Producing Explanations and Justifications of Expert Consulting Programs

William R. Swartout

January 1981

Laboratory for Computer Science

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

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

Producing Explanations and Justifications of Expert Consulting Programs

by

William R. Swartout

This report is a modified version of a thesis submitted to the Department of Electrical Engineering and Computer Science on December 18, 1980 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Traditional methods for explaining programs provide explanations by converting to English the code of the program or traces of the execution of that code. While such methods can provide adequate explanations of what the program does or did, they typically cannot provide justifications of the code without resorting to canned-text explanations. That is, such systems cannot tell why what the system is doing is a reasonable thing to be doing. The problem is that the knowledge required to provide these justifications is needed only when the program is being written and does not appear in the code itself. In the XPLAIN system, an automatic programming approach is used to capture some of the knowledge necessary to provide these justifications.

The XPLAIN system uses an automatic programmer to generate the consulting program by refinement from abstract goals. The automatic programmer uses a domain model, consisting of facts about the application domain, and a set of domain principles which drive the refinement process forward. By keeping around a trace of the execution of the automatic programmer it is possible to provide justifications of the code using techniques similar to the traditional methods outlined above. This paper discusses the system described above and outlines additional advantages this approach has for explanation.

Keywords: Explanation, Automatic Programming, Expert Systems

Thesis Supervisor: Dr. Peter Szolovits

Title: Associate Professor of Computer Science and Electrical Engineering

CONTENTS

FIGURES

Acknowledgements

I would like to thank all those who made this thesis possible, and in particular, the following:

Peter Szolovits for suggesting this thesis topic and for being an exemplary supervisor;

Randall Davis and Robert Fano for their helpful comments on early drafts of this document;

Bill Martin for introducing me to the area of knowledge based application systems;

Ramesh Patil for his considerable help in programming the pattern-matcher and for being a very insightful sounding-board for ideas;

Bill Long for his comments on automatic programming and his extensive knowledge of digitalis therapy;

Dr. Steven Pauker for advice on medical aspects and help in making arrangements at Tufts New England Medical Center;

Harold Goldberger and Ken Church for spirited and thought-provoking conversations while I was trying to formulate my ideas;

and, finally, Lisa Berlin and my parents for cheering me up when I was glum.

This research was supported (in part) by the National Institutes of Health Grant No. 1 P01 LM 03374-01 from the National Library of Medicine.

1. Introduction

Computers can be inscrutable. Too often, the person who tries to have his bank correct an error in his account, have a duplicate charge removed from his credit card statement, or stop the local department store from sending him a bill for zero dollars every month finds that dealing with a computer and the bureaucracy that surrounds it can be a mystifying, frustrating, and time-consuming process. In part, these difficulties have arisen because the designer of a data processing system is primarily concerned with processing efficiency. He relies on a staff of computer support personnel to deal with problems and questions as they occur. However, even in relatively simple areas such as accounting and billing this approach to system design has not been an overwhelming success, and it becomes less appropriate as we become more ambitious and attempt to use the computer to solve more sophisticated problems.

The area of medical consultant programs[1] provides a case in point. The design desiderata of a consultant program are quite different from those for an accounting program. In designing a consultant program, we must consider what sorts of capabilities we are trying to provide for the physician user. If we consider the interaction between a physician and a human consultant, we realize that it is not just a simple one-way exchange where the physician provides data and the consultant provides an answer in the form of a prescription or diagnosis. Rather, there is typically a lively dialog between the two. The physician may question whether some factor was considered or what effect a particular finding had on the final outcome. Viewed in this light, we realize that a computer program which only collects data and provides a final answer will probably not be found acceptable by most physicians. In addition to providing diagnoses or prescriptions, a consultant program must be able to explain what it's doing and justify why it's doing what it's doing.

---

1. Some medical consultant programs include: MYCIN—a program that aids physicians with antimicrobial therapy [Shortliffe76], INTERNIST—a program that makes diagnoses in internal medicine [Pople77] and PIP—a program that makes diagnoses primarily in the area of renal disease [Pauker76].

If a program can explain its reasoning processes accurately, user acceptance can be more easily obtained since the user can assure himself that the program is doing reasonable things. An explanatory capability can also serve as a pedagogical aid. A student or practitioner may use the system and improve his understanding of the program's area of expertise by comparing his own reasoning with that of the system. An explanation facility may also be able to elucidate any assumptions and simplifications built into the consultant system which may limit its applicability in certain special types of cases. Finally, as system designers, we have found that an explanatory capability often aids us in debugging the system.

The next section will describe the Digitalis Therapy Advisor, the program we have chosen as a testbed for our ideas about explanation, and some aspects of digitalis therapy which readers without medical backgrounds will probably need to understand the remainder of the thesis. While we have concentrated on the problem of providing explanations to medical personnel, we do not feel that the need for explanation is limited to medicine nor do we feel that the techniques we have developed for explanation and justification are limited to medical applications. Medical programs provide a good testbed for the general problem we are attacking, which is to be able to explain a consulting program to the audience it is intended to serve.

1.1 Digitalis Therapy and the Digitalis Advisor

The digitalis glycosides are a group of drugs that were originally derived from the foxglove, a common flowering plant. This group includes digoxin, digitoxin, ouabain, cedilanid 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 (CHF) 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 [Ogilvie72, Doherty73].

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 weight, electrolyte balance, and history of heart damage (to name a few) that may cause the patient to be more sensitive to digitalis and thus 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. More frequently, potentially life-threatening abnormal heart rhythms indicate digitalis intoxication.

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 [Doherty61, Doherty70, Doherty73]. Digitalis leaves the body through two routes. Much of the drug is excreted

in the urine, and the rest is metabolized in 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 twenty per cent of hospitalized patients receiving digitalis show toxic symptoms, and that the mortality rate among these patients may be as high as thirty per cent [Ogilvie72, Peck73].

1.1.1 Digitalis Sensitivities

In Chapter 3, we will describe how the XPLAIN system synthesizes the portion of a digitalis advisor that checks and corrects for increased sensitivity to digitalis. In this section, we will describe in a little more depth what the digitalis sensitivities are, and what causes them.

In administering digitalis (and many other drugs) a physician must deal with the possibility that his patient may be more sensitive to the drug (for whatever reason) than the average patient. If a physician knows those factors that make a patient more sensitive he can reduce the likelihood of overdosing (or underdosing) the patient by adjusting the dose depending on whether he observes the sensitizing factors or not.

Over the years, a number of factors have been identified that increase the

automaticity of the heart.[2] These include: a low level of serum potassium (hypokalemia), a high level of serum calcium (hypercalcemia), damage to the heart muscle (cardiomyopathy), and a recent myocardial infarction (among others). When these exist in conjunction with digitalis administration, the automaticity can be increased substantially. We will concentrate on just the first three in the program synthesis presented in Chapter 3.[3]

1.1.2 The Digitalis Therapy Advisor Testbed

A few years ago, a Digitalis Therapy Advisor was developed at MIT by Pauker, Silverman, and Gorry [Silverman75, Gorry78]. This program was later revised and given a preliminary explanatory capability [Swartout77a, Swartout77b]. The limitations of these explanations (and of those produced by similar techniques) will be discussed in the next section. This program differed from earlier digitalis advisors [Peck73, Jelliffe70, Jelliffe72, Sheiner72] in two important respects. First, when formulating dosage schedules, it anticipated possible toxicity by taking into account the factors that increased digitalis sensitivity and it reduced the dose when those factors were present. Second, the program made assessments of the toxic and therapeutic effects which actually occurred in the patient after receiving digitalis to formulate subsequent dosage recommendations. This program worked in an interactive fashion. The program would ask the physician for data about the patient and produce recommendations after that data was entered. When the dose of digitalis was being adjusted, the physician was asked to consult with the program again to assess the patient's response. This is the program we used as a testbed for our work in explanation and justification.

---

2. In the normal heart, there is a place in the left atrium called the sino-atrial (SA) node, which sets the pace for the heart. Under the right circumstances, other parts of the heart can take over the pace-setting function. Sometimes this can be life-saving if, for example, the SA node is damaged. But at other times it can be life-threatening, since several pace-makers operating simultaneously tend to increase the likelihood of setting up a dangerous arrhythmia. When we say that digitalis increases the automaticity of the heart, we mean that digitalis increases the tendency of other parts of the heart to take over the pace-setting function from the SA node.

3. The XPLAIN system currently only knows about the first three factors, although it would not be particularly difficult to expand it to cover the others.

1.2 Kinds of Questions

In the spring of 1979, we conducted a series of informal trials in an attempt to discover what sorts of questions occurred to medical personnel as they ran the Digitalis Advisor. In this trial, medical students and fellows were asked to run the program and ask questions (verbally) as they occurred to them. The author attempted to answer these questions. The interactions were tape recorded and later transcribed.

No formal analysis of the data was attempted, but examination of the transcripts did give us a good feeling for the sorts of questions that a doctor might have while running a consulting program.

One type of question asked directly about the methods the program employed:

> *Subject: "How do you calculate your body store goal? That's a little lower than I anticipated."*

This sort of question could be answered by the explanation routines of the old Digitalis Advisor. It can also be answered by the system presented in this thesis.

Another sort of question asks for a justification of what the program is doing:

> *Subject: (peruses recommendations) "Why do we want to make a temporary reduction?*
>
> *Experimenter: "We're anticipating surgery coming up, and surgery, even non-cardiac surgery can cause increased sensitivity to digitalis, so it wants to hold digitalis."*

This is exactly the sort of question we are concentrating on in this thesis.

Finally, there are some sorts of questions that came up that this thesis does not address. Most of these seem to involve confusion about the meaning of terms:

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

> *Subject: "now this question...I'm not really sure...'renal function stable' does*

it mean stable abnormally or...because I mean, the patient's is not normal it's stable at the present time."

Experimenter: "That's what it means"

1.3 Previous Approaches to Explanation

A number of different approaches have been taken to attempt to provide programs with an explanatory capability. The major approaches include using 1) previously prepared text to provide explanations and 2) producing explanations directly from the computer code and traces of its execution. These approaches will be discussed below.

The simplest way to get a computer to answer questions about what it is doing is to figure out what questions will be asked and then store the answers to those questions as English text. The computer can only display the text that has been stored. This is called *canned text*, and explanations produced by displaying canned text are called canned explanations. The simplest sorts of canned explanations are error messages which the computer displays when something goes wrong. For example, a medical program designed to treat adults might print the following message if someone tried to use it to treat an infant:

```
THE PATIENT IS TOO YOUNG TO BE TREATED BY THIS PROGRAM.
```

It is relatively easy to get a small program to provide English explanations of what it is doing using this canned text approach. First we write the program. Then we associate with each part of the program canned English text which explains what that part of the program is doing. Then when the user wants to know what's going on, the computer merely displays the text associated with what it's doing at the moment. However, the fact that the program code and the text strings that explain that code can be changed independently makes it difficult to guarantee consistency between what the program does and what it claims to do. Another problem with the canned text approach is that all questions and answers must be anticipated in advance and the programmer must provide answers for all the questions that the user might ask. For large systems, that is a nearly impossible task. Finally, the system has no conceptual model of what it is saying.

Fig. 1. Explanation of How the System Checks Hypercalcemia

```
TO CHECK SENSITIVITY DUE TO CALCIUM I DO THE FOLLOWING STEPS:
1. I DO ONE OF THE FOLLOWING:
    1.1 IF EITHER THE LEVEL OF SERUM CALCIUM IS GREATER THAN 10 OR IV[4]
    CALCIUM IS GIVEN THEN I DO THE FOLLOWING SUBSTEPS:
        1.1.1 I SET THE FACTOR OF REDUCTION DUE TO HYPERCALCEMIA TO 0.75.
        1.1.2 I ADD HYPERCALCEMIA TO THE REASONS OF REDUCTION.
    1.2 OTHERWISE, I REMOVE HYPERCALCEMIA FROM THE REASONS OF REDUCTION
    AND SET THE FACTOR OF REDUCTION DUE TO HYPERCALCEMIA TO 1.00.
```

Fig. 2. Code to Check for Increased Digitalis Sensitivity Due to Hypercalcemia

```
[(CHECK (SENSITIVITY (DUE (TO CALCIUM))))
 METHOD: (OR
          (IF-THEN
           (OR
            (GREATER-THAN 10. (QUANTA SERUM-CALCIUM))
            (IV-STATUS CALCIUM GIVEN))
           (BECOME (FACTOR REDUCTION-HYPERCALCEMIA 0.75)):1,
           (BECOME-ALSO
            (REASONS REDUCTION HYPERCALCEMIA)):)
          (AND:2
           (UNBECOME (REASONS REDUCTION HYPERCALCEMIA)):
           (BECOME (FACTOR REDUCTION-HYPERCALCEMIA 1.0)):2))]
```

That is, to the computer, one text string looks much like any other, regardless of the content of that string. Thus, it is difficult to use this approach if we want our system to provide more advanced sorts of explanations such as suggesting analogies or if we want to be able to give explanations at different levels of abstraction.

Another approach to explanation is to produce explanations directly from the program [Davis76, Shortliffe76, Swartout77a, Swartout77b, Winograd71]. That is, the explanation routines examine the program which is run by the computer. Then by performing relatively simple transformations on the code these explanation routines can produce explanations of how the system does things. For example, the Digitalis Advisor uses the code shown in Figure 2 to check for increased digitalis sensitivity caused by

4. Intravenous

increased serum calcium. The system can examine the code and produce an English explanation of what the code does (shown in Figure 1).

The Digitalis Advisor, like most similar systems, also keeps a trace of the execution of the code. That is, as the code is executing, the system records what happens. This trace can then be examined by the explanation routines to tell what the system did for a particular patient. Figure 3 explains how the system checked for myxedema. The system also has a limited ability to explain why it is asking the user a question. Figure 4 shows the system's response when the user wants to know why he is being asked about serum calcium.

Fig. 3. Explaining How Thyroid Function Was Checked

```
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 DEGRADABLE CONDITIONS.  THE
     DEGRADABLE 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.
```

Fig. 4. Telling Why a Question is Asked

```
WHAT IS THE LEVEL OF SERUM CALCIUM?

ENTER NUMERICAL VALUE ====> why? (The user types "why?" indicating that he
questions why he is being asked to supply the value of serum potassium.)

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 CALCIUM.
```

Since the explanation routines only perform simple transformations on the program code, the quality of the explanations produced in this manner depends to a great degree on how the system code is written. In particular, the basic structure of the program is not altered significantly, and the names of variables in the explanation are basically the same as those in the program. If the explanations are to be understandable, the expert system must be written so that its structure is easily understood by anyone familiar with its domain of expertise, and the variable and procedure names used in the program must represent concepts which are meaningful to the user.

This method of producing explanations has some advantages. It is relatively simple. If the right way of structuring the problem can be found, it does not impose too great a burden on the programmer; since the explanations reflect the code directly, consistency between explanation and code is assured. Despite these advantages, there are some serious problems with this technique.

It may be difficult or impossible to structure the program so that the user can easily understand it. The fact that every operation performed by the computer must be explicitly spelled out sometimes forces the programmer to program operations which a physician would perform without thinking about them. That problem is illustrated in Figure 3. Steps 2.1, 2.2, and 2.4 are somewhat mystifying. In fact, these steps are needed by the system so that it can record what sensitivities the patient had that made him more likely to develop digitalis toxicity. These steps are involved more with record keeping than with medical reasoning, but they must appear in the code so that the computer will remember why it made a reduction. Since they appear in the code, they are described by the explanation routines, although they are more likely to confuse a physician-user than enlighten him. An additional problem is that it's difficult to get an overview of what's really going on here. While the system is explicit about record keeping, it isn't very explicit about the fact that it's going to reduce the dose, though it hints at a reduction by saying that the "factor of reduction" is being set to 0.67.

An additional problem, and the primary one we will address in this thesis is that while this way of giving explanations can state *what* the system does or did, it can't state *why* the system did what it did. That is, the system can't give justifications for its actions. In the explanations given above, the system can't state that it reduces the dose because

increased calcium causes increased automaticity. The information needed to justify the program is the information that was used by the programmer to write the program, but it does not have to be incorporated into the program for the program to perform successfully—just as one can successfully bake a cake without knowing why baking powder appears in the recipe. Since it is desirable for expert programs to be able to justify what they do as well as do it successfully, we need to find a way of capturing the knowledge and decisions that went into writing the program in the first place. The remainder of this report will describe recent efforts we have made toward achieving that goal in the context of the Digitalis Therapy Advisor.

1.4 Providing Justifications

We need a way of capturing the knowledge and decisions that went into writing the program. One way to do this is to give the computer enough knowledge so that it can write the program itself and remember what it did. The notion of having one program write another program is not new. It is called automatic programming and has been researched considerably [Balzer77, Barstow77, Green79, Long77, Manna77]. Using an automatic programmer to help in producing explanations is a new idea. We will describe how the system works below.

1.4.1 System Overview

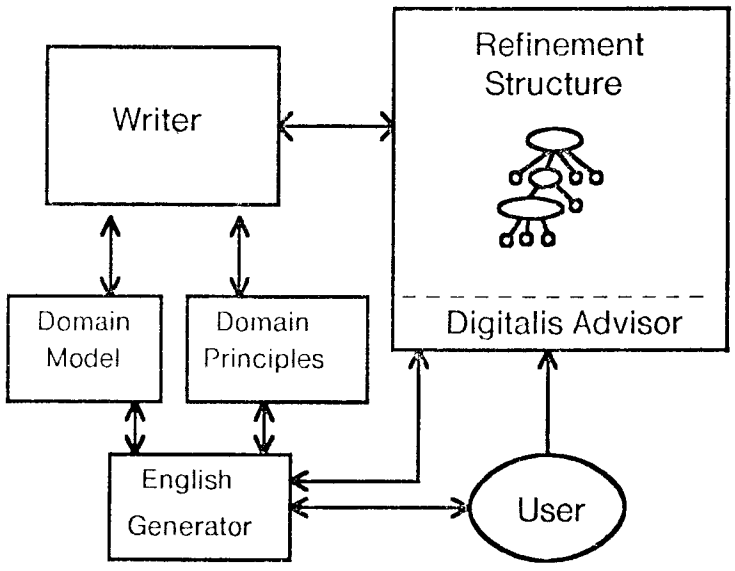
An overview of the system is given in Figure 5. The system has five parts: a Writer, a Domain Model, a set of Domain Principles, an English Generator, and a Refinement Structure. The Writer is an automatic programmer. It writes the Digitalis Advisor. The Domain Model and the Domain Principles contain knowledge about the domain of expertise. Thus, in this case, they contain information about digitalis and digitalis therapy. They provide the Writer with the knowledge it needs to write the Digitalis Advisor. The Refinement Structure can be thought of as a trace left behind by the Writer. It shows how the Writer develops the Digitalis Advisor. When a physician user runs the Digitalis Advisor, he can ask the system to justify why the program is doing what it is doing. The Generator gives him an answer by examining the Domain Model, the Domain Principles, the Refinement Structure, and the step of the Advisor currently

being executed. If we wanted to write a new program covering a new medical domain, we would have to change the Domain Model and the Domain Principles, but we should not have to change the Writer or the English Generator.

The Refinement Structure is created by the Writer from the top level goal (in this case Administer Digitalis) as it writes the Digitalis Advisor. The Refinement Structure is a tree of goals, each being a refinement of the one above it in the tree (see Figure 6). By "refining a goal" we mean taking a goal and turning it into more specific subgoals. For example, if we had the abstract goal of getting from New York to San Francisco, some refinements (or subgoals) of that goal would be getting to the airport, buying a ticket, flying to San Francisco, and so forth. Looking at Figure 6, we see that the top of the tree is a very abstract goal, in this case, Administer Digitalis. This goal is refined into less abstract steps by the Writer. These more specific steps are steps the system executes to administer digitalis. For example, one such step is to Anticipate Toxicity, that is, to anticipate whether the patient may become toxic due to increased digitalis sensitivity. The writer then refines this more specific goal to a still more specific goal. Eventually, the level of system primitives is reached. System primitives are operations which are built-in. Normally they are very basic, simple operations, so the fact that they cannot be

---

Fig. 5. System Overview

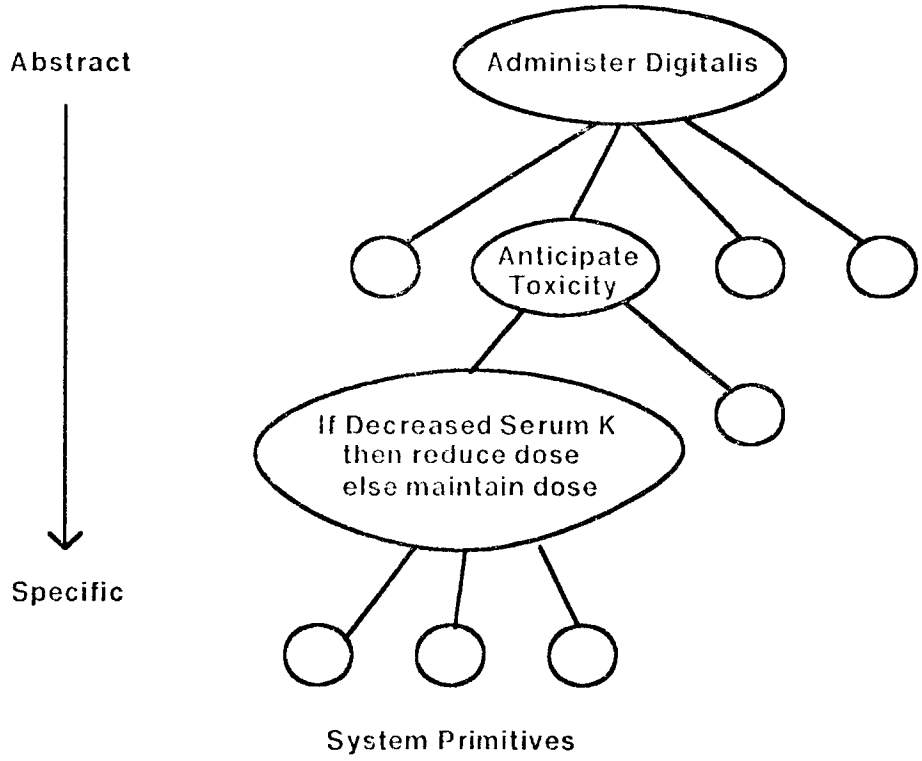


explained is usually not a problem. Typical primitives include those that perform arithmetic operations like PLUS and TIMES and those that set variables to a particular value. The structures at this primitive level are the Digitalis Advisor, the program that we wanted the automatic programmer to produce.

The Domain Model is a model of the facts of the domain. In this case, it is a model of the causal relationships in digitalis therapy. A simplified portion of the model is shown in Figure 7. In this model, the boxes are states, and the arrows represent causality. This model shows some of the effects of increased digitalis. It also shows that hypercalcemia and hypokalemia can cause increased automaticity. In a certain sense, these facts correspond to the sorts of facts that a medical student learns in class during the first two years of medical school. These facts are static. That is, they have no notion of process. The model says that increased digitalis can cause a change to ventricular fibrillation but it doesn't say what to do about it. Medical students go to medical school for an additional two years and acquire these procedures by observing more

---

Fig. 6. Refinement Structure



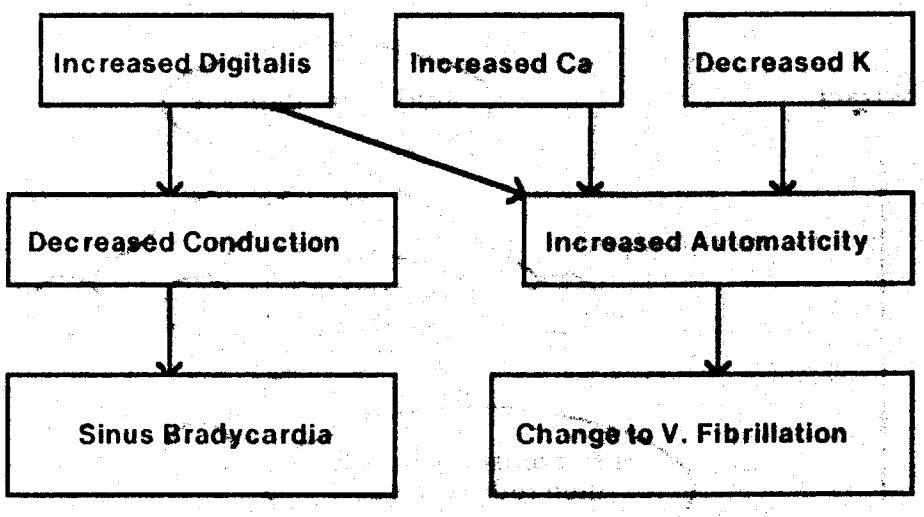
experienced personnel as they practice medicine on the wards. The set of Domain Principles provides the Writer with this sort of procedural knowledge.

Domain Principles tell the Writer how something (such as prescribing a drug or analyzing symptoms) should be done. They guide it as it refines abstract goals to more specific ones. A (somewhat simplified) Domain Principle appears in Figure 8.[5] This particular Principle helps the writer in anticipating digitalis toxicity. It represents the common sense notion that if one is considering administering a drug, and there is some factor that enhances the deleterious effects of that drug, then if that factor is present in the patient, less drug should be given. This principle has three parts: a Goal, a Domain Rationale, and a Prototype Method.

The goal tells the Writer what it is that the Principle can help it do. In this case, the Principle can help the Writer in anticipating toxicity. The Domain Rationale is a pattern which is matched against the Domain Model to see where it is appropriate to

---

Fig. 7. Domain Model



---

5. Domain Principles are composed of variables and constants. Variables appear in boldface in Figure 8. When the writer is matching, a variable in a pattern will match anything which is of the same kind as itself. Thus, the variable finding would match increased serum-Ca or decreased K, since increased serum-Ca and decreased K are both kinds of findings.

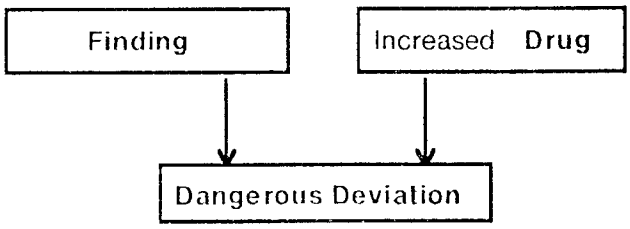
achieve the goal. In the example, the system will look in the Domain Model to match a **finding** (e.g. increased Ca) which causes some sort of a **dangerous deviation** (e.g. change to ventricular fibrillation) which is also caused by an increased level of the drug. By looking at the Domain Model, we can see both increased Ca and decreased K will match as findings, since both can cause a change to ventricular fibrillation.

The Prototype Method is an abstract method which tells the system how to accomplish the goal. Once the Domain Rationale has been matched, the Prototype Method is instantiated for each match of the Domain Rationale (see Figure 9). When we say that we instantiate the Prototype Method, that means that we create a new structure where the variables in the Prototype Method have been replaced by the things they matched. In this case, two structures would be created. In the first, **finding** would be replaced by increased serum Ca and **drug** would be replaced by digitalis. In the second, **finding** would be replaced by decreased serum K and **drug** would again be replaced by digitalis. Note that now, with these new structures, we have changed the single abstract problem of how to anticipate toxicity into several more specific ones, such as how to determine whether increased serum K exists, how to reduce the dose, and how to maintain it.

Fig. 8. An Example of a Domain Principle

Goal: Anticipate **Drug** Toxicity

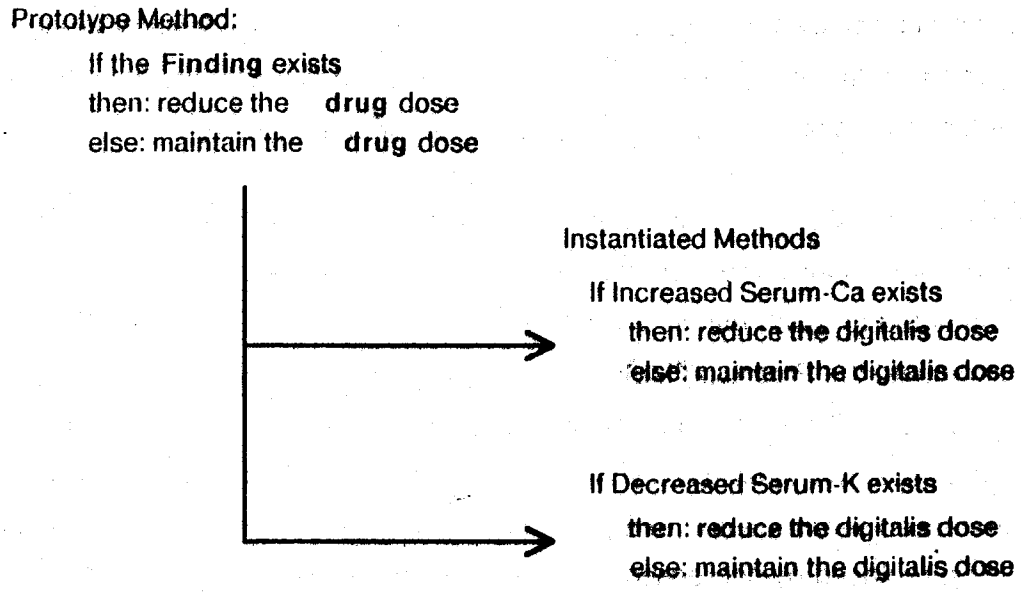
Domain Rationale:



Prototype Method:

If the **Finding** exists
then: reduce the **drug** dose
else: maintain the **drug** dose

Fig. 9. Instantiation of a Prototype Method



After instantiation, the more specific goals of the Prototype Method are placed in the Refinement Structure as sons of the goal being resolved. If we look at Figure 6, we can see that the instantiated Prototype Method that checks for decreased serum K has been placed below the Anticipate Toxicity goal. Once they have been placed in the Refinement Structure, the newly instantiated goals become goals for the writer to resolve. For example, after the Writer applied this Domain Principle, it would have to find ways of determining whether increased calcium existed in the patient, whether decreased potassium existed, and ways of reducing and maintaining the dose. The system continues in this fashion, refining goals at the bottom of the structure and growing the tree down and down until eventually the level of system primitives is reached. At that point, the system is finished, and the goals at the very bottom actually represent a runnable computer program. The system also has to deal with transformations of program structure and constraints on the refinement process and the selection of Domain Principles. We will deal with those aspects in Chapter 3.

Once the refinement process is complete, we have a working expert system. A sample interaction with the system is given in Figure 10. The explanations were produced by finding the Domain Principle which caused the step in question to appear in the program. The domain rationale associated with that principle was then converted to

Fig. 10. A Sample Interaction Providing Justifications

Please enter the value of serum-k: why?

The system is anticipating digitalis toxicity. Decreased serum-k causes increased automaticity, which may cause a change to ventricular fibrillation. Increased digitalis also causes increased automaticity. Thus, if the system observes decreased serum-k, it reduces the dose of digitalis due to decreased serum-k.

Please enter the value of serum-k: 3.7

Please enter the value of serum-ca: why?

(The system produces a shortened explanation, reflecting the fact that it has already explained several of the causal relationships in the previous explanation. Also, since the system remembers that it has already told the user about serum-K, it suggests the analogy between the two here.)

The system is anticipating digitalis toxicity. Increased serum-ca also causes increased automaticity. Thus, (as with decreased serum-k) if the system observes increased serum-ca, it reduces the dose of digitalis due to increased serum-ca.

Please enter the value of serum-ca: 9

English (with pattern variables replaced by the facts in the Domain Model they matched). That step produced the first two sentences of the explanation. The last sentence is just the instantiated version of the Prototype Method of the Domain Principle. These explanations should be compared with those presented in Figure 4 to appreciate the improvement that is possible with this approach.

1.5 A Summary of Major Points

First, we have argued that to be acceptable, consultant programs must be able to explain what they do and why. Second, we have described the various ways that traditional approaches fail to provide adequate explanations and justifications. Major failings include: 1) the inability of such approaches to justify what the system is doing because the knowledge required to produce justifications is not represented within the

system, and 2) a lack of distinction between steps required just to get the computer-based implementation to work, and those that are motivated by the application domain. Third, we have outlined an approach which captures the knowledge necessary to improve explanations. This involves using an automatic programmer to generate the performance program. As the program is generated, a refinement structure is created which gives the explanation routines access to decisions made during the creation of the program. The improvement in explanatory capabilities that is achieved is due more to the availability of this refinement structure than to the use of more sophisticated English generation functions, since the explanation routines used in this thesis do not differ greatly from those used in the old Digitalis Advisor.

In the remainder of the thesis: Chapter 2 outlines XLMS and the XLMS interpreter, the knowledge base tools used to build the system. Chapter 3 describes and motivates the design of the automatic programmer and traces its operation as it creates the part of the Digitalis Advisor that deals with digitalis sensitivities. Chapter 4 describes how a quite different part of the Digitalis Advisor, the code for assessing toxicity, is written. That chapter actually presents two different implementations to show the flexibility in implementation the XPLAIN system allows. Chapter 5 describes the routines that actually generate the explanations, and the thesis concludes with a discussion of the interrelationships between the automatic programmer, the performance program, and explanation.

2. System Building Tools: XLMS and the XLMS Interpreter

2.1 XLMS Notation

The XPLAIN system uses XLMS to manage its knowledge base. XLMS (which stands for e**X**perimental **L**inguistic **M**emory **S**ystem) was developed primarily by William Martin, Lowell Hawkinson, Peter Szolovits and members of the Clinical Decision Making and Automatic Programming Groups at MIT. Since it is not necessary to have a complete understanding of the intricacies of XLMS to understand the XPLAIN system, this section is intended to elucidate only as much of XLMS as is required to comprehend the remainder of the thesis. For a more complete discussion of the design goals and implementation of XLMS see [Hawkinson80].

For the purposes of this paper, perhaps the best way to think of XLMS is that it is an extension of LISP that allows one to use *structured names*. In LISP, atoms are used to name variables and functions. In the XPLAIN system, variables and procedures are named by XLMS concepts—the difference is that these concepts can have a substructure which can be taken apart and examined, while LISP atoms are indivisible.

2.1.1 XLMS Concepts

In XLMS, every concept is composed of an ilk, a tie and a cue and is written as:

```
[(<ilk>*<tie> <cue>)]
```

or, to pick an actual example from the XPLAIN system:

```
[(LEVEL*R DIGITALIS)]
```

The *ilk* of a concept is itself a concept. It tells what kind of a concept this is. Thus, the example concept is a kind of level. The *cue* of a concept is either a concept or a LISP atomic symbol (more about symbols later). It indicates what it is that makes this concept different from others with the same ilk. The example represents the "level of digitalis": a particular kind of level. Finally, the *tie* of a concept indicates the relationship between the ilk and the cue. In this case, the tie is R for "role". Role ties are used to indicate slots in concepts. Thus this concept represents the "level" slot in the concept

"digitalis". This is one implementation[6] of the notion of slots and frames as described by Minsky [Minsky75]. In the XPLAIN system, ties are used by the generator in determining how to convert a concept to English.

There are several other ties that are used extensively in the XPLAIN system. These are listed in Table I together with examples of their use.

Concepts may be given *labels*. The notation is:

```
[<label> = <concept>]
```

Table I. Types of Ties

Tie	Name	Example Use	English Form	Purpose
*f	function	`[(ball*f red)]`	(the) red ball	modifies
*r	role-in	`[(color*r ball)]`	(the) color of (the) ball	slot-filling
*i	individual	`[(ball*i 1)]`	ball	instantiates individual
*o	object	`[(treat*o patient)]`	treat (the) patient	verb-object
*s	species	`[dog = (animal*s "dog")]`	dog	(see text)
*measure		`[(pvcs*measure salvos)]`	salvos of pvcs	measure
*c	call	(these ties are discussed with the MINT interpreter)		
*d	definition			
*b	begin	(these ties are used to define links; see Chapter 5)		
*e	end			

6. See [Martin79] for a more complete discussion.

English words are defined in XLMS by creating a concept which has a tie of *s or *i and a cue which is a LISP atomic symbol which is the English word. The ilk of the concept indicates its kind. Additionally, the concept is usually assigned a label which is the English word. Thus, we could define *collie* in the following way:

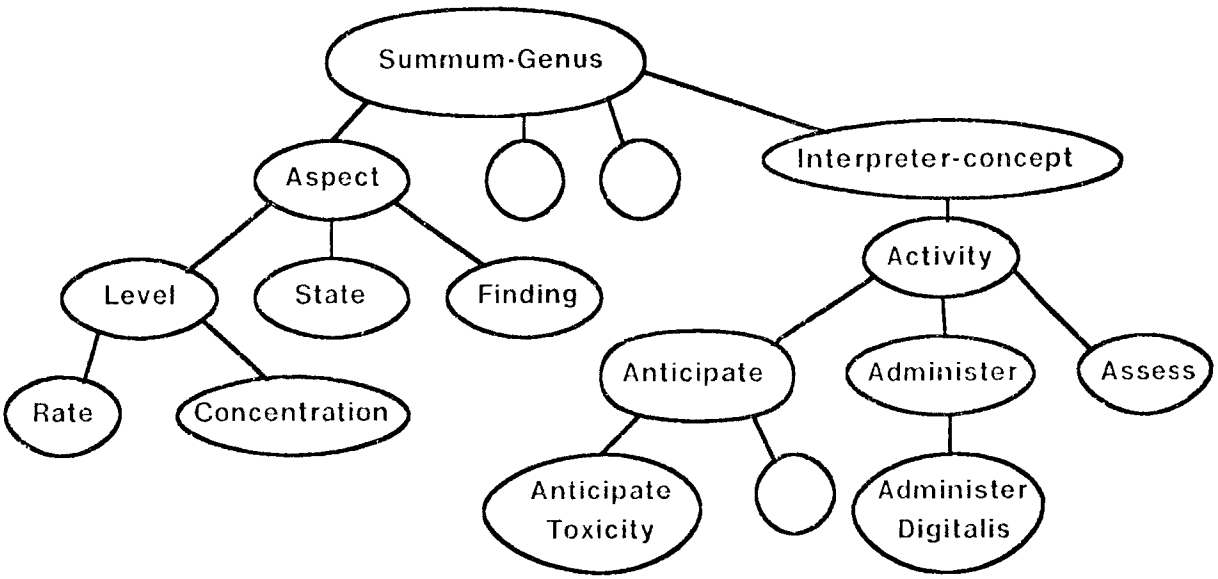
```
[collie = (dog*s "collie")]
```

As was indicated above, concepts in XLMS are organized into a kind hierarchy. The root concept is [summum-genus] (see Figure 11) and is pre-defined in XLMS. Like atomic symbols in LISP, concepts in XLMS are unique.

2.1.2 Attachments

In LISP, it is possible to associate lists and atoms relating to a particular atomic symbol with that symbol by placing them on that symbol's property list. In XLMS, one can associate concepts relating to a concept, with that concept, by *attaching* them. The XLMS notation is:

Fig. 11. The Kind Hierarchy



This figure only shows a portion of the hierarchy

```
[<concept> #<attachment-relation> <attached-concept1>...
                                          <attached-conceptN>]
```

or, for example:

```
[(input*r plus) #e  A B]
```

The attachment-relation specifies how the concept and the attached concept (called the *attachment*) are related. In the XPLAIN system, the most frequently used attachment-relations are `#e` (exemplar), `#f` (function), `#q` (equivalence), `#c` (characterization) and `#m` (meta-characterization). `#e` is primarily used for slot-filling in the XPLAIN system. The example above states that A and B are inputs to plus. `#f` is similar to the tie `*f` and is used to associate descriptors with concepts. For example, `[ball #f red]` is a red ball. `#q` indicates that a concept and the attached concept are equivalent. `#c` denotes that a concept can be characterized as the attached concept. For example, `[Boston #c metropolis]` says that Boston can be characterized as a metropolis. Note that characterizing A as B is very similar to placing A under B in the kind hierarchy. The differences between the two stem more from the effect they have on the XLMS knowledge base than from their intention. Placing A under B in the kind hierarchy creates a permanent structure, while attachments can be removed. The built-in functions of XLMS tend to make it easier to work with the kind hierarchy than with attachments. Typically, primary characterizations are placed in the kind hierarchy while secondary ones are indicated by attachments. `#m` is used to meta-characterize a concept, that is, to provide information about where the concept comes from or how it should be interpreted. Some additional attachments will be introduced later when their use is discussed.[7]

---

7. It is also possible to specify a reverse attachment. For example, `[Boston e#c metropolis]` characterizes Boston as a metropolis *and* says that one of the exemplars of a metropolis is Boston.

2.1.3 Sequences

Sequences, which are a lot like LISP lists, provide a means for grouping concepts together. They are indicated in XLMS notation by a list of concepts separated by commas:

```
[<concept-1>, <concept-2>, <concept-3>, <concept-n>]
```

Sequences are used extensively within the XPLAIN system to represent program fragments and sets of concepts.

2.1.4 XLMS Plexus

As the reader may have noticed, XLMS notation is delimited by square brackets. These brackets identify the concept as a piece of XLMS notation and delimit the scope of its attachments (if any). The first concept to appear after a left bracket is called the head of the plexus. If a plexus is contained within some piece of XLMS notation, the XLMS reader makes any attachments or builds any structure indicated by the plexus, and then replaces that plexus by the head of the plexus.[8]

2.1.5 Colon Anaphora

Colon anaphora provide a convenient shorthand for specifying the slots of a concept. If a concept appears in XLMS notation with a colon (or several colons) immediately following it (as in COLOR:) then the XLMS reader replaces that concept with a new concept whose ilk is the concept in the notation, whose tie is *r, and whose cue is the head of the plexus n levels in from the outside, where n is the number of colons in the notation. Thus, in the plexus:

```
[BALL
    [COLOR: #e RED]]
```

---

8. Thus [A #e [B #e C]] is equivalent to the two separate notations [A #e B] and [B #e C]. [[A]] is equivalent to [A].

COLOR: expands into (COLOR*R BALL). Uparrows ("↑") which appear in XLMS notation work like colons, except that the appropriate head is chosen by counting from the inside out instead of from the outside in.

2.2 The Phrase Generator

This section discusses the phrase generator, which is a low level English generator used by higher level generators. Although this section is very closely related to the higher level generators presented in Chapter 5, it has been placed here to give the reader some familiarity with manipulating XLMS concepts.

The phrase generator generates phrases from XLMS concepts. For example, given the XLMS expression:

```
[((pvcs*f dangerous)*f (induced*o (by*o digitalis)))]
```

the phrase generator would generate the phrase:

"dangerous pvcs induced by digitalis"

In XLMS, the tie of a concept indicates the relationship between the ilk and cue of the concept. Thus, *f indicates that the cue is a modifier of the ilk, while *o indicates that the cue is an object of the ilk, and *r indicates that the ilk is a role in the cue. Because the tie often determines the English form of a concept, the phrase generator has been organized around the types of ties.

The phrase generator is actually composed of a number of smaller generators. Each of these generators is capable of generating English for concepts with a particular tie. The top level generator first determines whether some other concept should be substituted for the concept passed to it as an argument. This could occur in several ways. If the audience is a medical audience and the argument is characterized as some other concept which is itself meta-characterized as a medical term, the other concept will be substituted for the original argument and a phrase will be generated for it. A flag

can also be set so that a pattern variable will be replaced by its value. If neither of these apply, the function determines whether the concept passed to it is an English word. This is a simple test. A concept corresponds to an English word if either its cue is a symbol or it is meta-characterized by the concept [use-label-as-name] which indicates that its label is the English word for the concept. If the concept does not correspond to an English word, the generator examines the tie of the concept and dispatches to the appropriate *tie-generator* for that tie. When a tie-generator is called, it breaks the concept apart and may invoke the top level generator recursively to generate English phrases for the parts of the concept.

2.2.1 Generator for *R

Most of the tie-generators are quite simple. The generator for concepts with ties of *r (for *role*) calls the phrase generator on the ilk of the concept, inserts the word "of" and calls the phrase generator again on the cue of the concept. Thus [(level*r serum-k)] is output as "the level *of* serum-k".[9]

2.2.2 Generator for *Measure

The tie of *measure is similar to *r. The difference is that we use this tie when we wish to express the concept of a certain amount or measure of something such as "a cup of sugar" or "a cup of coffee". Note that the phrase "cup of coffee" could refer to a *cup* filled with coffee, or to a certain amount of coffee. Concepts with ties of *measure are intended to represent only the latter meaning. Since we are primarily focusing on the coffee and not the cup, the ilk of the concept is coffee: [(coffee*measure cup)]. Note that the generator for *measure generates the cue before the ilk while the generator for *r does the reverse.

---

9. The generator inserts articles (i.e. the, a, an) where appropriate. The mechanism for accomplishing this is described later in the chapter.

2.2.3 Generator for *I

Unless the concept is a set, concepts with the tie of *i are converted to English by calling the phrase generator on the ilk of the concept, ignoring the cue of the concept. If the ilk of the concept is [set], then the concept is a set, and must be generated a little differently. The set of concepts A, B, and C is represented in XLMS as `[(set*i A,B,C)]`. To generate English for the set, the generator passes the cue of the concept to a function which generates conjunctions by making calls to generate phrase on the elements of the sequence `[A,B,C]` and inserting commas and "and" in the appropriate places.

To reduce the verbosity of the English, the generator for conjunctions factors the set where possible. For example, the approach outlined above would turn the concept:

```
[(set*i (assessment*r pvcs),
        (assessment*r av-block),
        (assessment*r bigeminy)]
```

into:

"The assessment of pvcs, the assessment of av-block, and the assessment of bigeminy"

However, the system notes that the elements of the set all have the same ilks and ties of *r. It factors the set by generating the ilk of the elements in plural form followed by the cues of the elements:

"The assessments of pvcs, av-block, and bigeminy"

2.2.4 Generator for *F

The generator for modifiers is a little more complex. Concepts with ties of *f are concepts which represent the modification of the ilk of the concept by the cue. In English, modifiers can either appear before or after the modified word. In the current implementation, if the cue of the concept is either a single word or an adjective, it is placed before the ilk. Otherwise, the cue follows the ilk. Thus, the concept [(pvcs*f

(severe*f extremely))] is "extremely severe pvcs", while [(block*f (on*o table))] is "the block on the table". Finally, as a special case, if the cue of the concept is [plural], the ilk is generated as a plural (e.g. [(book*f plural)] is generated as "books").

2.2.5 Generator for *O

In concepts with ties of *o, the ilk of the concept is something that takes an object (such as a verb or preposition) and the cue is the object. The generator first outputs the ilk, then calls the phrase generator to output the cue. If the ilk of the concept is a verb, the generator calls a special generator for verbs which constructs a verb with the appropriate tensed form. The form of the verb that should be generated is indicated either by modifying the verb by a *f tie[10] or by the setting of global registers.[11]

2.2.6 Generator for *Characterization

Normally, if a concept is characterized by a characterization attachment the characterization is not mentioned.[12] However, when a pattern is being described, if the concept being generated is the ilk of a concept with the tie of *characterization, then a relative clause is generated to describe the characterization.[13] Once the characterization is described, it is placed on a list of described characterizations and is not mentioned again. For example, if the system were generating English for the concept [pvcs] and the following concept existed in the knowledge base:

---

10. As in [(adjust*f -ed)]
11. Currently, the system can generate the infinitive form, the past and third person singular present tenses, and the present participle. It would not be difficult to extend this list if the need arose.
12. Although if the characterization is a more appropriate term to use as would be the case if it were a medical term and the answer was being directed at a medical audience the system will substitute the characterization for the original term.
13. The primary reason for making characterizations into concepts is that it *conceptualizes* the relationship between the object and its characterization. This allows us to make attachments to the relationship itself and so that we could indicate, for example, that this particular relationship should only be described to medical students but not to experienced doctors.

```
[(av-block*characterization
  ((finding*r (toxicity*f digitalis))*f (specific moderately)))]
```

the system would generate the phrase:

"av-block which is a moderately specific finding of digitalis toxicity."

2.3 The XLMS Interpreter

The XLMS interpreter (also called MINT, for Micro-INTerpreter) was written by the author to execute the code produced by the Program Writer. It was possible to make this interpreter quite simple since the complex and time consuming operations that take place within the XPLAIN system are performed while the Program Writer is creating the program—not while the interpreter is running. As the interpreter executes the code of the Digitalis Advisor, it creates an *event structure* which is a trace of the execution of the code. The syntax and semantics of the language will be briefly discussed below.

The BNF grammar for the interpreter is shown in Figure 12. And some examples of various types of calls are shown in Figure 13. As can be seen from the figures the interpreter differs from LISP in that subroutines can return multiple values—in this regard the language is similar to ALGOL or FORTRAN. However, functional calls are very similar to LISP: they return the value of the last expression evaluated.

The evaluation of variables and handling of arguments is also similar to (deep bound) LISP. The system associates values with variables by maintaining a list of variable/value pairs called an *association list*. In each pair, the first item is a variable and the second is its value. To find the value for a particular variable, the system examines pairs starting from the head of the association list and returns the value associated with the first pair whose variable is the one sought. If no value is found on the association list, the system examines the variable to see if a value has been attached to it using the #v (for value) attachment.[14] If no value is found, an unbound variable error message is displayed.

---

14. Since attachments to concepts are global, this mechanism allows us to give variables global values.

Fig. 12. BNF Grammar for the XLMS Interpreter

```
<subroutine-call> ::= [(<plan-name>*c <input-output-sequence>)]
<functional-call> ::= [(<plan-name>*c <input-specifier>,)]

<subroutine-definition> ::= [(<plan-name>*d <input-output-sequence>)
                              [method: #q <method-specifier>]]

<function-definition> ::= [(<plan-name>*d <input-output-sequence>)
                              [method: #q <method-specifier>]]

<method-specifier> ::= <method-step> | <method-step>,<method-specifier>

<method-step> ::= <subroutine-call> | <functional-call>

<plan-name> ::= xlms-concept

<input-output-sequence> ::= <input-specifier>,<output-specifier> |
                            <input-specifier>,

<input-specifier> ::= <null-seq> | <input-sequence> | <input-item>

<input-sequence> ::= [<input-sequence1>] | [<input-item>,]

<input-sequence1> ::= <input-item>,<input-sequence1> |
                      <input-item>,<input-item>

<input-item> ::= <xlms-variable> | <functional-call>

<output-specifier> ::= <null-seq> | <output-sequence> | <xlms-variable>

<output-sequence> ::= [<output-sequence1>] | [<xlms-variable>,]

<output-sequence1> ::= <xlms-variable>,<output-sequence1> |
                       <xlms-variable>,<xlms-variable>

<xlms-variable> ::= xlms-concept

<null-seq> ::= []
```

MINT provides a number of basic primitive operations for constructing and taking apart XLMS structures, for performing arithmetic operations, and for controlling program flow. Most of these functions are typical system primitives—their meanings should be clear and they will not be discussed. Two potentially confusing operators *will* be described here. Other more specialized operators will be discussed when they become relevant in the remainder of the thesis.

Fig. 13. Examples of Calls

`[(foo*c A,[B,C,D])]` — subroutine foo with input of A and outputs of B, C and D.

`[(foo1*c [A,B,C],)]` — functional subroutine with inputs A, B and C.

`[(foo2*c [],)]` — functional with no inputs.

`[(foo3*c [A,B],[])]` — subroutine with 2 inputs no outputs — rare

`[(foo6*c [],[])]` — subroutine with no inputs or outputs — even rarer

`[(foo4*c (foo5*c [A,]),B)]` — subroutine whose input is the output of the functional foo5 and whose output is b.

MSETQ is used for setting variables. It is similar to SETQ in LISP but with two differences. First, the order of the arguments is reversed since inputs precede outputs in the MINT interpreter. Second, when a new value is given to a variable, the new value is pushed onto the front of the association list. This amounts to rebinding a variable every time a new value is given to it. This constrains the programmer's ability to cause side-effects to variables and tends to encourage a programming style that changes variables explicitly rather than by side effect.

MIF-THEN is used to control program flow. The arguments to MIF-THEN consist of a two- or three-part sequence. The first element is a predicate which is evaluated. If it returns `[true]`, the second element of the sequence is evaluated. Otherwise, the third element (if present) is evaluated. Other control operators (such as case and cond statements) have been defined, and they can be used by the XPLAIN system, but the need to employ them has not arisen in the area of Digitalis Therapy that we have concentrated on.

When a call is made, the interpreter finds the appropriate plan to execute by searching up the kind hierarchy starting from the ilk of the call (that is, the plan-name) until it finds a subroutine-definition or a function-definition. The interpreter then binds up the input arguments, executes the plan, and if the plan is a subroutine, binds up the output arguments. Bindings are pushed on the association list when the plan is entered and popped off when it is exited.

As the interpreter executes programs, it can selectively create a trace of the execution of that program. Individual plans can be marked to indicate whether or not they should be traced, or to indicate that the plans they call should or should not be traced. A global variable may be set to denote that everything should (or should not) be traced. To record the execution of a plan, the interpreter creates a new individual instance of the concept [event]. The call and method used to execute the call are attached to the event, as well as the value of the system's association list on entrance and exit. This makes it easy to re-create the variable environment under which the plan was executed. Events also record the value returned by functional calls.

A simple XLMS method written by the automatic programmer for determining whether decreased serum potassium exists appears below:

```
[(((DETERMINE-WHETHER*O (DECREASED*O SERUM-K))*I 1)*D [[SERUM-K,],])
   [METHOD: #Q (MLESS-THAN*C
                        [[SERUM-K, (THRESHOLD*R (DECREASED*O SERUM-K))],])]]
```

This method has one input `[serum-k]` and no outputs, hence, it is a functional subroutine. The method has one step, which is a call to the system primitive `[mless-than]`. That function is passed two inputs: `[serum-k]` and `[(threshold*r (decreased*o serum-k))]`. Since it is the last (and only) call executed in the method, the value returned by `[mless-than]` will be the value returned by the subroutine for determining whether decreased serum-k exists.

3. Creating the Performance Program by Refinement

This chapter describes and motivates the design of the automatic programmer used by the XPLAIN system. The first section of the chapter describes the knowledge sources that the system needs to write the program. Later sections detail how the programmer itself works and show how it refines the portion of the Digitalis Advisor that checks for digitalis sensitivities.

The reader should realize that the primary motivation for this thesis was explanation, not automatic programming. The automatic programmer was only carried through to the extent required to show the feasibility of our ideas about explanation.[15] However, we feel that several interesting ideas have emerged from the synthesis of explanation and automatic programming. Most importantly, the use of two distinct interacting knowledge sources: the Domain Model and the Domain Principles.

3.1 Knowledge Sources: the Domain Model and Domain Principles

This section describes the Domain Model and the Domain Principles. As the name suggests, these components of the system depend on the application domain and are the parts of the system that would have to be changed if the application area changed.

The Domain Model represents the characteristics of the domain. In the case of digitalis therapy, these are the physiological effects of digitalis and other related substances on the patient. While this information is needed to figure out how to give digitalis, it is not enough. Another source of knowledge is needed—one that can outline the *process* of drug administration, subject to constraints imposed by the Domain Model.

---

15. For a more extensive discussion of issues such as the structure of automatic programming systems, plan selection, constraints and constraint propagation the reader should see [Barstow77, Barstow80, Balzer77, Green79, Long77, Manna77].

In the introductory chapter, we mentioned that medical students seem to experience the same sort of split between knowledge concerned with process and knowledge that is static. During the first two years of medical school, they learn a tremendous number of facts about the human body, yet it is hard for them to begin treating patients. What they lack is an understanding of the *process* of actually treating patients. During the last two years of medical school they acquire these processes by actually participating in the care of patients and by being instructed by senior members of the hospital staff. In a sense, they acquire a common sense understanding of what is involved in treating a patient. Domain principles are intended to supply knowledge of exactly this sort. They are used by the program writer as it creates the performance program by refinement.

3.1.1 The Domain Model

The Domain Model is a representation of what the system knows about the characteristics of its application area. In the version of the system being described here, the Domain Model is a (primarily causal) representation of the system's knowledge of the physiological actions of digitalis. Thus, it tells what an increased level of digitalis may be expected to cause, what factors may increase sensitivity to digitalis, and so forth. In the remainder of this subsection, the general characteristics of the Domain Model will be described, illustrated by examples from the domain of digitalis therapy where appropriate. The reader is cautioned that I do not yet regard the existing primitives of the Domain Model as a complete set in the sense that they are sufficient to represent any knowledge, rather the primitives presented here should be regarded as stepping stones on the way to a larger, more complete system.

Figure 14 is a simplified version of that part of the Domain Model concerned with factors that may make a patient more sensitive to digitalis. (The figure also appears in Chapter 1.) In this diagram, arrows represent causality. From this figure we can see that decreased potassium and increased digitalis both cause increased automaticity. While the figure may give some feeling for the sort of information we seek to represent and how we have represented it, there are a number of subtleties which must be dealt with but are not indicated in the figure. To understand how these situations are handled, we must take a closer look at the actual XLMS notations that are used.

The causal relationships in the Domain Model are represented as *causal links*. In the XPLAIN system, links have the general form:

[((<link-type>*b <source>)*e <destination>)]

<link-type> indicates the type of the link and <source> and <destination> have their obvious meanings. Thus, the causal relationship between decreased serum-k and increased automaticity would be expressed as:

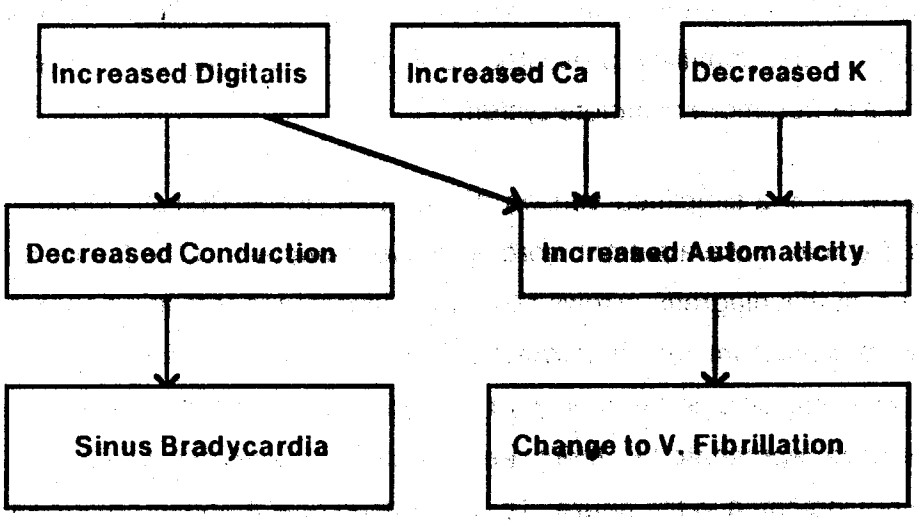
[((causal-link*b (decreased*o serum-k))*e (increased*o automaticity))]

Notice that the causal link above is actually an XLMS concept rather than just a pointer. The fact that it is a *conceptualized link* means that we can place attachments on the link, give it slots, and so forth, just as we can for any other concept and thereby further describe causal relationships using the same sorts of facilities used for other concepts.

For example, one thing we would like to be able to describe is the way that two causal links with a common destination interact. We know that increased serum calcium and decreased serum potassium both cause increased automaticity, and we can easily represent these in XLMS, but we need a way of expressing the relationships between

---

Fig. 14. The Domain Model



these relations. For example, if the two relationships are causally additive[16] we need a way of expressing that fact.

In general, in the XPLAIN system, we can express interrelationships between links by characterizing them as elements of that inter-relationship. For example, we can express the additivity of the causal links between serum potassium and serum calcium and increased automaticity in the following way:

```
[(additive-relation*i 1)
 [link-element: c#e ((causal-link*b (decreased*o serum-potassium))*e
                                    (increased*o automaticity))
                    ((causal-link*b (increased*o serum-calcium))*e
                                    (increased*o automaticity))]]
```

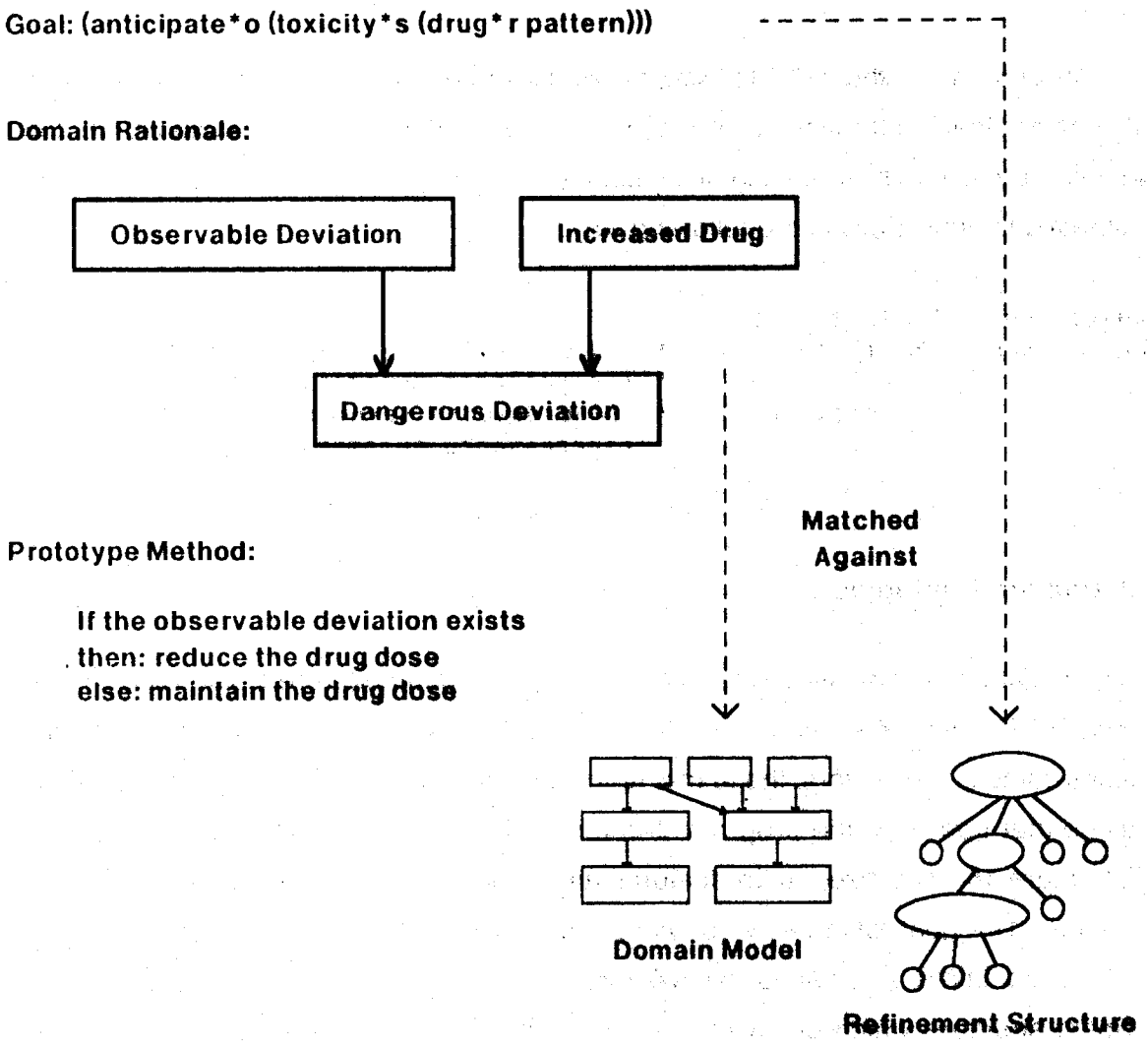
3.1.2 Domain Principles

The major features of domain principles will be briefly outlined here and detailed more extensively later when their use in relation to the writer is described. Currently, domain principles come in two flavors: those that refine a single node of the refinement structure and those that transform the program structure. The first type is depicted in graphical form in Figure 15. The three major features are outlined here, and described in greater detail below: 1) A goal, which may contain pattern variables, tells what this principle can do. The plan finder (described below) matches this goal against the steps in the refinement structure which are waiting to be refined. 2) A domain rationale, which is matched against the domain model by the pattern matcher. This is in a sense an additional specification for the program, telling the writer what cases must be considered for the given domain model in refining the goal. 3) The prototype method, which is a set of steps to be instantiated by the system. These are the refinement of the goal, and are placed under it in the refinement structure. The second type of domain principle is similar to the first with a few exceptions. This type of domain principle has no domain

---

16. If two causal relations have a common destination (such as increased automaticity) and the causal relations taken together cause more of the destination to occur (or cause it to be more likely to occur in the case of a state change) than either of the relations by itself, then we say that these relations are at least *causally additive*.

Fig. 15. A Domain Principle



rationale. The goal of this principle may make reference to steps waiting to be refined. The prototype method may contain active elements[17] which transform the execution order of the steps to be refined and which may add new steps to be refined. In addition

17. In the first type of domain principle, the prototype method was static. That is, it was instantiated by just creating a structure or structures with the pattern variables replaced by their value or values. In the second type, the prototype method may be a program to be interpreted which returns the instantiated structure.

to the features above, both forms have constraints associated with them (described below) which limit their applicability.

3.2 The Pattern Matcher

The pattern matcher in the XPLAIN system was written by Ramesh Patil and the author. It can find all the structures in the knowledge base that match a pattern structure. It can also be used to compare a pattern structure with a structure in the knowledge base to determine whether they match. The specification of various types of patterns is described below.

3.2.1 Specifying a Pattern

A pattern in XLMS is much like any other structure in the knowledge base. For example, a pattern that would match a causal-link between two different disease states might appear as:[18]

```
[(pattern*i 1)
 [structure: #e [((causal-link*b disease-state:1)*e disease-state:2)]]
 [predicate: #e
           [(mnot*c (mequal*c [disease-state:1,disease-state:2]),)]]]
```

Pattern 1

There are two variables in Pattern 1: `[disease-state:1]` and `[disease-state:2]`. A concept is a variable if and only if its tie is `*r` and its cue is a pattern. The ilk of the concept indicates where matching should start in the knowledge base. With some exceptions, to be a successful match, a variable must have the same attachments and be tied to the same concepts (recursively). The header of the pattern is the concept attached by `#e` to `[structure:]`.

---

18. The ties `*b` and `*e` will be explained in Chapter 5.

Different instances of variables matching individuals are indicated by placing a number after the : as in [disease-state:1] which expands to [((disease-state*r <pattern>)*i 1)] Any concept which is under [disease-state] will match this variable. For any pattern, only those variables which are tied or attached to the structure will be matched.

3.2.1.1 Sequences

The pattern matcher will also match sequences of variables or constants. For example, the pattern:

```
[(pattern*i 2)
 [structure: #e [(predicate:*c [(msum*c
                                [summum-genus:1,summum-genus:2],),])]]]
```

will match all calls to predicates which have the sum of two items as input. The variables are [predicate:] [summum-genus:1] and [summum-genus:2]. Due to a slight peculiarity in the way sequence matching is implemented, a sequence cannot be the top level item in a pattern.

3.2.1.2 Kleene Operators: Kstar, Kplus and Kor

To match arbitrary repeats of sequence elements or sub-sequences, the KSTAR and KPLUS notations may be used, which stand for Kleene star and Kleene plus, respectively. For example,

```
[(kstar*i foo),]
```

will match all sequences which consist solely of zero or more FOOs. [(kstar*i [a,b,c])] will match sequences such as [a,b,c,a,b,c,a,b,c] Note that some subtle things can occur if the structures of the pattern sequences are changed slightly. For example, [(kstar*i [a,b,c],)] matches [[a,b,c],[a,b,c],[a,b,c]] not [a,b,c,a,b,c,a,b,c]. KPLUS is the same as KSTAR but requires at least 1 match for the match to be successful.

KOR is the way of specifying a disjunction in the pattern. The general form is [(kor*i [alternate1,alternate2])]. In matching, the match will succeed if either alternate1 or alternate2 is found.

3.2.1.3 Predicates

After a structure is found that matches the pattern, the pattern matcher invokes the MINT interpreter to evaluate the predicates associated with the pattern. If all of them return TRUE, the structure is added to the list of successful bindings. Otherwise, it is rejected.

3.3 The Program Writer: How it Works

In this section we will trace the operation of the Program Writer as it writes the portion of the digitalis advisor that anticipates digitalis toxicity. To understand the example program synthesis to be presented, the reader may wish to review the section on Digitalis Sensitivities in Chapter 1.

3.3.1 Synthesizing the Performance Program

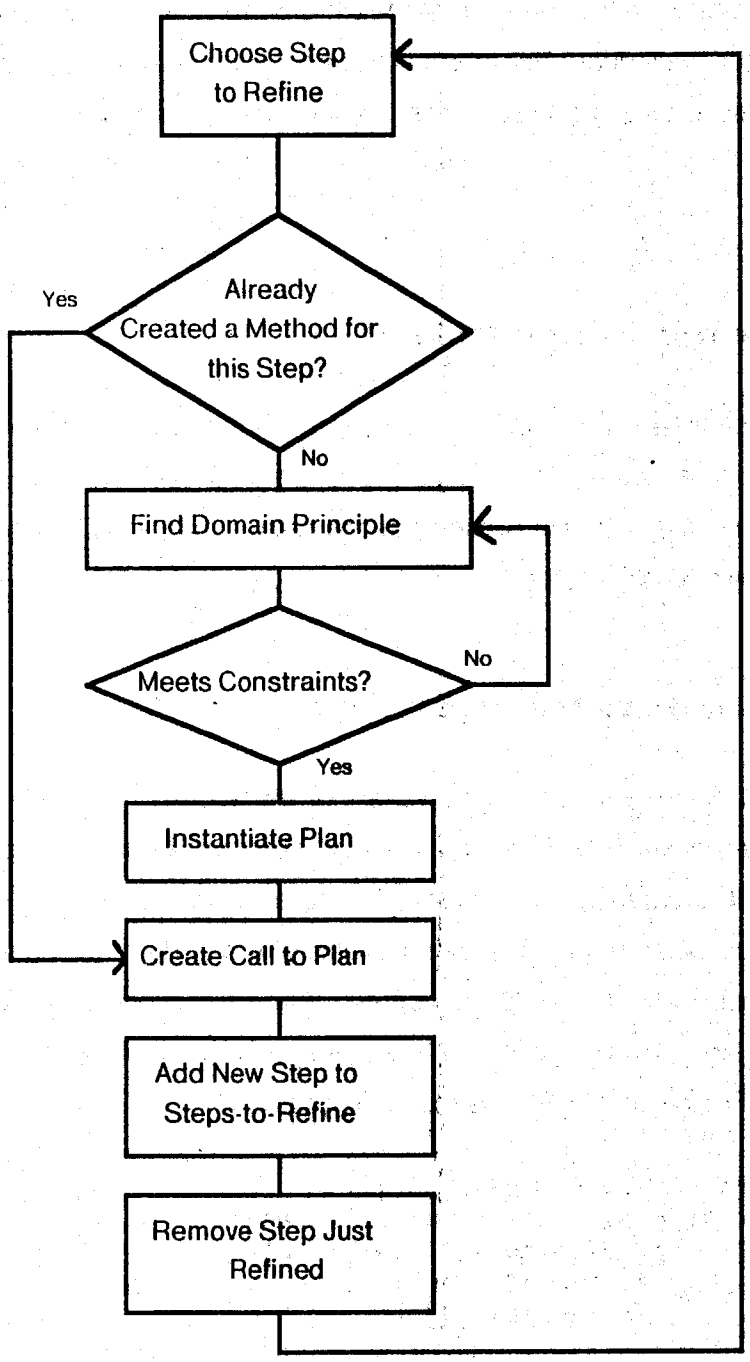
Figure 16 gives an overview flowchart of the Writer.[19] The Writer synthesizes the performance program by refinement from an abstract, high-level specification. As the Writer runs, it creates a refinement structure which is a goal tree tracing its refinement of the program. Consider the synthesis of the code required for adjusting the dose of digitalis because of possible toxicities. When the system starts this synthesis, there is just one node in the structure, a high level description of what the performance program is to do. For this example, the top level goal is:[20]

```
[(anticipate*o (toxicity*f digitalis))
 [input:  #e [(dose*r digitalis) #m variable]]
 [output: #e [((dose*r digitalis)*f adjusted)
                    #t (dose*r digitalis)
                    #m variable]]]
```

---

19. This figure leaves out a few details that will be discussed later in the chapter.
20. Remember that for this example, we are just synthesizing a portion of the digitalis advisor. If we were synthesizing the entire advisor, the top goal would be [(administer*o digitalis)], and its refinement structure would include the goal of anticipating toxicity.

Fig. 16. The Program Writer



This statement says that the system needs to write a procedure that will anticipate digitalis toxicity. The procedure will be supplied with one input, the dose of digitalis [(dose*r digitalis)], and will produce one output, an adjusted dose of digitalis [((dose*r digitalis)*f adjusted)]. In general, a call may have more than one (or no) inputs or outputs attached by #e to the input and output roles of the call. The ordering of the inputs and outputs does not matter to the system, because the binding of the input of a call to the input of a domain principle is done by pattern matching which does not depend on ordering.[21] #m is a *meta-characterization*. Here, both the dose of digitalis and the adjusted dose of digitalis are meta-characterized as variables (as opposed to constants, for instance).

Variable names in a procedure may be chosen according to different criteria. Sometimes, a programmer chooses a variable name that will indicate what the variable *is*. For example, the dose of digitalis is what the [(dose*r digitalis)] refers to. Sometimes, however, a variable name is chosen to indicate the role that the variable plays within the procedure, or its characteristics in relation to the procedure. In the goal above, the adjusted dose of digitalis is still a dose of digitalis, the fact it has been adjusted really relates more to its role within the calling procedure than to anything intrinsic within the variable itself. The choice of names is significant because the procedure will be translated into English, and these names will be used in that translation.

We would like to be able to give the person writing domain principles the same sort of flexibility in choosing variable names that a person writing procedures enjoys. That is, he should be able to describe a variable based on what it is or based on the role it plays in the procedure, as we described in the preceding paragraph. Allowing this flexibility can have a positive effect on the quality of the explanations. But there is a problem. Since a pattern matcher is used to match up arguments, if the arguments are named by the role they play within a procedure, then variables which are supposed to match may not if they play different roles in the called and calling procedures.

---

21. Note that this contrasts with the MINT interpreter where the ordering of input and output arguments *is* important since the binding of arguments is based on position. The program writer converts the more general, but slower, pattern matched form to the faster positional form of argument passing as it writes the performance program.

The #t attachment is called the *type* attachment, and was introduced to address this problem. The fact that [(dose*r digitalis)] is attached by the type attachment to [((dose*r digitalis)*f adjusted)] indicates that the adjusted dose of digitalis is still really just a dose of digitalis. Although the fact that it is adjusted may be useful in reasoning about the higher level procedure, the called procedure need only worry about the fact that it deals with a dose of digitalis. This information is used by the pattern matcher as it attempts to find a suitable domain principle for accomplishing the goal. That is, the pattern matcher performs matches based on the *intrinsic* nature of the variable, rather than the role it plays within a procedure. If there were no type attachment on [((dose*r digitalis)*f adjusted)], then for a successful match, the output of the domain principle would be have to be some sort of adjusted dose of digitalis, but with the type attachment it only has to be some sort of dose of digitalis.

The system maintains a list of the leaves of the refinement structure called *steps-to-refine*. The entries in this list are either calls or program fragments which must be transformed (called *transformations*). When the system starts, just the top level call is on the list, and the system halts when the list becomes empty.

When the system starts, an entry is selected from *steps-to-refine* to be refined. Selection is done as follows: If there are any transformations on the list, the first one is selected. If there are no transformations, the entry following the one selected on the previous iteration is selected.[22] Transformations are selected before other entries because their refinement may constrain the way in which the other entries may be refined. The system now has to find a domain principle to help it refine the entry.

3.3.2 Finding a Domain Principle

As was mentioned above, the header of a domain principle is a concept which contains pattern variables and which indicates what the domain principle can do. In this example, the Writer needs to find a domain principle whose header matches the goal of

---

22. This is a very simple scheme for selection, of course. If we had to deal with backup (which we don't, but see discussion at the end of this chapter) a more sophisticated algorithm would clearly be desirable to keep the system efficient.

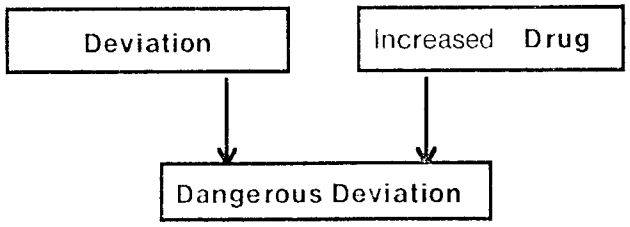
anticipating digitalis toxicity.

The system finds those domain principles that match in the following way: First, the system examines the concepts under the node [(prototype-method [*r])].[23] For each of these concepts, if the call is under the cue of the concept in the kind hierarchy,[24] and no other more specific concept[25] can be found, the pattern matcher is used to determine whether the domain principle matches the call. If the match succeeds, the pattern matcher returns a list of bindings showing how the pattern variables in the domain principle were bound to the literals within the call, and how the inputs and outputs of the call were bound to the inputs and outputs of the domain principle. The domain principle that matches in this case appears in graphical form in Figure 17 and the XLMS

Fig. 17. Domain Prinicple for Anticipating Drug Toxicity

Goal: Anticipate **Drug** Toxicity

Domain Rationale:



Prototype Method:

If the **Deviation** exists
then: reduce the **drug** dose
else: maintain the **drug** dose

23. Since all domain principles must have a prototype-method role, the goal (or header) of all of them will appear as the cue of some concept under this node.

24. Actually, the test referred to here cannot be done with the simple underp primitive of XLMS. Since the domain principle header will contain pattern variables, the check must be done by breaking apart each concept into its ilk, tie and cue, and performing the test recursively on the parts. When a pattern variable is encountered in the header, the test is modified to see whether the part of the call being tested is under the *ilk* of the pattern variable.

25. That is, one which is deeper in the kind hierarchy.

representation of the same principle appear in Figure 18. The individual parts of the domain principle will be discussed in the following paragraphs.

The matcher matches just the header and the inputs and outputs against the call:[26]

```
[(anticipate*o (toxicity*f (drug*r !pv)))

 [input↑ #e [(dose*r (drug*r !pv)) #m variable]]
 [output↑ #e [((dose*r (drug*r !pv))*f
                    (adjusted*for (get-all-matches*c aspect1,)))
              #m variable
              #t (dose*r (drug*r !pv))]]]
```

After the match succeeds, the pattern variable *(drug*r !pv)* will be bound to *digitalis*, and the input and output variables will be bound to the corresponding variables in the call. Note that the output variable has a type attachment. As was explained earlier, when a variable in a call or domain principle has a type attachment, the attachment is the object that is matched, rather than the variable itself.. If the match of the type attachment succeeds, then it is bound to the thing it matched, and the actual variables are also bound.

Even if the match is successful, there may be additional constraints that must be satisfied before the principle can be accepted as applicable to this situation. This principle illustrates one type, the *domain rationale*, others will be discussed later.

3.3.3 The Domain Rationale

The domain rationale is a pattern which is matched against the domain model. It serves two purposes. First, it is a constraint on the acceptability of the domain principle, because if no matches are found, the domain principle is rejected. Second, it can also be thought of as a further specification of the performance program. We will discuss the latter point in more detail later in this section.

---

26. Concepts with cues of `!pv` are pattern variables. Thus, `[(drug*r !pv)]` is a pattern variable which will match anything which is a kind of drug, just like `[(drug*r pattern)]`

Fig. 18. Domain Principle for Anticipating Drug Toxicity

(GOAL)

```
[(anticipate*o (toxicity*f (drug*r !pv)))
 [input↑ #e [(dose*r (drug*r !pv)) #m variable]]
 [output↑ #e [((dose*r (drug*r !pv))*f
                    (adjusted*for (get-all-matches*c aspect1,)))
              #m variable
              #t (dose*r (drug*r !pv))]]
```

(DOMAIN-RATIONALE)

```
 [domain-rationale↑ #q
   [(pattern*i 100)
    [structure↑ #e [chain1 = ((causal-chain*b deviation1)*e deviation2)]]
    [deviation1 = (deviation↑1*o [aspect1 = aspect↑↑1]) #c finding]
    [deviation2 = deviation↑2 #f dangerous]
    [chain0 = ((causal-chain*b (increased*o (drug*r !pv)))*e deviation2)]
    [predicate↑ #e (mand*c
                    [(mnot*c (pat-equal*c
                              [deviation1,
                               (mquote*c (increased*o (drug*r !pv)),)],),),
                     (new-match*c deviation1,),
                     ((determine-whether*o
                       ((additive*f causally)*f least))*c
                      (form-set*c [chain1,chain0],),)
                     ],)
                ]]]
```

(PROTOTYPE-METHOD)

```
 [prototype-method: #q [(if-then*c [(determine-whether*o
                                     (value*c deviation1,))],
                                   [((reduce*o (dose*r (drug*r !pv)))*due-to
                                     (value*c deviation1,))
                                    [input↑ #e [(dose*r (drug*r !pv))
                                                #m variable]]
                                    [output↑ #e [((dose*r (drug*r !pv))*f
                                                  (adjusted*for
                                                    (value*c deviation1,)))
                                                 #m variable
                                                 #t (dose*r (drug*r !pv))]
                                               ]],
                                   [(maintain*o (dose*r (drug*r !pv)))
                                    [input↑ #e (dose*r (drug*r !pv))]
                                    [output↑ #e [((dose*r (drug*r !pv))*f
                                                  (adjusted*for
                                                    (value*c deviation1,)))
                                                 #t (dose*r (drug*r !pv))]]
                                    #m computer-viewpoint]
                                   )]]]
```

Pattern variables are denoted either in the manner described above, or by having cues of !pv.

The domain rationale of this domain principle appears below:

```
[domain-rationale↑ #q
  [(pattern*i 100)
   [structure↑ #e [chain1 = ((causal-chain*b deviation1)*e deviation2)]]
   [deviation1 = (deviation↑1*o [aspect1 = aspect↑↑1]) #c finding]
   [deviation2 = deviation↑2 #f dangerous]
   [chain0 = ((causal-chain*b (increased*o (drug*r !pv)))*e deviation2)]
   [predicate↑ #e (mand*c
                  [(mnot*c (pat-equal*c
                           [deviation1,
                            (mquote*c (increased*o (drug*r !pv)),)],),),
                   (new-match*c deviation1,),
                   ((determine-whether*o
                      ((additive*f causally)*f least))*c
                    (form-set*c [chain1,chain0],),)
                   ],)
              ]]]
```

As was described above, patterns have several parts. The structure of this pattern is a causal chain, `[chain1]`, leading from a deviation, `[deviation1]`, which can be characterized as a finding to another deviation, `[deviation2]`, which is dangerous. In addition, `[deviation2]` is also involved in another chain, `[chain0]`. `[chain0]` requires that anything that matches deviation2 has to be caused by an increased level of the drug (in this case digitalis, of course). Put more simply, this pattern is looking for a finding which causes something dangerous where that something dangerous is also caused by an increased level of digitalis.

To place further restrictions on the match that would be difficult to express within the structure, this pattern has three predicates (embedded within an and) that must evaluate to true for a particular match to succeed.[27] The first predicate specifies that `[deviation1]` cannot be the increased drug level. We cannot allow that match because we are looking for *other* factors which increase the danger of giving the drug. The second predicate, `new-match`, fails if the current value of its argument `[deviation1]` is the same as the value it had on a previous match. This requirement ensures that for each successful match, the value of `[deviation1]` will be different from its value in all other successful matches. This predicate is necessary because a

---

27. `[mand]`, `[mnot]` and `[mquote]` are MINT primitives which perform similar functions to their LISP counterparts.

particular deviation may cause more than one dangerous deviation. For the purposes of this principle it is sufficient that it cause one dangerous deviation. The third predicate requires that the causal effects of the increased drug level and [deviation1] must be at least causally additive. This predicate checks to see whether the causal links in [chain1] and [chain0] are at least additive[28] at the node where the two chains merge together.

The notion of the domain rationale as a partial program specification is something that seems to be unique to the XPLAIN system. Generally, in other automatic programming systems, the specifications for the program are very closely tied to the eventual form of the program, and must be specified before the implementation of the program begins. Here, the specifications for the program are not supplied explicitly in advance by the user of the automatic programming system, but the specifications are derived by matching the domain rationale against the domain model as the refinement of the program progresses.

Taking this approach permits considerably more flexibility and generality. The creator of the domain model only has to worry about trying to encode the knowledge of the domain. He does not have to worry about how that knowledge will be used in the creation of a program (as he might if he were trying to create program specifications). New information can be added to this model and incorporated into a new version of the performance program by re-running the automatic programmer. A particular piece of knowledge might be used for several purposes (or not at all). For example, information about the effects of increased digitalis levels is used by the system both in anticipating toxicity and in assessing toxic and therapeutic reactions.

In some ways this is reminiscent of rule-based systems—one adds more knowledge by adding more rules, and the rule interpreter puts them together to solve some particular problem. But there are some key differences. In most rule-based systems, the rule interpreter is fixed. In the XPLAIN system, the interpretation of the knowledge in the domain model depends on the domain principles which are not fixed but can easily be changed. This makes it easy to experiment with different

---

28. Effects which are synergistic would be considered to be more than additive.

interpretations of the same knowledge (see Chapter 4). Additionally, if we include special-purpose domain principles together with general-purpose ones the system can easily bring to bear what amount to special-purpose interpreters to handle special situations. These features would be more difficult to implement within the framework of most rule-based systems.[29]

The domain rationale is one of the mechanisms used in the XPLAIN system for tying the independent domain model into the specification of the performance program. Yet, the domain rationales themselves can be quite general, and are really independent of the particular domain model. The domain principle used by the system for anticipating digitalis toxicity could be used with different domain models to accomplish the same task for a number of other drugs.

Returning to the problem of anticipating digitalis toxicity, when this domain rationale is matched against the digitalis domain model, there are three matches. Not surprisingly, these turn out to be the three sensitivities that were described above, namely, increased serum calcium, decreased serum potassium, and cardiomyopathy. When the pattern matcher returns the matches, it binds up not only the pattern variables, but it returns the entire structure that matched as well. Thus, it is easy to determine that decreased serum potassium matched because it causes increased automaticity which may in turn cause a change of the ventricular rhythm to ventricular fibrillation. Once all the matches have been obtained, the system is ready to instantiate the prototype method.

3.3.4 Instantiating the Prototype Method

The process of instantiating the prototype method is relatively simple: the system constructs a copy of the prototype method in the plan with the variables replaced by their values. This new structure is called the *instantiated method*. There are some

---

29. Davis [Davis76] introduced the notion of meta-rules, which bear some similarity to domain principles. However, meta-rules were used only for ordering and pruning the application of lower level rules within the context of a standard rule interpreter. Domain principles can control interpretation to a much greater degree.

special functions which are recognized by the instantiator. These functions are run by the instantiator using the MINT interpreter, and the concepts returned by the functions are placed in the instantiated method.

`[value]` is one such function. It takes one argument which is a variable and returns the value of that variable. This function is not needed by the instantiator (since variables appearing in the prototype method are automatically evaluated) but it is needed to prevent parts of the prototype method from being confused with the structure of the domain rationale pattern.[30] Another function is the `[get-all-matches]` function. This function has a pattern variable in the domain rationale as an input argument, and constructs an ordered set consisting of all the concepts that matched the variable.[31] Other functions will be discussed as they appear in other domain principles.

So far, things seem pretty simple. But what if the domain rationale matches several structures? How should the prototype method be instantiated then? If several matches have been found, then there are several situations where the prototype method is the appropriate thing to do, but these various actions must be integrated into a whole. (See Figure 19.)

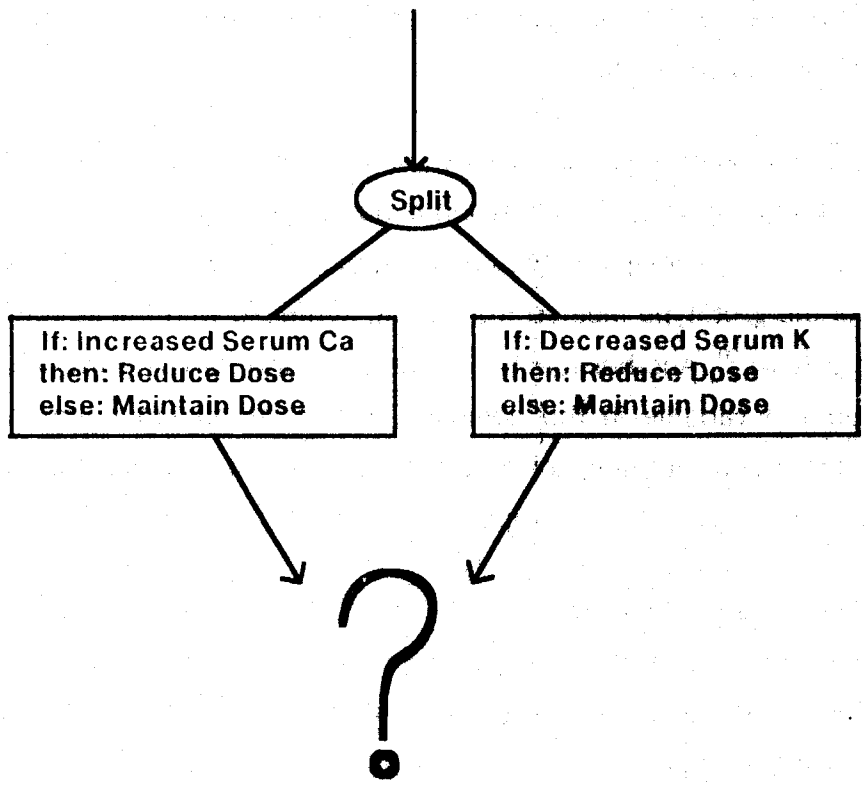
In the case under consideration, increased serum calcium, decreased potassium and cardiomyopathy are all situations where the dose of digitalis should be reduced from the standard dose. Yet the prototype method does not tell what should be done if a patient has *both* increased calcium and cardiomyopathy. The correct thing to do depends on the domain knowledge again. If the inter-relationship between increased calcium and cardiomyopathy is such that both of them together make the patient even more sensitive than either one of them by itself, then the appropriate thing to do is to make a bigger reduction when both of them are observed. On the other hand, it could be

---

30. This could occur if a pattern variable in a domain rationale pattern were placed in the prototype method without being an argument to the `[value]` function. The pattern matcher when checking the ilks and cues of the pattern variable against some candidate concept would find that the pattern variable was used as the ilk or cue of some concept in the pattern variable, and would require that the candidate have the same ilk or cue. That usually results in the rejection of otherwise correct matches. There is a similar notation in Conniver.

31. Sets in the XPLAIN system have the form `[(set*i a,b,c)]`. The elements of the set are ordered by the XLMS beforep relation. This simplifies comparisons of sets.

Fig. 19. A Split to be Resolved



Note: To keep the figure simple, only 2 sensitivities are shown

---

that the sensitivities aren't additive or that a reduction for one takes care of any of the others that may exist. In that case, a different sort of program structure is required. The system needs some representation for the current state so that it can reason about what sort of program structure might be appropriate.

For each of the matches of the domain rationale, the system instantiates the prototype method. It then places each of these instantiations into a larger structure called a *split-join*. The split-join is placed in the refinement structure and domain principles are used to transform it into executable program structure. A split-join is a concept whose ilk is **split-join** and whose cue is a sequence with as many elements as there were matches. The elements of this sequence are themselves sequences of two elements, where the first element is the instantiation of the prototype method and the second element is the corresponding structure that matched the domain-rationale. The split-join for the example appears in Figure 20. If only one match is found for the domain rationale, or if there is no domain rationale at all in the domain principle, the system just

instantiates the prototype method and does not make up a split join.

Fig. 20. An Example of a Split-join

```
[(SPLIT-JOIN*O
    [(MIF-THEN*C
        (DETERMINE-WHETHER*O CARDIOMYOPATHY),
        [(((REDUCE*O (DOSE*R DIGITALIS))*DUE-TO CARDIOMYOPATHY)*I 1)
        [INPUT↑ #E (DOSE*R DIGITALIS)]
        [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F (ADJUSTED*FOR (CONDITION*R
                        (MUSCLE*F HEART))))]],
        [((MAINTAIN*O (DOSE*R DIGITALIS))*I 1)
        [INPUT↑ #E (DOSE*R DIGITALIS)]
        [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F (ADJUSTED*FOR (CONDITION*R
                        (MUSCLE*F HEART))))]]),
     (((CAUSAL-CHAIN*B CARDIOMYOPATHY)*E (CHANGE*O (TO*O V-FIB)))*I
        CARDIOMYOPATHY, (INCREASED*O AUTOMATICITY),
        (CHANGE*O (TO*O V-FIB)))],
    [(MIF-THEN*C
        (DETERMINE-WHETHER*O (DECREASED*O SERUM-K)),
        [(((REDUCE*O (DOSE*R DIGITALIS))*DUE-TO (DECREASED*O SERUM-K))*I
           1)
         [INPUT↑ #E (DOSE*R DIGITALIS)]
         [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F (ADJUSTED*FOR SERUM-K))]],
        [((MAINTAIN*O (DOSE*R DIGITALIS))*I 2)
         [INPUT↑ #E (DOSE*R DIGITALIS)]
         [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F (ADJUSTED*FOR SERUM-K))]]),
     (((CAUSAL-CHAIN*B (DECREASED*O SERUM-K))*E
         (CHANGE*O (TO*O V-FIB)))*I
        (DECREASED*O SERUM-K), (INCREASED*O AUTOMATICITY),
        (CHANGE*O (TO*O V-FIB)))],
    [(MIF-THEN*C
        (DETERMINE-WHETHER*O (INCREASED*O SERUM-CA)),
        [(((REDUCE*O (DOSE*R DIGITALIS))*DUE-TO (INCREASED*O SERUM-CA))*I
           1)
         [INPUT↑ #E (DOSE*R DIGITALIS)]
         [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F (ADJUSTED*FOR SERUM-CA))]],
        [((MAINTAIN*O (DOSE*R DIGITALIS))*I 3)
         [INPUT↑ #E (DOSE*R DIGITALIS)]
         [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F (ADJUSTED*FOR SERUM-CA))]]),
     (((CAUSAL-CHAIN*B (INCREASED*O SERUM-CA))*E
         (CHANGE*O (TO*O V-FIB)))*I
        (INCREASED*O SERUM-CA), (INCREASED*O AUTOMATICITY),
        (CHANGE*O (TO*O V-FIB)))])]
```

3.3.4.1 Cleaning Up Some Details

Once the prototype method has been instantiated, there are still some details to be done. A procedure head must be defined, the steps of the instantiated method must be linked to it, a corresponding call must be created and transformed into something that can be executed by the interpreter, and those steps of the instantiated method that require further refinement must be identified and placed in the refinement structure.

To define the procedure head, the head of the domain principle is instantiated and a new instance is made of that concept. In the current case, it is:

```
[((anticipate*o (toxicity*f digitalis))*i 1)]
```

Next, the inputs and outputs associated with the domain principle are instantiated and collected into a input-output sequence which can be interpreted by the MINT interpreter. Using this input-output sequence together with the instantiated head of the domain principle, a procedure definition is created:

```
[(((anticipate*o (toxicity*f digitalis))*i 1)*d
      [(dose*r digitalis),],
      [((dose*r digitalis)*f
          (adjusted*for
             (set*i
                (condition*r (muscle*f heart)), serum-k, serum-ca))),])]
```

A [method] slot is created for the procedure definition, and the already instantiated steps of the procedure are attached to it.

A call is then created to correspond to the procedure. This is done by placing the variables in the original call (the one that was pattern-matched) into an input-output sequence so that the variables in this sequence match the ones in the input-output sequence of the procedure. A new call is then created whose ilk is the ilk of the procedure definition and whose cue is the input-output sequence:

```
[(((anticipate*o (toxicity*f digitalis))*i 1)*c
    [(dose*r digitalis),], [((dose*r digitalis)*f adjusted),])]
```

Naturally, (as this example illustrates) the variables in the call and procedure definition that correspond to one another are not necessarily identical, representing the fact that they may play different roles in their respective procedures. This newly created call is

then attached (by [# d]) to the old call that was in the steps to be refined.[32]

Finally, it is necessary to find those steps in the instantiated steps which need to be refined further. The system does this by examining the instantiated steps and their inputs. Each of these concepts can be either a variable, a call to a system primitive, a function or subroutine call which needs to be further refined, or a constant. Variables are easy to identify because they are meta-characterized as variables.[33] Variables do not have to be further refined so the system essentially ignores them at this point. System primitives are functions which do not have to be refined because they have already been written by someone else. The calls to system primitives have ties of [*c], and they are identified by those ties. Constants are also identified by being meta-characterized as such. Constants can either have a value or be unbound. Concepts with values are ignored. If an unbound constant is encountered, the writer asks the system builder[34] to supply a value for the constant. The writer then gives the constant the supplied value. Everything else is assumed to be a step which needs to be further refined, and is placed in the list of steps to be refined. Finally, the system removes the step that was just refined from the list of steps to be refined.

So where are we now? The system has just refined the goal of anticipating toxicity and has produced a split-join and a number of other goals which must be further refined. Although the list of steps to be refined is longer than it was when we started, we have made progress toward writing the performance program, and we're ready to begin the process of refining again by choosing a new goal and refining it.

---

32. When executing the program, the MINT interpreter looks to see if a step it is about to execute has a [# d] attachment. If so, it traverses the attachment and executes the attached step instead.
33. That is, the concept [variable] is attached to all variables using the [# m] attachment.
34. That is, the person running the writer program.

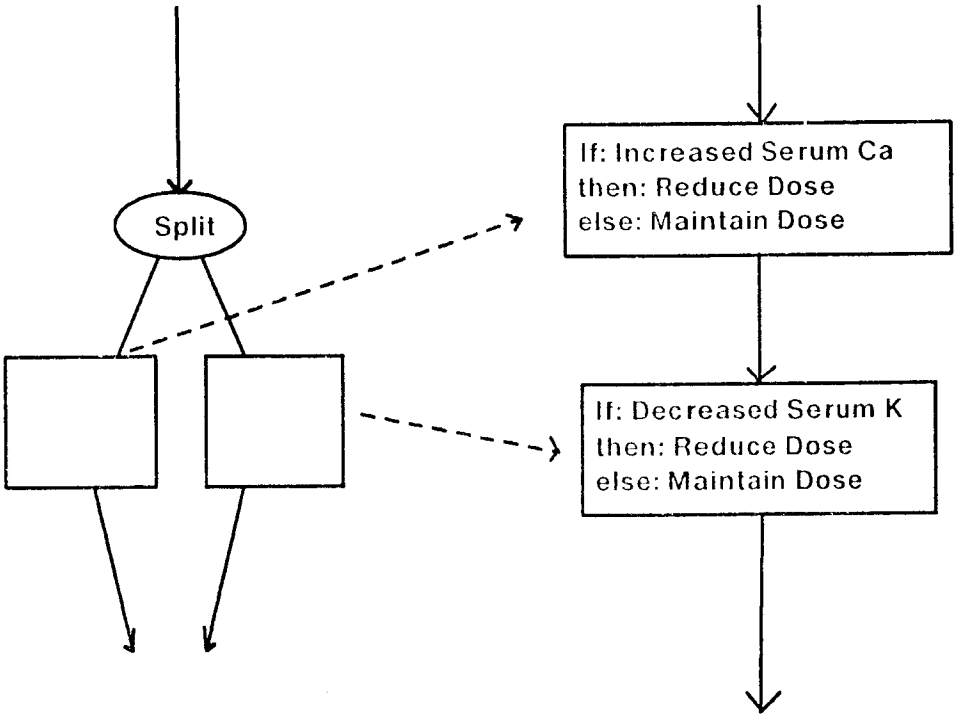
3.3.5 Refining a Split-join

The system chooses to refine the split-join next. The split-join is chosen because it is a transformation, that is, it will result in a transformation of the program structure. It is necessary to refine transformations first because they may impose constraints on the way other steps will be refined. Recall that the reason the split-join was created in the first place was that there were several matches for the domain rationale which resulted in several instantiations of the prototype method which have to be integrated into a unified whole. The split-join is an intermediate representation of the various instantiations which the system uses in reasoning about how to put things together. The goal of the domain principle that matches appears as:

```
[(split-join*o (kplus*t
                    [kps =
                      (if-then*c [(determine-whether*o
                                   ((deviation*r !pv)*o (aspect*R !pv)))],
                                 [adjust1 = (adjust*R !pv)
                                  [redi = input↑ #e [((level*r !pv)*I 1)
                                                     #m variable]]
                                  [redo = output↑
                                        #e [((level*r !pv)*I 2)
                                            #m variable]]],
                                 [adjust2 = ((adjust*r !pv)*i 2)
                                       [input↑ #e ((level*r !pv)*i 1)]
                                       [output↑ #e ((level*r !pv)*i 2)]]
                                 )],
                    (causal-chain*r !pv)),)]
```

Even though the pattern match succeeds, there are some constraints associated with this domain principle which must be satisfied before the domain principle can be used to transform the program structure. This domain principle will produce an executable piece of code by serializing the parts of the split join (see Figure 21). That is, the checks for increased serum calcium, decreased serum potassium, and cardiomyopathy will be performed in turn and the outputs for the first reduction will be connected to the second, and so forth. Thus, if multiple sensitivities exist, multiple reductions will be performed. The constraints check whether this serialization is a reasonable sort of resolution. There are two general types of constraints: *domain constraints* and *refinement constraints*.

Fig. 21. Resolving a Split By Serialization



Note: To keep the figure simple, only 2 sensitivities are shown

---

Domain constraints are tests to see whether a domain principle is applicable given the characteristics of the domain. In this principle, there are three domain constraints. These constraints are attached to the domain-constraint slot of the domain principle:

```
[domain-constraint↑ c#e [((independent*f causally)*c
                          (get-all*c ((deviation*R !pv)*o
                                      (aspect*R !pv)),),)]
                        [((determine-whether*o
                          ((additive*f causally)*f at-least))*c
                          (get-all*c (causal-chain*r !pv),),)]
                        [(mequal*c [(set*r !pv),
                          (get-all*c [(aspect*R !pv)],)
                          ],)]]
```

The first constraint checks that all the deviations (e.g. increased serum calcium, decreased serum potassium and cardiomyopathy) are causally independent in the sense that none of them causes the other. This is done by examining the domain model to see if there is a causal chain leading from any of the deviations to any other.

Get-all is a function which returns all the matches for a sequence pattern variable.[35]

The second constraint checks to see whether the effects of the causal chains are additive. That is, before we are willing to make multiple reductions for multiple sensitivities, we must be sure that the occurrence of two deviations is worse than just one by itself. Two chains are taken to be additive if they have a common terminus and if at the point where they join, the links leading into the joining node are noted as being in an additive relationship.

To understand the third constraint, we first have to describe the method-input and method-output roles in the split-join domain principle. When the system instantiated the prototype method for anticipating digitalis toxicity, none of the prototype method fragments had output variables which corresponded to the output variable for the method as a whole. The output for the method as a whole was:

```
[((dose*r digitalis)*f
          (adjusted*for
             (set*i
                (condition*r (muscle*f heart)), serum-k, serum-ca)))]
```

But, as can be seen in Figure 20, the outputs from the program fragments are doses individually adjusted for serum-k, serum-ca, and the condition of the heart muscle. As part of the process of integrating these fragments into a whole, the split-join also has to indicate what should be the output variable for the method. As of now, the output variable for the method is never set by any of the fragments. In the split-join domain principle, the method-input and method-output are matched against the input and output for the recently defined method for anticipating digitalis toxicity. The third constraint is placed here to ensure that the aspects that are being checked are the same ones that the dose claims to be adjusted for.

Refinement constraints are the other type of constraints used in this domain principle. Like domain constraints, refinement constraints determine whether or not a particular domain principle is applicable, but if it is found to be applicable, they also

---

35. See the section on the pattern matcher for a discussion of matching sequences and sequence pattern variables. (Section 3.2)

constrain the way in which further refinements may be made.

In this particular case, we are resolving the split-join by serializing the reductions. Whether or not this is a valid way to proceed depends in part on how the reductions themselves are refined. If the reductions are to be performed by subtracting some quantity from the dose, then there is some possibility that the dose will eventually become negative. That, of course, doesn't make sense. So the principle for resolving the split-join would have to insert a step at the end which would check for a negative dose and do whatever was proper. On the other hand, if the dose is reduced by multiplying the original dose by some constant less than one to produce a new dose, then the dose can never become negative, and no check is required. If we are going to resolve the split-join now, we need a way of constraining the resolution of the steps that perform the reduction. The refinement constraint for this principle is:

```
[refinement-constraint↑ c#e [(for-each*i 1)
                             [constrained-call↑ #e adjust1]
                             [predicate↑ #e
                                 (mor*c
                                  [(look-for-c-attachment*c
                                       [found-plan,
                                        (mquote*c identity-operator,)],),
                                      (look-for-c-attachment*c
                                       [found-plan,
                                         (mquote*c
                                            multiplicative-operator,)],)
                                      ],)]]
                            [(for-each*i 2)
                             [constrained-call↑ #e adjust2]
                             [predicate↑ #e
                                 (mor*c
                                  [(look-for-c-attachment*c
                                       [found-plan,
                                        (mquote*c identity-operator,)],),
                                      (look-for-c-attachment*c
                                       [found-plan,
                                         (mquote*c
                                            multiplicative-operator,)],)
                                      ],)]]]
```

To check this constraint, the system tries to find principles for refining the three instances of [adjust1] and [adjust2] (which are the calls to reduce and maintain the dose respectively) so that the principle is characterized as being either a multiplicative-operator or an identity-operator. If a principle is found for each of the calls, the refinement constraint is satisfied.

As the principles are found, the system remembers them by associating them with the appropriate entry in the list of steps-to-be-refined. Each entry in the list of steps-to-be-refined is itself a list with the following format:

```
(<step-to-refine> <domain-principle> <refinement-bindings>
                        <domain-matches> . <constraint-list>)
```

Only the <step-to-refine> entry is required. The <domain-principle> is the domain principle which should be used to refine the <step-to-refine>. The <refinement-bindings> are the bindings made by the pattern-matcher when the step and domain principle were refined. The <domain-matches> are the matches the system found for the domain-rationale of the domain principle (if it has one). Finally, the <constraint-list> is a list of refinement constraints imposed by higher level domain principles upon the selection of this <domain-principle> for refining the <step-to-refine>. An entry in the constraint list has this structure:

```
(<step-imposing-constraint>
        <principle-imposing-constraint> . <constraint>)
```

In designing the writer, it was decided to retain the step and principle imposing the constraint to allow the system to extricate itself from blind refinement paths, but the system really does not get into predicaments of this sort in refining the digitalis advisor. Accordingly, the implementation of a "backup" capability has not been completed. The addition of such a capability will be discussed in the conclusion of this chapter.

3.3.5.1 Instantiating the Prototype Method of a Split-Join

The prototype method of the principle for anticipating drug toxicity was a template. It contained no active elements. The prototype method for the split-join is different because it contains elements which must be interpreted, and which actively transform the split-join into an executable program. The instantiator recognizes these special forms. When evaluated by the MINT interpreter, they return a concept that the

instantiator puts in their place.[36] The various types of active elements are described below:

[(create-sequence*c [<from-number>, <to-number>, <index>, <concept-to-instantiate>],)]

This is an iterative function which creates a sequence of instantiations of the <concept-to-instantiate>. Starting from <from-number>, the value of the <index> is incremented on each iteration until the <to-number> is reached. The value of the <index> may be examined while the <concept-to-instantiate> is being instantiated.

[(get-all*c <sequence-pattern-variable>,)]

This function returns a set of all the matches of <sequence-pattern-variable> where <sequence-pattern-variable> is a pattern variable embedded within a kstar or kplus.[37]

[(get-all-matches*c <domain-rationale-variable>,)]

This function returns a set of all the values that a <domain-rationale-variable> had in all the successful matches of the domain rationale.

[(pappend-element-to-sequence*c [<sequence>,<element>],)]

This function appends <element> to the end of <sequence> creating a new sequence. One use for it is when the resolution of a split-join requires that some additional step be inserted.

[(index*c [<sequence-pattern-variable>,<n>],)]

This function selects the nth match of a <sequence-pattern-variable> embedded in a kstar or kplus.

---

36. This distinction is similar to that between simple pattern replacement and computed patterns in macro expansion.
37. See section on matcher (Section 3.2).

[(make-set*c [<i>,<j>,<sequence-pattern-variable>],)]

This function returns a set of the <i> through <j>-th matches of <sequence-pattern-variable>.

[(pif-then*c <predicate>, <concept-to-instantiate1>, <concept-to-instantiate2>)]

If <predicate> evaluates to [true] then the first concept is instantiated otherwise the second.

[(pcase*c [<pred1>, <con1>], [<pred2>, <con2>], ..., [<predN>, <conN>])]

The predicates of each pair are evaluated in turn until one returns [true]. The concept of that pair is then instantiated.

The prototype method of this particular split-join uses the create-sequence function to create a new method sequence which is a serialized version of the original split-join as shown in Figure 22.

Generally, an attempt has been made to minimize the use of prototype methods with active elements because they are much harder to explain. The reason for this seems to be that one is forced to talk about two kinds of activities: the activity of constructing the method, and the activity that the method is intended to perform.

Fig. 22. Method After Split-Join Resolved

```
[(MIF-THEN*C
    (DETERMINE-WHETHER*O CARDIOMYOPATHY),
    [(((REDUCE*O (DOSE*R DIGITALIS))*DUE-TO CARDIOMYOPATHY)*I 2)
     [INPUT↑ #E (DOSE*R DIGITALIS)]
     [OUTPUT #E ((DOSE*R DIGITALIS)*F
         (ADJUSTED*FOR (CONDITION*R (MUSCLE*F HEART))))]],
    [((MAINTAIN*O (DOSE*R DIGITALIS))*I 4)
     [INPUT↑ #E (DOSE*R DIGITALIS)]
     [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F
         (ADJUSTED*FOR (CONDITION*R (MUSCLE*F HEART))))]]),
 (MIF-THEN*C
    (DETERMINE-WHETHER*O (DECREASED*O SERUM-K)),
    [(((REDUCE*O (DOSE*R DIGITALIS))*DUE-TO (DECREASED*O SERUM-K))*I 2)
     [INPUT↑ #E ((DOSE*R DIGITALIS)*F
                  (ADJUSTED*FOR (CONDITION*R (MUSCLE*F HEART))))]
     [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F
       (ADJUSTED*FOR (SET*I (CONDITION*R (MUSCLE*F HEART)), SERUM-K)))]],
    [((MAINTAIN*O (DOSE*R DIGITALIS))*I 5)
     [INPUT↑ #E ((DOSE*R DIGITALIS)*F
         (ADJUSTED*FOR (CONDITION*R (MUSCLE*F HEART))))]
     [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F
      (ADJUSTED*FOR (SET*I (CONDITION*R (MUSCLE*F HEART)), SERUM-K)))]]),
 (MIF-THEN*C
    (DETERMINE-WHETHER*O (INCREASED*O SERUM-CA)),
    [(((REDUCE*O (DOSE*R DIGITALIS))*DUE-TO (INCREASED*O SERUM-CA))*I 2)
     [INPUT↑ #E ((DOSE*R DIGITALIS)*F
         (ADJUSTED*FOR (SET*I (CONDITION*R (MUSCLE*F HEART)), SERUM-K)))]
     [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F
                  (ADJUSTED*FOR
                   (SET*I (CONDITION*R (MUSCLE*F HEART)),
                                       SERUM-K, SERUM-CA)))]],
    [((MAINTAIN*O (DOSE*R DIGITALIS))*I 6)
     [INPUT↑ #E ((DOSE*R DIGITALIS)*F
        (ADJUSTED*FOR (SET*I (CONDITION*R (MUSCLE*F HEART)), SERUM-K)))]
     [OUTPUT↑ #E ((DOSE*R DIGITALIS)*F
                  (ADJUSTED*FOR
                   (SET*I (CONDITION*R (MUSCLE*F HEART)),
                                       SERUM-K, SERUM-CA)))]])]
```

3.3.6 Completing the Implementation

Several steps remain to be refined. The reduce steps for the various factors must be refined, as well as the steps which determine whether a factor exists or not, and those which just maintain the dose. The reduce steps are all refined using the same domain principle:

```
[[((reduce*o ((dose*R !pv)*I 1))*due-to ((deviation*R !pv)*I 1))
    #c multiplicative-operator]
 [input↑ #e [doser1 =
             (dose*f (before*o (adjusting*for ((deviation*R !pv)*I 1))))
                  #t ((dose*R !pv)*I 2) #m variable]]
 [output↑ #e [doser2 =
             (dose*f (after*o (adjusting*for ((deviation*R !pv)*I 1))))
                   #t ((dose*R !pv)*I 3) #m variable]]
 [prototype-method↑
  #q [(msetq*c [(mproduct*c [doser1,((constant*f reduction)*for
                                ((deviation*R !pv)*I 1))],),doser2])]]]
```

Although all are refined by the same domain principle, each of the refinements is different and results in a different method. This is because the reason for the reduction ((deviation*R !pv)*I 1) is part of the call. The difference between the methods is in the reduction constant that is used. Each method has its own, tailored for the particular sensitivity. As the system instantiates the prototype method for each reduction, it uses the reason for the reduction to create a unique reduction factor and asks the user for the value of the factor. That is, when the system instantiates the template:

```
((constant*f reduction)*for ((deviation*R !pv)*I 1))
```

the pattern variable ((deviation*R !pv)*I 1) is bound by the pattern matcher to the particular reason for this reduction (i.e. increased calcium, decreased potassium, etc.). Since the newly created constant (i.e. [((constant*f reduction)*f (increased*o serum-ca))]) is unbound, the person running the writer system is asked to enter a value for it. The constant can also be pre-specified in the knowledge base, in which case no question will be asked.

The refinement of the steps to determine whether the various conditions such as increased calcium, decreased potassium, and cardiomyopathy exist proceeds in an uneventful fashion. Unlike the reduction steps however, a different domain principle is used to refine each of the steps. The first two conditions are refined by similar principles: the first one checks whether some aspect (here calcium) has exceeded a threshold supplied by the user of the Writer program, while the second checks whether the aspect (here potassium) has fallen below a threshold. The selection of which principle to employ is based on whether "increased" or "decreased" appears in the call.

Fig. 23. Principles to Determine If Increased or Decreased Conditions Exist

```
[(determine-whether*o (decreased*o (aspect*r !pv)))
 [input↑ #e (aspect*R !pv)]
 [prototype-method↑ q#q
          [(mless-than*c [[(aspect*R !pv) #m variable],
                          [(threshold*r (decreased*o (aspect*r !pv)))
                           #m constant]],)]]]


[(determine-whether*o (increased*o (aspect*R !pv)))
 [input↑ #e (aspect*R !pv)]
 [prototype-method↑ q#q
          [(mgreater-than*c [[(aspect*R !pv) #m variable],
                             [(threshold*r (increased*o (aspect*R !pv)))
                              #m constant]],)]]]
```

The problem of determining whether cardiomyopathy exists or not is a little different, because there is no level that can be measured to determine whether the state exists. Thus, the system uses a method which yields a program that just asks the user whether or not cardiomyopathy is present. The system doesn't select this principle when trying to determine whether a state of increased calcium or decreased potassium exists because the system always picks the most specific principle that it can find. For those steps, the other principles are more specific because they appear lower in the kind hierarchy of XLMS.

One additional interesting thing occurs in refining these steps. The reader may have noted that the calls that appeared earlier in the chapter to determine whether these various conditions existed never had any inputs. Yet the domain principles require inputs. When matching, the pattern matcher does not require that all the inputs in the principle be supplied by the call (although it does require that all the inputs supplied by the call have some matching input argument in the principle). The system determines the missing input by placing a call to the ask-user function when it creates a MINT-level

---

Fig. 24. Principle to Determine If a Condition Exists

```
[(determine-whether*o (deviation*r !pv))
 [input↑ #e [(status*r (deviation*r !pv)) #m variable]]
 [prototype-method↑ q#q
                  [(mequal*c [(status*r (deviation*r !pv)),
                              (mquote*c present,)],)]]]
```

version of the pattern-matched call. That is, the pattern matched call:

```
(DETERMINE-WHETHER*O (INCREASED*O SERUM-CA))
```

becomes:

```
[(((DETERMINE-WHETHER*O (INCREASED*O SERUM-CA))*I 1)*C
    [[(ASK-USER*C [(MQUOTE*C [SERUM-CA,]),]),],])]
```

To refine the step that maintains the dose, we just need a domain principle that will set its output to its input. The simple principle shown in Figure 25 suffices. Notice that this principle and the call don't make any mention of the reason for maintaining the dose. It makes sense to talk about reducing the dose due to something or other, but it makes less sense to give a reason for maintaining the dose. That seems to be because maintaining the dose is the normal case, hence in a certain sense, the maintain step could be thought of as a no-op.[38] Since there is no need to distinguish the methods based on the reason for maintaining the dose, all three calls could use the same method. To allow this to take place, the method for finding a domain principle to employ is actually a little different than what we have described so far. Before the system searches for a domain principle that matches the step it's trying to refine, the system looks to see if a method has already been refined which would work. This is fairly simple to do. The system looks for the definition of a method which is inferior (in the kind hierarchy) to the call.[39] The system then checks to see whether the principle used to refine the found method could also be used to refine the current call. This is done through the usual mechanisms of pattern-matching and constraint checking as described above. If the

---

Fig. 25. Principle to Maintain the Dose

```
[[(maintain*o ((dose*r !pv)*i 1)) #c identity-operator]
 [input↑ #e [dosem2 = ((dose*r !pv)*i 2) #m variable]]
 [output↑ #e [dosem3 = ((dose*r !pv)*i 3) #m variable]]
 [prototype-method↑
  #q [(msetq*c dosem2,dosem3)]]]
```

---

38. And in fact, the explanation routines don't mention it for exactly that reason.
39. Or to the ilk of the call if the call is an individual.

principle is suitable, the system does not instantiate it, but rather it creates a MINT-level call to the found method and links that to the call that was to be refined. The call to be refined is then removed from the list of steps to refine. In addition to saving some time and space by preventing the system from performing needless refinements, this scheme also allows the system to refine recursive domain principles.[40]

3.4 Future Needs

This section has outlined the automatic programmer used by the XPLAIN system. This programmer has proved to be adequate to synthesize major portions of the Digitalis Advisor, and to demonstrate that the use of an automatic programmer can significantly enhance the explanatory capabilities of a system. However, while it has shown that the approach is feasible, the automatic programmer itself is not complete. Currently, the programm writer has no backup (or fix-up) capability. If it could not find a suitable domain principle to refine a step, the current version of the writer would be stuck. It seems that some sort of dependency-directed backtracking [Doyle79, Stallman76] which treated defined procedures in a manner analogous to the way assertions are treated in Doyle's Truth Maintenance System might be a reasonable way to attack this problem. Another lack in the current system is that it cannot fix plans which are nearly correct [Sussman75]. For example, there are times when it may be necessary to modify methods which have already been refined based on something which comes up in the refinement of a deeper node. At least one such instance does come up in the Digitalis Advisor, where it is discovered that it is necessary to determine whether the patient is experiencing nausea, anorexia, or blurred vision (possible toxic signs) before giving any digitalis so that a baseline comparison will be possible. The need for the baseline information (which must be gathered during the initial session) doesn't become clear until toxicity is being evaluated (which occurs during the feedback session).

---

40. However, the need for recursive principles never came up in this domain.

4. Assessing Toxicity

Whenever the dosage of digitalis is being adjusted, it is necessary to monitor the patient closely to see what effect (if any) the change is having on the patient. In the original Digitalis Advisor, there were two sets of routines for assessing the drug's effects. One set was concerned with the harmful toxic effects of digitalis, while the other dealt with the beneficial or therapeutic effects. Each set of routines produced an assessment of the degree to which the patient was showing toxic and therapeutic effects. Based on these assessments, the system recommended corrective actions if they were appropriate.

This chapter describes how the portion of the Digitalis Advisor that assesses toxicity was implemented using the XPLAIN system. Actually, two implementations are described, the first is based more on causality, while the second is more empirical. Interestingly, the domain model used in the previous chapter only required a few additions to be used for assessing toxicity.

The same overall plan was used in the two implementations described here. The user is asked whether the various toxic effects that digitalis may cause have been observed in the patient. The assessments of these individual findings are then combined into an overall assessment of toxicity. The assessment is a number representing the degree of toxicity and the individual assessments are combined together using numerical techniques. The two implementations differ in the way the combination functions work, and they differ in what gets combined with what.

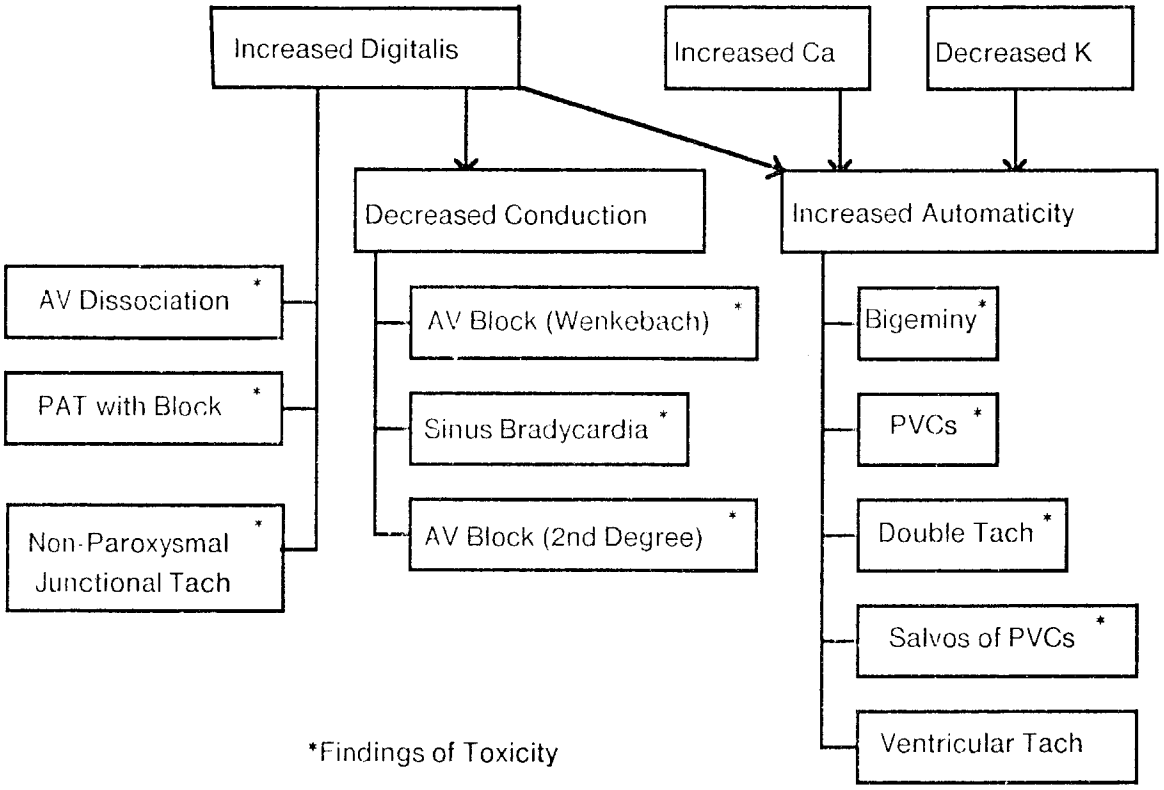
4.1 The Causal Implementation

The first implementation (which we ultimately rejected) was actually the more ambitious of the two. In this implementation, we wanted the causality network of the domain model to help dictate what should be combined with what and how they should be combined. The domain model used by both implementations appears in Figure 26.[41]

---

41. This figure (being graphical) leaves out a few important details which will be supplied later on.

Fig. 26. Domain Model For Toxicity



Note that this domain model is just an augmented version of the one used in the previous chapter.

In this implementation, the idea was to have the system recursively descend the causal paths which emerge from increased digitalis and produce code which would ask for the findings and then combine these findings at the next higher level in the domain model. For example, the assessments of those things caused by increased automaticity such as pvcs, bigeminy, and so forth would be combined together to form an assessment of increased automaticity. Increased automaticity and decreased conduction would then be combined to produce an overall assessment of digitalis toxicity.

The top level method, the one to assess findings of digitalis toxicity, has the following domain rationale:

```
[domain-rationale↑ #q
   [(pattern*i 200)
     [structure↑ #e [((clink*b (increased*o (drug*r !pv)))*e deviation6)]]
     [deviation6 = deviation↑1]
     [predicate↑ #e (mor*c
                     [(look-for-f-attachment*c
                       [deviation6,(mquote*c dangerous,)],),
                      (exists*c
                       [(pattern*i 201)
                        [structure↑ #e
                           ((causal-chain*b (value*c deviation6,))*e
                              [deviation↑↑ #f dangerous])]],)],)]
                    ]]
```

This pattern finds those deviations which are directly caused by increased digitalis[42] which are themselves dangerous or may cause something dangerous to occur. The `exists` predicate returns true if a match can be found for its pattern argument. The `look-for-f-attachment` looks to see whether its second argument is attached to its first by `#f`. The prototype method just sets up calls to assess the matches for `[deviation6]`:

```
[prototype-method↑ #q
  [(assess*o ((value*c deviation6,)*f (induced*o (by*o (drug*r !pv)))))
   [output↑ #e [(assessment*r ((value*c deviation6,)*f
                               (induced*o (by*o
                                 (drug*r !pv))))) #m variable]]]]
```

As was the case when anticipating toxicity, there may be several matches for the domain rationale, and if there are, the system creates a split-join. The system resolves the split-join by serializing the various assessments and then adding an additional step at the end to combine the assessments together.

Having moved one step down the causal links, there are three possibilities for each of the deviations. First, the deviation may be a finding, in which case the system can ask the user directly whether it exists or not. Second, the deviation may not be a finding, but it may cause another deviation to occur. In that case, the system should check the other deviation. Finally, it may be that the deviation to be assessed is neither a

---

42. Note the *causal-link* rather than causal-chain.

finding nor does it cause any other deviations. In that case, the system looks for warning signals—things which lead to the dangerous deviation. For example, for the purposes of a computer system, ventricular fibrillation cannot be considered a finding[43] nor does it cause anything of interest to the program. It is extremely dangerous, however. There are some warning signs which indicate that there is danger of entering ventricular fibrillation.[44] To assess the danger of ventricular fibrillation the system assesses these warning signs.[45]

To deal with the three situations outlined above, the system has three domain principles. Each principle has the same goal, but the predicates and the domain rationales associated with them make the system select the correct plan. The method used in the first case, when the deviation is a finding, is to just ask the user.[46] In the second case, the system just recurses down the causal links another step and assesses the deviations there. The actions taken in the third case have already been described.

The only problem that remains is to find a way of combining the various assessments together. We planned to use two different types of combination functions here. For combining the assessments of deviations which were directly caused by the same higher level deviation, we planned to use a function which would return the maximum (or worst) assessment. Our reasoning was that a doctor would feel that it was appropriate to reduce the dose if *any* of the dangerous things that can result from digitalis administration occurred. Taking the maximum reflects this reasoning better than using, say, a weighted-sum. However, when combining the assessments of

---

43. While ventricular fibrillation is observable using an EKG, no doctor is ever likely to enter it as a finding on a computer terminal. When the heart is fibrillating, it ceases to pump blood, and the patient with untreated ventricular fibrillation will die in less than five minutes. Thus, if a doctor observes ventricular fibrillation, his responsibility is to attempt to de-fibrillate the patient, NOT to enter findings on a computer terminal.

44. For example, premature ventricular contractions (PVCs) are usually not particularly dangerous to the patient. However, an increased number of PVCs in conjunction with digitalis administration is dangerous in the sense that it indicates that level of digitalis in the patient is reaching the point where a dangerous arrhythmia may occur.

45. The warning signs are required to be findings.

46. A more sophisticated method could be used here which would attempt to determine the correct answer using the other two methods if the user responded that he didn't know the answer. (This approach would be very similar to that used in MYCIN)

warnings signs, we felt that a weighted-sum was the appropriate thing to use, since it reflects to some degree the process of combining evidence.[47] Unfortunately, this part of the system was only partially implemented, since we decided to adopt the empirical approach before this one was completed.

4.2 The Empirical Implementation

When we outlined our design for the causal implementation to Dr. Pauker, the cardiologist we consult with, he was uncomfortable with it, feeling that it did not capture the reasoning he performed in trying to assess toxicity in a patient. He felt that when assessing digitalis toxicity he looked for signs in three general classes: highly specific signs of digitalis toxicity, moderately specific signs, and signs with low specificity (also called non-specific signs).[48] The original Digitalis Advisor followed this strategy and weighted the various findings according to their type to produce an assessment of digitalis toxicity. To implement this algorithm using the XPLAIN system, the only changes that had to be made in the domain model were to indicate (by characterization ties) the specificity of the findings. The domain principles required more extensive modification. This should not be at all surprising since we aren't changing the medical knowledge, we are changing our *interpretation* of that knowledge.

For this implementation, the top level method to assess digitalis toxicity just sets up the calls to assess the highly specific signs, moderately specific signs and non-specific signs and to then combine them together. The three different assessment steps are all refined by the same domain principle because that principle has the degree of specificity as a variable in its goal:

```
[(assess*o ((findings*r (toxicity*f (drug*r !pv))}*f (specific*R !pv)))]
```

The domain rationale looks for all findings which are caused by increased digitalis and

---

47. Of course, a number of other plans (such as certainty factors [Shortliffe76]) could have been used as well. This automatic programming approach lends itself well to experimentation with different methodologies because the use of domain principles allows the user to make changes at a higher conceptual level.

48. Viewed from a probabilistic perspective, we would say that the probability of digitalis toxicity would be high given the observation of a highly specific sign, moderate given the observation of a moderately specific sign and low given the observation of a non-specific sign.

have the degree of specificity specified in the call:

```
[domain-rationale↑ #q
     [dr3 = (pattern*i 300)
      [structure↑ #e ((causal-chain*b (increased*o (drug*R !pv)))*e
                                    [dev2 = deviation*↑])]
      [(dev2*characterization
        ((finding*r (toxicity*f (drug*R !pv)))*f (specific*R !pv)))]]]
```

The prototype method then sets up calls to assess these various findings. For most of the findings, the system just asks the user whether the finding is present or not. The code to do this is meta-characterized as having a computer-viewpoint because it is too low level to be of interest to doctors. Thus, this code is not normally explained to doctors.[49] The only exception is PVCs. PVCs are premature ventricular contractions. The system has special knowledge concerning how to assess them. A computer generated explanation of how that routine works appears in the next chapter.

---

49. Although the system can be instructed to explain it if an explanation is desired. Viewpoints are more thoroughly described in the next chapter.

5. Generating Explanations

This chapter describes the operation of the routines that generate English text from XLMS representations. By design, the knowledge structures left behind by the automatic programmer made it possible to achieve quite high quality English output with a simple generator. The generator should really be viewed more as an engineering effort that attempts to produce acceptable English rather than as a generation system that encodes deep linguistic principles. The main thrust of this thesis has been to investigate ways of representing the knowledge necessary to justify expert consulting systems. A generator is necessary to demonstrate the capabilities of the approach being espoused here, but the generator itself has not been the focus of the research.[50] The generator is really composed of two types of generators at two levels. The low level *phrase generator*, which has already been partially described in Chapter 2, constructs phrases directly from the XLMS representation. Higher level *answer generators* set up the appropriate environments and call the lower level generators in an appropriate order to produce answers to specific questions. The reader may wish to review the section in Chapter 2 on the phrase generator before continuing.

5.1 The Phrase Generator Revisited

The tie generators described in Chapter 2 are general purpose in the sense that they are not oriented toward a particular application domain. The generators described here are more domain specific in that they are oriented to the problem of explaining program structures (although they are *not* oriented to the domain of digitalis therapy at all). In addition to the generators, some of the additional facilities of the phrase generator are described at the end of this section.

---

50. See [Mann80, McDonald80]

5.1.1 Generator for *C

*C is the attachment used to indicate a call in the syntax of the XLMS interpreter. The ilk is the name of the called procedure and the cue is a sequence of input/output arguments. If the call is not a special form (such as [mif-then], [msetq], etc.) the system just generates the English form of the call preceded by "the system". If "the system" has already been established as a referent the pronoun "it" is used instead. For example, the call:

```
[(((assess*o (toxicity*f digitalis))*i 1)*c
    [], [(assessment*r (toxicity*f digitalis)),])]
```

is output as:

"The system assesses digitalis toxicity"

If the ilk is [if-then], English for the predicate is generated preceded by "if" and followed by the action to be taken if the predicate evaluates to true and the action taken (if any) if it is false. Thus, the concept:

```
[(MIF-THEN*C
    (DETERMINE-WHETHER*O (DECREASED*O SERUM-K)),
    (((REDUCE*O (DOSE*R DIGITALIS))*DUE-TO (DECREASED*O SERUM-K))*I 2),
    ((MAINTAIN*O (DOSE*R DIGITALIS))*I 5))]
```

Generates the phrase:

"If the system determines that decreased serum-k exists, it reduces the dose of digitalis due to decreased serum-k."[51]

Other special forms such as [mless-than], [mquote], and so forth are handled in a similar straightforward manner. [msetq] is also handled in the obvious way unless the input is a mathematical function. In that case a special routine is invoked which is

---

51. This example also illustrates the suppression of computer details. The [maintain] step, which appears in the XLMS code, does not appear in the generated English text. That is because the step has been meta-characterized with the flag [computer-viewpoint]. This feature will be described in greater detail later.

described in section 5.2.5.

5.1.2 Generator for *E

The *e tie is used with the *b tie to represent links and chains of various types.[52] The form of a link is:

```
[((causal-link*b cardiomyopathy)*e (increased*o automaticity))]
```

This says that cardiomyopathy causes increased automaticity (review chapter 2 for a more extensive discussion of the representation.) Chains are a representation for a series of links—a path from one object to another:

```
[((causal-chain*b cardiomyopathy)*e
          (change*o (to*o (fibrillation*f ventricular))))]
```

This concept indicates a causal chain from cardiomyopathy to a change (of the ventricular rhythm) to ventricular fibrillation. By design, concepts with ties of *e always have ilks that are concepts with ties of *b, and concepts with *b ties are not used in isolation, so a generator for *b is not really needed: the generator for *e can take care of the whole affair.

If the concept to be generated is a link, the system generates a phrase for the head of the link, followed by a phrase generated from the type of the link (the ilk of the ilk of the concept) followed by the tail of the link (which is the cue of the concept). For example, the link (not the chain) given above generates the phrase "Cardiomyopathy causes increased automaticity."

In the current version of the system, chains occur only in the patterns of domain rationales. That is not to say that we would envision limiting them to that use, but only that that is the only use we found for them in the application area. Since chains are found in patterns, there are two situations that arise in converting them to English.

---

52. As it turns out, in the current version of the system, all chains are causal chains because other types of chains were not needed to get a working system. While it would not be difficult to program the generator to work with other types of chains, the current generator only handles causal chains.

Either we wish to describe the pattern as a pattern, or we wish to describe the sequence of links that the pattern matched. Chains which are to be described as patterns look much like links, and they are described using a similar process to the one described for links. If we wish to describe the sequence of links that matched the pattern, the situation is a little more complicated.

When the system finds a match for a chain, it creates a new concept. The ilk of this concept is a copy of the pattern chain with the head and tail replaced by the concepts that form the head and tail of the found chain. The cue of this concept is a sequence of concepts which are on the path from the start of the chain to its end. For example, when the system is looking for a path from increased digitalis to a moderately specific finding of toxicity, it creates the following concept when it finds such a path between increased digitalis and increased pvcs:

```
[(((CAUSAL-CHAIN*B (INCREASED*O DIGITALIS))*E (INCREASED*O PVCS))*I
    (INCREASED*O DIGITALIS),
    (INCREASED*O AUTOMATICITY), (INCREASED*O PVCS))]
```

The pattern matcher binds this concept to the pattern chain as its value. The generator describes the path by constructing sentences detailing the chain link by link. If there are just two items in the path the generator produces:

A causes B.

If there are 3 items, it adds a relative clause:

A causes B, which causes C.

If there are more than three, it breaks things up into sentences:

A causes B, which causes C. C causes D.

The system locates the actual links that link the items together. Thus, if one of the links indicates possibility rather than definite causality, it is possible to say so:

A may cause B.

In addition, the system keeps around a list of the link relationships that have been

described. If a link is being described which causes the same thing as another recently described link, the system inserts "also":

A causes B. C also causes B.

This list of described relationships also allows the system to stop describing a path if it has already explained the remainder of it. The sample session in the introductory chapter gives a further illustration of these features.

5.1.3 Dealing with Articles

The generator is designed to insert articles where it is appropriate to do so. In general, it can be quite complex to decide whether or not to use an article and whether the definite or indefinite article is appropriate. A relatively simple heuristic states that objects which are mass-nouns do not take articles while those that are not do. Unfortunately, one can easily think of numerous violations of this rule. Fortunately, this rule has been adequate for explaining program structures of the Digitalis Advisor.

When a word is defined in the knowledge base, a flag is placed on it if an article should be used with the word. If no flag is found on the concept being generated, the system examines concepts above to see if they are flagged. Thus, lower level concepts inherit the flag from higher concepts, however, inheritance occurs only up to the level of a concept with a tie of *s. If no flag is found or an over-riding flag is found, the system does not insert an article.[53]

The described solution still has a problem. If the concept `[block]` is flagged to indicate that an article should be inserted, the system will generate the phrase "the red the block" when asked to generate a phrase for the concept `[(block*f red)]`. This occurs because the system inserts an article because it inherits the flag from `[block]` when generating a phrase for `[(block*f red)]` then it inserts another article when generating a phrase for `[block]`. To get around this problem, there are actually two top

---

53. The inheritance is limited because this appears to reduce the number of explicit over-rides that must be inserted in the knowledge base.

level phrase generators used by the system. One of them checks whether an object is flagged and inserts an article if it is, and the other does not. By having the generators for the various ties make calls to the appropriate top level generator, it is possible to avoid the problem of inserting articles in inappropriate places. For example, by having the generator for *f call the phrase generator which does not insert articles when it generates a phrase for the ilk of a concept with a *f tie, the problem described above is avoided.

The problem of deciding whether to use the definite or indefinite article has been addressed by examining those situations where each is appropriate. So far, it seems to be appropriate to use the indefinite article only when describing patterns. In all other cases, the definite article is called for. While a more sophisticated system might require a greater number of distinctions, this solution has proved adequate for our needs.

5.1.4 Viewpoints

The reader may recall that one problem with previous explanation systems was the problem of *computer artifacts*. Computer artifacts are parts of the program which appear mainly because we are implementing an algorithm on a computer. If these steps are described to physicians, they are likely to be uninteresting and potentially confusing. The introductory chapter gave some examples of these computer artifacts. In the XPLAIN system, steps in prototype methods can be meta-characterized by certain viewpoints.[54] When a prototype method is instantiated, the instantiated steps will share these viewpoints. As the XPLAIN system is generating an explanation for a step it compares the viewpoint(s) of the step (if any) against a list of viewpoints which should be filtered out and another list of viewpoints which should be included. If one of the step's viewpoints appears on the include list, that step is included in the English explanation. If not, and one of the viewpoints appears on the exclude list, the step is excluded from the explanation. If the step has no viewpoint, it is included in the explanation. This approach allows us to separate those steps that are appropriate for a

---

54. This can occur either during the refinement of a step from a higher goal, or during a transformational refinement.

particular audience from those that are not. Of course, the exclude and include lists may be changed to reflect a changing understanding of the user's viewpoint.

While this is a simple solution from the standpoint of generation, it is a feasible one because we are employing an automatic programmer. In the domain principles, we bring together and define for the system to use, computer implementation knowledge and medical reasoning knowledge. A domain principle is thus the appropriate place to indicate what viewpoint show be taken on the knowledge that it is composed of. By placing a viewpoint on a step in a prototype method, we cause all the instantiations of that step (and there are usually several) to share that viewpoint. If we were to try to do the same thing at the level of the performance program (without an automatic programmer) we would have to annotate each individual step—we could not capture as high a level of abstraction.

This result is consistent with the observation we made in Chapter 1, where we stated that improvements in the quality of the explanations generated resulted more from the use of an automatic programmer than from increases in the sophistication of the generation routines. It should be pointed out, however, that while this solution allows the system to customize its explanations based on a particular viewpoint or set of viewpoints, the problem of deciding which viewpoint to present to a particular user remains open and is beyond the scope of this thesis.

5.2 The Answer Generators

This section will describe the higher level answer generators. When the user asks for an explanation these generators find the things which should be explained and set up the appropriate environment for explaining them.

5.2.1 Answering "Why" Questions

One of our chief goals in this research was to have the XPLAIN system explain why the performance program was doing what it was doing. In producing answers to such questions, the system makes use of the knowledge in the Domain Model and the Domain Principles as well as traces left behind by the automatic programmer resulting form its creation of the performance program. Some of the capabilities of the system were illustrated in the introductory chapter when the performance program was anticipating digitalis toxicity. Additional capabilities are illustrated below from those portions of the advisor that anticipate toxicity and assess toxicity.[55]

Is the patient showing signs of cardiomyopathy? (yes or no): why?

The system is anticipating digitalis toxicity. Cardiomyopathy causes increased automaticity, which may cause a change to ventricular fibrillation. Increased digitalis also causes increased automaticity. Thus, if the system determines that cardiomyopathy exists, it reduces the dose of digitalis due to cardiomyopathy.

This explanation is similar to those in Chapter 1.

The remaining explanations are produced while assessing toxicity.

Is the patient showing signs of paroxysmal atrial tachycardia with block? (yes or no): why?

The system is assessing the highly specific findings of digitalis toxicity. Increased digitalis may cause paroxysmal atrial tachycardia with block which is a highly specific finding of digitalis toxicity.

Is the patient showing signs of paroxysmal atrial tachycardia with block? (yes or no): overview

The system repeats the question, but the user wants an overview. This is produced by describing the method for assessing highly specific findings which was mentioned in the previous explanation.

To assess the highly specific findings of digitalis toxicity:

---

55. These examples all show the user questioning the system by entering a "why?" or "overview" when it requests an input. It is also possible to obtain a justification of a particular event by calling the LISP function JUSTIFY with the event as an argument.

1. The system assesses paroxysmal atrial tachycardia with block.
2. It assesses double tachycardia.
3. It assesses av-dissociation.
4. It combines the assessments of paroxysmal atrial tachycardia with block, double tachycardia and av-dissociation.

This produces the assessment of the highly specific findings of digitalis toxicity, which is used when the system combines the assessments of the highly specific findings of digitalis toxicity, the moderately specific findings of digitalis toxicity and the non-specific findings of digitalis toxicity.

To avoid leaving the user hanging, the system describes how the output of the method will be used in a higher context. This is done whenever the output of the method is something which is not likely to be familiar to the user. Thus, if the output of a method were a drug dose, no additional explanation would be generated (at least for a medical audience) because the user would be presumed to be familiar with the notion of a drug dose.

Is the patient showing signs of paroxysmal atrial tachycardia with block? (yes or no): why?

Still curious, the user asks "why?" again. This causes the system to give an explanation of the procedure that calls the procedure for assessing the highly specific findings. This explanation is much shorter, because the domain principle used to refine the higher level procedure had no domain rationale.[56]

The system is assessing digitalis toxicity. One step in doing that is to assess the highly specific findings of digitalis toxicity.

Is the patient showing signs of paroxysmal atrial tachycardia with block? (yes or no): overview

This time the description is offered at the level of the higher procedure.

To assess digitalis toxicity:
1. The system assesses the highly specific findings of digitalis toxicity.
2. It assesses the moderately specific findings of digitalis toxicity.
3. It assesses the non-specific findings of digitalis toxicity.
4. It combines the assessments of the highly specific findings of digitalis toxicity, the moderately specific findings of digitalis toxicity and the non-specific findings of digitalis toxicity.

---

56. This is explained in more detail later in this section.

This produces the assessment of digitalis toxicity, which is used when the system adjusts the dose of digitalis.

Is the patient showing signs of paroxysmal atrial tachycardia with block? (yes or no): no
Is the patient showing signs of double tachycardia? (yes or no): no
Is the patient showing signs of av-dissociation? (yes or no): no
Please enter the number of pvcs: why?

The system is assessing the moderately specific findings of digitalis toxicity. Increased digitalis causes increased automaticity. Increased automaticity may cause increased pvcs which is a moderately specific finding of digitalis toxicity.

Please enter the number of pvcs: overview

To assess the moderately specific findings of digitalis toxicity:
1. **The system assesses increased pvcs.**
2. **It assesses bigeminy.**
3. **It assesses salvos of pvcs.**
4. **It assesses second-degree av-block.**
5. **It assesses Wenkebach av-block.**
6. **It combines the assessments of increased pvcs, bigeminy, salvos of pvcs, second-degree av-block and Wenkebach av-block.**

Please enter the number of pvcs: 3
Is the patient showing signs of bigeminy? (yes or no): no
Is the patient showing signs of salvos of pvcs? (yes or no): yes
.
.
.

When a "why" question is entered, control passes to the routine that produces justifications. This routine determines at what level the description should be given, states what's going on in general, describes the domain rationale (if any) used in refining the step being described, and finally describes the step.

5.2.1.1 Choosing the Level of Description

The system uses the viewpoint attachments to determine where to start the explanation. The control stack of the MINT interpreter is available to the explanation modules. The justification routine goes up the stack looking for the first procedure which is not meta-characterized as an excluded viewpoint and which has no procedure meta-characterized as an excluded viewpoint above it. If that procedure happens to be a system primitive[57] with a system primitive above it, then the system keeps going up the stack until it finds a procedure which does not have a system primitive above it. For example, in the sample session above, when the second question is asked, the control stack is:

```
[((ASSESS*O (TOXICITY*F DIGITALIS))*I 1)]
[((ASSESS*O ((FINDINGS*R (TOXICITY*F DIGITALIS))*F
                                    (SPECIFIC*F HIGHLY)))*I 2)]
[((ASSESS*O PAT-WITH-BLOCK)*I 3)]
[(MIF-THEN*C
  (MEQUAL*C [[(ASK-USER*C [(MQUOTE*C [(STATUS*R PAT-WITH-BLOCK),]),]),
              (MQUOTE*C [PRESENT,])],]),
  (MSETQ*C 1, (ASSESSMENT*R PAT-WITH-BLOCK)),
  (MSETQ*C A-ABS, (ASSESSMENT*R PAT-WITH-BLOCK)))
   #M COMPUTER-VIEWPOINT]
[(MEQUAL*C [[(ASK-USER*C [(MQUOTE*C [(STATUS*R PAT-WITH-BLOCK),]),]),
             (MQUOTE*C [PRESENT,])],])]
[(ASK-USER*C [(MQUOTE*C [(STATUS*R PAT-WITH-BLOCK),]),])]
```

In this case, the [mif-then] has been meta-characterized as having a computer-viewpoint. Therefore, the system will start giving its explanation at the next level up, at the procedure that assesses PAT with block. This will be referred to as the *current description level.* In contrast, if the exclude list had not contained `[computer-viewpoint]` explanation would have begun at a lower level, producing the following explanation:

> **Is the patient showing signs of paroxysmal atrial tachycardia with block? (yes or no):** why?
>
> **The system is assessing paroxysmal atrial tachycardia with block. If the status of paroxysmal atrial tachycardia with block is equal to present, the assessment of**

---

57. System primitives include: `[mif-then]`, `[msetq]`, `[mless-than]`, etc.

paroxysmal atrial tachycardia with block is set to the assessment level for present findings (1), otherwise the assessment of paroxysmal atrial tachycardia with block is set to the assessment level for absent findings (0).

And, in answer to the question about pvcs, the following would have been produced:

Please enter the number of pvcs: why?

The system is assessing increased pvcs. If the number of pvcs is greater than the baseline number of pvcs (5), the assessment of increased pvcs is set to the assessment level for present findings (1), otherwise the assessment of increased pvcs is set to the assessment level for absent findings (0).

This is the sort of information a person maintaining the advisor might wish to know, but that a medical audience would probably not want to see.

5.2.1.2 Stating What's Going On in General

To give the user an overview of what the system is trying to accomplish, the system finds the next procedure above the current description level in the control stack. This will be called the *higher level procedure*. It then generates a phrase using the name of the procedure to describe what's going on:

The system is assessing the highly specific findings of digitalis toxicity.

5.2.1.3 Explaining the Domain Rationale

If the higher level procedure was refined using a domain principle which had a domain rationale, then the procedure at the current description level must be the result of one of the matches of the domain rationale. The system finds the domain rationale and the particular match of it that resulted in the procedure at the current description level. Flags are set to indicate to the tie-generators that they should replace occurrences of pattern variables with the objects they matched. After this environment has been set up, the complete pattern is found and converted to English using the tie-generators. For example, the domain principle that refined the procedure to assess highly specific toxic findings contained the following domain rationale:

```
[domain-rationale↑ #g
     [dr3 = (pattern*i 300)
      [structure↑ #e ((causal-chain*b (increased*o (drug*R !pv)))*e
                                   [dev2 = deviation↑↑])]
      [(dev2*characterization
        ((finding*r (toxicity*f (drug*R !pv)))*f (specific*R !pv)))]]]
```

When the appropriate environment was set up, the tie-generators produced this description for the pattern:

> **Increased digitalis may cause paroxysmal atrial tachycardia with block which is a highly specific finding of digitalis toxicity.**

5.2.1.4 Finishing Up the Explanation

Finally, the system uses the tie-generators to produce a description of the step at the current level of description. So in answer to the first question the system prints:

> **Thus, if the system determines that cardiomyopathy exists, it reduces the dose of digitalis due to cardiomyopathy.**

The system then re-iterates its original question. If the user asks "why?" again, the system moves the current description level up to the level of the higher level procedure and repeats the explanation process.

The reader may have noticed that the system did not generate a similar sentence in answer to the second question. That is, the system did not produce the sentence:

> **"Thus, the system assesses paroxysmal atrial tachycardia with block."**

as the last sentence of its answer to the second question. The reason is that such a sentence would have been redundant. The user already knows that the system is assessing paroxysmal atrial tachycardia with block, because he has just been asked a question about it. Following the general principle that the user should not be told something he already knows, the system deletes this part of the explanation if the step about to be described is a type of assessing step and the object of that step is the same as the thing the user has been asked about.

5.2.2 Explanation of Methods

Given the system we've described so far, it's relatively easy to get it to generate descriptions of methods by translating them directly into English. There are only a few subtleties. A function is needed to number the steps of the method sequence and pass them to the phrase generator. The system must also distinguish between functional subroutines, where a single value is passed back as the value of the subroutine, and conventional subroutines, with input and output arguments. The latter case may be handled quite simply, but in the former case some special things must be done.

We have to recognize that a functional subroutine is treated much like a "read-only" variable in programming. Reflecting that fact, when the system is describing one of the exiting steps[58] from a functional subroutine, it converts the name of the method to a noun and describes it as a variable which is set to the result of evaluating the exiting step. If the method being described is a kind of [determine-whether], the system re-arranges things a bit to improve readability. For example, one of the methods written by the automatic programmer is:

```
[(((DETERMINE-WHETHER*O (DECREASED*O SERUM-K))*I 1)*D [[SERUM-K,],])
 [METHOD: #Q (MLESS-THAN*C
              [[SERUM-K, (THRESHOLD*R (DECREASED*O SERUM-K))],])]]
```

The system generates the following description of that method:

> **If serum-k is less than the threshold of decreased serum-k (3.4), the system determines that decreased serum-k exists.**

The explanations given by the "overview" command in the previous section were produced by passing the current higher level procedure to the function that describes methods. However, as we pointed out in the introduction, this particular style of explanation has some limitations. In the next section, we present a different way of explaining methods which provides a richer sort of abstraction which can not be done in explanations produced directly from the code.

---

58. Note that there may be several exits from a functional subroutine.

5.2.3 Prototype Method Explanations

In the original version of the Digitalis Advisor, when we wanted to give a more abstract view of what was going on, we just described a higher level procedure [Swartout77a, Swartout77b]. In this regard, we were following the principles of structured programming. While this approach was often reasonable, there were times when it was considerably less than illuminating. The general method for anticipating digitalis toxicity was called "Check Sensitivities" in the old version of the Digitalis Advisor. An explanation of it appears in Figure 27. While this explanation does tell the user what sensitivities are being checked,[59] it does not say what will be done if sensitivities are discovered nor does it say why the system considers these particular factors to be sensitivities. Finally, it is much too redundant and verbose. The first objection can be dealt with by removing the calls to lower procedures and substituting the code of those procedures in-line. This results in the somewhat improved explanation produced by XPLAIN when it is asked to describe the method for anticipating digitalis toxicity (see Figure 28). However, while this explanation shows what the system does, it

---

Fig. 27. An Explanation From the Old Digitalis Therapy Advisor

(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.
```

---

59. The reader may notice that there were more sensitivities checked in the original version of the program than in the current version. We now feel that some of these, such as thyroid function and advanced age, should not be treated as sensitivities per se because they tend to have an effect on reducing renal function and hence slowing excretion, rather than on increasing sensitivity to digitalis. The other sensitivities would be easy to add by including the appropriate causal links in the domain model.

doesn't say why things like increased calcium, cardiomyopathy and decreased potassium are sensitivities, and if anything, it's even more verbose than the original explanation.

The reason we can't get the sorts of explanations we want by producing explanations directly form the code is that much of the sort of reasoning we want to explain has been "compiled out." Thus, we are forced into explaining at a level that is either too abstract or too specific. The intermediate reasoning which we would like to explain was done by a human programmer in the case of the old Digitalis Advisor. However, because this performance program was produced by an automatic programmer, we have a handle on that reasoning. For example, if we were to explain the domain principle that produced the code for anticipating digitalis toxicity rather then the code itself we would get the explanation that appears in Figure 29. This explanation is produced by first describing the domain rationale with the refinement pattern variables [60] replaced by what they matched, but with the domain pattern variables[61] described as themselves rather than as what they matched. Thus while the system says "increased digitalis" rather than "increased drug", it also says "finding" rather than "increased serum-K". The next part of the explanation is produced by describing the prototype

---

Fig. 28. An Explanation From the Code for Anticipating Toxicity

(describe-method [((ANTICIPATE*O (TOXICITY*F DIGITALIS))*I 1)])

To anticipate digitalis toxicity:
1. **If the system determines that cardiomyopathy exists, it reduces the dose of digitalis due to cardiomyopathy.**
2. **If the system determines that decreased serum-k exists, it reduces the dose of digitalis due to decreased serum-k.**
3. **If the system determines that increased serum-ca exists, it reduces the dose of digitalis due to increased serum-ca.**

---

60. Those are the variables in the head of the domain principle that were bound during plan finding by the automatic program writer.

61. The pattern variables that are matched against the domain model.

Fig. 29. Explanation of a Domain Principle

(describe-proto-method [(anticipate*o (toxicity*f digitalis))])

The system considers those cases where a finding causes a dangerous deviation and increased digitalis also causes the dangerous deviation. If the system determines that the finding exists, it reduces the dose of digitalis due to the finding.

The findings considered are increased calcium, decreased potassium and cardiomyopathy.

---

method. Finally, the set of values is given for the domain variable used in the prototype method. Thus, the use of an automatic programmer not only allows us to justify the performance program, but it also allows us to give better descriptions of methods by making available intermediate levels of abstraction which were not previously available.

5.2.4 Explaining Events

The MINT interpreter can be set up to leave behind a trace of its execution. As it executes a procedure, it creates an *event object* (see Chapter 2). This event object records the call and method used, the variable environments on entrance and exit, and the value returned if the procedure is a functional subroutine. These events can be examined by the system after execution is completed to produce an explanation of what the system did for a particular patient.

Once we have the mechanisms in place to explain methods, it turns out to be quite easy to explain events. Basically, it's done by having the system examine the event to be explained, generate a heading sentence using the call of the event, and then generate phrases for the immediate subevents of the event.[62] The major changes that have to be made are that a flag has to be set so that verbs are generated in the past tense, and the generator for [if-then] has to be modified to indicate the choice taken.

---

62. As when explaining methods, the subevents are filtered by their viewpoints.

This is done by first having the generator check that there was an action taken by the [if-then][63] and then having it generate English for the predicate and the action taken.[64] Finally, the system generates a phrase indicating the final output values of the routine. Some examples are presented in Figure 30.

Fig. 30. Examples of Event Explanations

"How did the system anticipate digitalis toxicity?"

*(describe-event [(event*i "e0002")])*

To anticipate digitalis toxicity:
1. The system determined that cardiomyopathy did not exist.
2. The system determined that decreased serum-k did not exist.
3. Since the system determined that increased serum-ca existed, the system reduced the dose of digitalis due to increased serum-ca.

The adjusted dose of digitalis was set to 0.20.

"How did the system determine that serum-ca was increased?"

*(describe-event [(event*i "e0016")])*

Since serum-ca (13) was greater than the threshold of increased serum-ca (11) the system determined that increased serum-ca existed.

---

63. For example, the [if-then] would not have taken an action if its predicate evaluated to false and there was no "else" clause in the [if-then], or if the action taken was of a viewpoint that was filtered out. If no action was taken, or the action taken is filtered out, the system just generates English for the predicate (for example, see steps 1 and 2 in Figure 30.)
64. If the action taken is the "else" clause, then the system inverts the logic of the predicate when converting it to English.

5.2.5 Non-English Explanation

Although it might not be clear from this chapter, I feel that there are many situations in which English is not the only or best way to give an explanation. There are many situations where explanations are much more effective when English text is supplemented with figures, charts, drawings and so forth.

A case in point is explaining mathematical formulas. Mathematical formulas expressed in English are not only verbose; they are ambiguous as well [Swartout77a]. As a small step in moving toward a larger investigation of non-English explanations, the XPLAIN system describes arithmetic expressions using mathematical notation. This is done by choosing shortened variable names for the variables and converting the prefix MINT form for arithmetic expressions to an infix form which is printed. Figures 31 and 32 show some examples.

Fig. 31. Describing Events with Arithmetic Expressions

"How did you reduce the dose for increased serum-ca?"

(describe-event [(event*i "e0019")])

The dose after adjusting for increased serum-ca was set according to the following formula:

D2 = D1 C

where:
C = the reduction constant for increased serum-ca (0.8)
D1 = the dose before adjusting for increased serum-ca (0.25)
D2 = the dose after adjusting for increased serum-ca (0.20).

The dose of digitalis adjusted for the condition of the heart muscle, serum-k and serum-ca was set to 0.20.

Fig. 32. Describing Methods with Arithmetic Expressions

"How does the system combine the assessments of highly specific, moderately specific, and non-specific findings of toxicity?"

```
(describe-method
 [((COMBINE*O
   (SET*I
     (ASSESSMENT*R
       ((FINDINGS*R (TOXICITY*F DIGITALIS))*F (SPECIFIC*F HIGHLY))),
     (ASSESSMENT*R ((FINDINGS*R (TOXICITY*F DIGITALIS))*F
             (SPECIFIC*F MODERATELY))),
     (ASSESSMENT*R
       ((FINDINGS*R (TOXICITY*F DIGITALIS))*F NON-SPECIFIC))))*I
  2)])
```

The combined assessment of the highly specific findings of digitalis toxicity, the moderately specific findings of digitalis toxicity and the non-specific findings of digitalis toxicity is set according to the following formula:

$$C = F1\,A1 + F2\,A2 + F3\,A3$$

where:

A1 = the assessment of the highly specific findings of digitalis toxicity
A2 = the assessment of the moderately specific findings of digitalis toxicity
A3 = the assessment of the non-specific findings of digitalis toxicity
C = the combined assessment of the highly specific findings of digitalis toxicity, the moderately specific findings of digitalis toxicity and the non-specific findings of digitalis toxicity
F1 = the weighting factor of the highly specific findings of digitalis toxicity (4)
F2 = the weighting factor of the moderately specific findings of digitalis toxicity (2)
F3 = the weighting factor of the non-specific findings of digitalis toxicity (1).

6. A Discussion of the Automatic Programming Approach to Explanation

This chapter addresses several issues that have occurred to me while implementing the system. Most of these deal with the interrelationships between the automatic programmer, the performance program, and the explanations that can be produced.

6.1 Does Automatic Programming Affect the Performance Program?

We attempted to get the XPLAIN system to write procedures that captured the intent of the corresponding sections of the original Digitalis Advisor as much as possible. However, there were situations where we decided to adopt different strategies. Usually this occurred because the attempt to find domain principles to generate the program forced us to look more closely at the methods we were adopting, and occasionally we discovered that the original program was flawed or inconsistent.

For example, in the original Digitalis Advisor, myxedema[65] was considered as one of the digitalis sensitivities (like increased calcium or decreased potassium). In creating the domain model and domain principle to anticipate toxicity in the new system, we realized that a problem existed because myxedema was not causally additive with the other sensitivities and hence would not meet all the refinement constraints required by the domain principles in refining the program (see chapter 3). To resolve the problem, we dug deeper into the medical literature and discovered that myxedema should not really be considered a sensitivity at all! In fact, myxedema reduces the excretion of digitalis through the kidneys and hence tends to make digitalis accumulate in the body rather than making the patient more sensitive to digitalis. Thus, the appropriate way to handle myxedema is not as a sensitivity, but rather as a factor which modifies the excretion rate in the pharmacokinetic model.

---

65. Myxedema is a disease caused by decreased thyroid function. Signs of the disease include dry skin, swellings around the lips and nose, mental deterioration, and a subnormal basal metabolic rate.

One of the advantages of the automatic programming approach is that it forces the user to think harder about the performance program and its implementation. Just as the implementation of any theory on a computer forces one to work out the details and think about the consistency of the theory, working out the implementation of the implementation carries the process one step further and forces one to think that much harder about the entire undertaking.

6.2 Is this Approach to Explanation Compatible with Others?

The approach to explanation espoused in this thesis is compatible with other approaches such as using canned text or producing explanations by translating the program code (see Chapter 1). It should be regarded as an extension of these earlier approaches rather than a replacement for them. This is important because there may be times when it is not feasible to get an automatic programmer to produce the code. The XPLAIN system allows the user to hand code parts of the system and can generate the remainder automatically. Those parts of the system that are hand-coded can be translated to English just like the parts that are automatically generated. The current implementation of the XPLAIN system does not support canned-text explanations (mainly because they haven't been needed) but could easily be modified to do so.

6.3 Is Automatic Programming Too Hard?

One possible objection to the whole approach to explanation advocated in this thesis is that it is just too hard to get an automatic programmer to write the performance program. When I first began this research, I thought that was the case. The original plan for producing better explanations was to create structures detailing the development of the performance program, but these structures would be created by hand rather than automatically. It was feared that automatic programming was just too hard. However, as the research progressed, it became clear that if we had sufficiently powerful representations available so that it could be said that in some sense explanations were being produced from an understanding of the program, then actually writing the program in the first place wouldn't be all that much more difficult. I suspect this is true in general. It seems that the primary difficulty in both explanation and automatic programming is a

knowledge representation problem, and that the kinds of knowledge to be represented in both cases are similar so that a solution to one case makes the other much easier. Furthermore, if this conjecture is correct, it implies that we are not likely to find easier approaches to explanation than the one presented here (if we require that our explanations be based on an understanding of the program as opposed to, say, canned-text.)

6.4 Levels of Language

Computer science has developed a variety of different types of languages for describing algorithms. At the lowest level, there is machine language. As we move to more abstract languages we encounter assembly languages, high level languages, very high level languages, and finally automatic programmers. At each level, various features are introduced which make it easier to use the same piece of code in different places. Often, these same features make it easier to understand and explain the code too.

Even at machine language level, most computers have a special instruction or set of instructions to facilitate writing subroutines. Subroutines are a very powerful idea because they allow the same piece of code to be used in several parts of a program. From the standpoint of explanation, subroutines are powerful for two reasons. First, if a subroutine is called several times, the fact that the same code is being used is very clearly indicated. (It might not be so clear if the code for the subroutine was inserted everywhere in place of the call.) This means that the code for the subroutine only needs to be understood once. More importantly from the explanation standpoint, subroutines can be used to partition a task into easy to understand pieces. Furthermore, these pieces can themselves be broken down into still more specific subroutines so that a hierarchy of subroutines is formed. This, of course, is one of the main thrusts behind structured programming [Dahl72] and for explanation it is important because it allows us to suppress the details of a calculation within a lower level subroutine. However, to really take advantage of this procedural hierarchy, we need to be able to give them names.

Assembly language allows us to give names to subroutines, data objects and labels.[66] From an explanatory viewpoint, this is valuable because it allows us to give program objects names which relate their role in the program to their role in the application domain of the program. This feature is often misused however, and one frequently finds programs where a variable that has been used in one way is later used in another way so that its name, which was given to reflect its earlier use, conflicts with its later use. This is often done in a mis-guided effort to save memory, and is symptomatic of the problems that result from trying to optimize code and describe an algorithm at the same time.

Macro packages are typically thought of as extending assembly languages. For the purposes of this discussion, however, they tend to sully the waters a bit because it is possible (in theory, at least) to build arbitrarily high level languages using a macro package since most of them can perform powerful transformations on program structures. Typically, however, macros are used so that an operation which is done repeatedly (possibly with slight variations) only has to be coded once. In that respect, they are similar to subroutines. However, unlike subroutines, macros are usually not organized into hierarchies so that the output of a macro expansion contains other macros to be expanded. Usually the results of macro expansion are not intended to be examined by the programmer (unless, of course, he is debugging the macro expansion itself).

Higher level languages such as LISP, PL/1, Algol, and so forth provide additional features. These languages provide a set of more natural operators[67] and data types [68] that free the user from concerns about implementation details such as register allocation, saving and restoring state when calling subroutines, implementation of strings and arrays, and so forth. One of the most important features of these languages is that they allow the user to input *expressions* while lower level languages only permitted him to input sequences of instructions. By moving to higher level operators and removing some implementation details, these languages make it easier to explain

---

66. Although the labeling allowed by typical assembly languages is often very restrictive.
67. i.e. arithmetic operators, logical operators, string operators, etc.
68. i.e. arrays, lists, strings

programs written in them because the operators are closer to the sorts of operations people are familiar with and irrelevant details of the implementation are suppressed. Explanation packages have been developed which work with languages at this level [Roberts79].

Very high level languages attempt to provide still higher level and more natural operators and control structures but usually within the context of a limited domain. For example, the Business Definition Language (BDL) developed by Hammer, Howe and Wladawsky [Hammer74] is intended to make it possible for a non-programmer businessman to define business application programs, such as order handling and invoicing, with only minimal (if any) training. To achieve such a goal, the language must clearly have operators, data types, and control structures which are familiar to businessmen. If the goal is realized, it should be relatively easy to explain such programs, and some efforts have been made in that direction [Mikelsons75]. The main weakness of this approach seems to be its domain dependence. While the approach itself is general, the languages are domain-oriented. The knowledge that a particular language has about a particular domain is compiled into the language itself and would make it quite difficult to extend the language or apply it to other domains.

Some other very high level languages achieve a somewhat broader range of applicability by dealing with a broader domain. SETL [Schwartz74], for example, allows the user to specify various operations on sets which the language then implements by choosing an appropriate implementation from several possible ones. While its application may be a broader, SETL is primarily limited to sets, just as BDL is limited to business programs.

One of the main approaches to automatic programming has been what might be called the transformational approach. In this approach, various transformations are repeatedly applied to an initial input-output specification or algorithmic sketch until an executable program is produced. Superficially, these transformations are somewhat analogous to macros, however, unlike macros, the structures produced by these transformations and the transformations themselves are intended to mimic those of human programmers and hence to be meaningful to programmers.

One of the transformational approaches is based on theorem proving [Manna77]. The automatic programmer is provided with the input-output specifications of the desired program, which are written in mathematical notation. The input-output specifications are usually not given in terms of the primitive operators of the target language (the language the desired program is to be written in). The automatic programmer also has available a set of equivalence-preserving transformations. Some of these transformations will contain primitive operators. The transformations are applied to the input-output specifications, using certain rules, until an executable program is produced. The transformations themselves contain relatively little algorithmic information, so the system "discovers" the algorithm as it writes a program meeting the specifications. While the discovery aspect is appealing from an explanatory point of view, the fact that programs are essentially derived from basic principles each time makes the synthesis of any but the simplest programs extremely time-consuming. Additionally, since the transformations are at such a low level, the program produced may not be well-structured.

The refinement approach [Barstow77, Balzer77, Green79, Long77, Rich79] rests on the assumption that the abilities of human programmers come more from their knowledge of a large number of plan templates which can be customized for a particular application than from their use of general purpose deductive rules. These plans are used as the transformations in such systems. Since the plans are usually organized into a hierarchy based on their specificity, the resulting programs tend to be well-structured. Also, since the transformations are from abstract to specific, a directionality is imposed on the search for an executable program, improving the efficiency of the automatic programmer. From an explanatory standpoint, this approach allows explanations to be offered at different levels of abstraction which should be meaningful to the user, and it can improve the structure of the program, but at the expense of not being able to explain everything down to basic principles.

Recently, Barstow [Barstow80] has suggested a hybrid approach. The system would basically be a refinement system, but a theorem prover would be used to prove that constraints associated with particular transformations held in a particular situation, thus indicating the applicability of the transformation. If there were an explanatory facility associated with this system, it would have the characteristics of the refinement

systems with the additional capability to explain in detail why a particular transformation was chosen.

The XPLAIN system uses a refinement-type automatic programmer. The major difference between it and other programmers lies in the Domain Model and the Domain Rationale. The Domain Rationale is essentially an additional program specification that is not stated at the outset. It allows us to cleanly represent the fact that some additional program specifications may be needed if certain refinement paths are taken but they may not be needed in others. From an explanatory viewpoint, it is valuable because it allows us to make the domain principles more abstract and more independent of the application domain, and the domain rationale represents the criteria under which phenomena in the domain model must be considered in the refinement of a particular step.

It is easy to get an automatic programmer to leave behind a trace of its reasoning in creating a program. Structured programming is also designed in part to aid human programmers in capturing the process of creating a program. The difference is that in structured programming much of the reasoning remains in the head of the programmer. In the introductory chapter, we showed how this can adversely affect the quality of explanations. In structured programming, one kind of procedural abstraction is available. The name of a procedure is supposed to be a summary of the actions it performs. In the automatic programmer used in the XPLAIN system, this sort of abstraction is available, but another sort of abstraction, based on the domain principles, is also available (and is described in Chapter 5). Since the domain principles may be used repeatedly to produce different procedures, the XPLAIN system can capture abstractions that go across procedure boundaries. These sorts of abstractions, together with the refinement structure left behind by the program writer, contribute substantially to the explanations the system can produce.

6.5 Is a Top-down Approach Really Necessary?

The XPLAIN system can produce good justifications in part because it has access to the refinement structure produced by the automatic programmer in a top-down fashion. A natural question to ask is whether a bottom-up approach might not work equally well. In other words, one could envision a system that analyzed an existing

program structure into higher principles, and explained it at that level. This system would need to employ knowledge structures much like the domain principles and domain model, but they would be used in reverse to parse the existing performance program into a parse tree (which would correspond to XPLAIN's refinement structure). [69] This approach is enticing: it seems that if it can be made to work in general then any program can be explained whether or not it was written with explanation in mind. While such an approach might be attractive in principle, I feel there are several obstacles that make its implementation difficult. First, as was pointed out earlier, the process of writing a program is a process that distills "how-to-do" something out of a much larger body of knowledge. Given that, the analyzer will not be able to explain a program without knowledge structures similar to the domain principles and domain model used by XPLAIN, and furthermore, these structures will have to be similar in both size and scope to those used by XPLAIN. While XPLAIN works deductively this recognizer would have to work by induction and the possibility of ambiguity would exist. In the XPLAIN system, the major intellectual effort involved figuring out what the domain principles and domain model were and how they should be represented. Once they existed, it was relatively easy to get the program writer to use them to write the performance program. Since both require similar knowledge structures, and once they exist it's easy to synthesize the performance program, the top-down approach would appear to have the edge.

6.6 Limitations and Extensions of the XPLAIN System

While the explanations presented in this paper provide an indication of the power of the automatic programming approach to explanation, they do not exhaust its possibilities. The current system can be extended in several areas:

6.6.1 What Can the Current Implementation Do?

The current domain model and domain principles contain enough knowledge to generate all the examples within this document. They can also produce additional examples, although these are quite similar to those that appear here. There are three

---

69. see also [Clancey79] for a discussion of this approach

things that would have to be done to complete the implementation of the Digitalis Advisor. First, it would be necessary to implement routines to assess therapeutic improvement. These should not be too difficult because they can be very similar to the routines that assess toxicity. Second, it would be necessary to develop domain principles to adjust the dose based on the therapeutic and toxic assessments. Third, it would be necessary to implement various utility functions for gathering data and the pharmacokinetic model of digitalis excretion. These would probably be implemented in LISP, as they were in the old digitalis advisor. While there would be a fair amount of programming involved, I do not foresee any major conceptual hurdles. Once this implementation was completed, the domain principles of that program could be used with different domain models to develop similar consulting programs (i.e. programs that offered advice about drug therapy). I think that from the standpoint of programs of this type the implementation of the Writer and the generation routines is complete (or nearly so). That is, I would not expect to have to make major modifications to them to complete the implementation of the digitalis advisor.

6.6.2 Improved Answer Generators

Additional answer generators could be employed to provide the user with: 1) improved access to the domain model so that the domain model itself could be explained as well as its use in the development of the program; 2) improved explication of the decisions made by the automatic programmer; 3) an ability to assess the significance of the program's recommendations.

1) Currently, the explanation routines make use of the domain model to justify a piece of program structure. It would be nice (particularly in a teaching environment) to have answer generators which focused on the domain model so that a user could enhance his understanding of the domain. In addition, it would not be particularly difficult to cross-reference the domain model with the refinement structure to indicate where the domain knowledge was used in the program. This would allow the system to answer questions such as, "How does the system take increased calcium into account?" The answer would be produced by finding those places in the system where the concept

increased calcium was used[70] and then displaying the appropriate pieces of code.

2. The current system has no ability to explain domain or refinement constraints. In part, this is because the implementation of the XPLAIN system has concentrated on offering explanations to medical users and it was felt that the constraints have more of a computer than medical viewpoint. But that is not entirely correct. Recall that when the system was refining the split-join associated with anticipating toxicity it was necessary to assure that all the factors involved were at least causally additive. Whether or not the factors are additive is a question that clearly involves medical knowledge, and it is something which should be explainable to a medical audience in terms of its medical significance.

3. The system should also be able to explain the advice of the performance program in terms of its medical significance. For example, the advisor might conclude that no digitalis should be given for 3 hours and then 0.25 mg should be administered. If the advice was given at 11pm, the patient would have to be awakened at two in the morning if the attending physician wished to follow the program's recommendation to the letter. However, since digitalis has a relatively long half-life, the precise timing of doses (within a few hours) is not thought to be terribly important. In this case, the inconvenience and discomfort involved in waking the patient would probably dictate that the patient receive the drug at an earlier time. While we could program the system so that it does not give drugs during sleeping hours, it seems that that approach might eventually result in a program which knew substantially more about hospital procedure than about digitalis therapy. A better approach might be one where the system could indicate to the user the importance of its recommendations. For example, in this case, the system could mention that a variation of a few hours in drug administration would not be significant.

---

70. For example, increased calcium could be a match for a pattern variable used in a domain rationale or as an argument to a domain constraint.

6.6.3 Telling White Lies

Currently, the system can describe what it does and why at various levels of abstraction by describing the methods it uses, the refinement structure, the domain principles and the domain model. While it can leave out details based on viewpoint or by using a higher abstraction, it does not deliberately distort its explanations. Yet sometimes human teachers do exactly that to make their explanations easier to understand. The XPLAIN system could easily be modified to tell these "white lies" by linking alternate refinements (the white lies) to the existing refinement structure along with the differences between them and the refinement structure actually used by the program. In this way the system could offer the user the (presumably easier to understand) inexact explanation first, and then use the difference links to explain how things really work.

But where do these white lies come from? Sometimes a teacher may create them from scratch, but often they are just earlier versions of what was thought at the time to be the complete, final version of the theory, program or whatever. For example, the old Digitalis Advisor used to adjust the dose for sensitivities using a simple threshold model: if the level of serum potassium (say) was below a certain threshold, the dose was reduced by a fixed percentage. The current version of the Digitalis Advisor[71] makes a sliding reduction depending on how depressed the serum potassium is. The threshold model is more understandable, but the sliding reduction is more accurate. Rather than throwing away old versions of the performance program, it might be interesting if the program writer kept them around and noted the differences between the refinement of the old program and the new and where these differences arose (i.e. new principle, different domain model, etc.). The explanation routines could then use the old program fragments as a source for white lies[72] and after the old version was understood, the difference links could be used to indicate how things really worked. Additionally,

---

71. Not the version written by the XPLAIN system, but a LISP-based, medically more advanced version (which cannot justify itself) which was developed by Bill Long in parallel with the XPLAIN system.

72. Of course, the system would have to be careful. Sometimes new program fragments would result from a better understanding which resulted in a simpler *and* more accurate program. In that case, referring to the old program would gain nothing.

recording the changes between versions would allow the system to offer effective explanations about those changes to a user who hadn't used the system for a while. To continue the example above, suppose a user who had last used the performance program when it made reductions by a threshold used the new version with sliding reductions. If his patient were only slightly hypokalemic, he might wonder why the reduction for decreased potassium was much smaller than before. The system can justify the difference only if it has access to the differences between the two versions and the reasons for those differences.

6.6.4 Telling the User What He Wants to Know

While the current system has a limited ability to tailor the explanation to the interests of the user and to model what has been explained to him, the quality of the explanations could be substantially improved if the results of other research efforts could be integrated with the XPLAIN system. These include: 1) having the system model what it believes the user knows [Genesereth79], 2) developing tutorial strategies giving the system a more global view of its interaction with the user and allowing it to take part in directing it [Carr77, Clancey79], 3) on the opposite end of the scale, improving the low level English generators so they are more firmly grounded on linguistic principles [McDonald80, Mann80] and 4) improving the system's understanding of its own explanatory capabilities and the user's question so that it can reformulate the user's request into what it can deliver [Mark80].

6.7 Conclusions

I feel that the major contribution of this research is that it brings together automatic programming and program explanation. The use of an automatic programmer to generate the performance program and keep around a trace of decisions made during the refinement of that program makes it possible to justify the consulting system in a more flexible and more powerful way than other existing methodologies. This approach also allows the system to employ abstractions not otherwise available. Finally, from the standpoint of automatic programming, I feel that the notions of a domain model and domain rationale have interesting implications it terms of program specification, because

the specification of the performance program is interlaced with its refinement.

7. References

[Barstow77]
Barstow, D., "A Knowledge-Based System for Automatic Program Construction," Proceedings of the Fifth International Conference on Artificial Intelligence, 1977

[Barstow80]
Barstow, D., "The Roles of Knowledge and Deduction In Algorithm Creation," Yale University, Department of Computer Science, Research Report # 178, April 1980

[Balzer77]
Balzer, R., Goldman, N., Wile, D., "Informality in Program Specifications," Proceedings of the Fifth International Conference on Artificial Intelligence, 1977

[Carr77]
Carr, B., "Overlays: a Theory of Modelling for Computer Aided Instruction," MIT AI Laboratory Memo 406, February 1977.

[Clancey79]
Clancey, W.J., "Transfer of Rule-based Expertise Through a Tutorial Dialogue", Stanford University, Department of Computer Science, STAN-CS-79-769. 1979

[Dahl72] Dahl O.J., Dijkstra E.W., Hoare C.A.R., *Structured Programming*, Academic Press, 1972

[Davis76]
Davis, R., "Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases," PhD thesis, Stanford Artificial Intelligence Laboratory Memo 283(1976).

[Doherty61]
Doherty J.E., Perkins W.H., Mitchell G.K., "Tritiated digoxin studies in human subjects," Arch. Intern. Med. 108:531-539, 1961

[Doherty70]
Doherty J.E., Flanigan W.J. et al, "Tritiated Digoxin XIV, Enterohepatic circulation, absorption and excretion in human volunteers," *Circulation* 42:867-873, 1970

[Doherty73]
Doherty J.E., "Digitalis Glycosides: Pharmacokinetics and their Clinical Implications," Ann. Intern. Med. 79:229-238, 1973.

[Doyle79]
Doyle,J., "A Truth Maintenance System," *Artificial Intelligence* Volume 12 (231-272) 1979

[Genesereth79]
Genesereth, M.R., "The Role of Plans in Automated Consultation," *Proceedings of the Sixth International Conference on Artificial Intelligence*, 1979

[Gorry78]
Gorry, G. A., Silverman, H., and Pauker, S. G., Capturing Clinical Expertise: A Computer Program that Considers Clinical Responses to Digitalis, *American Journal of Medicine* 64:452-460, (March 1978).

[Green79]
Green, C.C., Gabriel, R.P., Kant, E., Kedzierski, B.I., McCune, B.P., Phillips, J.V., Tappel, S.T., Westfold, S.J., "Results in Knowledge Based Program Synthesis," Proceedings of the Sixth International Joint Conference on Artificial Intelligence, 1979

[Hammer74]
Hammer, M.M.. Howe, W.G., Wladawsky, I., "An Interactive Business Definition System," *Proceedings of a Symposium of Very High Level Languages*, SIGPLAN Notices Volume 9, Number 4, April 1974

[Hawkinson80]
Hawkinson, L.B., "XLMS: A Linguistic Memory System," MIT Laboratory for Computer Science TM-173, 1980

[Jelliffe70]
Jelliffe R.W., Buell J., Kalaba R. et al, "A Computer Program for Digitalis Dosage Regimens," Math. Biosci. 9:179-193, 1970

[Jelliffe72]
Jelliffe R.W., Buell J, Kalaba R, "Reduction of digitalis toxicity by computer-assisted glycoside dosage regimens," Ann. Intern. Med. 77:891-906, 1972

[Long77]
Long, W.J., "A Program Writer," MIT Laboratory for Computer Science, TR-187, 1977

[Mann80]
Mann, W.C., Moore, J.A., "Computer as Author—Results and Prospects," USC Information Sciences Institute ISI/RR-79-82, 1980

[Manna77] Manna, Z., Waldinger, R., "The Automatic Synthesis of Systems of Recursive Programs," Proceedings of the Fifth International Conference on Artificial Intelligence, 1977

[Mark80]
Mark, W., "Rule-Based Inference in Large Knowledge Bases," *Proceedings of the First Annual National Conference on Artificial Intelligence*, 1980

[Martin79]
Martin, W.A., "Roles, Co-Descriptors and the Formal Representation of Quantified English Expressions," MIT Laboratory for Computer Science TM-139, September 1979

[McDonald80]
McDonald, D.D., "Natural Language Production as a Process of Decision-making Under Constraints," MIT PhD Thesis, 1980

[Mikelsons75]
Mikelsons, M., "Computer Assisted Application Description," Second ACM Symposium on Principles of Programming Languages, 1975

[Minsky75]
Minsky, M., "A Framework for Representing Knowledge," in *The Psychology of Computer Vision*, P.H. Winston (ed), McGraw-Hill, 1975

[Ogilvie72]
Ogilvie R.I., Reudy, J., "An Educational Program in Digitalis Therapy," *Journal of the American Medical Association*, 222:50-55, 1972

[Peck73]
Peck C.C., Sheiner L.B. et al: "Computer-assisted Digoxin Therapy," *New England Journal of Medicine* 289:441-446, 1973.

[Pauker76]
Pauker, S.G., Gorry, G.A., Kassirer, J.P., and Schwartz, W.B., "Toward the Simulation of Clinical Cognition: Taking a Present Illness by Computer," The American Journal of Medicine 60:981-995 (June 1976).

[Pople77]
Pople, H.E. Jr., "The Formation of Composite Hypotheses in Diagnostic Problem Solving: an Exercise in Synthetic Reasoning," Proceedings of the Fifth International Joint Conference on Artificial Intelligence (1977).

[Rich79]
Rich, C., Shrobe, H., Waters, R., "Overview of the Programmer's Apprentice," *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, August 1979

[Roberts79]
Roberts, B., "Building English Explanations from Function Descriptions," MIT AI Laboratory Working Paper 185. 1979

[Schwartz74]
Schwartz, J., "Automatic and Semiautomatic Optimization of SETL," *Proceedings of a Symposium of Very High Level Languages*, SIGPLAN Notices Volume 9, Number 4, April 1974

[Sheiner72]
Sheiner L.B., Rosenberg B., Melmon K., "Modelling of Individual Pharmacokinetics for Computer-aided Drug Dosage," *Computers and Biomedical Research* 5:441-459, 1972

[Shortliffe76]
Shortliffe, E.H., **Computer Based Medical Consultations: MYCIN,** Elsevier North Holland Inc. (1976)

[Silverman75]
Silverman, H., "A Digitalis Therapy Advisor," MIT Project MAC TR-143, 1975

[Stallman76]
Stallman, R.M.,Sussman, G.J., "Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. MIT AI Laboratory Memo 380, 1976.

[Sussman75]
Sussman, G.J., *A Computer Model of Skill Acquisition*, American Elsevier Publishing Co. 1975

[Swartout77a]
Swartout, W.R., "A Digitalis Therapy Advisor with Explanations," MIT Laboratory for Computer Science TR-176, February 1977

[Swartout77b]
Swartout, W.R., "A Digitalis Therapy Advisor with Explanations," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 1977

[Winograd71]
Winograd, T., "A Computer Program for Understanding Natural Language," MIT Artificial Intelligence Laboratory TR-17, 1971