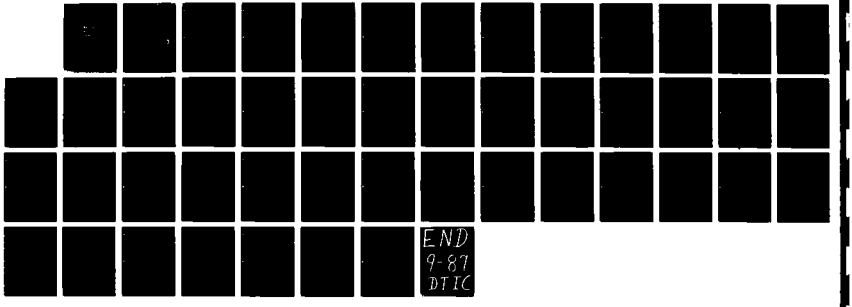
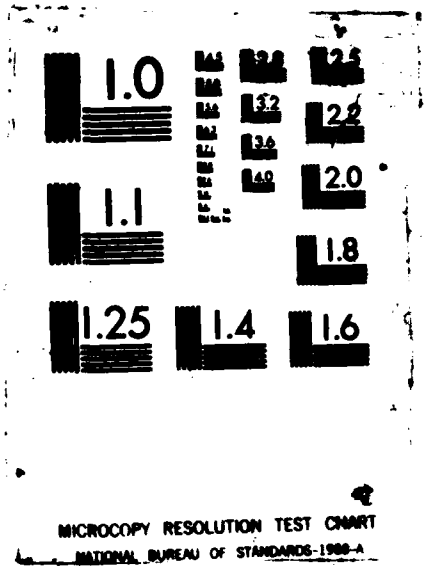
AD-A183 918 THE PROGRAMMER'S APPRENTICE: A PROGRAM SYNTHESIS SCENARIO(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB C RICH ET AL. NOV 86 1/1

UNCLASSIFIED AI-M-933 N00014-85-K-0124 F/G 12/5 NL





REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 933	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): The Programmer's Apprentice: A Program Synthesis Scenario	5. TYPE OF REPORT & PERIOD COVERED: AI-Memo
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Charles Rich & Richard C. Waters	8. CONTRACT OR GRANT NUMBER(s): NSF Grant#IRI-8616644; IBM & Sperry Corp.; ARPA/ONR Grant#N0001485K0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: November 1986 13. NUMBER OF PAGES: 45
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, VA 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programmer's Apprentice
Automatic Programming
Software Engineering
Program Synthesis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A scenario is used to illustrate the capabilities of a proposed Synthesis Apprentice. Given a specification, the Synthesis Apprentice will be able to make many of the design decisions needed to synthesize the required program. The Synthesis Apprentice will also be able to detect various kinds of contradictions and omissions in a specification.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 933 November 1986

The Programmer's Apprentice: A Program Synthesis Scenario

by

Charles Rich
Richard C. Waters

Abstract

A scenario is used to illustrate the capabilities of a proposed Synthesis Apprentice. Given a specification, the Synthesis Apprentice will be able to make many of the design decisions needed to synthesize the required program. The Synthesis Apprentice will also be able to detect various kinds of contradictions and omissions in a specification.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the Sperry Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor implied, of the National Science Foundation, of the IBM Corporation, of the Sperry Corporation, or of the Department of Defense.

The Programmer's Apprentice Project

The Programmer's Apprentice project uses programming as a domain for studying and attempting to duplicate human problem solving skills. Recognizing that it will be a long time before it is possible to fully duplicate human abilities in this domain, the near-term goal of the project is the development of a system, called the Programmer's Apprentice, which provides intelligent assistance in various phases of the programming task.

Viewed at the highest level, software development is a process that begins with the desires of an end user and ends with a program that can be executed on a machine. The first step of this process is traditionally called requirements acquisition, while the last step is called implementation. Figure 1 shows how the current and proposed demonstration systems in the Programmer's Apprentice project support these activities.

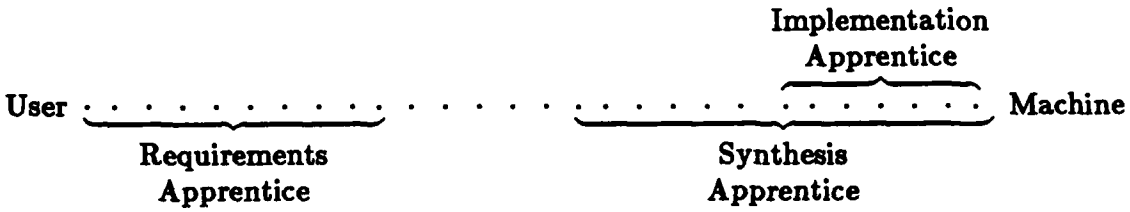


Figure 1: Parts of the Programmer's Apprentice along the spectrum of software development activities.

To date, most of the research in the project has focused on the development of an Implementation Apprentice. This has resulted in a working demonstration system called the Knowledge-Based Editor in Emacs (KBEmacs)[6]. The principal benefit of KBEmacs is that it allows a programmer to construct a program rapidly and reliably by combining algorithmic fragments stored in a library. An additional benefit of the knowledge-based editing approach is that it provides a basis for intelligent program modification and maintenance.

This paper describes a prototype Synthesis Apprentice, which we are now beginning to develop. In comparison with the Implementation Apprentice, the Synthesis Apprentice will have increased reasoning abilities and will be able to assist in a greater portion of the programming process. In particular, the Synthesis Apprentice will be able to detect errors and inconsistencies in the programmer's specifications and design decisions, which the Implementation Apprentice cannot do. It will also be able to automatically make many straightforward implementation choices.

We are also beginning to develop a prototype Requirements Apprentice[4] to assist a systems analyst in the creation and modification of software requirements. Research on the Requirements Apprentice establishes a second beachhead from which to attack the problem of automating the programming process. Requirements acquisition is an opportune place for such a beachhead because, like implementation (and unlike the middle parts of the programming process), it is constrained by contact with a fixed boundary. Among other things, this means that the Requirements Apprentice can

be used as a separate module before the entire Programmer's Apprentice is completed. Producing a requirements document with a high degree of confidence in its completeness and consistency is very useful in and of itself.

Our plan for the future is to link up the Requirements Apprentice with the Synthesis Apprentice to provide support for the entire programming process. However, there is currently a significant gap between the proposed capabilities of the two prototypes. This gap corresponds to what is sometimes called high-level system design. Work on the Synthesis Apprentice will focus on the problem of detailed (low-level) design, and yield insight into the nature of the gap between that and requirements.

This paper describes the proposed Synthesis Apprentice via an extended scenario. Our main goal is to give a clear picture of the desired capabilities of the Apprentice. The underlying knowledge representation and reasoning techniques that will be used to achieve these capabilities are described elsewhere[2,3], as is the relationship between our approach and other related work[6,5].

Introduction to the Scenario

The scenario is presented as a sequence of scenes showing specifications written by the programmer, dialog between the programmer and the Apprentice, and programs synthesized by the Apprentice. Before beginning, however, it is important to set the stage by discussing what the Apprentice does and does not know at the beginning of the interaction. The scenario shows the construction of a device driver program. It is assumed that the Apprentice has extensive knowledge about device drivers in general.

Device drivers were chosen as the domain for the scenario for two reasons. First, device drivers are of significant practical importance. Second, they are an example of a kind of domain where the Apprentice can be particularly helpful, namely one in which large numbers of similar programs need to be written, but these programs are not similar enough to allow a program generator approach.

In domains where the set of similar programs is very well understood, it is often possible to write a completely automatic program generator. When this is possible, it has many advantages over using an assistant system. However, the applicability of current program generators is quite limited. The Synthesis Apprentice extends the range of automation to domains in which programs, despite their basic similarity, can differ in unpredictable ways.

The first step in applying the Apprentice to any domain is to develop a comprehensive store of knowledge about the domain. This is an unavoidably costly task. However, if there are many programs which need to be written in the domain, then the benefits of using the Apprentice should amply return the investment in developing the knowledge store.

In the domain of device drivers, significant unpredictability comes both from the idiosyncratic nature of various hardware devices, and from variability in the high-level operations which have to be supported by drivers. A key feature of our approach is that the Apprentice can help with the stereotyped aspects of a program, even if the program also contains significant aspects about which the Apprentice knows little.

Clichés

The Apprentice's knowledge of programming in general, and device drivers in particular, is represented as a library of *clichés*. A cliché is a standard method for doing something. It consists of three parts: *roles*, a *matrix* in which the roles are embedded, and *constraints* on the roles. Roles correspond to parts of the cliché which are expected to change from one use of the cliché to another. The matrix provides the fixed context for the roles — it shows how they fit together. The constraints specify what kind of objects can be used to fill the roles and the required relationships between them.

For example, consider a cliché for the format of a business letter. This cliché would have roles for the sender's address, the recipient's address, the greeting, the body of the letter, etc. The matrix would specify the physical arrangement of these roles on the page and certain fixed aspects of the letter (e.g., that the date should appear after the sender's address). Constraints would specify that whatever fills the recipient's address must be an address, that the person in the greeting should be the same as the person in the recipient's address, etc.

In the Synthesis Apprentice, there are clichés for standard kinds of specifications, standard algorithms, and standard kinds of hardware. These are organized into taxonomic hierarchies, with additional information about how one cliché can be used to implement another.

Device Drivers

The way a device driver is written varies depending on the operating system it is designed to fit into. The driver program in this scenario is being written in the context of the XINU operating system[1]. XINU is a textbook example of an operating system. It is written in C and runs on the PDP/11. The scenario assumes that the target machine has the same basic architecture as the PDP/11 (i.e., memory mapped I/O and 16-bit data paths). However, the driver in this scenario is written in Common Lisp rather than C. This reflects the fact that, although the approach taken by the Synthesis Apprentice is essentially programming-language independent, Common Lisp will be the first target language supported.

In XINU, all device drivers have the standard architecture shown in Figure 2. The top layer in XINU always consists of the nine device-independent I/O functions shown in Figure 3. The bottom layer of the architecture is determined by the physical structure of the given hardware device. The middle two layers are the new driver code that needs to be written for each new type of device. This driver code acts as an interface between

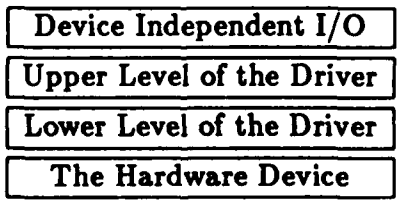


Figure 2: The architecture of a XINU device driver.

the hardware and the device independent functions.

Each of device-independent I/O functions shown in Figure 3 takes as its first argument a device identifier (e.g., (PUTC *device character*)). The device identifier is an index into a device table. Each device table entry contains information about a specific device and pointers to driver functions which support the various I/O operations. Indirecting through the device table makes it possible for one fixed set of functions to support I/O to every kind of device.

The CONTROL function is a catchall which is used to support various idiosyncratic operations—in particular, device specific ones. Its second argument is a code that indicates what operation to perform. For example, (CONTROL *device* :CLEAR) might be used to blank the screen on a terminal.

Each driver is divided into an upper and a lower level in order to simplify the task of writing drivers and the task of verifying that the operating system as whole possesses certain important properties, such as deadlock avoidance. The upper level of the driver contains one function for each device-independent I/O operation supported by the driver. The lower level of the driver contains functions that communicate directly with the hardware device.

The clichés in the Apprentice focus on the lower three levels of Figure 2. To start with, there is an abstract cliché *driver* which contains information which is common to all drivers. In addition to encoding the information presented above, it contains a number of constraints on driver functions.

Neither upper-level nor lower-level driver functions are allowed to have any internal state. All state variables must be in the driver table. (This allows one copy of the driver code to be used for multiple instances of a hardware device.) Interrupts from a hardware device cannot be used to trigger upper-level driver functions. They must all be fielded by lower-level functions. Lower-level functions are not allowed to wait. (This restriction reduces the likelihood that deadlock will occur.) Upper-level functions can wait, but they should not busy-wait. Instead, they should use semaphores and suspend themselves when waiting. (Busy-waiting squanders system resources.) In addition, upper-level functions should not, in general, communicate directly with the hardware. All communication should be routed through lower-level functions.

GETC	Get a character.
PUTC	Put out a character.
READ	Read in a block of characters.
WRITE	Write out a block of characters.
INIT	Initialize the device.
OPEN	Open the device (e.g., connect to a file).
CLOSE	Close the device (e.g., disconnect from a file).
SEEK	Move to a particular position (e.g., on a disk).
CONTROL	Catchall for other operations.

Figure 3: Standard device-independent I/O functions in XINU.

In addition to the abstract cliché *driver*, the Apprentice knows a number of specific clichés for particular kinds of drivers. Consider, for example, the clichés *printer driver* and *interactive display driver*. The printer driver cliché specifies things such as the fact that printer drivers usually only support the operations PUTC and WRITE and that complex output padding is sometimes required after characters which cause large movements of the printing head or platen.

Much of the content of a typical cliché is in the form of links to other clichés which are intended to be used with it. For example, the printer driver cliché is linked to clichés which specify how to buffer up characters before printing them and how to keep track of the line number and page number in the output. Families of specialized clichés exist for dealing with particular classes of printers. For example, there are simple clichés for dealing with printers that do not require special output processing, such as padding. More elaborate clichés specify how to support padding and how to reorder output to take advantage of bidirectional printers.

Much of the following scenario revolves around the interactive display driver cliché. This cliché captures the standard structure of a driver for a terminal with keyboard. It specifies that both input and output is supported and what the typical control operations are. It has links to clichés that specify how to buffer characters on input and output, and how to implement echoing of the input. Families of supporting clichés specify how to implement rubout processing in the input stream, and how to take advantage of direct cursor positioning and screen editing commands (e.g., commands which delete lines and characters) in the output.

The Apprentice also knows a number of hardware clichés, which can be used to describe particular devices. Two examples of this are the clichés *interactive display device* and *serial line unit* (SLU).

An interactive display device is a terminal with keyboard. The cliché includes information such as that the terminal typically has a screen of some fixed height and that I/O with the display is typically carried on in terms of characters in one the standard coding schemes (e.g., ASCII or EBCDIC).

The similarity in name between the clichés interactive display driver and interactive display device reflects the fact that these clichés are closely related. Each driver cliché has links to the hardware clichés for compatible classes of devices.

The SLU cliché specifies the structure of a standard kind of bus interface that is used by many different kinds of PDP/11 hardware devices. For example, it is used by both printers and interactive displays.

The basic structure of an SLU is shown in Figure 4. There are four registers in consecutive words. The address of the first register is used as the address of the SLU as a whole. The *receiver control and status register* (RCSR) contains bits which indicate the state of the receiver side of the SLU and which allow the receiver to be controlled. For example, bit 7 is set to 1 by the SLU whenever a character is received. In addition, bit 6 controls the signalling of interrupts. If bit 6 is set to 1 then an interrupt occurs whenever a character is received. The *transmitter control and status register* (XCSR) has analogous bits referring to the transmission side of the SLU.

When bit 7 of the RCSR is 1, then the *receiver data buffer* (RBUF) contains the char-

Register	High-Order Byte	Low-Order Byte
RCSR	unused	receiver control
RBUF	receiver errors	receiver data
XCSR	unused	transmit control
XBUF	unused	transmit data

Figure 4: The registers in a serial line unit.

acter last received. Reading the RBUF clears bit 7 of the RCSR. The top half of the RBUF contains bits which are set when errors such as bad parity and overrun occur. Bit 15 is set to 1 if any error occurs.

When bit 7 of the XCSR is 1, writing a character in the *transmitter data buffer* (XBUF) triggers its transmission. After a character is written, bit 7 of the XCSR becomes 0 until the transmission is complete. At that time bit 7 is reset to 1 and an interrupt will occur if bit 6 is 1. There are no transmission error bits.

Scene 1: The Initial Driver and Its Specification

In the firs scene, input from the programmer comes in two forms: the specification for a driver and a dialog with the Apprentice which clarifies the specification. The specification is shown in Scenes 1A and 1B. This specification may have been created directly by the programmer without the assistance of the Apprentice. The Apprentice can also help a programmer evolve a specification just as it can help him evolve a program (see Scene 8).

When looking at Scene 1, the reader should bear in mind that the Apprentice does not support interaction in free-form English. The input language has not been designed in detail. However, from a linguistic point of view, it will be very simple. The text in Scene 1 is designed to illustrate the straightforward nature of the input language. Keeping the input language simple is important, because it keeps the work focussed on the problems inherent in programming, without getting bogged down in natural language understanding.

A key feature of the input language is that it makes heavy use of extra-linguistic features and conventions to simplify the syntax of individual sentences and paragraphs. For example, much of the text is in the form of outlines, which use nesting to indicate the structure of the text.

Another feature of the input language is that the vocabulary allowed is quite restricted. Only a few simple verbs are allowed. The only nouns allowed are the names of clichés, roles, and programmer-defined concepts. The definition of a new noun is signaled by using quotes, as illustrated in the first line of Scene 1A. Except for a few acronyms, such as SLU or K7, upper case is used to indicate an identifier from the program space, such as the name of a function or variable.

The specification in Scene 1A consists of two parts. The first part describes an

```
The "K7" is an interactive display device where:
  The screen height is 4 lines.
  The screen width is 40 characters.
  The character format is ASCII.
  Direct cursor positioning is not supported.
  The keyboard has three keys:
      key          character
    ACKNOWLEDGE    ACK  #0006
    YES            Y    #0131
    NO             N    #0116.
  The bus interface is a standard SLU except that:
    Writing a 1 in Bit 1 of the XCSR initializes the device.
      Initializing the device blanks the screen and homes the cursor.
      Completion of initialization is signaled in the same way as the
        transmission of a character.
      Sending characters to the K7 and initializing the K7 cannot be
        done at the same time.

The "engine diagnosis display" is a K7 where:
  The device address is #0177120.
  The receive interrupt address is #0640.
  The transmit interrupt address is #0642.
```

Scene 1A: Specification for the K7 device type, and an instance.

imaginary device called the K7. The first line of the K7 specification names the device and says that it is an instance of the interactive display device cliché. The remaining lines specify how various roles of the interactive display device cliché are filled in. For example, the screen of the K7 is 4 by 40 and the character format is ASCII.

The K7 is a very simple kind of interactive display device. In particular, the screen is very small and there are only three keys that the user can press. The intended use of the K7 is for a message to be displayed on the screen and for the user to provide a simple response.

The K7 specification contains both positive information, e.g., how particular roles are filled in, and negative information. For example, the K7 does not support commands to move the cursor to arbitrary locations. There are other aspects of the interactive display device cliché about which nothing is. For example, the specification does not indicate which output characters are actually displayed on the screen and which are ignored.

In reality, specifications are almost always incomplete. The Apprentice is expected to operate in an environment of incomplete information. To assist with this, each cliché is annotated with information about what parts of the cliché are *mandatory*, *likely*, and only *possible*. For example, it is mandatory that the screen have some height and that some output characters be supported. Therefore, even though the programmer has said nothing about them, the Apprentice can depend on the fact that there are some output characters which are supported. In contrast, it is only likely that an interactive display device will have some cursor positioning command, but not mandatory. In this case, the Apprentice makes no assumptions at all unless the programmer says something. Finally, it possible that an interactive display will have commands for editing the screen (e.g., deleting characters and lines), but not likely. The Apprentice assumes that features which are only possible are not present unless the programmer says that they are.

The features of a cliché typically depend on each other in a number of ways. As a result, making specific statements about the presence or absence of a few of the features usually makes it possible to infer which other features are present. For example, interactive displays which do not support cursor positioning do not support screen editing.

The first nine lines of the K7 specification describe various external features of the device. The last five lines specify how the device communicates. This description relies heavily on the SLU cliché . The phrase "except that" is used to describe the K7 by modifying the SLU cliché rather than merely filling in roles. The exception describes a novel feature which is not present in standard SLUs. The exception makes use of a number of terms (i.e., "XCSR", "initialize the device", "blank the screen", "home the cursor", "completing an operation", etc.) which are defined in the context of hardware interfaces in general and the cliché SLU in particular. The availability of this rich vocabulary makes it possible for the programmer to describe succinctly what happens when a 1 is written in bit 1 of the XCSR.

The second part of Scene 1A describes the "engine diagnosis" display. This display is a particular K7 which has its device and interrupt addresses jumpered to the addresses

shown. The scenario assumes that this is the specific device the programmer wants a driver for. (The expression "#0177120" is the standard Common Lisp construct for specifying an octal number.)

Scene 1B is the specification for a driver for the K7. This specification relies heavily on the interactive display driver cliché . As with the K7 specification, most of the driver specification says how various roles of the cliché are filled in. The first two lines indicate that, in keeping with the simplicity of the K7 itself, only a simple driver is desired.

The next segment of the specification indicates which device-independent I/O functions should be supported. Each of these functions is so stereotyped that nothing needs to be said about it other than whether it should be supported or not. The term "ignored" means that calling the specified functions should do nothing, but not cause an error. In contrast, calling an unsupported function causes an error.

The various control codes to be supported are specified in greater detail, because they are more idiosyncratic in nature. Each control code is followed by a brief specification. For example, calling CONTROL with the code :LINE-NUMBER returns the current cursor line (i.e., the number of the line the cursor is on). The cliché interactive display driver contains information about how to support various common control codes. However, there is no limit to the kind of operations which a programmer might desire.

A particularly interesting part of the driver specification is the implementation guidelines section at the end. The Apprentice uses these guidelines to decide which specific algorithms to pick when implementing the driver. The first guideline states that the algorithms chosen must not do dynamic storage allocation. The second guideline states that the programmer considers storage efficiency to be more important than time efficiency.

The third guideline instructs the Apprentice that consideration of error checking is to be deferred until a later time. This makes it possible for the programmer to produce a testable driver quickly, without worrying about specifying the program's behavior under exceptional conditions. As discussed in conjunction with Scene 7, this is an example of a way in which the Apprentice supports rapid prototyping.

There are a variety of other guidelines the programmer could have selected. For

```
The K7 driver is an interactive display driver where:
  Echoing is not supported
  Rubout handling is not supported.
  The basic software functions are:
    Supported: PUTC, WRITE & GETC.
    Ignored: OPEN & CLOSE.
    Not supported: READ & SEEK.
  The control codes are:
    :CLEAR - blanks the screen.
    :RESET - does a :CLEAR and flushes pending input.
    :LINE-NUMBER - returns the cursor line.
  The implementation guidelines are:
    1- Do no consing.
    2- Favor space efficiency.
    3- Defer error checking.
```

Scene 1B: Specification for the K7 driver.

example, he could have told the Apprentice to favor time efficiency. Alternately, he could have told the Apprentice to use only simple, straightforward algorithms. By making the programs produced easier to understand, this guideline facilitates both rapid prototyping and future modification.

Note that the specifications in Scene 1 are relatively detailed. They deal quite specifically with what needs to be implemented, rather than stating criteria to be satisfied. For example, it is no accident that the specification for :CLEAR uses the imperative phrase "blank the screen" rather than providing a non-constructive specification such as "cause there to be no characters shown on the screen". By referring to the interactive display driver cliché , the specification indicates indirectly (through the connections in the knowledge base) what families of algorithms should be used. In addition, except for the implementation guidelines (which are drawn from a small set of fixed choices), there are no global constraints in the specification; everything is relatively local in nature.

The detailed nature of the specification in Scene 1 makes it much easier for the Apprentice to create a program corresponding to the specification. However, it also illustrates the fact that a "design apprentice" will be required to bridge the gap between the output of the Requirements Apprentice and the input of the Synthesis Apprentice.

Having completed the specification, the programmer now asks the Apprentice to write code for the driver. The dialog which ensues is shown in Scene 1C. (The input typed by the Programmer is shown in a slanted font.)

```
>Write code for the K7 driver and a device table entry for the engine diagnosis display.

Request for Advice: I do not know how to support :CLEAR.
Note:
  (1) :CLEAR blanks the screen.
  (2) The only K7 operation that blanks the screen is initialization.
Unfortunately:
  (3) Initializing the K7 homes up the cursor.
  (4) A device should not be initialized except by the init function.

>:CLEAR should home the cursor.
 Support :CLEAR by initializing the device.

Request for Advice: I do not know how to support :LINE-NUMBER.
Note:
 (1) :LINE-NUMBER returns the cursor line.
 (2) No K7 operation returns the cursor line.

>Postpone.

Constraint Violation: There should be a lookahead operation.
Note: The following are the standard types of lookahead operation:
  (1) :ARE-CHARS-WAITING returns T if there are pending input characters.
  (2) :NUMBER-CHARS-WAITING returns the number of pending input characters.
  (3) :PEEK returns NIL if there are no pending input characters and
      returns the next character (without removing it from the input
      buffer) if a character is pending.

>Use (2).
```

Scene 1C: Dialog between the Apprentice and the Programmer.

The programmer's first command tells the Apprentice to implement the K7 driver and create a device table entry for the engine diagnosis display. Before creating any code, the Apprentice analyzes the specification. It then asks a series of questions to clarify various aspects of the specification. These questions fall into three categories: requests to fill in essential parts of the specification which have been left unspecified, requests for advice on how to implement parts of the specification, and requests for the programmer to fix inconsistencies in the specification. Examples of each type of request are found in Scene 1C.

The first request concerns :CLEAR. The Apprentice notes that there are difficulties with implementing this operation. It has determined that, since the K7 does not support cursor positioning or screen editing, the only plausible way to blank the screen is to initialize the device. However, there are two problems with this solution.

First, the specification for what happens when the K7 is initialized does not match what the programmer said the clear operation should do. This problem stems from an error in the specification. The user resolves it by changing the specification.

This problem may seem somewhat artificial. Assumedly, in addition to the terms "blank the screen" and "home the cursor", the Apprentice understands that the term "clear the screen" means blank the screen and home the cursor. If the programmer had used the term "clear the screen" when describing both initialization of the K7 and the clear operation, then this problem would not have arisen. However, this problem is a good example of the kind of small difficulty which comes up all the time in programming, no matter how careful a programmer is.

The second problem with implementing :CLEAR by initializing the device is that it violates a general constraint on the use of the interactive display device cliché: Since initialization typically resets every piece of state in a device, it should only be used when starting up the device. Here, however, the programmer decides that this general constraint is not applicable. The ability to override constraints is an important feature of the Apprentice.

The second request in Scene 1C illustrates how the Apprentice responds in situations where it has no idea of what to do. The problem here concerns :LINE-NUMBER. Since the K7 device does not have a hardware command to determine the current cursor line, the Apprentice has no idea how to implement the line number operation.

In response to the Apprentice's request, the programmer tells the system to postpone worrying about the line number operation. The ability to defer questions posed by the Apprentice is essential in order for the programmer to maintain control over the interaction. The Apprentice keeps track of the fact that the question has been deferred and inserts a stub corresponding to the line number operation.

The third request in Scene 1C illustrates how constraints attached to a cliché can be used to check the reasonableness of a specification. A constraint in the interactive display driver cliché says that if any input functions are supported, then there should be some control operation which allows a user of the driver to determine whether or not there are any characters waiting to be read. The Apprentice notes that this constraint is violated and proposes three ways in which the specification could be fixed. These alternatives are generated based on the list of common control operations which are

also part of the interactive display driver cliché.

The programmer decides to follow one of the Apprentice's recommendations. He could, however, have specified that the constraint does not apply to this particular driver and that there is therefore no lookahead operation. He could alternatively have specified some idiosyncratic operation which satisfies the constraint.

In general, when the Apprentice finds a problem, it initially reports it with relatively little explanation. For example, the third request in Scene 1C states that there should be a lookahead operation without saying why. (The Apprentice does suggest a few potential solutions.) If the programmer requests an explanation, however, the Apprentice can provide one. Scene 1D shows the output which would be produced if the programmer requested an explanation immediately after seeing the third request in Scene 1C.

```
>Explain.

(4) If any input operations are supported, then a lookahead operation
    should be supported.
(5) GETC is an input operation.
(6) There is no lookahead operation.

>Explain (4).

(4) is a constraint of interactive display driver cliche, which is
inherited from the driver cliche.
```

Scene 1D: Example of explanation.

After the requests in Scene 1C have been resolved, the Apprentice creates code for the K7 driver. This code is shown in Scene 1E. For the most part, the code is standard Common Lisp. However, a few special functions are used. These functions will be described as they come up.

The code is divided into four segments. The first segment of the code defines the device control block (DCB) for the K7. The function `CREATE-K7` allocates a K7 DCB and fills it in with appropriate data. The DCB contains the address of the device and a number of fields which are used when communicating with the device. `IDLE-SEM` is a semaphore that specifies whether or not the device is idle and ready to accept output. `IN-BUFFER` is a queue which is used to store characters that come from the device. `IN-SEM` is a semaphore that keeps track of the number of characters in the input buffer. `OUT-BUFFER` is a queue that is used to store characters which are to be sent to the device. `OUT-SEM` keeps track of the number of empty slots in the output buffer. (The reason for the asymmetry between `OUT-SEM` and `IN-SEM` is discussed below.) `SENDING?` specifies whether or not the driver is in the middle of sending a group of characters to the device. The fields `WATERMARK` and `PENDING` are used to support watermark processing.

(Watermark processing increases the efficiency of the output functions. Suppose that the user writes a string of characters which is longer than the size of the output buffer. Since characters can only be transmitted to the device slowly, the output buffer will eventually fill up and the `WRITE` function will have to wait. From that point on, each time a character is sent, `WRITE` will be reactivated and put another character

```
;CONTROL BLOCK DEFINITION FOR K7

(defconstant K7-init #0002)
(defstruct K7
  addr idle-sem in-buffer in-sem out-buffer out-sem
  sending? watermark pending)

(defun create-K7 (addr)
  (make-K7 :addr        addr
           :idle-sem    (create-sem 1)
           :in-buffer   (create-fast-q 10)
           :in-sem      (create-sem 0)
           :out-buffer  (create-fast-q 20)
           :out-sem     (create-sem 20)
           :sending?    nil
           :watermark   8
           :pending     0))

(defun K7-init (dcb)
  (flush-fast-q (K7-in-buffer dcb))
  (setf (sem-count (K7-in-sem dcb)) 0)
  (flush-fast-q (K7-out-buffer dcb))
  (setf (sem-count (K7-out-sem dcb)) (fast-q-size (K7-out-buffer dcb)))
  (setf (K7-sending? dcb) nil)
  (setf (K7-pending dcb) 0)
  (%store CSR-int-disable (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store K7-init (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (setf (sem-count (K7-idle-sem dcb)) 1)
  (%store CSR-int-enable (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (RCSR (K7-addr dcb))))

;DRIVER TABLE FOR THE ENGINE DIAGNOSIS DISPLAY

(defconstant *engine-diagnosis-display* 14)

(setf (get-device *engine-diagnosis-display*)
      (make-device
        :in-int-addr  #0640          :out-int-addr  #0642
        :in-int-fn    #'K7-receive   :out-int-fn    #'K7-done
        :init         #'K7-init      :dcb           (create-K7 #0177120)
        :open         #'io-noop      :close         #'io-noop
        :putc         #'K7-putc      :write         #'K7-write
        :getc         #'K7-getc      :control       #'K7-control))
```

Scene 1E: Initial code created by the Apprentice (part 1).

```
;UPPER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-control (dcb code &rest ignore)
  (without-interrupts
    (case code
      (:CLEAR (flush-fast-q (K7-out-buffer dcb))
              (setf (sem-count (K7-out-sem dcb))
                    (fast-q-size (K7-out-buffer dcb)))
              (sem-wait (K7-idle-sem dcb))
              (K7-clear dcb))
      (:RESET (flush-fast-q (K7-in-buffer dcb))
              (setf (sem-count (K7-in-sem dcb)) 0)
              (K7-control dcb :CLEAR))
      (:LINE-NUMBER {not-yet-implemented})
      (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb))))))

(defun K7-getc (dcb)
  (without-interrupts
    (sem-wait (K7-in-sem dcb))
    (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-write (dcb buffer length)
  (dotimes (pointer length)
    (K7-putc dcb (aref buffer pointer))))

(defun K7-putc (dcb char)
  (without-interrupts
    (if (not (K7-sending? dcb)) (sem-wait (K7-idle-sem dcb)))
    (sem-wait (K7-out-sem dcb))
    (fast-enq (K7-out-buffer dcb) (char-code char))
    (when (not (K7-sending? dcb))
      (setf (K7-sending? dcb) T)
      (K7-send dcb))))

;LOWER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-receive (dcb)
  (without-interrupts
    (fast-enq (K7-in-buffer dcb) (%get (RBUF (K7-addr dcb))))
    (sem-signal (K7-in-sem dcb))))

(defun K7-done (dcb)
  (without-interrupts
    (if (K7-sending? dcb) (K7-send dcb))
    (if (not (K7-sending? dcb)) (sem-signal (K7-idle-sem dcb)))))

(defun K7-send (dcb)
  (let ((count (sem-count (K7-out-sem dcb))))
    (cond ((= count (fast-q-size (K7-out-buffer dcb)))
           (setf (K7-sending? dcb) nil))
          (T (%store (fast-deq (K7-out-buffer dcb)) (XBUF (K7-addr dcb)))
             (cond ((> count (K7-watermark dcb))
                    (sem-signal (K7-out-sem dcb)))
                   ((= (incf (K7-pending dcb)) (K7-watermark dcb))
                    (setf (K7-pending dcb) 0)
                    (sem-signal (K7-out-sem dcb) (K7-watermark dcb)))))))))

(defun K7-clear (dcb)
  (%store (logand CSR-int-enable K7-init) (XCSR (K7-addr dcb))))
```

Scene 1E: Initial code created by the Apprentice (part 2).

in the output buffer. Operating on one character at a time in this fashion leads to a considerable amount of scheduling overhead per character. Watermark processing changes things so that no characters are entered into the output buffer unless a certain watermark is reached in which case at least WATERMARK characters can be entered at once.)

In a production environment, one would expect to see comments in the code containing information such as that discussed above. The Apprentice will be capable of generating such comments based on the clichés used to construct a program. However, to keep the figures in this scenario manageably small, such comments are not shown.

The function K7-INIT is used to initialize a DCB and the associated device at system startup. The first six lines initialize fields in the DCB. The next four lines initialize the K7 associated with the DCB. They turn off the K7 transmit interrupt, busy-wait until the device is idle, set the initialization bit, and then busy-wait until the initialization is complete. The last three lines of K7-INIT initialize IDLE-SEM and turn on the K7 receive and transmit interrupts.

Note that the functions CREATE-K7 and K7-INIT call a number of special Lisp functions which are assumed to be part of the operating system hosting the driver. The function (CREATE-SEM *initial-value*) creates a semaphore. The form (SEM-COUNT *semaphore*) returns the value stored in a semaphore. If the form is used as the destination of a SETF, it sets the value of the semaphore triggering pending waits if necessary. Both of these functions are part of a package of functions which support standard semaphores. This package also includes the function (SEM-WAIT *semaphore*) and (SEM-SIGNAL *semaphore* {*amount*}). SEM-WAIT suspends itself until the indicated semaphore has a positive value and then decrements the semaphore. SEM-SIGNAL increments the indicated semaphore by an amount which defaults to 1, possibly reawakening one or more SEM-WAITs.

The functions (CREATE-FAST-Q *size*) and (FLUSH-FAST-Q *queue*) are part of a package of functions which operate on queues of characters. These functions are discussed in conjunction with Scene 1F below.

The macro (XCSR *address*) takes the address of an SLU and returns the address of its XCSR register. Analogous macros are provided for accessing the other SLU registers. In addition special constants such as CSR-INT-ENABLE and CSR-INT-DISABLE are defined which allot bits in the various SLU registers to be set and cleared without using undocumented literal constants in the code. It is assumed in the scenario that these macros and constants already exist, since other device drivers access SLUs. An additional constant K7-INIT is defined which corresponds to the initialization bit of the XCSR for the K7.

The function (%STORE *value address*) stores the value into the memory location specified by the address. The companion function (%GET *address*) gets the value from the indicated memory location. The form (BUSY-WAIT-NOT *mask address*) repeatedly reads the indicated memory location and busy-waits until all the bits corresponding to the bits in the are zero.

The second segment of code in Scene 1E is a device table entry for the engine diagnosis display. A constant *ENGINE-DIAGNOSIS-DISPLAY* is defined which holds the

index of the entry. An entry is then created and stored in this position. (The macro (GET-DEVICE *device*) is used to access device table entries.)

The device entry contains a DCB for the engine diagnosis display. The DCB is created using the function CREATE-K7 described above. The device entry hold the interrupt addresses for the device and the functions to be triggered when interrupts occur. At system startup, these functions are installed in the interrupt handling routine. The remainder of the device entry contains pointers to the appropriate device dependent I/O functions. The function IO-NOOP is used to implement device independent functions which are ignored.

For example, the following function call would be used to clear the engine diagnosis display.

```
(control *engine-diagnosis-display* :clear)
```

As shown below, the function CONTROL indirects through the device table to obtain the appropriate DCB and device specific control function.

```
(defun control (device code arg)
  (let ((entry (get-device *engine-diagnosis-display*)))
    (funcall (device-control entry) (device-dcb entry) code arg)))
```

Part 2 of Scene 1E shows the code the Apprentice has written for the upper and lower levels of the K7 driver. The first upper-level function, K7-CONTROL, supports the device independent function CONTROL. To support :CLEAR, K7-CONTROL first flushes any pending output characters from the output buffer. (It would be a waste of time to send characters to the device immediately before blanking the screen, because they would not be in view long enough for the user to see.)

After flushing the output buffer, the code supporting :CLEAR waits to make sure that the device is in a state where it can process an initialization request. The code then calls the lower-level driver function K7-CLEAR to initialize the device.

To support the reset operation, K7-CONTROL flushes the input buffer and does a clear. The form "{NOT-YET-IMPLEMENTED}" is used to indicate that :LINE-NUMBER has not yet been implemented. :NUMBER-CHARS-WAITING reports the number of characters in the input buffer.

The function K7-GETC waits until there is an input character available and then returns it. (Waiting with interrupts disabled does not cause deadlock because the wait suspends the entire process. Other processes then run which can get interrupted.) The function K7-WRITE (see Scene 1E) writes out a group of characters by repetitively calling K7-PUTC.

The function K7-PUTC enters a character into the output buffer. It operates in one of two modes. If SENDING? is T then it merely adds the new character into the buffer. To do this, it uses the semaphore OUT-SEM to make sure that there is space for the character in the buffer.

However, if SENDING? is NIL then K7-PUTC needs to take control of the device before starting to output characters. It does this by waiting on IDLE-SEM. In addition, after

putting the character into the buffer, K7-PUTC sets SENDING? to T and calls K7-SEND to get the actual transmission going. After the transmission of characters starts, K7-SEND is called repeatedly by K7-DONE until the output buffer is empty.

It is important not to be misled by the apparent complexity of the algorithm used by K7-PUTC and the way it interacts with the other functions in the driver. Viewed from first principles, it is a complex algorithm. However, the Apprentice does not have to construct it from first principles. It is a combination of two quite standard pieces. The wait on IDLE-SEM is used to arbitrate between K7-PUTC and :CLEAR.

The rest of the code in K7-PUTC is a standard method for sending batches of characters to a device—it is part of the cliché interactive display driver. This code is somewhat convoluted because of the restriction that lower-level driver functions cannot wait. If K7-SEND could wait until a character was ready to send, then the code in K7-PUTC would be as simple as the code in K7-RECEIVE. Instead, K7-PUTC waits until there is an empty slot in the output buffer. The processing centered around the DCB field SENDING? is then needed because K7-PUTC has to call K7-SEND to initiate transmission when it is not already underway.

The second half of the driver code in Scene 1E is composed of the lower-level driver functions. The function K7-RECEIVE is called whenever the K7 signals a receive interrupt indicating that a new character is available. K7-RECEIVE moves the character to the input buffer. As per the programmer's request, no error checking is performed.

The function K7-DONE is called whenever the K7 signals a transmit interrupt. If SENDING? is T it calls K7-SEND. Otherwise, it resets IDLE-SEM.

The function K7-SEND sends characters to the K7. It assumes that it will only be called when the K7 is idle. If there are no characters to send, it sets SENDING? to NIL (after which K7-DONE signals IDLE-SEM). If there is a character to send, K7-SEND puts it in XBUF and updates OUT-SEM. The task of updating OUT-SEM is somewhat complicated due to the watermark processing.

The function K7-CLEAR clears the K7 by setting the initialization bit. It assumes that it will only be called when the K7 is idle. The WAIT has to be in K7-CONTROL because lower-level driver functions cannot wait.

The code in Scene 1E makes extensive use of a group of functions which operate on queues. These functions are shown in Scene 1F. If necessary, the Apprentice would write code for these functions as part of coding up the K7 driver. However, the queue functions might already exist for some other reason as part of the operating system.

Several features of the queueing functions are worthy of note. In accordance with the implementation guidelines in Scene 1B, they do no consing (except when queues are created at system startup). Instead, they operate on queues of fixed maximum sizes. They also do not support a full range of operations. In particular, it is not possible to reliably determine how many characters are stored in a queue because it is not possible to discriminate between an empty queue and a full queue. (In the driver code, semaphores are used to record how many characters are in the queues.) A final specialized aspect of the queue functions is that they do no error checking. The dequeue function assumes that it will never be called on an empty queue and the enqueue function assumes that it will never be called on a full one.

```
(defstruct FQ head tail buffer)

(defun create-fast-q (size)
  (make-FQ :head 0 :tail 0 :buffer (make-array size)))

(defun fast-q-size (queue)
  (length (FQ-buffer queue)))

(defun flush-fast-q (queue)
  (setf (FQ-head queue) (FQ-tail queue)))

(defun fast-deq (queue)
  (let ((item (aref (FQ-buffer queue) (FQ-head queue))))
    (if (minusp (decf (FQ-head queue)))
        (setf (FQ-head queue) (1- (length (FQ-buffer queue)))))
    item))

(defun fast-enq (queue item)
  (setf (aref (FQ-buffer queue) (FQ-tail queue)) item)
  (if (minusp (decf (FQ-tail queue)))
      (setf (FQ-tail queue) (1- (length (FQ-buffer queue))))))
```

Scene 1F: Functions supporting fast queues.

These features make the queueing functions simpler and more efficient at the cost of a loss of generality. The Apprentice is able to derive this specialized queue data abstraction based on the way queues are used in the driver code. This is a good example of the classic automatic programming problem of data structure selection.

Scene 2: Changing a Design Decision

Once the Apprentice produces the driver code, the programmer looks it over. He may test parts of it and/or ask the Apprentice questions about parts of it. Scene 2A shows the programmer asking about the watermark processing.

```
>Why are you using watermark processing?

Because:
  (1) Watermark processing should be used unless the output buffer is large.
  (2) The output buffer is small.

>Why (2) ?

Because:
  (2.1) The specification states that space efficiency should be favored.

>Make the size of the output buffer equal to the total screen size.

Watermark processing is no longer advisable.  I have removed it.
```

Scene 2A: The programmer asks to have watermark processing removed.

The programmer asks why the Apprentice chose to use watermark processing. The Apprentice responds by giving the reasoning behind its decision. The response is a formatted printout of the dependencies which form a record of its reasoning.

Note that the Apprentice's response in Scene 2A does not include a tutorial description of the watermark algorithm. (It assumes that the programmer already knows what watermark processing is.) In general, work on the Apprentice will focus on supporting explanation at the level of design decisions, rather than deep understanding of algorithms. The Apprentice should have a lot of knowledge about when various algorithms should be used, but it is not intended to be an aid to algorithm design.

After considering the Apprentice's reasons for using watermark processing, the programmer decides that it is a false economy in this situation. When using the K7, one never outputs more than the number of characters on the screen without pausing to wait for input. Since the screen is quite small, there is a relatively small bound (only 160) on the number of characters which will ever have to be buffered. (Allocating a buffer of this size probably does not take more space then is occupied by the watermark processing code.)

The programmer tells the Apprentice to increase the size of the output buffer. (The cliché interactive display device defines the term "total screen size" to be the product of the screen height and width.)

After changing the output buffer size, the Apprentice checks all of its design decisions in light of the change. It notices that watermark processing is no longer desirable because the buffer is large. (The interactive display driver cliché contains a statement that an output buffer which is as large as the whole screen is a large one.)

Thus the programmer's command to change the buffer size has the effect of getting rid of the watermark processing. The programmer could have done this directly by simply telling the system to stop using watermark processing. However, this would be

```
;CONTROL BLOCK DEFINITION FOR K7

(defconstant K7-init #0002)
(defconstant K7-height 4)
(defconstant K7-width 40)
(defstruct K7
  addr idle-sem in-buffer in-sem out-buffer out-sem sending?)

(defun create-K7 (addr)
  (let ((out-size (* K7-height K7-width)))
    (make-K7 :addr        addr
             :idle-sem    (create-sem 1)
             :in-buffer   (create-fast-q 10)
             :in-sem      (create-sem 0)
             :out-buffer  (create-fast-q out-size)
             :out-sem     (create-sem out-size))
             :sending?    nil))

(defun K7-init (dcb)
  (flush-fast-q (K7-in-buffer dcb))
  (setf (sem-count (K7-in-sem dcb)) 0)
  (flush-fast-q (K7-out-buffer dcb))
  (setf (sem-count (K7-out-sem dcb)) (fast-q-size (K7-out-buffer dcb)))
  (setf (K7-sending? dcb) nil)
  (%store CSR-int-disable (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store K7-init (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (setf (sem-count (K7-idle-sem dcb)) 1)
  (%store CSR-int-enable (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (RCSR (K7-addr dcb))))

;DRIVER TABLE FOR THE ENGINE DIAGNOSIS DISPLAY

(defconstant *engine-diagnosis-display* 14)

(setf (get-device *engine-diagnosis-display*)
      (make-device
        :in-int-addr  #0640          :out-int-addr  #0642
        :in-int-fn    #'K7-receive   :out-int-fn    #'K7-done
        :init         #'K7-init      :dcb           (create-K7 #0177120)
        :open         #'io-noop      :close         #'io-noop
        :putc         #'K7-putc      :write         #'K7-write
        :getc         #'K7-getc      :control       #'K7-control))
```

Scene 2B: Code with watermark processing removed (part 1).

```
;UPPER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-control (dcb code &rest ignore)
  (without-interrupts
    (case code
      (:CLEAR (flush-fast-q (K7-out-buffer dcb))
              (setf (sem-count (K7-out-sem dcb))
                    (fast-q-size (K7-out-buffer dcb)))
              (sem-wait (K7-idle-sem dcb))
              (K7-clear dcb))
      (:RESET (flush-fast-q (K7-in-buffer dcb))
              (setf (sem-count (K7-in-sem dcb)) 0)
              (K7-control dcb :CLEAR))
      (:LINE-NUMBER {not-yet-implemented})
      (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb))))))

(defun K7-getc (dcb)
  (without-interrupts
    (sem-wait (K7-in-sem dcb))
    (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-write (dcb buffer length)
  (dotimes (pointer length)
    (K7-putc dcb (aref buffer pointer))))

(defun K7-putc (dcb char)
  (without-interrupts
    (if (not (K7-sending? dcb)) (sem-wait (K7-idle-sem dcb)))
    (sem-wait (K7-out-sem dcb))
    (fast-enq (K7-out-buffer dcb) (char-code char))
    (when (not (K7-sending? dcb))
      (setf (K7-sending? dcb) T)
      (K7-send dcb))))

;LOWER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-receive (dcb)
  (without-interrupts
    (fast-enq (K7-in-buffer dcb) (%get (RBUF (K7-addr dcb))))
    (sem-signal (K7-in-sem dcb))))

(defun K7-done (dcb)
  (without-interrupts
    (if (K7-sending? dcb) (K7-send dcb))
    (if (not (K7-sending? dcb)) (sem-signal (K7-idle-sem dcb)))))

(defun K7-send (dcb)
  (cond ((= (sem-count (K7-out-sem dcb))
            (fast-q-size (K7-out-buffer dcb)))
         (setf (K7-sending? dcb) nil))
        (T (%store (fast-deq (K7-out-buffer dcb)) (XBUF (K7-addr dcb)))
           (sem-signal (K7-out-sem dcb)))))

(defun K7-clear (dcb)
  (%store (logand CSR-int-enable K7-init) (XCSR (K7-addr dcb))))
```

Scene 2C: Code with watermark processing removed (part 2).

unwise if the output buffer size were not increased. More importantly, the illustrated sequence of events makes it possible for the Apprentice to be more helpful if a related change is made in the future. For example, the Apprentice will reintroduce watermark processing if something comes up which indicates that the output buffer is, in reality, not large.

Scene 2B shows the code which results after the changes requested by the programmer. Change bars ("|") in the left margin indicate which lines of code are different from the code in Scene 1.

The DCB fields `WATERMARK` and `PENDING` have been removed, which changes the functions `CREATE-K7` and `K7-INIT`. The function `K7-SEND` is modified by removing the code which does the watermark processing.

The function `CREATE-K7` is also changed to increase the size of the output buffer. The constants `K7-HEIGHT` and `K7-WIDTH` are introduced so that `CREATE-K7` will not contain the integer 160 as a mystery constant. This is done on the theory that constants which are derived from something concrete like the screen size should reflect this fact in the way they are coded. The code uses two constants and an expression (which the compiler computes at compile time) because total screen size is a derived concept.

Scene 3: Making an Orthogonal Addition

At this point in the scenario, the programmer is testing the K7 driver. This experimentation is oriented toward testing the specification rather than testing the code.

To a considerable extent, the Apprentice guarantees that the code corresponds to the specification. There are only two main exceptions to this. First, there may be parts of the specification which the Apprentice cannot understand and does not produce any code for (e.g., the line number operation). Second, there may be complex logical consequences of the specification which the Apprentice's reasoning abilities are not powerful enough to detect.

In contrast to the code, the specification is almost certainly not correct. For example, it is incomplete in many ways (it does not say anything about what output characters are supported by the K7). Further, experimentation may well reveal that some additional (or different) operations should be supported.

During the testing, the programmer decides that it would be useful to keep a trace of the situations in which the driver sends information to the K7. In Scene 3A, the programmer asks the Apprentice to insert tracing code. An interesting aspect of this command is that it describes what should be done somewhat indirectly. Instead of saying exactly where tracing should be inserted, the command describes the logical properties of the place(s) where tracing should be inserted. Instead of saying exactly what information to record, the command merely says that the "state of the driver" should be saved.

>*Collect a trace of the state of the driver at each point where information is sent to the hardware device.*

Scene 3A: The programmer asks to have tracing added.

The tracing instrumentation added by the Apprentice is shown in Scene 3B. Code is added in three places: right after `K7-INIT` sets the initialization bit, right after `K7-CLEAR` sets the initialization bit, and right after `K7-SEND` writes a character in the transmission buffer. Each section of instrumentation code pushes a string identifying the action being performed, the time, and a copy of the DCB onto a list of debugging information. The function `COPY-ALL` is part of a package of instrumentation functions. It copies a structure and all of its substructures. This is necessary so that subsequent execution of the driver will not alter stored debugging information.

Instrumentation is a domain of expertise which is orthogonal to the driver domain. The fact that knowledge about instrumentation is separate from knowledge about drivers makes it possible for the Apprentice to assist in the instrumentation of many kinds of programs. In addition, instrumentation provides a second domain in which to investigate the kind of facilities and reasoning capabilities which are needed to assist a programmer.

```
;CONTROL BLOCK DEFINITION FOR K7

(defconstant K7-init #0002)
(defconstant K7-height 4)
(defconstant K7-width 40)
(defstruct K7
  addr idle-sem in-buffer in-sem out-buffer out-sem sending?)

(defun create-K7 (addr)
  (let ((out-size (* K7-height K7-width)))
    (make-K7 :addr       addr
             :idle-sem   (create-sem 1)
             :in-buffer  (create-fast-q 10)
             :in-sem     (create-sem 0)
             :out-buffer (create-fast-q out-size)
             :out-sem    (create-sem out-size))
             :sending?   nil))

(defun K7-init (dcb)
  (flush-fast-q (K7-in-buffer dcb))
  (setf (sem-count (K7-in-sem dcb)) 0)
  (flush-fast-q (K7-out-buffer dcb))
  (setf (sem-count (K7-out-sem dcb)) (fast-q-size (K7-out-buffer dcb)))
  (setf (K7-sending? dcb) nil)
  (%store CSR-int-disable (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store K7-init (XCSR (K7-addr dcb)))
  (push (list :action "K7-init set initialization bit"
              :time (get-universal-time)
              :dcb (copy-all dcb))
        *debugging-info*)                          ;TEMPORARY INSTRUMENTATION
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (setf (sem-count (K7-idle-sem dcb)) 1)
  (%store CSR-int-enable (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (RCSR (K7-addr dcb))))

;DRIVER TABLE FOR THE ENGINE DIAGNOSIS DISPLAY

(defconstant *engine-diagnosis-display* 14)

(setf (get-device *engine-diagnosis-display*)
      (make-device
        :in-int-addr  #0640          :out-int-addr  #0642
        :in-int-fn    #'K7-receive   :out-int-fn    #'K7-done
        :init         #'K7-init      :dcb           (create-K7 #0177120)
        :open         #'io-noop      :close         #'io-noop
        :putc         #'K7-putc      :write         #'K7-write
        :getc         #'K7-getc      :control       #'K7-control))
```

Scene 3B: Code with tracing added (part 1).

```
;UPPER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-control (dcb code &rest ignore)
  (without-interrupts
    (case code
      (:CLEAR (flush-fast-q (K7-out-buffer dcb))
              (setf (sem-count (K7-out-sem dcb))
                    (fast-q-size (K7-out-buffer dcb)))
              (sem-wait (K7-idle-sem dcb))
              (K7-clear dcb))
      (:RESET (flush-fast-q (K7-in-buffer dcb))
              (setf (sem-count (K7-in-sem dcb)) 0)
              (K7-control dcb :CLEAR))
      (:LINE-NUMBER {not-yet-implemented})
      (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb))))))

(defun K7-getc (dcb)
  (without-interrupts
    (sem-wait (K7-in-sem dcb))
    (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-write (dcb buffer length)
  (dotimes (pointer length)
    (K7-putc dcb (aref buffer pointer))))

(defun K7-putc (dcb char)
  (without-interrupts
    (if (not (K7-sending? dcb)) (sem-wait (K7-idle-sem dcb)))
    (sem-wait (K7-out-sem dcb))
    (fast-enq (K7-out-buffer dcb) (char-code char))
    (when (not (K7-sending? dcb))
      (setf (K7-sending? dcb) T)
      (K7-send dcb))))

;LOWER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-receive (dcb)
  (without-interrupts
    (fast-enq (K7-in-buffer dcb) (%get (RBUF (K7-addr dcb))))
    (sem-signal (K7-in-sem dcb))))

(defun K7-done (dcb)
  (without-interrupts
    (if (K7-sending? dcb) (K7-send dcb))
    (if (not (K7-sending? dcb)) (sem-signal (K7-idle-sem dcb)))))

(defun K7-send (dcb)
  (cond ((= (sem-count (K7-out-sem dcb))
            (fast-q-size (K7-out-buffer dcb)))
         (setf (K7-sending? dcb) nil))
        (T (%store (fast-deq (K7-out-buffer dcb)) (XBUF (K7-addr dcb)))
           (push (list :action "K7-send sent char"
                       :time (get-universal-time)
                       :dcb (copy-all dcb))
                 *debugging-info*)                    ;TEMPORARY INSTRUMENTATION
           (sem-signal (K7-out-sem dcb)))))

(defun K7-clear (dcb)
  (%store (logand CSR-int-enable K7-init) (XCSR (K7-addr dcb)))
  (push (list :action "K7-clear set initialization bit"
              :time (get-universal-time)
              :dcb (copy-all dcb))
        *debugging-info*))                            ;TEMPORARY INSTRUMENTATION
```

Scene 3B: Code with tracing added (part 2).

Scene 4: Changing a Major Design Decision

In this scene, the programmer makes a fundamental change in the way the driver functions interact. Looking at the code, the programmer notices that a considerable amount of complexity revolves around IDLE-SEM. At the start of Scene 4, the programmer asks the Apprentice to explain why this semaphore is being used. As in Scene 2A, the Apprentice relates the chain of reasoning which led to this design decision.

Given the Apprentice's explanation, the programmer's first thought is that there really is no excuse for the way the hardware guys designed the K7. However, hardware devices never seem to be defined exactly the way you want them to be. After mentally shrugging his shoulders, the programmer tries to figure out some way to get around the ugliness of the hardware and streamline the driver code.

```
>Why are you using IDLE-SEM?

Because:
  (1) If two pieces of code initiate device operations that cannot be done
      at the same time then a semaphore must be used to arbitrate between
them.
  (2) PUTC sends characters to the K7.
  (3) :CLEAR initializes the K7.
  (4) Sending characters to the K7 and initializing the K7 cannot be done at
      the same time.

>Have the clear operation busy-wait before and after calling K7-CLEAR.

Constraint Violation: Driver functions other than the init function
  cannot busy-wait.

>Merge the functions K7-CLEAR and K7-SEND by multiplexing the clear
 operation through the function K7-PUTC.

Request For Clarification: What character should I use for encoding :CLEAR ?

>#\Page.
```

Scene 4A: The programmer introduces multiplexing.

The programmer first tries to simplify the driver by modeling the function K7-CLEAR after the function K7-INIT. He reasons that if the clear operation busy-waits before and after calling K7-CLEAR, then there will be no risk of contention with the function K7-PUTC. Unfortunately, this approach cannot be used because driver functions other than the init function are not allowed to busy-wait.

The programmer then decides to use multiplexing to simulate the way the K7 should have been designed. The idea here is to combine K7-SEND and K7-CLEAR into one function. A special character code will then be used to tell the combined function when to initialize the device as opposed to sending a character. The Apprentice asks the programmer for a character to use.

Scene 4B shows the driver code after multiplexing has been added. There are two major changes. First, IDLE-SEM has been removed. This results in minor changes to the functions CREATE-K7, K7-INIT, K7-PUTC, and K7-CONTROL. In addition, the function

```
;CONTROL BLOCK DEFINITION FOR K7

(defconstant K7-init #0002)
(defconstant K7-height 4)
(defconstant K7-width 40)
(defstruct K7
  addr in-buffer in-sem out-buffer out-sem sending?)

(defun create-K7 (addr)
  (let ((out-size (* K7-height K7-width)))
    (make-K7 :addr       addr
             :in-buffer  (create-fast-q 10)
             :in-sem     (create-sem 0)
             :out-buffer (create-fast-q out-size)
             :out-sem    (create-sem out-size))
             :sending?   nil))

(defun K7-init (dcb)
  (flush-fast-q (K7-in-buffer dcb))
  (setf (sem-count (K7-in-sem dcb)) 0)
  (flush-fast-q (K7-out-buffer dcb))
  (setf (sem-count (K7-out-sem dcb)) (fast-q-size (K7-out-buffer dcb)))
  (setf (K7-sending? dcb) nil)
  (%store CSR-int-disable (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store K7-init (XCSR (K7-addr dcb)))
  (push (list :action "K7-init set initialization bit"
              :time (get-universal-time)
              :dcb (copy-all dcb))
        *debugging-info*)                          ;TEMPORARY INSTRUMENTATION
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (RCSR (K7-addr dcb))))

;DRIVER TABLE FOR THE ENGINE DIAGNOSIS DISPLAY

(defconstant *engine-diagnosis-display* 14)

(setf (get-device *engine-diagnosis-display*)
      (make-device
        :in-int-addr  #0640          :out-int-addr  #0642
        :in-int-fn    #'K7-receive   :out-int-fn    #'K7-send
        :init         #'K7-init      :dcb           (create-K7 #0177120)
        :open         #'io-noop      :close         #'io-noop
        :putc         #'K7-putc      :write         #'K7-write
        :getc         #'K7-getc      :control       #'K7-control))
```

Scene 4B: Code with multiplexing added (part 1).

```
;UPPER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-control (dcb code &rest ignore)
  (without-interrupts
    (case code
      (:CLEAR (flush-fast-q (K7-out-buffer dcb))
              (setf (sem-count (K7-out-sem dcb))
                    (fast-q-size (K7-out-buffer dcb)))
              (K7-putc dcb #\Page))
      (:RESET (flush-fast-q (K7-in-buffer dcb))
              (setf (sem-count (K7-in-sem dcb)) 0)
              (K7-control dcb :CLEAR))
      (:LINE-NUMBER {not-yet-implemented})
      (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb))))))

(defun K7-getc (dcb)
  (without-interrupts
    (sem-wait (K7-in-sem dcb))
    (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-write (dcb buffer length)
  (dotimes (pointer length)
    (K7-putc dcb (aref buffer pointer))))

(defun K7-putc (dcb char)
  (without-interrupts
    (sem-wait (K7-out-sem dcb))
    (fast-enq (K7-out-buffer dcb) (char-code char))
    (when (not (K7-sending? dcb))
      (setf (K7-sending? dcb) T)
      (K7-send dcb))))

;LOWER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-receive (dcb)
  (without-interrupts
    (fast-enq (K7-in-buffer dcb) (%get (RBUF (K7-addr dcb))))
    (sem-signal (K7-in-sem dcb))))

(defun K7-send (dcb)
  (without-interrupts
    (if (= (sem-count (K7-out-sem dcb))
           (fast-q-size (K7-out-buffer dcb)))
        (setf (K7-sending? dcb) nil)
      (let ((char (fast-deq (K7-out-buffer dcb))))
        (cond ((= char (char-code #\Page))
               (%store (logand CSR-int-enable K7-init) (XCSR (K7-addr dcb)))
               (push (list :action "K7-send set initialization bit"
                           :time (get-universal-time)
                           :dcb (copy-all dcb))
                     *debugging-info*))          ;TEMPORARY INSTRUMENTATION
              (T (%store char (XBUF (K7-addr dcb)))
                 (push (list :action "K7-send sent char"
                             :time (get-universal-time)
                             :dcb (copy-all dcb))
                       *debugging-info*)))       ;TEMPORARY INSTRUMENTATION
        (sem-signal (K7-out-sem dcb))))))
```

Scene 4C: Code with multiplexing added (part 2).

K7-DONE is no longer needed. The function K7-SEND is directly installed as the function triggered by transmit interrupts.

Second, the functions K7-SEND and K7-CLEAR have been combined into one. The new version of K7-SEND has to operate without interrupts because it responds directly to an interrupt request rather than being called by K7-DONE. The instrumentation inserted in scene 3 is moved and changed appropriately.

The clear operation is supported by calling K7-PUTC with the special character #\Page. (Flushing the output queue before doing this promotes efficiency in exactly the same way that it did before.) K7-SEND initializes the device when it receives a #\Page.

Multiplexing is an interesting cliché, which is somewhat different from the other clichés described above. To a considerable extent, it is a cliché about how to transform a program rather than a cliché about how to create a program. Like knowledge about instrumentation, it does not have anything to do with drivers in particular—it is useful in a wide range of contexts.

Scene 5: Making a Low-Level Addition

At this point, the programmer feels that the driver code is complete and asks the Apprentice to install it in the operating system. The Apprentice points out that there are a number of reasons why it would not be a good idea to install the driver code as is, and asks the programmer to confirm the fact that he really wants to install the driver.

The programmer could force the Apprentice to install the driver as is. However, he decides that it would be a good idea to fix a few things first. He starts by telling the Apprentice to remove the instrumentation. He then sets about telling the Apprentice how to support :LINE-NUMBER.

```
>Install the driver.

Warning: It is not a good idea to install the driver.
Because:
 (1) There is debugging instrumentation in it.
 (2) :LINE-NUMBER has not been implemented.
 (3) There is no error checking.

>Remove the instrumentation.
 Add a DCB field LINE-NUMBER. LINE-NUMBER tracks the cursor line.
 Make :CLEAR set LINE-NUMBER to 0.
 When K7-PUTC is called with #\Return, increment LINE-NUMBER by 1.

Request For Clarification: What should LINE-NUMBER be initialized to?

>0.
```

Scene 5A: The programmer says how to calculate the line number.

In this scene, it is assumed that the Apprentice does not understand very much about the concept "cursor line" (the line the cursor is on). The purpose of this is to illustrate the fact that a programmer can give very specific directions to the Apprentice, if desired. At this level of interaction, the Apprentice operates in much the same way as the current KBEmacs system.

The programmer tells the Apprentice to add a DCB field named LINE-NUMBER. He then says exactly where and how to modify this field. The only thing which links the LINE-NUMBER field up with the concept "cursor line" is the programmer's assertion that the new field tracks the cursor line.

Examining the programmer's commands at a superficial level, the Apprentice notices that the value of LINE-NUMBER will be undefined until the first clear operation is performed. It therefore asks the programmer to provide an initial value.

Scene 5B shows the code which results from the programmer's commands. The instrumentation has been removed from K7-INIT and K7-SEND. The LINE-NUMBER field has been added along with code which increments it and sets it to zero in the places specified by the programmer. Because the programmer said that the LINE-NUMBER field tracks the cursor line, its value is returned as the result of the line number operation.

```
;CONTROL BLOCK DEFINITION FOR K7

(defconstant K7-init #0002)
(defconstant K7-height 4)
(defconstant K7-width 40)
(defstruct K7
  addr in-buffer in-sem out-buffer out-sem sending? line-number)

(defun create-K7 (addr)
  (let ((out-size (* K7-height K7-width)))
    (make-K7 :addr       addr
             :in-buffer  (create-fast-q 10)
             :in-sem     (create-sem 0)
             :out-buffer (create-fast-q out-size)
             :out-sem    (create-sem out-size))
             :sending?   nil))

(defun K7-init (dcb)
  (flush-fast-q (K7-in-buffer dcb))
  (setf (sem-count (K7-in-sem dcb)) 0)
  (flush-fast-q (K7-out-buffer dcb))
  (setf (sem-count (K7-out-sem dcb)) (fast-q-size (K7-out-buffer dcb)))
  (setf (K7-sending? dcb) nil)
  (setf (K7-line-number dcb) 0)
  (%store CSR-int-disable (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store K7-init (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (RCSR (K7-addr dcb))))

;DRIVER TABLE FOR THE ENGINE DIAGNOSIS DISPLAY

(defconstant *engine-diagnosis-display* 14)

(setf (get-device *engine-diagnosis-display*)
      (make-device
        :in-int-addr  #0640          :out-int-addr  #0642
        :in-int-fn    #'K7-receive   :out-int-fn    #'K7-send
        :init         #'K7-init      :dcb           (create-K7 #0177120)
        :open         #'io-noop      :close         #'io-noop
        :putc         #'K7-putc      :write         #'K7-write
        :getc         #'K7-getc      :control       #'K7-control))
```

Scene 5B: Code with line number computation added (part 1).

```
;UPPER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-control (dcb code &rest ignore)
  (without-interrupts
    (case code
|     (:CLEAR (setf (K7-line-number dcb) 0)
              (flush-fast-q (K7-out-buffer dcb))
              (setf (sem-count (K7-out-sem dcb))
                    (fast-q-size (K7-out-buffer dcb)))
              (K7-putc dcb #\Page))
      (:RESET (flush-fast-q (K7-in-buffer dcb))
              (setf (sem-count (K7-in-sem dcb)) 0)
              (K7-control dcb :CLEAR))
|     (:LINE-NUMBER (K7-line-number dcb))
      (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb))))))

(defun K7-getc (dcb)
  (without-interrupts
    (sem-wait (K7-in-sem dcb))
    (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-write (dcb buffer length)
  (dotimes (pointer length)
    (K7-putc dcb (aref buffer pointer))))

(defun K7-putc (dcb char)
  (without-interrupts
|   (if (equal char #\Return) (incf (K7-line-number dcb)))
    (sem-wait (K7-out-sem dcb))
    (fast-enq (K7-out-buffer dcb) (char-code char))
    (when (not (K7-sending? dcb))
      (setf (K7-sending? dcb) T)
      (K7-send dcb))))

;LOWER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-receive (dcb)
  (without-interrupts
    (fast-enq (K7-in-buffer dcb) (%get (RBUF (K7-addr dcb))))
    (sem-signal (K7-in-sem dcb))))

(defun K7-send (dcb)
  (without-interrupts
    (if (= (sem-count (K7-out-sem dcb))
           (fast-q-size (K7-out-buffer dcb)))
        (setf (K7-sending? dcb) nil)
      (let ((char (fast-deq (K7-out-buffer dcb))))
|       (if (= char (char-code #\Page))
|           (%store (logand CSR-int-enable K7-init) (XCSR (K7-addr dcb)))
|           (%store char (XBUF (K7-addr dcb))))
        (sem-signal (K7-out-sem dcb))))))
```

Scene 5B: Code with line number computation added (part 2).

Scene 6: Making a High-Level Addition

It is interesting to consider what the interaction in Scene 5 would have looked like if the Apprentice had known a lot about the concept "cursor line". This scene repeats the interaction concerning :LINE-NUMBER under the assumption that the Apprentice knows a lot rather than only a little.

To start with, the Apprentice would have responded quite differently to the programmer's initial specification of the line number operation. Scene 6A shows what the second request in Scene 1C would have been.

```
Request for Clarification: How do output characters change the cursor
  line?
```

Scene 6A: Alternate portion of Scene 1C.

Here we assume that the interactive display device cliché contains knowledge about what the cursor line is and how it is affected by various operations on the device. It is also assumed that the interactive display driver cliché contains knowledge about how to determine the value of a quantity such as the cursor line: If there is no way to directly query the device for the value of the quantity, you must simulate the device to determine how the quantity changes when output is sent to the device.

In order to simulate a device you have to know how each operation on the device alters the quantity. The specification for the K7 says that initializing the K7 homes the cursor and therefore sets the cursor line to zero. However, the specification says nothing about the effects of output characters. Therefore, the Apprentice asks the programmer for clarification. (As discussed in conjunction with Scene 1D, the programmer could ask for an explanation of all this if he desired.)

In Scene 6B the programmer tells the Apprentice how output characters affect the cursor line. The Apprentice integrates the programmer's statements with what it knows about interactive display devices in general and comes up with a contradiction. As described by the programmer, there is no upper limit on the cursor line.

```
>The only output character that changes the cursor line is #\Return.
 When a #\Return is output, the cursor moves to the next line.

Constraint Violation: An output character cannot cause the cursor line to
  become greater than or equal to the screen height.

>If a #\Return is output when the cursor is already on the last line,
 the cursor stays on the last line.
```

Scene 6B: The programmer describes how the K7 changes the line number.

The programmer fixes the bug by amending his description of the effect of #\Return on the cursor line. He could also have relieved the contradiction by saying that the K7 allows the cursor line to become greater than the screen height.

There is a fundamental difference between Scene 5 and Scene 6. In Scene 5 the Apprentice has almost no understanding of what the programmer is doing. The goal

```
;CONTROL BLOCK DEFINITION FOR K7

(defconstant K7-init #0002)
(defconstant K7-height 4)
(defconstant K7-width 40)
(defstruct K7
  addr in-buffer in-sem out-buffer out-sem sending? cursor-line)

(defun create-K7 (addr)
  (let ((out-size (* K7-height K7-width)))
    (make-K7 :addr       addr
             :in-buffer  (create-fast-q 10)
             :in-sem     (create-sem 0)
             :out-buffer (create-fast-q out-size)
             :out-sem    (create-sem out-size))
             :sending?   nil))

(defun K7-init (dcb)
  (flush-fast-q (K7-in-buffer dcb))
  (setf (sem-count (K7-in-sem dcb)) 0)
  (flush-fast-q (K7-out-buffer dcb))
  (setf (sem-count (K7-out-sem dcb)) (fast-q-size (K7-out-buffer dcb)))
  (setf (K7-sending? dcb) nil)
  (%store CSR-int-disable (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store K7-init (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (setf (K7-cursor-line dcb) 0)
  (%store CSR-int-enable (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (RCSR (K7-addr dcb))))

;DRIVER TABLE FOR THE ENGINE DIAGNOSIS DISPLAY

(defconstant *engine-diagnosis-display* 14)

(setf (get-device *engine-diagnosis-display*)
      (make-device
        :in-int-addr  #0640          :out-int-addr  #0642
        :in-int-fn    #'K7-receive   :out-int-fn    #'K7-send
        :init         #'K7-init      :dcb           (create-K7 #0177120)
        :open         #'io-noop      :close         #'io-noop
        :putc         #'K7-putc      :write         #'K7-write
        :getc         #'K7-getc      :control       #'K7-control))
```

Scene 6C: Code with improved line number computation added (part 1).

```
;UPPER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-control (dcb code &rest ignore)
  (without-interrupts
    (case code
      (:CLEAR (flush-fast-q (K7-out-buffer dcb))
              (setf (sem-count (K7-out-sem dcb))
                    (fast-q-size (K7-out-buffer dcb)))
              (K7-putc dcb #\Page)
              (setf (K7-cursor-line dcb) 0))
      (:RESET (flush-fast-q (K7-in-buffer dcb))
              (setf (sem-count (K7-in-sem dcb)) 0)
              (K7-control dcb :CLEAR))
      (:LINE-NUMBER (K7-cursor-line dcb))
      (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb))))))

(defun K7-getc (dcb)
  (without-interrupts
    (sem-wait (K7-in-sem dcb))
    (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-write (dcb buffer length)
  (dotimes (pointer length)
    (K7-putc dcb (aref buffer pointer))))

(defun K7-putc (dcb char)
  (without-interrupts
    (sem-wait (K7-out-sem dcb))
    (fast-enq (K7-out-buffer dcb) (char-code char))
    (if (and (equal char #\Return)
             (< (K7-cursor-line dcb) (1- K7-height)))
        (incf (K7-cursor-line dcb)))
    (when (not (K7-sending? dcb))
      (setf (K7-sending? dcb) T)
      (K7-send dcb))))

;LOWER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-receive (dcb)
  (without-interrupts
    (fast-enq (K7-in-buffer dcb) (%get (RBUF (K7-addr dcb))))
    (sem-signal (K7-in-sem dcb))))

(defun K7-send (dcb)
  (without-interrupts
    (if (= (sem-count (K7-out-sem dcb))
           (fast-q-size (K7-out-buffer dcb)))
        (setf (K7-sending? dcb) nil)
      (let ((char (fast-deq (K7-out-buffer dcb))))
        (if (= char (char-code #\Page))
            (%store (logand CSR-int-enable K7-init) (XCSR (K7-addr dcb)))
            (%store char (XBUF (K7-addr dcb))))
        (sem-signal (K7-out-sem dcb))))))
```

Scene 6C: Code with improved line number computation added (part 2).

of that scene is to show that the programmer can make an addition to a program even if the Apprentice is not able to understand anything about it. In contrast, in Scene 6 the Apprentice has a detailed understanding of what the user says. The improved understanding of the Apprentice is illustrated by the fact that the Apprentice is able to detect a bug which it did not catch in Scene 5.

Based on the programmer's description, and what it already knows, the Apprentice can figure out how to simulate changes in the cursor line. The driver code which results is shown in Scene 6C. For the most part, the code is very similar to the code in Scene 5B. However, there are a number of differences. For example, the DCB field in Scene 6C is called CURSOR-LINE rather than LINE-NUMBER. (The Apprentice named it based on what it is simulating.) In addition, the pieces of code which change the CURSOR-LINE field are positioned directly after the computation which actually causes the cursor line to change, rather than merely being placed in the correct function.

Some of the most interesting aspects of Scene 6C are places where the code is equivalent to Scene 5 but for quite different reasons. For example, CURSOR-LINE is set to zero in K7-INIT because K7-INIT initializes the K7, rather than merely because the programmer said to initialize CURSOR-LINE to zero. In addition, CURSOR-LINE is incremented in K7-PUTC rather than in K7-SEND because the Apprentice realizes that this kind of simulation has to be done in the upper-level driver functions. This placement is necessary because, when a user executes the line number operation he presumably wants to know the line the next character he is going to send will go on. He is not interested in whether or not all of his previous I/O operations have been completely communicated to the device.

Scene 7: Adding Error Checking

In this scene, the programmer tells the Apprentice to add error checking into the driver. The Apprentice responds to this by asking for clarification about how error checking should be done.

The programmer tells the Apprentice not to worry about output errors at all. He then says that input errors (e.g., bad parity, and input buffer overflow) should be treated as if the bad character was never received at all. (The Apprentice suggests this method of handling input errors because the cliché interactive display driver specifies that it is the standard method to use.)

```
>Introduce error checking.

Request for Advice: I do not know how to validate output characters.

>Do not validate output characters.

Request for Advice: I do not know how to respond to reception errors.
Note: the standard way to deal with input errors is:
  (1) Treat reception errors as reception failures.

>Use (1).
```

Scene 7A: The programmer asks for error checking to be added.

Scene 7B shows the error checking code which is added to the driver. A clause is added to K7-CONTROL to check for bad control codes. Code is added to K7-RECEIVE to discard bad input characters and prevent the input buffer from becoming over filled. Two (previously omitted) fields of the device table entry for the engine diagnosis display are filled in with special functions which report errors if a user tries to call the device independent functions READ or SEEK.

It is interesting to note that error checking does not have to be added to the special queue manipulation functions used by the driver. Error checking is unnecessary in these functions because, with the changes to K7-RECEIVE, it can be proven that the driver will never call a queueing function erroneously.

The fact that error checking can be deferred until now (without it being forgotten altogether) is an example of a way in which the Apprentice can assist with rapid prototyping. The key benefit of this delay is not that it saves coding time, but rather that it saves thinking time. The only coding time that was saved was time that the Apprentice would have spent, not time that the programmer would have spent. However, the fact that the programmer did not have to think about the questions in Scene 7A until now is a very real saving.

A much more important way in which the Apprentice supports rapid prototyping revolves around the use of clichés. There are two basic ways in which to use the Apprentice. The first approach is to start from the clichés it knows and modify them adding new pieces of information until the system understands the exact specification you have in mind. This approach was illustrated in the scenario above.

The second approach is to select the clichés which are closest to the specification you

```
;CONTROL BLOCK DEFINITION FOR K7

(defconstant K7-init #0002)
(defconstant K7-height 4)
(defconstant K7-width 40)
(defstruct K7
  addr in-buffer in-sem out-buffer out-sem sending? cursor-line)

(defun create-K7 (addr)
  (let ((out-size (* K7-height K7-width)))
    (make-K7 :addr       addr
             :in-buffer  (create-fast-q 10)
             :in-sem     (create-sem 0)
             :out-buffer (create-fast-q out-size)
             :out-sem    (create-sem out-size))
             :sending?   nil))

(defun K7-init (dcb)
  (flush-fast-q (K7-in-buffer dcb))
  (setf (sem-count (K7-in-sem dcb)) 0)
  (flush-fast-q (K7-out-buffer dcb))
  (setf (sem-count (K7-out-sem dcb)) (fast-q-size (K7-out-buffer dcb)))
  (setf (K7-sending? dcb) nil)
  (%store CSR-int-disable (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (%store K7-init (XCSR (K7-addr dcb)))
  (busy-wait-not XCSR-busy (XCSR (K7-addr dcb)))
  (setf (K7-cursor-line dcb) 0)
  (%store CSR-int-enable (XCSR (K7-addr dcb)))
  (%store CSR-int-enable (RCSR (K7-addr dcb))))

;DRIVER TABLE FOR THE ENGINE DIAGNOSIS DISPLAY

(defconstant *engine-diagnosis-display* 14)

(setf (get-device *engine-diagnosis-display*)
      (make-device
        :in-int-addr  #0640          :out-int-addr  #0642
        :in-int-fn    #'K7-receive   :out-int-fn    #'K7-send
        :init         #'K7-init      :dcb           (create-K7 #0177120)
        :open         #'io-noop      :close         #'io-noop
        :putc         #'K7-putc      :write         #'K7-write
        :getc         #'K7-getc      :read          #'io-error
        :control      #'K7-control   :seek          #'io-error))
```

Scene 7B: Code with error checking added (part 1).

```
;UPPER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-control (dcb code &rest ignore)
  (without-interrupts
    (case code
      (:CLEAR (flush-fast-q (K7-out-buffer dcb))
              (setf (sem-count (K7-out-sem dcb))
                    (fast-q-size (K7-out-buffer dcb)))
              (K7-putc dcb #\Page)
              (setf (K7-cursor-line dcb) 0))
      (:RESET (flush-fast-q (K7-in-buffer dcb))
              (setf (sem-count (K7-in-sem dcb)) 0)
              (K7-control dcb :CLEAR))
      (:LINE-NUMBER (K7-cursor-line dcb))
      (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb)))
      (OTHERWISE (error "the control code ~A is not supported by~
                         the K7 display" code)))))

(defun K7-getc (dcb)
  (without-interrupts
    (sem-wait (K7-in-sem dcb))
    (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-write (dcb buffer length)
  (dotimes (pointer length)
    (K7-putc dcb (aref buffer pointer))))

(defun K7-putc (dcb char)
  (without-interrupts
    (sem-wait (K7-out-sem dcb))
    (fast-enq (K7-out-buffer dcb) (char-code char))
    (if (and (equal char #\Return)
             (< (K7-cursor-line dcb) (1- K7-height)))
        (incf (K7-cursor-line dcb)))
    (when (not (K7-sending? dcb))
      (setf (K7-sending? dcb) T)
      (K7-send dcb))))

;LOWER-LEVEL FUNCTIONS FOR K7 DRIVER

(defun K7-receive (dcb)
  (without-interrupts
    (let ((data (%get (RBUF (K7-addr dcb)))))
      (when (and (not (logtest RBUF-error data))
                 (< (sem-count (K7-in-sem dcb)))
                    (fast-q-size (K7-in-buffer dcb)))
        (fast-enq (K7-in-buffer dcb) data)
        (sem-signal (K7-in-sem dcb))))))

(defun K7-send (dcb)
  (without-interrupts
    (if (= (sem-count (K7-out-sem dcb))
           (fast-q-size (K7-out-buffer dcb)))
        (setf (K7-sending? dcb) nil)
      (let ((char (fast-deq (K7-out-buffer dcb))))
        (if (= char (char-code #\Page))
            (%store (logand CSR-int-enable K7-init) (XCSR (K7-addr dcb)))
            (%store char (XBUF (K7-addr dcb))))
        (sem-signal (K7-out-sem dcb))))))
```

Scene 7B: Code with error checking added (part 2).

want and then modify your specification until it fits into these clichés. This of course does not give you the exact program you wanted, but it gives you a program very rapidly. Experimenting with this approximately correct program can be very useful for answering questions about the program actually desired.

Scene 8: The Driver Specification Revisited

The scenario is presented as a series of modifications to the code in Scene 1. However, it could just as well have been presented as a series of modifications to the specifications. In fact, both things were happening at once.

As with KBEmacs, the program code is shown to the programmer so that he can get down as close to the details as he wants. (He could edit the program directly using a text editor if he wanted to.)

However, the Apprentice attempts to interpret everything the programmer does as changes to the specification. If asked to do so, the Apprentice can print out a new specification showing the net results of the programmer's commands. This is shown in Scene 8.

Most of the additions in the new specification are taken directly from the commands given by the programmer. (The new specification incorporates the commands in Scene 6 rather than Scene 5.)

A new section of detailed implementation directions has been added which contains information corresponding to the various commands given by the user.

```
The "K7" is an interactive display device where:
  The screen height is 4 lines.
  The screen width is 40 characters.
  The I/O is in terms of ASCII characters.
  Direct cursor positioning is not supported.
    Outputting a #\Return moves the cursor to the next line unless the
      cursor is already on the last line.
  The keyboard has three keys.
      key          character
    ACKNOWLEDGE    ACK  #0006
    YES            Y    #0131
    NO             N    #0116
  The bus interface is a standard SLU except that:
    Writing a 1 in Bit 1 of the XCSR initializes the device.
      Initializing the device blanks the screen and homes the cursor.
      Completion of initialization is signaled in the same way as the
        transmission of a character.
      Sending characters to the K7 and initializing the K7 cannot be
        done at the same time.

The "engine diagnosis display" is a K7 where:
  The device address is #0177120.
  The receive interrupt address is #0640.
  The transmit interrupt address is #0642.

The K7 driver is an interactive display driver where:
  Echoing is not supported
  Rubout handling is not supported.
  The basic software functions are:
    Supported: PUTC, WRITE & GETC.
    Ignored: OPEN & CLOSE.
    Not supported: READ & SEEK.
  The control codes are:
    :CLEAR - blanks the screen and homes the cursor.
    :RESET - does a :CLEAR and flushes pending input.
    :LINE-NUMBER - returns the cursor line.
    :NUMBER-CHARS-WAITING - returns the number of pending input characters.
  The implementation guidelines are:
    1- Do no consing.
    2- Favor space efficiency.
  The detailed implementation directions are:
    Support :CLEAR by initializing the device.
    Make the size of the output buffer equal to the total screen size.
    Multiplex the clear operation through the upper-level character output
      function.  Do this using the character #\Page.
    Do not validate output characters.
    Treat reception errors as reception failures.
```

Scene 8: Amended specification for K7 driver.

Acknowledgments

Discussions with many of the members of the Programmer's Apprentice group were helpful in formulating the ideas presented here. The authors would particularly like to acknowledge the assistance of Yishai Feldman and Jeremy Wertheimer.

References

[1] D. Comer, "Operating System Design: The XINU Approach", Prentice-Hall, 1984.

[2] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", IJCAI-81, pp. 1044-1052, August 1981.

[3] C. Rich, "The Layered Architecture of a System for Reasoning about Programs", IJCAI-85, pp. 540-546, August 1985.

[4] C. Rich & R.C. Waters, "Toward a Requirements Apprentice: On the Boundary Between Informal and Formal Specifications", MIT/AIM-907, July 1986.

[5] C. Rich & R.C. Waters (eds.), *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman, 1986.

[6] R.C. Waters, "The Programmer's Apprentice: A Session with KBEmacs", IEEE Transactions on Software Engineering, V11 #11, pp. 1296-1320, November 1985.

END

9-87

DTIC