO POTENTIAL POTASSIUM LOSS DUE TO DIURETICS (1.00), THE FACTOR OF REDUCTION DUE TO HYPOXEMIA (0.67), THE FACTOR OF REDUCTION DUE TO MYXEDEMA (0.67), AND THE FACTOR OF REDUCTION DUE TO CARDIOMYOPATHY-MI (1.00). THE FACTOR OF ALTERATION DUE TO SENSITIVITIES WAS SET TO 0.45. (BEFORE UPDATING, THE VALUE WAS 0.67.)

4. I SET THE FACTOR OF ALTERATION TO THE PRODUCT OF THE FACTOR OF ALTERATION DUE TO SENSITIVITIES (0.45) AND THE QUOTIENT OF THE WEIGHT OF THE PATIENT (72) AND 70.00. THE FACTOR OF ALTERATION WAS SET TO 0.46. (BEFORE UPDATING, THE VALUE WAS 0.69.)

5. WHILE COMPUTING THE BODY-STORES GOAL, I SET THE BODY-STORES GOAL TO THE PRODUCT OF THE FACTOR OF ALTERATION (0.46) AND THE BASIC BODY-STORES GOAL (0.80). THE BODY-STORES GOAL WAS SET TO 0.37. (BEFORE UPDATING, THE VALUE WAS 0.55.)

6. WHILE GIVING RECOMMENDATIONS, I PRINTED THE PARAMETERS.

7. I MADE THE PRESCRIPTION.

Now the user would like to change the value of serum potassium.

<>[(hypothesize (quanta serum-potassium))]

YOU STATED THAT THE LEVEL OF SERUM POTASSIUM WAS 4. IF YOU WISH TO CHANGE THAT ANSWER, PLEASE GIVE YOUR NEW RESPONSE BELOW.

WHAT IS THE LEVEL OF SERUM POTASSIUM?

ENTER NUMERICAL VALUE ----> 4.2

UPDATE COMPLETED.

It appears that nothing was changed by the update. The reason why becomes apparent when the user asks the system to describe the update.

(describe-update)

THE PRINCIPAL RESULTS OF THE UPDATE WERE:

1. WHILE CHECKING SENSITIVITY DUE TO POTASSIUM, I ASKED THE USER THE LEVEL OF SERUM POTASSIUM. THE USER RESPONDED THAT THE LEVEL OF SERUM POTASSIUM WAS 4.20.

2. I MADE THE SAME DECISION AS BEFORE THE UPDATE. SINCE THE LEVEL OF SERUM POTASSIUM (4.20, FORMERLY 4) WAS NOT LESS THAN 3.70, I DID THE FOLLOWING:

    2.1 I SET THE FACTOR OF REDUCTION DUE TO HYPOKALEMIA TO 1.00. (THIS VALUE WAS NOT CHANGED BY THE UPDATE.)

4.3 An Outline of the Issues in Updating

The OWL Digitalis Advisor attempts to minimize the number of steps that it re-executes in performing an update. It finds those portions of OWL methods which are directly or indirectly affected by the update and re-executes them alone. Since usually only a few steps are affected, it is relatively easy to produce concise explanations of the effects of the update. This section outlines some of the factors that must be taken into account in designing such an update mechanism.

4.3.1 Restrictions

First, we will state some restrictions. These were imposed to make programming easier and the following discussion clearer.

The Digitalis Advisor consults with the doctor during several sessions. The time between sessions can be as long as several days. The system does not allow updates to have effects in sessions other than the current session. That is, the doctor may not change an answer that was given during a previous session and observe the effects of that change. This restriction may be justified on medical grounds. It seems likely that a doctor would use a special set of methods to deal with the problem of a changed answer to a question in a prior session, since the patient would have already received a prescription based on the data given before the update. The update mechanism discussed here is not designed to invoke special procedures, and hence it would not be an appropriate solution to this problem.

Another restriction is that OWL procedures in the Digitalis Advisor are not allowed to pass arguments. In the context of the Digitalis Advisor, this is not really a restriction since the Advisor does not need procedures that pass parameters—all procedures communicate

through semantically meaningful global variables. It is likely that more sophisticated systems will need to be able to pass parameters, hence more research will be required to resolve the problem of passing arguments.

4.3.2 Types of Time

In most programming systems, the value of a variable depends upon when it is examined. For example, if we examine a variable used as a counter in a loop before the loop is executed, its value may be undetermined. While the loop is being executed, the value of the variable will depend on the number of completed iterations. Thus, the relationship between time and value is very important for a correct updating strategy.

In the OWL Digitalis Advisor, there are three different kinds of time which affect values. The first type is called *real-world time*. It refers to the time that a fact became true or was observed in the real world. For instance, the serum calcium level of a particular patient may have been observed to be 4.2 on September 27, 1976 at 2:00 pm. The second type is *computation time*. It refers to the time that a fact was entered into the data base, as the result either of a user's answer to a question, or of a calculation. To continue the example, the Digitalis Advisor may have been informed of the patient's serum calcium level at 5:00 pm on the day it was observed. The third type of time is called *precedence time-order*. It refers to the relationships between the values of a variable and the events that assert the values. Informally, it expresses the notion that some steps must be done before others may be performed. In the loop example above, the counter takes on many values. Each of these values is correct at some time, but only one is correct at any given time. Precedence time-order stresses the relational quality of time rather than the notion of time as a point on a time-line. In a programming environment which does not allow updating, the precedence time-

order is modelled by the computation time. If updating is allowed, the computation time of one event may precede the computation time of another, yet their precedence time-order may be reversed. When updating is allowed, the value of a variable depends solely on precedence time-order. If we had a computer that allowed parallel computation, the precedence time-order would be isomorphic to the dependencies between statements. In a serial machine, the precedence time-order is actually a stronger ordering than the dependency ordering, since for a serial machine, the precedence time-order is a total ordering, while the dependency ordering is only a partial ordering.

The OWL Digitalis Advisor performs updates (which require knowledge of dependencies) without requiring the programmer to explicitly indicate the dependencies. The system can discover a precedence time-order as it executes which can be used as a model for the dependencies, and, therefore, as a basis for updating.

4.3.3 Special Data Structures for Updating

During execution, the OWL I interpreter creates data structures which are used by the updating mechanism to find which steps should be re-executed, to determine the value of variables, and so forth. These data are not needed for normal interpretive execution. If a system designer anticipates that updates will not be needed, he may set a switch so that the data structures are not created.

Whenever the value of a variable is used in computing a value, determining the truth of a predicate in a conditional branch, or making a pattern match, the event evaluating the variable is linked to the variable. This link is called a *function-evaluation-use* of the variable. Such links are used to find all the events that might be changed by a change in the value of the variable.

When making updates, it is vital that the interpreter be able to evaluate the variables of

some previously computed step so that all the variables (except those changed by the update) will evaluate to the values they had when the step was originally executed. This feature is also used by the routines that describe events. It is necessary to create a data structure which will model the precedence time-order relationships between the values of a variable at various times, and the values of other variables. The GWA Digitalis Advisor uses an *environment list* to accomplish this. The environment list is similar to the association list used in some implementations of LISP for variable binding, except the environment list is never "popped". As new assertions are made, they are placed at the front of the environment list. Each assertion contains both the variable and its value (like a dotted pair in LISP). To find the current value of some variable, we merely go down the environment list until we find the first assertion involving that variable. The value associated with that first assertion is the current value of the variable. If we wish to find the value of a variable before the execution of some event, we find that point in the environment list which corresponds to the start of the event, and start our search for the first assertion involving the variable from there. Note that the precedence time-order relationships between assertions are intrinsic to the structure of the environment list itself. It is possible to maintain this structure during an update by *editing* it. When a step is re-executed, any assertions resulting from the original execution of the step are deleted from the environment list, while those resulting from the re-execution are inserted in place of the originals. The mechanisms involved in maintaining this structure are described in more detail below.

4.4 Updating: the Algorithm

4.4.1 An English Description

This section provides a description of how the updating mechanism works. After the advisor has presented its recommendations, the user may indicate to the system that he wishes to change an answer he gave previously.

For the purposes of illustration, suppose that the variable *a* in the program[12] listed in Figure 4.1 has been changed from 0 to 2. The updating algorithm would then proceed in the following manner:

1. First, the system creates an update-event. After the update is completed, the update-event will contain links to all the updated events, as well as a link to the variable that was changed by the user. These links are used in mining explanations.

2. The system finds all the uses of the variable to be changed by examining the function-evaluation-use links associated with the variables. These events are placed in an update-list so that they are in ascending precedence time-order (Figure 4.2):

```
b := 2;
if a < 1 then[13]
   b := 1;
c := 5;
d := c * b;
   .
   .
   .
```

4.1 The Program

```
if a < 1 then
   b := 1;
```

4.2 The Update List

3. The system finds that point in the environment-list which corresponds to the start of the original execution of the first event on the update-list. When finding values, the system will start examining the environment-list from this point so that the corresponding step will be re-executed in the original environment as

12. For clarity, this program is listed in pedagogic ALGOL rather than OWL I.

13. Assume that the variable *a* does not have a value before this step is executed, so that the interpreter will ask the user the value using the mechanism described in Chapter 3.

modified by prior updates [14] (Figure 4.3). To find the correct point, the system goes down the environment list from its head until it comes to the first assertion that precedes the event being updated in procedural time-order.

```
b = 2 <-- environment-list pointer
a = 0
c = 5
d = 10
```

4.3 The Environment List

4. As the system re-executes the step, it places any assertions made by the step on a *temporary-environment-list*. To find the value of a variable, the system examines the temporary-environment-list before looking at the environment-list (Figure 4.4). The system also places an update link between the event created by the original execution of the step and the event created by re-executing the step. By checking for this link, it is easy to tell whether or not an event has been updated.

```
b = 2 <--environment list pointer      a = 2
a = 0                                  b = 1
c = 5
d = 10
```

Environment List Temporary Environment List

4.4 The Environment List and Temporary Environment List

5. After the step has been re-executed, the system determines which variables were changed by the update of that step. There are several ways that a variable may be given a *new* value. A new assertion may be made during the re-execution of a step that is different from the assertion (if any) made during the original execution of the step, or the re-execution of a step may not make an assertion about a variable that was asserted during the original execution of the step. If the value of a variable is affected in either way, all the events which used that variable that have not been updated and have a greater precedence time-order than the step just executed are merged into the update-list. If an event being merged in is superior to an event already on the list, the subevent is removed (Figure 4.5).

6. The assertions on the temporary-environment-list are merged into the environment-list (Figure 4.6). The environment-list and temporary-environment-list are kept separate until this step to facilitate the comparisons in step 5.

---

14. On the first iteration of the algorithm, the first event in the list will be either a step which will directly ask the user for the new value of the variable or a conditional statement which will cause the system to re-ask the user in the process of computing truth of the predicate.

7. The whole process is repeated, starting at 3, with the next event on the update list. The process stops when there are no more events on the update-list.

```
d := c * b;
```

```
b = 2
a = 2
b = 1
c = 5 <--- pointer[15]
d = 10
```

4.5 The New Update List

4.6 The New Environment List

4.4.2 The Program

In this section, a LISP implementation of the top-level updating function is given. Those functions which depend heavily on OWL data base functions are not shown, but are described in English below.

```
(DEFUN UPDATE (VARIABLE-TO-BE-CHANGED)
       (PROG (CHANGED-VARIABLES EVENT-BEING-UPDATED)
        (SETQ *UPDATE-LIST* NIL)
        (CREATE-UPDATE-EVENT)
        (SETQ CHANGED-VARIABLES (LIST VARIABLE-TO-BE-CHANGED))
    A   (INSERT-EVENTS (EXAMINE-FUNCTION-EVALUATION-USE CHANGED-VARIABLES))
        (COND ((NULL *UPDATE-LIST*)
               (SETQ *ENVIRONMENT-LIST-POINTER* NIL)
               (RETURN 'DONE))
              (T
               (SETQ EVENT-BEING-UPDATED (CAR *UPDATE-LIST*))
               (SETQ *UPDATE-LIST* (CDR *UPDATE-LIST*))
               (SET-ENVIRONMENT-LIST-POINTER EVENT-BEING-UPDATED)
               (RE-EXECUTE-STEP (FIND-CALL-FOR-EVENT EVENT-BEING-UPDATED))
               (SETQ CHANGED-VARIABLES
                     (COMPARE-ENVIRONMENT-LISTS EVENT-BEING-UPDATED))
               (MERGE-ENVIRONMENT-LISTS EVENT-BEING-UPDATED)
               (GO A))))))
```

UPDATE-LIST is a global list that is the list of events that need to be updated.

---

15. Prior to the re-execution of the step associated with the next event on the update list.

ENVIRONMENT-LIST-POINTER is a global pointer into the environment list. If it is not null, the evaluation routines use it in determining where to start looking for values. If it is null, the evaluation routines start from the head of the environment list.

CREATE-UPDATE-EVENT is a function that creates an update event. All events re-executed during the update will be linked to the update event. These links are used to explain the update.

EXAMINE-FUNCTION-EVALUATION-USE is a function that accepts a list of variables as input. It returns a list of events which are the events that used the variables in the input list. These events are found by examining the FUNCTION-EVALUATION-USE link on the reference lists of the variables.

INSERT-EVENTS is a function that merges the events that need to be updated into the *UPDATE-LIST*. The events are merged in procedural time-order so that the order of the *UPDATE-LIST* is always maintained. Before one of the input events is inserted into the *UPDATE-LIST* a few checks are performed. If an event to be inserted precedes the last step updated in procedural time-order, or if the event has already been updated, the event is not inserted in the update list. In addition, if the event is a subevent of an event already on the *UPDATE-LIST* it is not inserted. Conversely, if the event to be inserted is superior to an event already on the *UPDATE-LIST* the subevent is removed and replaced by its superior.

SET-ENVIRONMENT-LIST-POINTER is a function which sets the *ENVIRONMENT-LIST-POINTER* to the first assertion on the environment list which precedes the step being updated in procedural time order.

FIND-CALL-FOR-EVENT finds the call associated with an event.

RE-EXECUTE-STEP causes the OWL interpreter to re-execute a particular step.

COMPARE-ENVIRONMENT-LISTS maps down the temporary environment-list and the environment-list to determine which variables have changed during the re-execution of the step.

MERGE-ENVIRONMENT-LISTS splices the environment-list and temporary environment list together. The temporary environment-list is re-set to nil.

4.5 The Nitty-Gritty

This section describes in some detail how some of the things described in the preceding section are actually implemented. The casual reader may skip this section without loss of continuity.

4.5.1 Determining Precedence Time-Order

In the section above, it was stated that events are merged into the update-list in ascending precedence time-order. A predicate is needed that can determine the precedence time-order of events. Computation time alone cannot be used, and there are no explicit links in the interpreter that indicate the precedence time-ordering of two events. However, if the subevent structures associated with events are examined *in conjunction with* their computation times it is possible to determine the precedence time-ordering of the events.

The algorithm for determining precedence time-order during updates makes a few assumptions. Assume that neither event has been updated. If either event has already been updated, it is not necessary to insert it in the update list. Further assume that neither event is a subevent of the other. If it were, only the superior event needs to be placed on the update list. The algorithm works in the following way:

1. To begin with, the depth in the subevent structure is determined for each of the two events being compared. If one event is deeper than the other, the system goes up the subevent structure from the deeper event until it finds an event at the same level as the shallower event. These are the two events that will be compared. Call them A and B.

2. If either A or B has an update link on its reference list, it is replaced by the corresponding original event, found by following update links backward.

3. Next, the events immediately superior to A and B are compared. If they are the same (i.e. if A and B are both immediate subevents of the same event), the computation times of the events are compared. The one with the earlier computation is the earlier event in precedence time-order.

4. If the events immediately superior to A and B are different, the algorithm sets A and B to their immediate superiors and loops back to step 2.

This algorithm works by going up the subevent structure from each of the two events being compared until it finds two events that are immediate subevents of the same event. After the original events are found[16], a comparison may be made on the basis of computation time, since all the effects of updates have been removed. Earlier we pointed out that if no updating has occurred, the computation time may be used as a model for precedence time-order. By going back to the original events, the updates are essentially "removed", and we can use their computation times to determine the precedence time order.

4.5.2 Editing Environment Lists

In performing an update, it is necessary to be able to splice new assertions into the environment list. Although the actual splicing is easy enough, the process of finding the points where the splice should start[17] and stop is a little more involved. If the event being updated is just an assertion, it's easy to find the start and stop points: the assertion is found in the environment list and removed. On the other hand, the step being updated may contain several assertions, conditional expressions, and so forth. To find where the splice should start, the system finds the last assertion made before the start of the event being updated. The last assertion is found by examining the subevents of the events immediately preceding the event being updated. Finding the stop point of the splice is easier: it is just the last assertion made by the event being updated. If no assertions were made, then the stopping point is just the same as the starting point since no assertions need to be removed from the environment list.

---

16. Tracing over update links does not affect the depth.

17. The starting point of the splice is also the point (referred to in step 3 of the algorithm, page 74) at which the functions that evaluate variables start looking at the environment list.

4.5.3 A Proof of Correctness

In this section, we will prove that the updating algorithm described above produces correct results. Our proof will be by induction. We will show that if certain conditions are true before the re-execution of some OWL I program step, the algorithm assures that they will remain true after the step has been re-executed.

Suppose that a variable *A* which originally had the value x has been changed to y. We will assume that the following statements are true at the end of step 3 (page 74), after any number of iterations of the updating algorithm:

1. The next event *E* which must be re-executed is at the head of the update list.

2. The pointer into the environment list has been set so that all variables evaluate to the values they would have had at this point in the program if the value of the variable *A* had originally been y.

3. All events preceding *E* in precedence time-order which must be re-executed to obtain correct results have already been re-executed.

4. The update list contains, in precedence time-order, all those events that are known (prior to the re-execution of *E*) to need updating, that is, the update list contains all events which involve variables that have already been changed by the update.

Basis:

First, we need a basis for our induction. Suppose that steps 1, 2, and 3 of the update algorithm have each been executed exactly once. Lemma A (below) states that *E* will be on the function-evaluation-use links of the variable *A* being changed by the update. Then, by the action of step 2 of the algorithm, assumption 1 must be true. The action of step 3 assures that assumption 2 holds. Since the first event to be re-executed must depend on *A*, there can be no events prior to *E* in procedural time-order which must be re-executed. Thus, assumption 3 is true. Since no events have been re-executed, only those events depending on *A* are known to need updating, thus assumption 4 is correct.

Induction:

Now that we have a basis, we suppose that the assumptions are true after *n* iterations of the algorithm. We need to show that they will be true after *n*+1 iterations.

Claim 1:

> The re-execution of *E* will produce exactly the same results as it would have if *A* were originally set to y.

There are only two ways this claim may be false.

> 1. If variables used during the re-execution of *E* but set before *E* evaluate incorrectly then the claim is false. However, assumption 2 contradicts this statement.
>
> 2. If variables set within *E* evaluate incorrectly during the execution of *E*. This cannot happen because the updating algorithm uses a temporary-environment-list which contains all those assertions made during the re-execution of *E*. When evaluating variables during the re-execution of *E*, the system examines the temporary-environment-list before examining the environment list[18].

---

18. See the description of the algorithm above for a more complete explanation of the temporary-environment-list.

After re-execution, and the removal of *E* from the update list, the assertions on the temporary-environment-list are compared with those on that portion of the environment list corresponding to the original execution of *E* to determine which variables changed. A variable may change in three ways:

1. It may be assigned a different value than it originally received.
2. It may be set during the re-execution of the event, although it was not set during the original execution.
3. It may not be set during the re-execution, although it was set before.

In all cases, the events making use of variables whose values have changed are inserted in precedence time-order into the update list. These events are found by examining the function-evaluation-use links associated with the changed variables. Lemma A (below) shows that all events which must be updated as the result of a change in a variable may be found on the function-evaluation-use list of that variable. An event is not inserted if:

1. The event is before *E* in precedence time-order. (Not necessary by assumption 3)
2. The event has already been updated.

If an event being merged in is superior to an event already on the list, the subevent is removed.

Since assumption 4 was true before *E* was re-executed and all events associated with variables changed by *E* were added to the update list, assumption 4 remains valid. Assumption 1 also remains valid since assumption 4 is true, and since all events associated with variables changed by *E* were inserted in precedence time-order. Assumption 3 is valid since if there were any events between *E* and *E'* (the event at the head of the update list after the re-

execution of *E*) which required updating, these events would have been inserted into the update list.

All that remains, then, is to show that assumption 2 remains valid. Recall from the section on editing environment lists that the environment list pointer is set to the first assertion in the environment list that is earlier in precedence time-order than *E'*. If assumption 2 were no longer valid, the values that invalidate it must appear on the environment list between the points corresponding to the starts of *E* and *E'*. However, since all values asserted by *E* are spliced in and the old values are spliced out, and since no events between the end of *E* and the start of *E'* are affected, assumption 2 must still hold.

Claim 2:

> If the OWL program being updated terminates under all conditions, then the update also terminates. That is, the update mechanism will not introduce any endless loops into a program that always terminates.

This claim is true, since steps corresponding to events from the update list are executed in ascending procedural time-order, and there are only a finite number of steps that may be updated.

Furthermore, at termination, by assumption 3, the results are correct. By assumption 2, evaluation of any variable will give the correct value. Finally, by claim 1, the actions taken by the update are the same as would have been taken if *A* had been set to y.

Lemma A:

> Whenever an event *e* must be re-executed due to a change in the value of a variable, either *e* or an event superior to *e* will found on the function-evaluation-use links of that variable.

Whenever a step uses a variable either in an assignment, a computation, or a predicate, that step is linked to the variable by the function-evaluation-use link. Suppose that there is some step which must be re-executed due to a change in the value of a variable then either:

1. The step was executed before the update, so that it must be on the function-evaluation-use link.

2. The step was not executed before. Then it can only be executed if a superior predicate evaluates differently than originally. This implies that some superior event whose plan contains the predicate will be on the function-evaluation-use link.

4.6 Comparison of Different Updating Strategies

In comparing the various methods of updating it is difficult to analyze them quantitatively, since their performance is very dependent on the state of the knowledge base and the interdependencies among steps of the OWL code. However, it is possible to state certain general characteristics of each approach and indicate which ones would be most suited to various types of applications.

In the introduction to this chapter, two broad types of updating were listed--the recomputation method and the "support" approach. Each approach has a number of interesting variations.

The most primitive way to do recomputation is given in the introduction: to change the value of a variable, the system starts over from the very beginning and recomputes everything. Since this method throws away all the results of the session (except possibly the user's answers to questions), it has the advantage that no data structures need to be kept around to indicate intermediate states of the program. Thus, when the interpreter is running normally, there is no additional cost associated with having an update capability. A significant disadvantage of this approach is that it is very difficult to write an explanation routine to describe the changes resulting from the change of a variable value, since the results prior to the update are thrown away before the update begins. Another disadvantage is that the entire session must be recomputed, which means that many statements which are unaffected by the change in the value of the variable will be nonetheless re-executed. This approach to recomputation, then, is most appropriate when updates are done infrequently, when the purpose of the update is primarily to correct an answer rather than understand the behavior of the program if a parameter is varied, or when there is little storage available to record the extra data structures required by other methods.

There is also a less primitive recomputation approach to which we are willing to use some storage to record the state of the system at various points (slowing down the interpreter slightly to record them) we can speed up the updating process by re-creating the state of the system at some point prior to the change in the variable and letting the interpreter re-execute from that point. This approach speeds up the updating process at the cost of decreased normal execution speed and increased storage costs. How often we record the state of the system will directly determine both the speed-up we may expect in updating and the increased cost in storage and normal running time.

The other approach is the "support" method. In this method, dependencies between various parts of a system are used to determine what must be re-computed when a change occurs. The dependencies may be explicitly hand-coded by the system designer, or they may be automatically generated by the system itself. Bouyj [19] has recently produced a system for the design of procurement systems which uses hand-coded dependencies. The advantages of this approach are that the system overhead imposed is quite low, and the updating strategy itself is relatively simple. The disadvantage is that programming the system is more cumbersome with a greater chance for error.

In the remainder of this section, we will compare the procedure used in the OWL Digitalis Advisor with the approach used in EL, a system using automatically generated dependencies recently described by Stallman and Sussman [15]. This system uses a set of rules to analyze dc circuits. It makes as many conclusions about the circuit as it can, and then, if the circuit is not completely solved, it assumes values for the remaining unknown parameters in the circuit. If assumed values lead to contradictions, they are changed, and the analysis continues. EL, like the Digitalis Advisor, attempts to avoid recomputing unaffected deductions when a change occurs. EL links conclusions to the assertions that were used to deduce them. All deductions are based on information given by the user, assumptions, or

other deductions. All information given by the user is linked to a special node called GIVEN. When the user wishes to change an assertion he has made, the system breaks the link between the old assertion and the GIVEN node. Then the system traces over all links, starting from the GIVEN node, marking the assertions that are still valid[19]. Those assertions that are not marked are removed from the data base. Those facts that remain are guaranteed to be valid. The facts that have been removed are saved in a special area. Those facts that are considered valid are said to be *in*, while those that have been removed are said to be *out*.

In comparing Stallman and Sussman's approach to the approach used in the OWL Digitalis Advisor, it should be pointed out that EL is a rule-based system, while the Digitalis Advisor is a procedural system. This difference in basic system design is reflected in the updating strategies each system uses. There are, however, some interesting comparisons to be made between the two.

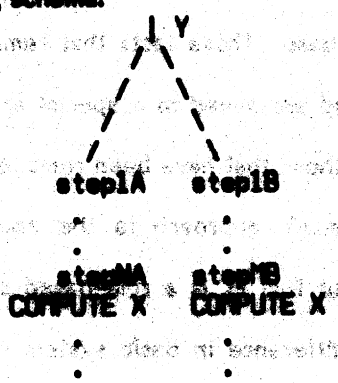
After the fact garbage collector has been run, EL is free to use a valid assertion in making new conclusions, independent of the order in which the original computations were made. It is not possible to do this in the OWL Digitalis Advisor -- and it is possible to imagine a few situations (described below) in which some statements which were not affected by a change would be unnecessarily recomputed[20]. However, the reason EL can use assertions independently of the order in which they were computed is that it makes the assumption that the order of computation does not matter. Although this assumption may be valid for rule-based systems operating in the world of circuits that EL analyzes, it is not always valid in a procedural system. For example, the value of the body stores goal in the Digitalis Advisor is very much dependent on its relationship to the order of computation of other steps. The OWL Digitalis Advisor can model these relationships through use of the environment list.

---

19. This phase is very similar in concept to the mark phase used by the LISP garbage collector. For that reason, Stallman and Sussman refer to this routine as the fact garbage collector.

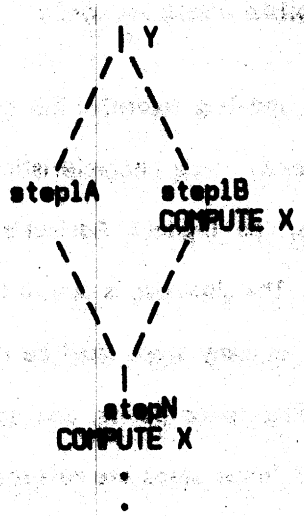
20. No such situations have, to my knowledge, ever come up in the Digitalis Advisor.

It was noted above that situations could be imagined where the updating scheme used in the Digitalis Advisor would re-execute steps which were not affected by the update. A few examples will illustrate how this problem may occur. Suppose that a portion of an OWL method has the following schema:



That is, the program makes a decision at point Y, and executes some steps, then it computes the value of the variable X. Let us suppose that the first time this code was executed the statements on the left branch were executed, but that the user has changed a variable which affects the decision at Y so that now during the update, the statements on the right branch are to be executed. Let us further suppose that the value of X computed by COMPUTE X is not affected by the update. In this case, COMPUTE X will be re-executed (since it is a substep of the right branch), even though the results it gives are unaffected by the changed variable.

Let us look at another example. Suppose we have this schema:



Again suppose that the first time this section was executed the left branch was taken, and that now a variable has been changed so that during the update the right branch will be executed. Also suppose that the function COMPUTE X is unaffected by the changed variable. As the system executes the right branch, the value of X computed during the first execution will be unavailable to it, since it was computed later in precedence time-order during the original execution of this fragment, so that COMPUTE X will have to be unnecessarily re-executed.

What should be done about this? Would it be a good idea to have some sort of mechanism analogous to the fact garbage collector of EL that would go through the code of the program and mark all values that might have changed? I feel the answer is "probably not". First, the situations described above are relatively rare. They have never occurred in the Digitalis Advisor[21]. Second, the updating mechanism would have to be more complicated, so that the hoped for gains in speed might not materialize. It is probably better to allow a small amount of "unnecessary" recomputation to take place than go to great lengths to eliminate it.

---

21. In fact, if it is true that COMPUTE X always computes the same value regardless of the path taken, then these are just examples of poor programming practice. In both cases, the call to COMPUTE X could be executed before the decision point Y, eliminating the extra call. If that were done, no "unnecessary" recomputation would result.

4.7 Current Performance and Possible Improvements

Although no formal analysis of the updating algorithm has been attempted, it is possible to describe qualitatively some of the performance characteristics. In programs which have many interdependencies between steps, the Digitalis Advisor's updating scheme is often slower than the recomputation approach. The slowness is due to the fact that when a variable changes in a highly interconnected program many steps must be re-executed. The process of inserting events in the update list, finding splice points, and so forth adds a considerable overhead to the interpreter. Even though fewer steps are re-executed, the fact that each one takes longer results in slower computations. Fortunately, the Digitalis Advisor is relatively sparsely interconnected.

Another potential problem with the updating scheme involves the environment list. There is a danger that as programs become more complex, the process of evaluating a variable will take an intolerably long time as the environment list becomes lengthy[22]. To improve performance, the interpreter could use the first position of the reference list of a variable as a value cell for that variable's current value (as it now does when not running in "updatable" mode). During normal execution of the interpreter, the value cell would be examined to evaluate the variable, although the environment list would still be maintained to allow updating. During an update, the environment list would be used for evaluation purposes, but the value cells of changed variables would be altered to reflect their new values. This approach improves the speed of evaluation without eliminating the advantages of an environment list.

22. This problem is similar to the one that occurs in LISP interpreters which do not use shallow-binding. In the current Digitalis Advisor, the cost of using the environment list exclusively is only about a 2% increase in execution time.

4.8 Explaining Updates

Since the update process re-executes only significant steps, it is quite easy to provide the user with a concise explanation of the update. An update-event is linked to all the steps re-executed during the update, so that finding the relevant events is easy.

The system can use the routines for explaining events described in Chapter 3 with just a few changes to take account of the special nature of updates. One change is that at certain decision points, the system compares the decision made during the update with the decision made before the update, and informs the user if the decisions differed or were the same. In addition, if these decisions involve variables with numerical values, the values of the variable before and after updating are displayed for the user. Note that it is not always possible to compare decisions, since the system may go down a different path during an update. The system compares decisions only if the update step making the decision is directly linked to the step it updates. This approach makes sense since these decisions will tend to be the most important ones.

Normally, the explanation system does not display a conditional statement if the predicate of the statement was false and the statement did not perform any action. However, if in the course of an update, some conditional statement which set some variables before the update does not now set those variables, we must explain this to the user, since the values of the variables have been changed.

As the system performs an update, it pulls out separate steps from OWL methods. Merely reciting these steps could result in somewhat confusing explanations, since the structure of the methods would not be apparent. For that reason, the explanation system prefaces its explanation of a step with the OWL method that the step was part of. As an example, the explanation of step 3 of the sample session uses the name of the OWL method

that contains the step to state "WHILE COMPUTING THE FACTOR OF ALTERATION...". Step 4 is not prefaced since it comes from the same method.

4.9 Procedures, Rules and Updating

A common criticism leveled by those who advocate the use of rule-based systems against those using procedural systems is that the knowledge embedded within procedures is trapped in those procedures: it cannot be taken out a bit at a time and used in new situations. Thus, they argue that procedural systems are less flexible than rule-based systems. The proceduralists respond that it is difficult to impose any structure on a rule-based system, so that each rule must indicate exactly those conditions under which it is applicable. Thus, they say, there is no notion of being able to apply knowledge within some context.

Recently, Davis has used the notion of strategies to impose some structure on rule-based systems[12]. I feel that the updating mechanism outlined in this chapter represents a move toward freeing the knowledge embedded within procedures, thereby bringing some of the flexibility of rule-based systems to a procedural system. Note that in normal operation, the ONYX Digitalis Advisor is a structured procedural system. This structure makes it easy to produce clear explanations, and carry on interviews with a physician in an orderly fashion. However, when an update is performed, the system uses individual steps from procedures, and puts them together dynamically creating what is, in essence, a new procedure for updating. Thus, the knowledge contained within the procedures is extracted from them and put together in a new way to perform a new task. The original structure remains, however, and (as was shown in the preceding section) is still very useful in making explanations of the update. Thus, the ONYX Digitalis Advisor overcomes some of the limitations of a procedural system, while retaining the advantages of its structure.

Chapter 5: Conclusions and Suggestions for Further Research

A very desirable capability for any expert problem solving system is the ability to explain its reasoning processes. User acceptance is more easily obtained if the user can assure himself that the program makes reasonable deductions which result in reasonable conclusions. An explanation feature may be a valuable pedagogical tool. Finally, it can be very useful in debugging the problem solving system itself.

The OWL Digitalis Advisor can explain, in English, the procedures it uses and the actions they take. It can also explain how its variables are set and used. In addition, the Advisor allows the user to change answers he has given to determine the effect on the recommendations produced by the system. The Advisor can produce a concise explanation of the changes introduced by a change in an answer. The explanations are produced directly from the code it executes. The Advisor is structured in a manner that attempts to model the understanding a cardiologist would have of digitalis therapy. The system is not designed to replace physicians, rather, it is designed to assist them in prescribing digitalis.

5.1 Further Research

There are a number of interesting issues involving explanation that remain unresolved. The OWL Digitalis Advisor can be extended in a number of ways.

It still remains to be determined how adequate the explanations are that the Digitalis Advisor provides. The limited experience we have had in demonstrating the program to doctors and medical students indicates that they generally find the explanations understandable, but they are occasionally confused by some of the terms it employs. A clinical trial is planned in the near future which should provide some answer to this question.

The clinical trial should also shed some light on the problem of constructing a model of the user. It would be good if the Digitalis Advisor could take into account a user's sophistication and experience when constructing explanations. An explanation that is appropriate for a medical student might be much too tedious for a cardiologist.

If the Digitalis Advisor is to be used in a clinical setting it will have to be able to accept questions from the user in English. The problem of accepting English input can be attacked in three stages. For the immediate future, a simple parser could be constructed similar to the one used in MYCIN[2]. After the OWL parser becomes operational, work could begin on a more sophisticated understanding module that could be more precise in its understanding of English. Finally, one could envision a quite complex system that would attempt to understand a user's confusions. It would then use the model of the user to produce an explanation.

The current system produces explanations which are based on the code it executes. These explanations often lack a notion of the medical reasons behind the actions. That is, the system can explain that it reduced the dose because the patient's level of serum calcium was below 3.7, but it cannot explain why the level 3.7 is significant. In fact, 3.7 is an arbitrary figure to some degree, yet the system should be able to explain that. It is possible that these explanations could be provided by making sophisticated use of a "medical alternate model".

Currently, the system can explain why it performed a particular action. A medical model might aid it in answering the corresponding question: "Why didn't you _____?". If the medical model were reasonably complete, the system might be able to use it to deal with new situations. The current program is quite rigid and cannot deal with conditions that were not anticipated when the program was written. Some of the ideas developed by Carbonell[17] might be useful in solving this problem.

References

1. Winograd T: Computer program for understanding natural language. AI TR-17, 1971

2. Shortliffe EH: MYCIN: A rule-based computer program for advising physicians regarding antimicrobial therapy selection. SAIL AIM 251, 1974

3. Silverman H: A digitalis therapy advisor. MAC TR-143, 1975

4. Ogilvie RI, Reudy J: An educational program in digitalis therapy. JAMA 222:50-55, 1972

5. Doherty JE: Digitalis Glycosides: Pharmacokinetics and their clinical implications. Ann Intern Med 79:229-238, 1973

6. Doherty JE, Flanigan WJ et al: Tritiated Digoxin XIV, Enterohepatic circulation, absorption and excretion studies in human volunteers. Circulation 42:867-878, 1970

7. Doherty JE, Perkins WH, Mitchell GK: Tritiated digoxin studies in human subjects. Arch Intern Med 108:531-539, 1961

8. Peck CC, Sheiner LB et al: Computer-assisted digoxin therapy. N Eng J Med 289:441-446, 1973

9. Jelliffe RW, Buell J, Kalaba R et al: A computer program for digitalis dosage regimens. Math Biosci 9:179-193, 1970

10. Jelliffe RW, Buell J, Kalaba R: Reduction of digitalis toxicity by computer-assisted glycoside dosage regimens. Ann Intern Med 77:891-906, 1972

11. Sheiner LB, Rosenberg B, Melmon K: Modelling of individual pharmacokinetics for computer-aided drug dosage. Computers and Biomedical Research 5:441-459, 1972

12. Davis R: Applications of meta level knowledge to the construction, maintenance and use of large knowledge bases. SAIL AIM-283, 1976

13. Long W: Question answering in Owl. Automatic Programming Group Internal Memo

14. Hawkinson L: The representation of concepts in Owl. Proceedings of the Fourth IJCAI, 1975

15. Stallman RM, Sussman GS: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. AI Memo 380, 1976

16. Mikelsons M: Computer assisted application description. Second ACM Symposium on Principles of Programming Languages, 1975

17. Carbonell JR, Collins AM: Natural semantics in artificial intelligence. Third IJCAI, 1973

18. Dahl OJ, Dijkstra EW, Hoare CAR: Structured Programming. Academic Press, 1972

19. Bosyj M: A program for the design of procurement systems. MIT Laboratory for Computer Science TR-160, 1976

20. Martin WA: A theory of English grammar. MIT Laboratory for Computer Science Technical Memo (in preparation)

21. Sunguroff A: OWL interpreter reference manual. MIT Automatic Programming Group Internal Memo, 1976

22. Hewitt C: Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. MIT AI TR-258, 1972

23. Whorf BL: Language, thought and reality. JB Carroll (ed). MIT Press, Cambridge, Mass. and John Wiley and Sons, New York, 1956

CS-TR Scanning Project
Document Control Form

Date : 11/3/95

Report # LCS-TR-176

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 98 (104 IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): FOLLOW TITLE & ABSTRACT PAGES.

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-98) UNNUMBERED TITLE, BLANK, ABST, BLANK, CONTENTS (2), ILLUSTRATIONS, 6-96	
(99-104) SCANCONTROL, COVER, SPINE, TRGTS (3)	

Scanning Agent Signoff:
Date Received: 11/3/95 Date Scanned: 11/25/95 Date Returned: 11/27/95

Scanning Agent Signature: Michael W. Cooper

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 11/25/1995

M.I.T. Libraries
Document Services