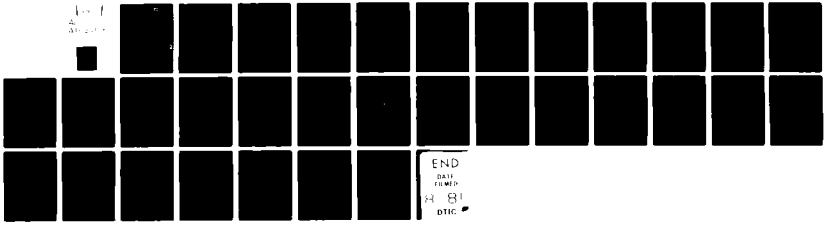
AD-A102 157 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
ABSTRACTION, INSPECTION AND DEBUGGING IN PROGRAMMING.(U)
JUN 81 C RICH, R C WATERS N00014-80-C-0505
UNCLASSIFIED AI-M-634 NL



LEVEL II

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
AI Memo # 634	AD-A102 157	

4. TITLE (and Subtitle)
Abstraction, Inspection and Debugging in Programming

5. TYPE OF REPORT & PERIOD COVERED
Memorandum

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Charles Rich & Richard C. Waters

8. CONTRACT OR GRANT NUMBER(s)
N00014-80-C-0505
MCS-7912179

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Advanced Research Projects Agency
1400 Wilson Blvd
Arlington, Virginia 22209

12. REPORT DATE
June 1981

13. NUMBER OF PAGES
30

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Office of Naval Research
Information Systems
Arlington, Virginia 22217

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Distribution of this document is unlimited.

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

DTIC ELECTE JUL 29 1981 S D F

18. SUPPLEMENTARY NOTES
None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Artificial Intelligence
Programmer's apprentice
Automatic programming
Program editor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
We believe that software engineering has much to learn from other mature engineering disciplines, such as electrical engineering, and that the problem solving behaviors of engineers in different disciplines have many similarities. Three key ideas in current artificial intelligence theories of engineering problem solving are: Abstraction -- using a simplified view of the problem to guide the problem solving process. Inspection -- problem solving by recognizing the form ("plan") of a solution. Debugging -- incremental modification of an almost satisfactory solution to a more satisfactory

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. one. These three techniques are typically used together in a paradigm which we call AID (for Abstraction, Inspection, Debugging): First an abstract model of the problem is constructed in which some important details are intentionally omitted. In this simplified view inspection methods are more likely to succeed, yielding the initial form of a solution. Further details of the problem are then added one at a time with corresponding incremental modifications to the solution.

This paper states the goals and milestones of the remaining three years of a five year research project to study the fundamental principles underlying the design and construction of large software systems and to demonstrate the feasibility of a computer aided design tool for this purpose, called the programmers apprentice.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 634 June 1981

ABSTRACTION, INSPECTION AND DEBUGGING IN PROGRAMMING

Charles Rich
Richard C. Waters

Abstract

We believe that software engineering has much to learn from other mature engineering disciplines, such as electrical engineering, and that the problem solving behaviors of engineers in different disciplines have many similarities. Three key ideas in current artificial intelligence theories of engineering problem solving are:

Abstraction -- using a simplified view of the problem to guide the problem solving process.

Inspection -- problem solving by recognizing the form ("plan") of a solution.

Debugging -- incremental modification of an almost satisfactory solution to a more satisfactory one.

These three techniques are typically used together in a paradigm which we call AID (for Abstraction, Inspection, Debugging): First an abstract model of the problem is constructed in which some important details are intentionally omitted. In this simplified view inspection methods are more likely to succeed, yielding the initial form of a solution. Further details of the problem are then added one at a time with corresponding incremental modifications to the solution.

This paper states the goals and milestones of the remaining three years of a five year research project to study the fundamental principles underlying the design and construction of large software systems and to demonstrate the feasibility of a computer aided design tool for this purpose, called the programmer's apprentice.

Adapted from proposals submitted to the National Science Foundation and the Advanced Research Projects Agency of the Department of Defense.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505 and in part by National Science Foundation grant MCS-7912179. The views and conclusions contained in this paper are those of the authors, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, or the United States Government.

Introduction

In an earlier proposal [32] we discussed the nature of current productivity and reliability problems in software engineering and suggested how current theories of problem solving in artificial intelligence can provide the conceptual basis for a solution. We also outlined a five year research project to study the fundamental principles underlying the design and construction of large software systems and to demonstrate the feasibility of a computer aided design tool for this purpose, called the *programmer's apprentice.* This proposal restates the goals and milestones of the remaining three years of this project updated by the insight gained in the past two years.

We believe that software engineering has much to learn from other mature engineering disciplines, such as electrical engineering, and that the problem solving behaviors of engineers in different disciplines have many similarities. Three key ideas in current artificial intelligence theories of engineering problem solving are:

(i) *Abstraction* -- using a simplified view of the problem to guide the problem solving process [39]. An example of using abstraction in programming is to first implement the desired behavior of a program on typical data before worrying about exception handling.

(ii) *Inspection* -- problem solving by recognizing the form (plan) of a solution. In programming, a plan may be a particular control strategy with unspecified primitive actions or an abstract data structure with an unspecified implementation.

(iii) *Debugging* -- incremental modification of an almost satisfactory solution to a more satisfactory one. In programming, this evolution is triggered as often by a change in the problem specification as by an error in the initial solution.

These three techniques are typically used together in a paradigm which we call AID (for Abstraction, Inspection, Debugging):[1] First an abstract model of the problem is constructed in which some important details are intentionally omitted. In this simplified view inspection methods are more likely to succeed, yielding the initial form of the solution. Further details of the problem are then added one at a time with corresponding incremental modifications to the solution.

This paradigm is a powerful mental tool used by engineers of all kinds for dealing with complex design problems for which there are no general solutions. For example, consider the following advice found in a textbook on designing integrated circuits [18].[2]

> Probably the worst way to proceed in analyzing the behavior of a small-signal circuit is to use the most complex model available for the device. Not only does such a procedure lead to lengthy computation, but by its complexity it also obscures the really important aspects of the circuit behavior. Instead, one should begin with a highly idealized simple model for the device. One can then focus on important circuit behavior and obtain some idea as to reasonable approximations; one can usually determine also what features of detailed device behavior will next be important.

---

1 Sussman [46] originally named this paradigm "problem solving by debugging almost right plans".

2 Page 402

We feel that the myth, current in some circles, that approximation and debugging have no place in the programming process has inhibited progress in developing more effective tools for programmers. Uniform general methods for automatic programming do not appear to be forthcoming in the near future. We therefore believe that the most reasonable alternative for now is the disciplined use of the AID paradigm.

Note that the AID paradigm is a *knowledge based* approach to the problem. Inspection methods are the embodiment of an experienced engineer's knowledge of the standard problems and corresponding solution forms in his field. A major part of formalizing the AID paradigm in programming is to compile a library of standard program and specification forms. We intend not only to provide tools for describing the structure of individual programs, but also a pre-analyzed set of standard forms out of which many different programs can be constructed. These standard forms can be thought of as programming "clichés".[1] As we will see later, this is a much more flexible notion than macros or subroutines. We will also see how the task of compiling such a library can be naturally divided up, starting with the most fundamental programming knowledge and expanding to include more specialized areas of programming.

The next section illustrates the issues and goals of our proposed research by presenting a scenario of program evolution aided by the programmer's apprentice. Section 2 describes the progress we have already made towards achieving this scenario. Our two principle achievements to date are the construction of an initial feasibility demonstration [51] and the development of an improved representation for knowledge about programs and programming. Sections 3 and 4 describe our present and future research topics, which include program analysis and synthesis by inspection, further codification of programming knowledge, the use of dependency-directed reasoning in programming, and automated learning of new programming clichés.

We are currently working on a second demonstration system based on the improved knowledge representation. Following this, we propose to complete a third system which demonstrates the feasibility of all of the essential capabilities shown in the scenario below. We also propose to construct a well-engineered prototype of the programmer's apprentice (based on the level of technology of the second feasibility demonstration) which will for the first time allow us to experiment with the actual use of such a system.

Section 1 - A Goal Scenario

We envision the programmer's apprentice (PA) as an expert programmer's junior partner and critic. The expert programmer makes the key decisions in the program development, while the apprentice helps with more limited tasks. The PA is not intended to be used by novice programmers or by non-programmers. This restriction sets a more modest research goal than automatic programming, but one which we believe is more realistic in the short term and which allows a smooth transition to more automatic systems in the future.

The scenario below shows a programmer adding a user accounts system to an existing operating system. We assume here that the existing operating system was originally implemented using the PA, so that the PA already understands much of its structure and function. The user accounts system is a simple one which keeps track of a balance for each user and prevents him from logging in when his balance is overdrawn.

An important theme to keep in mind when reading this scenario is what the PA knows as opposed to what the programmer knows. The PA has several different kinds of knowledge. First, it has extensive specific knowledge about the existing operating system. It has a simple understanding of the basic entities and operations in the operating system, such as users, user ids, passwords, logging in, etc., and it knows how the various functions of the system are implemented. Second, the PA has a lot of knowledge about programming

---

1 Perlis [28] uses the term "idiom" for a similar notion

in general. This includes knowing how to use standard data structures and algorithms (those you expect every good programmer to know) and also some that are more specialized, such as the manipulation of indexed files. Finally, the PA has some knowledge in non-programming domains. In this scenario the PA does not know what a user accounts system is *per se*, but it does know some simple accounting concepts, such what a balance is, what overdrawn means, etc. As will be discussed in following sections, we use a variety of artificial intelligence techniques to represent this knowledge and bring it to bear in appropriate situations.

The programmer's knowledge in this scenario partially overlaps with the PA's knowledge and is partially different. Both the programmer and the PA have a significant body of knowledge about basic programming technique, and some more specialized knowledge that is particular to operating systems programming. It is this shared knowledge which makes communication and cooperation between the programmer and the PA possible. What the programmer lacks is the ability to keep track of the details of the implementation of the existing operating system -- this is the PA's responsibility. On the other hand, what the PA lacks (and what the programmer provides) is an overall view of what needs to be done to create the new user accounts system.

Given this division of knowledge, there are seven basic facilities illustrated in the scenario by which the PA assists the programmer:

Programming with clichés -- The programmer can construct programs faster and with fewer errors by combining and modifying entries from the PA's library of standard algorithm and data structure fragments, as compared with using a standard text or structure editor.

Flexible display -- The PA can display programs at different levels of detail and from different points of view.

Propagation of design decisions -- As the programmer makes more and more design decisions, other decisions become forced. When this happens, the PA makes them automatically.

Bug detection -- Whenever the programmer makes a design decision which contradicts earlier decisions, the PA reports this fact as a bug.

Automatic modification -- The PA has explicit knowledge about many common kinds of modifications and can perform them automatically when requested.

Automatic optimization -- The PA knows how to perform a number of common optimizations, and is able to recognize situations in which they should be applied.

Escape to the surrounding environment -- At any time the programmer can step outside of the PA and use the tools in the surrounding environment.

Much of the scenario revolves around the concept of a *master file system*. This is a common programming concept known to the PA. The PA knows that a master file is an indexed file which contains most (if not all) of the information about a group of entities (such as users). It knows that there is a cluster of programs which operate on this file, typically including an interactive program for updating the file, report programs which print out selected information from the file, and various other programs which read and write individual records in the file. The PA knows how these programs are structured and how they interact. In addition, it knows the types of exceptional conditions which need to be handled in this kind of system and about how this kind of system is typically modified.

The scenario is presented as a sequence of screen images. The screen is divided into two parts. In the top part, the programmer types commands (in lower case) and the PA responds (in upper case). The PA uses the lower portion of the screen to display relevant sections of code. The programmer-PA interaction shown in the scenario is carried on in free form English; in the demonstration systems we implement, however, a simple formal command language will be used. The code shown in the scenario is Lisp; however, the basic PA

concept, as well as the actual implementation, is substantially programming language independent.

The Scenario

The scenario begins with the programmer creating a new master file system in which the master file is keyed by user ids.

```
> Let user-accounts be a master file system.
> The keys of the file are user ids.

USER PASSWORDS ARE STORED IN THE FILE <SYS>PASSWORD.
SHOULD ALL INFORMATION ABOUT USERS BE STORED IN THE USER MASTER FILE? yes

MODIFICATIONS:
 (1) THE PASSWORD FIELD HAS BEEN MOVED FROM THE FILE <SYS>PASSWORD TO
     THE FILE <SYS>USER.
 (2) THE PROGRAMS SYS-QUERY-PASSWORD AND SYS-CHANGE-PASSWORD HAVE BEEN
     MODIFIED TO REFLECT THIS CHANGE.
 (3) THE FILE <SYS>PASSWORD AND THE PROGRAM UPDATE-PASSWORD-FILE HAVE
     BEEN ELIMINATED SINCE THE FILE HAS NO OTHER FIELDS.
```

Since the file is indexed by user ids, the PA infers that the file contains information about users. Since the file is a master file, it expects that the file will contain all the information about a user. The PA looks to see if there is any user information stored anywhere else and discovers that there is already a file containing users' passwords. After checking with the programmer, it moves passwords into the user master file. (The PA informs the programmer whenever it makes a change which affects some code which is not currently displayed.) This is an example of a modification the PA can do automatically because it has specific knowledge about this kind of modification as part of its general knowledge about files.

The programmer next looks at the existing login program and modifies it so that overdrawn users are not allowed to log in.

```
> Show login.

(DEFUN LOGIN (ID)
  (COND ((SYS-QUERY-PASSWORD ID)
         (SYS-PRINT-ANNOUNCEMENTS)
         (SYS-GRANT-SYSTEM-ACCESS))
        (T (SYS-TERMINAL-DISCONNECT))))
```

```
> If a user's balance is overdrawn, then disconnect.

MODIFICATION: A BALANCE FIELD HAS BEEN ADDED TO THE USER MASTER FILE.

(DEFUN LOGIN (ID)
  (COND ((AND (SYS-QUERY-PASSWORD ID) (NOT (USER-OVERDRAWN ID)))
         (SYS-PRINT-ANNOUNCEMENTS)
         (SYS-GRANT-SYSTEM-ACCESS))
        (T (SYS-TERMINAL-DISCONNECT))))

(DEFUN USER-OVERDRAWN (ID)
  (LET* ((FILE (OPEN "<SYS>USER" '(IN KEYED)))
         (RECORD (KEYED-READ FILE ID)))
    (CLOSE FILE)
    (MINUSP (USER-BALANCE RECORD))))
```

Notice that the programmer here only needs to specify the essential logical content of the change to be made, while the PA takes care of the details of appropriately modifying the code. The PA highlights parts of the displayed code (rendered here as underlining) to indicate what changes have been made. (LET* is a primitive Lisp form which sequentially binds variables.)

Because the programmer referred to a user's balance, the PA adds a balance field to the user file. The PA also constructs the appropriate code to read the balance out of the file. (The functions OPEN, CLOSE, and KEYED-READ are standard functions for accessing files.) The PA has already chosen an external name for the file and the selector function USER-BALANCE to access the balance field. It is convenient to let the PA choose all of the names it needs, since the programmer is always free to change any name he doesn't like. Notice that because the PA knows what a balance is and what overdrawn means, it can construct the appropriate test in the definition of USER-OVERDRAWN. Finally, it uses its understanding of conditionals to combine this new test with the test already in LOGIN.

In its role as critic, the PA constantly looks to see if there are any obvious optimizations which can be made.

```
OPTIMIZATION:
  MERGE SYS-QUERY-PASSWORD AND USER-OVERDRAWN IN ORDER TO
  REMOVE DUPLICATE CALLS TO OPEN, CLOSE, AND KEYED-READ. -- SHOW RESULT? yes
```

```
(DEFUN LOGIN (ID)
  (COND ((USER-LOGIN-TEST ID)
         (SYS-PRINT-ANNOUNCEMENTS)
         (SYS-GRANT-SYSTEM-ACCESS))
        (T (SYS-TERMINAL-DISCONNECT))))

(DEFUN USER-LOGIN-TEST (ID)      ;merge of SYS-QUERY-PASSWORD and USER-OVERDRAWN.
  (LET* ((FILE (OPEN "<SYS>USER" '(IN KEYED)))
         (RECORD (KEYED-READ FILE ID)))
    (CLOSE FILE)
    (AND (EQUAL (SYS-PROMPT-WITHOUT-ECHO "PASSWORD:")
                (USER-PASSWORD RECORD))
         (NOT (MINUSP (USER-BALANCE RECORD))))))
```

Here the PA notices that an inter-program optimization can be performed because SYS-QUERY-PASSWORD (shown below) opens and closes the same file and reads the same record as USER-OVERDRAWN. The comment on the definition of USER-LOGIN-TEST above indicates that although the PA has combined two other functions to form this one, it continues to remember that *logically* they are two separate functions. As a result, if either SYS-QUERY-PASSWORD or USER-OVERDRAWN is later modified, the change will be reflected in USER-LOGIN-TEST.

```
(DEFUN SYS-QUERY-PASSWORD (ID)
  (LET* ((FILE (OPEN "<SYS>USER" '(IN KEYED)))
         (RECORD (KEYED-READ FILE ID)))
    (CLOSE FILE)
    (EQUAL (SYS-PROMPT-WITHOUT-ECHO "PASSWORD:") (USER-PASSWORD RECORD))))
```

Note that when the programmer moved passwords to the user file, SYS-QUERY-PASSWORD was modified as shown above to read the user master file instead of the old password file.

```
> Construct a report program.

(DEFUN ..report-name.. ()
  (LET ((FILE (OPEN "<SYS>USER" '(IN SEQUENTIAL)))
        (REPORT (OPEN "PRINTER" 'OUT)))
    (FORMAT REPORT "~5% ..report-title.. ~|")
    (PROG (RECORD LINES PAGENO)
          (SETQ LINES 0 PAGENO 0)
       LP (SETQ RECORD (READ FILE))
          (COND ((NULL RECORD) (RETURN NIL)))
          (COND ((..selection-predicate.. RECORD)
                 (COND ((ZEROP LINES)
                        (SETQ PAGENO (1+ PAGENO))
                        (SETQ LINES (- ..lines-per-page.. 2))
                        (FORMAT REPORT "..headings.. Page: ~D~2%" PAGENO)))
                 ..print-line..
                 (SETQ LINES (1- LINES))))
          (GO LP))
    (CLOSE FILE)
    (CLOSE REPORT)))
```

The PA knows the structure of a typical report program in a master file system: it is a program which looks at all of the records in the file, selects some subset of these records, and prints some of the information in the selected records along with a title and headings. The PA does not, however, use Lisp code as its internal representation of this structure. Programs are represented internally as composed of abstract parts, each of which is named by its logical *role* in the overall function of the program (e.g. the selection predicate). In response to the programmer's request, the PA above has produced from its internal representation a "sketch" of the code for the report program. This code has a number of gaps in it because several of the roles are unfilled. Unfilled roles are indicated in the code by "..<role>..". (The FORMAT function is a system function for creating formatted output. Like the Fortran construct it is modelled after, it is inscrutable but convenient.)

Thus the code shown by the PA here is only the tip of an iceberg. It knows many things about a program, such as its specifications, constraints on its logical parts, and its design history, which are not explicit in the code. The PA produces code only as a summary of what is going on, and in order to communicate with the programmer and other tools in the programming environment. Also, the code produced is based on the PA's idea of what a report is and what good programming style is. The programmer may well have somewhat different ideas. However, just as with a human assistant, the programmer gains more from not having to write the code himself than he loses from not being able to totally control its initial form.

```
> Select the users with overdrawn balances.
> Print each user's balance, name, and address.
MODIFICATION: TWO NEW FIELDS, NAME AND ADDRESS, HAVE BEEN ADDED TO
  THE USER MASTER FILE.
> Sort the selected users alphabetically by name.
> Do not print headings.
> Edit title format.          [Programmer directly edits the format]
```

```
(DEFUN REPORT-OVERDRAWN-USERS ()
  (LET ((RECORDS (SELECT-OVERDRAWN-USERS))
        (REPORT (OPEN "PRINTER" 'OUT)))
    (SETQ RECORDS (SORT RECORDS
                        '(LAMBDA (RECORD1 RECORD2)
                           (SYS-NAME-LESSP (USER-NAME RECORD1) (USER-NAME RECORD2)))))
    (FORMAT REPORT "~5% Overdrawn Users as of ~A~|" (DATE))
    (MAPC '(LAMBDA (RECORD)
             (FORMAT REPORT "~5D ~8,T ~A ~25,T ~A %"
                     (USER-BALANCE RECORD)
                     (USER-NAME RECORD)
                     (USER-ADDRESS RECORD)))
          RECORDS)
    (CLOSE REPORT)))
(DEFUN SELECT-OVERDRAWN-USERS ()
  (PROG (FILE RECORD RECORDS)
        (SETQ FILE (OPEN "<SYS>USER" '(IN SEQUENTIAL)))
     LP (SETQ RECORD (READ FILE))
        (COND ((NULL RECORD) (CLOSE FILE) (RETURN RECORDS)))
        (COND ((MINUSP (USER-BALANCE RECORD)) (SETQ RECORDS (CONS RECORD RECORDS))))
        (GO LP)))
```

The programmer issues a sequence of commands to tailor the report to fit his needs. (In order to save space, only a single display showing the code *after the final* command is shown.) These commands illustrate the basic ways in which the programmer can modify a program: fill in a role, add something new, delete something, and edit the program as text. This interaction shows how the PA's understanding of the typical report program for a master file system goes beyond simply having a template or macro for it. The PA has specifications for each role and constraints on how they interact.

In the first command line above, the programmer specifies the selection predicate for the report. A constraint in the internal representation of the program linking the selection predicate with the function name then causes the PA to automatically choose the name REPORT-OVERDRAWN-USERS for the function. Next, the programmer fills in the print line by specifying what to print, namely the user's name, address and balance. This causes two new fields (name and address) to be added to the user file.

Third, the programmer adds something new to the program by specifying that the report should be sorted by user names. This is a simple change in the PA's internal representation of the program, but results in conspicuous changes in the code, which now has two loops: one to select the appropriate records (which are then sorted) and another to print the report. The PA puts the first loop into a separate subroutine in order to make the program easier to read.

In the fourth command line, the programmer stipulates that headings should not be printed. The immediate effect of this is to delete the call to FORMAT which prints the headings. The PA then prunes out the computations involving page numbers and line counting which are no longer used (assuming that the printer automatically goes to the next page after the maximum number of lines that will fit on a page are printed). In general, the code initially produced by the PA contains features (such as headings) which are not strictly necessary, on the theory that as long as there aren't too many such features, it is easier for the programmer to prune things out than to think them up.

Finally, the programmer escapes to the surrounding environment and directly edits the title format as a text string. When the edit is completed, the PA re-analyzes the code in order to integrate the change with its understanding of the program as a whole.

```
> Construct the file update program.
IMPLEMENTATIONS:
   (1) INTELLIGENT TERMINAL
   (2) FAST DISPLAY TERMINAL
   (3) SLOW TERMINAL            -- CHOOSE? 3
```

```
(DEFUN UPDATE-USER-FILE ()
  (LET ((FILE (OPEN "<SYS>USER" '(UPDATE KEYED))))
    (PROG (COMMAND ID) ;command loop with quit command.
       LP (SETQ COMMAND (SYS-PROMPT "type Edit, Delete, Create, or Quit:"))
          (COND ((EQ COMMAND 'QUIT) (RETURN NIL)))
          (SETQ ID (SYS-PROMPT "User id:"))
          (CASEQ COMMAND
            (EDIT (KEYED-WRITE FILE ID (EDIT-USER-RECORD (KEYED-READ FILE ID))))
            (DELETE (KEYED-DELETE FILE ID))
            (CREATE (KEYED-WRITE FILE ID (EDIT-USER-RECORD
              (MAKE-USER-RECORD 'USER-ID ID 'USER-PASSWORD "" 'USER-BALANCE 0
                                'USER-NAME "" 'USER-ADDRESS ""))))))
          (GO LP))
    (CLOSE FILE)))
(DEFUN EDIT-USER-RECORD (RECORD)
  (FORMAT T "~|Id: ~A~%Password: ~A~%Balance: ~D~%Name: ~A~%Address: ~A~%"
          (USER-ID RECORD)
          (USER-PASSWORD RECORD)
          (USER-BALANCE RECORD)
          (USER-NAME RECORD)
          (USER-ADDRESS RECORD))
  (PROG (COMMAND VALUE) ;command loop with quit command.
     LP (SETQ COMMAND (SYS-PROMPT "Field Name (or Quit):"))
        (COND ((EQ COMMAND 'QUIT) (RETURN NIL)))
        (SETQ VALUE (SYS-PROMPT "New Value:"))
        (CASEQ COMMAND
          (NAME (SETF (USER-NAME RECORD) VALUE))
          (ADDRESS (SETF (USER-ADDRESS RECORD) VALUE))
          (PASSWORD (SETF (USER-PASSWORD RECORD) VALUE))
          (BALANCE (SETF (USER-BALANCE RECORD) VALUE)))
        (GO LP))
   RECORD)
```

The PA assumes that every master file system will have an interactive update program and it knows a lot about what this program should look like. The first thing it knows is that the structure of the program varies greatly depending on the kind of terminal which will be used: an intelligent terminal which can do some of the work itself, a fast display terminal that can support rapid modifications to arbitrary parts of the screen, or a slow terminal where interactions are restricted to a line at a time. In order to find out what kind of terminal is intended, the PA asks the programmer.

When the PA's general knowledge about update programs is combined with what it knows about the user accounts system in particular, the result is code which has no unfilled roles at all. Notice that the code has a two-level nested structure. The top level routine, UPDATE-USER-FILE, contains a command loop with three commands, EDIT, DELETE, and CREATE, each of which selects a record in the file to be acted upon in the appropriate way. The nested routine, EDIT-USER-RECORD, is used to modify an individual record. It prints out the record on the terminal and then lets the user change the fields one at a time. Each of these routines contains an instance of a *command loop*. This is an example of the PA automatically tailoring a standard programming cliché according to the constraints of two different usages.

```
> Replace each command in edit-user-record with its first letter.
```

```
(DEFUN EDIT-USER-RECORD (RECORD)
  (FORMAT T "~|Id: ~A~%Password: ~A~%Balance: ~D~%Name: ~A~%Address: ~A~%"
          (USER-ID RECORD)
          (USER-PASSWORD RECORD)
          (USER-BALANCE RECORD)
          (USER-NAME RECORD)
          (USER-ADDRESS RECORD))
  (PROG (COMMAND VALUE) ;command loop with quit command.
     LP (SETQ COMMAND (SYS-PROMPT "First Letter of Field Name (or Q to Quit):"))
        (COND ((EQ COMMAND 'Q) (RETURN NIL)))
        (SETQ VALUE (SYS-PROMPT "New Value:"))
        (CASEQ COMMAND
          (N (SETF (USER-NAME RECORD) VALUE))
          (A (SETF (USER-ADDRESS RECORD) VALUE))
          (P (SETF (USER-PASSWORD RECORD) VALUE))
          (B (SETF (USER-BALANCE RECORD) VALUE)))
        (GO LP))
   RECORD)
```

Looking at the code for the update program, the programmer find it satisfactory except for the names of the commands in EDIT-USER-RECORD, which are too long (the PA by default used the names of the fields). He therefore reduces them each to a single letter.

[The programmer modifies the logout program to charge for connect time.]

The programmer has finished a first pass through the implementation of the new user accounts system. He is now ready to make a second pass, paying attention to some details that were suppressed the first time through.

```
> Consider exceptional conditions.
EXCEPTIONAL CONDITIONS:
 (1) MISSING FILE (OPEN) -- USER-LOGIN-TEST, UPDATE-USER-FILE,
                            REPORT-OVERDRAWN-USERS, LOGOUT.
 (2) MISSING RECORD (KEYED-READ) -- USER-LOGIN-TEST, UPDATE-USER-FILE, LOGOUT.
 (3) READ/WRITE CONFLICT -- UPDATE-USER-FILE, LOGOUT.
 (4) INVALID COMMAND (COMMAND LOOP) -- UPDATE-USER-FILE, EDIT-USER-RECORD.
 (5) INVALID FIELD VALUE (USER RECORD)-- EDIT-USER-RECORD.
                                                  -- SHOW USER-LOGIN-TEST? yes
```

```
(DEFUN USER-LOGIN-TEST (ID)     ;merge of SYS-QUERY-PASSWORD and USER-OVERDRAWN.
  (LET ((FILE (OPEN "<SYS>USER" '(IN KEYED))))
    (COND ((NULL FILE) ..missing-file-handler..)
          (T (LET ((RECORD (KEYED-READ FILE ID)))
               (CLOSE FILE)
               (AND RECORD
                    (EQUAL (SYS-PROMPT-WITHOUT-ECHO "PASSWORD:")
                           (USER-PASSWORD RECORD))
                    (NOT (MINUSP (USER-BALANCE RECORD)))))))))
```

In the first part of the scenario, the programmer ignored many details such as the fact that OPEN and KEYED-READ can fail (in which case they return NIL). The PA supported this abstraction by suppressing the code concerned with these conditions in its displays. This made it easier for the programmer to concentrate on what he wanted the programs to do under normal conditions. Now the programmer is ready to start considering exceptional conditions. The PA responds by listing the places where such conditions have been ignored.

The display of USER-LOGIN-TEST above shows two underlined sections of code concerned with error handling which were suppressed until now. The first section tests whether the open failed and includes an unspecified action to take when it does. The other specifies that USER-LOGIN-TEST should return NIL if there is no record corresponding to the input user id. USER-LOGIN-TEST is a combination of the two functions USER-OVERDRAWN and SYS-QUERY-PASSWORD. In order to understand exactly where the error handling code above comes from, we need to examine the error handling code in the two original functions, as shown underlined below. (In the previously displayed versions of these functions, this detail was suppressed.)

```
(DEFUN USER-OVERDRAWN (ID)
  (LET ((FILE (OPEN "<SYS>USER" '(IN KEYED))))
    (COND ((NULL FILE) ..missing-file-handler..)
          (T (LET ((RECORD (KEYED-READ FILE ID)))
               (CLOSE FILE)
               (COND ((NULL RECORD) ..missing-record-handler..)
                     (T (MINUSP (USER-BALANCE RECORD)))))))))

(DEFUN SYS-QUERY-PASSWORD (ID)
  (LET ((FILE (OPEN "<SYS>USER" '(IN KEYED))))
    (COND ((NULL FILE) ..missing-file-handler..)
          (T (LET ((RECORD (KEYED-READ FILE ID)))
               (CLOSE FILE)
               (AND RECORD
                    (EQUAL (SYS-PROMPT-WITHOUT-ECHO "PASSWORD:")
                           (SYS-PASSWORD RECORD))))))))
```

When USER-OVERDRAWN was first created, the PA inserted the standard tests for failure of OPEN and KEYED-READ with unspecified error handlers. When SYS-QUERY-PASSWORD was converted to use the user master file instead of the system password file, a missing file test with an unspecified handler was added because the PA did not assume that the user file would be handled the same way as the system password file. SYS-QUERY-PASSWORD already had a test which returned NIL when no record with a given id was found. This test is moved to USER-LOGIN-TEST in preference to the unspecified missing record handler from USER-OVERDRAWN.

```
> Check that the master file exists at system startup and
  protect it from deletion.

MODIFICATIONS:
  (1) FILE EXISTS CHECK ADDED TO SYS-STARTUP.
  (2) <SYS>USER MARKED AS NON-DELETABLE.
  (3) MISSING FILE TESTS REMOVED FROM USER-LOGIN-TEST, UPDATE-USER-FILE,
      REPORT-OVERDRAWN-USERS, AND LOGOUT.     -- SHOW SYS-STARTUP? no
```

```
(DEFUN USER-LOGIN-TEST (ID)      ;merge of SYS-QUERY-PASSWORD and USER-OVERDRAWN.
  (LET* ((FILE (OPEN "<SYS>USER" '(IN KEYED)))
         ;file checked at startup and protected from deletion.
         (RECORD (KEYED-READ FILE ID)))
    (CLOSE FILE)
    (AND RECORD
         (EQUAL (SYS-PROMPT-WITHOUT-ECHO "PASSWORD:")
                (USER-PASSWORD RECORD))
         (NOT (MINUSP (USER-BALANCE RECORD))))))
```

The programmer instructs the PA to add code to the operating system to check that the master file exists at system startup, and to protect it from subsequent deletion (e.g. by setting a a file descriptor bit). The PA deduces that this means that OPEN's on the user file cannot fail and therefore removes all the relevant exception handling.

[The programmer completes the system and it goes into use.]

Much later someone decides to add a project hierarchy to the user data base. Specifically, a new field is to be added to each user record which contains the user id of that person's project leader. A programmer (most likely different from the one above) is given the task of making this change and also creating a new per-project report on overdrawn users.

```
> Describe user accounts.

USER-ACCOUNTS IS A MASTER FILE SYSTEM
   MASTER FILE: <SYS>USER
      KEY: ID
      FIELDS: NAME, ADDRESS, BALANCE, PASSWORD
   READ-ONLY ROUTINES: USER-LOGIN-TEST
      REPORTS: REPORT-OVERDRAWN-USERS
   UPDATE ROUTINES: UPDATE-USER-FILE, LOGOUT
```

The PA can help the new programmer by giving him a top level summary of the structure of the existing user accounts system. It can also go into much more detail if requested. Notice that the division of the various system programs into those which only read the master file and those which read and write (update) the master file was not something the original programmer ever had to tell the PA. This division is part of the PA's knowledge of master file systems in general. Furthermore, the PA was able to analyze the programs the programmer wrote to determine which fell into each category.

```
> There is a tree structure on users.
> A user's project leader is the user id of its parent node in the user tree.

PROBLEM: ADDING A PROJECT-LEADER FIELD IMPLIES ADDING THE COMMAND "P" TO THE
   COMMAND LOOP IN EDIT-USER RECORD, SINCE THE FIRST LETTER OF EACH FIELD NAME
   IS A COMMAND.  THIS VIOLATES THE CONSTRAINT THAT A NEW COMMAND MUST NOT BE
   EQUAL TO ANY EXISTING COMMAND.  EXISTING COMMANDS ARE:
   N(NAME), A(ADDRESS), B(BALANCE), P(PASSWORD), Q(QUIT).
                                          -- SHOW EDIT-USER-RECORD? no

> Rename project leader to supervisor.

MODIFICATIONS:
 (1) A SUPERVISOR FIELD HAS BEEN ADDED TO THE USER MASTER FILE.
 (2) EDIT-USER-RECORD HAS BEEN MODIFIED SO THAT IT CAN UPDATE THIS FIELD.
```

The programmer specifies that the tree structure on users is to be encoded using *up* pointers stored in a field in each record. As a consequence of adding the new field, the PA runs into a problem. Because the first letter of each field name is a command in EDIT-USER-RECORD (the PA remembers this as a constraint rather than a one-time change), adding a field named PROJECT-LEADER violates the constraint on command loops that all of the commands must be different. The PA keeps track of a large number of constraints of this type and reports a bug whenever one is violated. In this case the programmer takes the expedient of simply changing the name of the new field so that its first letter doesn't conflict with any of the existing commands.

```
> Construct a program based on the overdrawn users report.
> Copy the master file into memory using property lists.
> Invert the supervisor tree.
> Change the loop which selects overdrawn users so that it only enumerates
  the subtree rooted at a given user id.
```

```
(DEFUN REPORT-OVERDRAWN-USERS-SUBTREE (ID)
  (LET ((RECORDS (SELECT-OVERDRAWN-USERS-SUBTREE ID))
        (REPORT (OPEN "PRINTER" 'OUT)))
    (SETQ RECORDS (SORT RECORDS
                        '(LAMBDA (RECORD1 RECORD2)
                           (SYS-NAME-LESSP (USER-NAME RECORD1) (USER-NAME RECORD2)))))
    (FORMAT REPORT "~5% Overdrawn Users as of ~A~|" (DATE))
    (MAPC '(LAMBDA (RECORD)
             (FORMAT REPORT "~5D ~8,T ~A ~25,T ~A %"
                     (USER-BALANCE RECORD)
                     (USER-NAME RECORD)
                     (USER-ADDRESS RECORD)))
          RECORDS)
    (CLOSE REPORT)))

(DEFUN SELECT-OVERDRAWN-USERS-SUBTREE (ID)
  (LET ((*RECORDS* NIL) ;special.
        (FILE (OPEN "<SYS>USER" '(IN SEQUENTIAL))))
    (PROG (RECORD)
       LP (SETQ RECORD (READ FILE))
          (COND ((NULL RECORD) (RETURN NIL)))
          (PUTPROP (USER-ID RECORD) RECORD 'USER-RECORD) ;copy file in memory.
          (PUTPROP (USER-SUPERVISOR RECORD)              ;invert supervisor tree.
                   (CONS (USER-ID RECORD)
                         (GET (USER-SUPERVISOR RECORD) 'SUPERVISOR-OF))
                   'SUPERVISOR-OF)
          (GO LP))
    (CLOSE FILE)
    (SELECT-OVERDRAWN-USERS-SUBTREE1 ID)
    *RECORDS*))

(DEFUN SELECT-OVERDRAWN-USERS-SUBTREE1 (ID)
  (LET ((RECORD (GET ID 'USER-RECORD)))
    (COND ((MINUSP (USER-BALANCE RECORD)) (SETQ *RECORDS* (CONS RECORD *RECORDS*))))
    (MAPC 'SELECT-OVERDRAWN-USERS-SUBTREE1 (GET ID 'SUPERVISOR-OF))))
```

As the last step of the scenario, the programmer constructs the required new report program. He decides to start with the code for REPORT-OVERDRAWN-USERS and modify it in order to get the new report he wants. Comparison of the program above with REPORT-OVERDRAWN-USERS shows that the key changes are confined to the SELECT-OVERDRAWN-USERS-SUBTREE subroutine.

The new report program is derived from the old one in three steps. First, the programmer specifies that the user file should be read into memory and stored using property lists. Next, the programmer specifies that the supervisor tree links should be inverted. This part of the scenario is the most in-depth demonstration of the PA's understanding of common data structures and algorithms. Finally, the programmer specifies that instead of enumerating every record in the file, only those records in the subtree with a given user id as root are to be enumerated. The PA makes the given user id be an input to the report and defines a new recursive subroutine, SELECT-OVERDRAWN-USERS-SUBTREE1, which traverses the subtree starting at the given user id, accumulating the selected records in the special variable *RECORDS*. As before, the name for the new report program is chosen to reflect the users selected.

Section 2 - Accomplishments to Date

This section reviews the progress we have already made towards achieving the level of performance illustrated by the scenario. This includes several developments in the underlying theory and an initial feasibility demonstration.

The theoretical component of our research centers on two fundamental questions: what knowledge (about a particular program and about programming in general) does a programmer employ in scenarios such as the one above; and how should this knowledge be represented so that it can be effectively manipulated in a computer.

The Plan Calculus

Our first step was to develop the notion of the *plan* for a program. Like blueprints and block diagrams in other engineering disciplines, the plan for a program is a centralized record of many different kinds of information useful for design and maintenance. Some of this information (such as the flow of data and control between the parts of a program) is evident in the program text. Other information (such as the logical constraints between parts) is conspicuously absent from most program text. Furthermore, the plan describes the program at various levels of abstraction, spanning the spectrum from abstract specifications to the implementation language.

Our earlier proposal [32] describes an initial plan representation. A plan in this representation is essentially like a flow chart, except that data flow as well as control flow is represented by explicit arcs. Using this representation, many common programming cliché's such as report programs, command loops and tree enumeration can be expressed in a canonical, programming language independent fashion.

More recently [33] the plan representation has been refined into a formal *plan calculus* with an underlying axiomatic semantics [37]. The axioms for plans are written in a situational calculus, a variant of predicate calculus similar to the one used by Green [15] and McCarthy and Hayes [26], in which certain variables and constants (called *situations*) are taken to denote different states of a computation. This provides a formal system within which rules of inference, equivalence, and other relations on plans can be defined in a rigorous way.

In addition to providing a formal basis for plans, Rich extended the representation in two important ways. First, the plan notion was generalized to include data abstraction as well as control abstraction in a unified framework. Second, a mechanism was introduced for representing multiple points of view.

A plan is defined formally by a set of named, typed parts (called *roles*) and constraints between them. When the roles of a plan are data objects and the constraints are logical invariants, it is called a *data plan*. A data plan expresses a data abstraction similar to a data algebra [17,14] or an abstract data type [23]. For example, *indexed-vector* is a data plan with two roles, named *base* (of type vector) and *index* (of type integer), and a constraint requiring that the index must be within the bounds of the base vector. Indexed-vector is a common data structure configuration which is used in the implementation of many other data abstractions such as stacks, buffers and queues.

In contrast with the algebraic and abstract data type approaches, data structures in the plan calculus are *mutable*. Although this introduces extra complexity into both the formalization of data plans and reasoning about them, we feel that side-effects are an important feature of real programming which should not be ignored. For example, the most natural way to think of the effect of the EDIT command in UPDATE-USER-FILE

in the scenario is as a side-effect to the file.[1]

When the roles of a plan are operations and tests, and the constraints are control flow and data flow connections between them, it is called a *temporal plan.* Temporal plans correspond to our original flowchart-like idea of a plan. Data plans and temporal plans can be combined in order to express standard programming forms which include both data and control abstraction. For example, the prototypical structure of a master file system is expressed by a plan with a role for the master file (which is a data plan) and roles for various temporal plans which typically go along with it, such as report generators, and the interactive file update program.

Multiple Points of View

In the original plan representation, as in most knowledge representations, much use is made of the ability to view an individual object as an instance of a more general abstract class. It was not however possible in the original plan representation for a single object to be viewed as an instance of more than one class. For example, it is often useful to view a single data object as an instance of several different data abstractions. In particular, a Lisp list can be viewed alternatively as a singly recursive data structure, as a labelled directed graph, or as a sequence. Each of these points of view emphasizes a different aspect of a Lisp list. Each one provides a different vocabulary for specifying properties of the data structure and a standard set of manipulations on it:

Singly recursive data structure -- a data structure with two roles, head and tail, where the tail is defined recursively. Two standard operations on this data abstraction are push and pop. A Lisp list can be viewed as a singly recursive data structure by taking the CAR of the list to be its head and the CDR to be its tail. This viewpoint is appropriate for understanding CONS and CDR as the implementation of push and pop.

Directed graph -- a set of nodes and an edge relation. Two standard manipulations on directed graphs are splicing in and splicing out nodes. A Lisp list can be viewed as a directed graph by taking the individual CONS cells to be nodes connected by the edge relation CDR. This is the natural viewpoint for understanding RPLACD as the implementation of a splicing operation.

Sequence -- a mapping from the natural numbers to some set of objects. A list can be viewed as a sequence in which the first element of the list is the first element of the sequence, etc. This point of view is the most natural one in which to specify ordering properties on a list.

Rich added to the plan calculus the notion of a formal mapping between plans, called an *overlay*, which can be used to represent multiple points of view such as these. For each of the three views of a list above, there is a separate overlay which specifies how properties of a data structure viewed one way correspond with properties expressed in another view. Furthermore, overlays are specified formally in such a way that several may be used simultaneously for the same object. For example, all three may be used to describe how a single list is manipulated at different points in a single Lisp program.

---

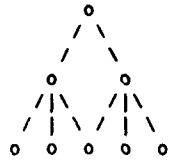
1 Recent work on functional programming has shown how certain side-effects of this type can be modelled with a primitive non-deterministic merge operation added to an otherwise side-effect free system [20].

Overlapping Implementations

A closely related kind of multiple viewpoints occurs when there is overlap between the implementations of different procedural abstractions, as illustrated by the program below.

```
(DEFUN MAX-MIN (L)
  (LET ((MAX (CAR L))
        (MIN (CAR L)))
    (MAPC '(LAMBDA (N) (COND ((> N MAX) (SETQ MAX N)))
                       (COND ((< N MIN) (SETQ MIN N))))
          (CDR L))
    (CONS MAX MIN)))
```

The program MAX-MIN computes both the maximum and the minimum of a non-empty list of numbers. The standard plan for finding the maximum (or minimum) element of a list has three principal parts: an initialization (here (CAR L)), an enumeration of the elements of the list (here MAPC), and an accumulation which tests each element to see if it is the largest (or smallest) found so far. The diagram below indicates how MAX-MIN can be analyzed in terms of this plan.



The top node in this diagram represents the entire program. At the next level, the program is viewed as the combination of two plans, one which finds the maximum and one which finds the minimum. The third level shows how the more primitive components of the program are grouped and viewed as the implementation of these two plans. There are only five nodes at this level rather than six because the list enumeration is shared between the implementation of maximum and of minimum. It must be simultaneously viewed as filling a role in both plans.

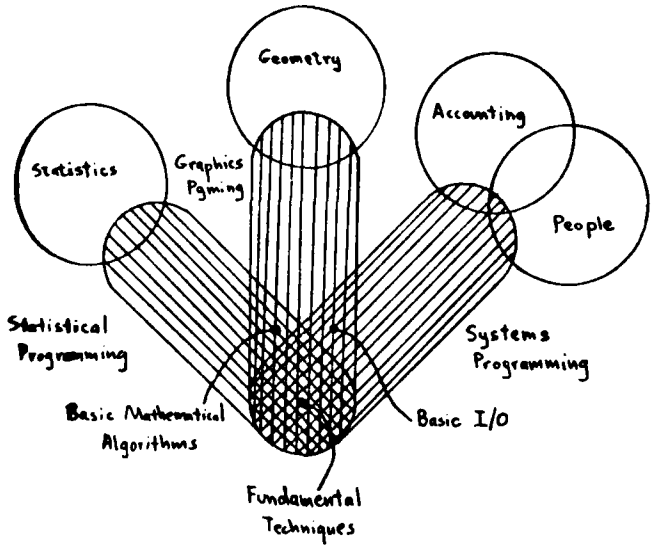
This type of analysis is a violation of strictly hierarchical decomposition, which is currently the dominant technique in program design. We have found, however, that it is not always possible to maintain a strictly hierarchical analysis and at the same time capture the appropriate generalizations.

The Plan Library

Using the plan calculus, we have begun the task of building up a library of standard plans in various areas of programming. Fig. 1 shows our view of how this knowledge is typically factored. The three large ovals in the figure represent three general areas of programming: statistics, graphics, and systems programming. The intersection of all three ovals in the center represents fundamental programming techniques; the areas of overlap between each pair of ovals represent programming knowledge of intermediate breadth; the remaining area in each oval represents specialized knowledge in that area.

Our initial focus of attention in the library has been on fundamental programming techniques. We have at this point codified several hundred plans in this area [33], for example: abstract input-output specifications, such as set addition, removal from a graph, and inverting a mapping; data plans involving integers, sets, sequences, trees, labelled directed graphs, and hashing; and temporal plans, such as generating, searching, splicing out, maximum, retrieve (by key), and discriminate and retrieve (in a hash table).

Fig. 1. The Factorization of Programming Knowledge



Examples of programming knowledge of intermediate breadth are basic input-output techniques (e.g. files, streams and displays), which are used in graphics and systems programming, but not statistics; and basic mathematical algorithms (e.g. successive approximation and matrix manipulations), which are used in statistics and graphics, but not systems programming.

Finally, there is programming knowledge which is specific to just one narrow application area, such as the plans associated specifically with operating systems, statistical or graphics..

As is evident from the scenario, understanding programs also often requires knowledge in domains which intrinsically have nothing to do with programming (signified by circles in the figure), such as branches of mathematics, business, and knowledge about people, places and things in the real world. (Neighbors [27] calls these the "problem domains"). We propose to do a minimum of representation work in these areas, since many other researchers in artificial intelligence are already working on them.

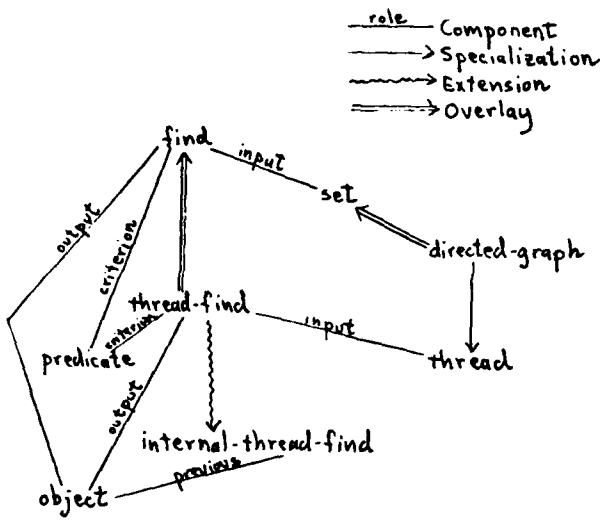
The many plans in the library are interconnected so that they can be efficiently retrieved and reasoned about. There are currently four kinds of taxonomic relations which connect plans in the library as illustrated by the example in Fig. 2. (For a detailed discussion of the plans referred to in this figure, see [37]).

The first relation (*component*) indexes which plans are used as components in the definition of other plans. This relation is particularly useful in program analysis For example, if some data object has been determined to be a set, then this relation can be used to retrieve various plans and specifications which involve sets (for example, the find operation).

The next two relations in Fig. 2 are inheritance relationships by which a plan inherits the roles and constraints of another plan and then adds additional constraints (*specialization*), or additional roles and constraints between them (*extension*). These inheritance relations greatly reduce redundancy in the library. In addition, these links can be used to retrieve the standard variations of a plan which might be useful during program synthesis.

Finally, *overlays* between plans in the library encode the details of how one abstraction may be implemented (or viewed) in terms of another and the conditions under which this is possible. Many of a programmer's design decisions are equivalent to deciding to use one overlay rather than another. The overlays themselves are used both to record the decisions made and to evaluate whether or not the decision

Fig. 2. An Example of the Taxonomic Relationships in the Plan Library



leads to a contradiction. Together with the specialization and extension links, overlays are also involved in propagating design decisions. For example, these links can be used to automatically select different variations of a plan depending on which implementation of a data object is chosen.

Knowledge Based Editing

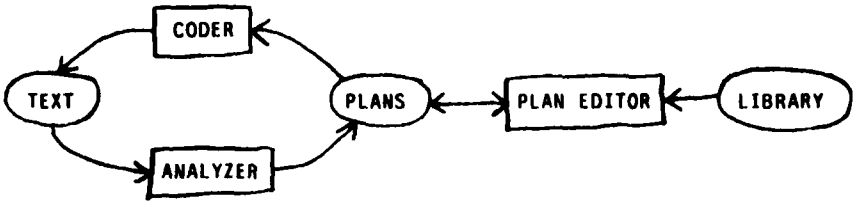
Starting with the program analysis system described in [48] and [49], Waters constructed a knowledge based editing system which demonstrates the feasibility of some of the basic capabilities of the PA. The architecture of this system is shown in Fig. 3. (The system is described in more detail in [51])

The important internal operations of this system are entirely based on the plan representation. However, it must be noted that since this system is the culmination of work begun several years ago, it uses the original plan representation described in [30] and [48] rather than the extended plan calculus discussed above. Similarly, although there is a library containing plans for a number of basic programming cliché's, there are no taxonomic links between the plans.

The central module in the system is the *plan editor* (see Fig. 3). The interface presented to the user is similar to a programming language structure editor such as Mentor [8] or Gandalf [12], in the sense that it treats the program as a hierarchical structure rather than merely a text string. However, many of the facilities offered by the plan editor are very different because it operates on the plan for a program rather than on its parse tree. The plan editor allows the programmer to construct a program by instantiating plans from the library and to modify a program by modifying its plan. We call this *knowledge based* editing because it makes use of knowledge in the library and in the plan for the program which is only implicit in the program text.

Whenever the programmer uses the plan editor to modify the plan for a program, the *coder* module is invoked to create program text (in Lisp) corresponding to the resulting plan. The programmer can also use an ordinary text editor to modify a program. In this case the *analyzer* module is used to create an updated plan so that the programmer can continue with the plan editor. Analyzers have been implemented for subsets of Fortran and Cobol, as well as for Lisp. The ability to work with several different languages demonstrates the language independence of the technology being developed here.

Fig. 3. Architecture of the Knowledge Based Editor.



The current analyzer module is a restricted version of our eventual goal in that it analyzes programs in terms of only a few basic plan fragments. This analysis decomposes a program into useful units and makes it possible to use the knowledge based editor on a program which was not originally constructing using the system. However, it does not make it possible to bring the full force of knowledge in the library to bear on such a program. Section 3 describes an improved analyzer which will be able to analyze a program in terms of all of the plans in the library.

Of the seven main facilities of the PA discussed at the beginning of the scenario, the knowledge based program editor more or less completely provides two of them (programming with cliché's and escape to the surrounding environment) and provides parts of two more (flexible display and automatic modification).

The most important facility provided by the current system is programming with cliché's. The library is capable of representing many complex program forms and the plan editor provides commands enabling the programmer to manipulate them in essentially the manner illustrated in the scenario.

Due to the existence of the coder and analyzer modules, the current system is also able to provide for escape to the surrounding environment. The programmer has available at all times an up-to-date version of the code being worked on, which he can then manipulate with other tools in the programming environment. Similarly he can take a program produced with some other tool and introduce it into the knowledge based editing system.

The current system partially addresses the issue of flexible display in that the coder is capable of producing the kind of sketchy code shown in the scenario for incomplete plans. However, the current coder does not have the ability to suppress classes of details, such as error handling. The current system provides some facilities which partially automate modification. For example, the plan editor provides several commands (such as "share these two roles in the plan") which the programmer can use to modify a program more easily than editing the text directly.

As an example to compare the current knowledge based editor with our goal, consider the program REPORT-OVERDRAWN-USERS. In the scenario this program is constructed via the following commands:

```
> Construct a report program.
> Select the users with overdrawn balances.
> Print each user's name, address, and balance.
> Sort the selected users alphabetically by name.
> Do not print headings.
> Edit title format.          [Programmer directly edits the format]
```

Using the current system the programmer could say the following. (The plan editor implements a simple English-like command syntax which is capable of processing these commands exactly as shown.)

```
> Define a program SELECT-OVERDRAWN-USERS.
> Implement it as a report.
> Implement the input-file as '"<SYS>USER".
> Implement the selection-predicate as overdrawn of '(USER-BALANCE RECORD).
> Implement the print-line as '(FORMAT REPORT "~A ~A $~D"
  (USER-NAME RECORD) (USER-ADDRESS RECORD) (USER-BALANCE RECORD)).
> Add a sort of the output of the filter by '(LAMBDA (RECORD1 RECORD2)
  (SYS-NAME-LESSP (USER-NAME RECORD1) (USER-NAME RECORD2))).
> Remove the printout of headings.
> Edit title format.          [Programmer directly edits the format]
```

The principal difference between the two sets of commands above is that there is no automatic propagation of information in the current system. There is also no knowledge about data structures and no bug detection. It is left completely up to the programmer to make consistent design choices.

This means, for example, that although in the current system the programmer can invoke the same plan for a report program that was shown in the scenario, none of the roles are filled in automatically. Also, when it comes to specifying the selection predicate and the print line in the current system, the programmer has to use literal text because there are no data plans. Finally, the current system does not know how to automatically add new fields to a file when they are mentioned.

Other features of the scenario which the current system *is* capable of supporting include: breaking the single loop in the SELECT-OVERDRAWN-USERS into two loops when the sort is introduced (the current coder would not break the program into two subroutines however); and removing the calculation of PAGENO and LINES when the printing of headings is removed.

Other Applications of Plans

Faust recently completed a Master's thesis [11] exploring the application of the ideas described above to the problem of maintaining existing software written in Cobol. He implemented a system based on the analyzer module of the knowledge based editor which shows the feasibility of automatically translating Cobol programs into a high level business data processing language called Hibol [38]. Hibol is a single assignment language without explicit looping constructs in which data processing applications are specified by a set of simple statements defining how the values in one data set are to be derived from other data sets. Once a program has been converted from Cobol to Hibol, it can be more easily maintained because the Hibol program embodies the functional specifications of the program much more clearly.

Frank (a visiting scientist from Schlumberger-Doll Research) conducted an initial exploration [13] into the feasibility of automatically generating program documentation based on recognition of standard programming cliche's. His study pointed out issues for future research which are currently being pursued by other members of the group.

For her Master's thesis, Steele [45] constructed a source-to-source program transformation system which keeps track of the reasons why each transformation is or is not applied at particular points in the program, and of the exact effect of each transformation. Her system does not use the plan representation; however, it focuses on several issues which are of fundamental importance to the PA. In particular it is very important for the PA to keep track of the decisions, modifications and optimizations it performs and the reasons behind them so that it can justify them to the user and so that it can determine what to do when a decision is changed.

Finally, Shapiro [41] has completed a Master's thesis in which he developed an interactive debugging aid which exhibits a deep understanding of a narrow class of bugs. The debugging knowledge in this system is organized as a collection of independent experts which know about particular bugs. These experts operate by applying a feature recognition process to the plan for a program and to the events which took place during

the execution of the code. Shapiro's system, like Faust's mentioned above, makes use of the analysis module of the knowledge based editor system.

Section 3 - Ongoing Research

The main focus of our current research is extending the knowledge based editor described above to take advantage of the improved plan calculus and plan library. Our first milestone in this work will be to complete implementation of two modules which demonstrate the utility and generality of the new plan library. The first module supports automated program synthesis using the library; the second one automatically analyzes programs according to plans in the library. It is an important feature of our methodology to study programming cliches from the perspective of both synthesis and analysis. These two modules will then be combined with the existing knowledge based editing system to become a second level feasibility demonstration for the PA as a whole.

Synthesis by Inspection

There have been many approaches to automating program synthesis. For example, powerful deductive methods have been applied to construct small but potentially quite complex algorithms from examples [19,42,44] or directly from specifications [24]. Another general approach, typically applied to the synthesis of very large programs, are so-called "configurable" programs [47] in which one of a class of programs for some particular application (typically in business data processing) is assembled out of a parameterized super-program by specifying the values of some precomputed set of switches or options.

Another major approach to program synthesis is program transformations, of both the correctness preserving and knowledge based variety. Work on correctness preserving transformations, such as the folding-unfolding transformations of Burstall and Darlington [4], tends to focus on a small set of very general transformations which are formally adequate to derive a large class of programs, but which must be composed in long sequences to achieve intuitively natural implementation steps. The knowledge based approach to program transformations, as exemplified by Barstow [2], Balzer [1] and Cheatham [6], de-emphasizes formal program correctness in favor of representing larger transformations which are more specific to the particular program being developed.

We feel that none of these approaches (with the exception of knowledge based program transformations) directly addresses the heart of the current software crisis, i.e. the construction of programs (or systems of programs) which are large and varied, but fundamentally straightforward. In our view, the method which gives the most leverage on this kind of programming is *synthesis by inspection*. By this we mean synthesis based on recognizing standard specification forms and choosing one of the corresponding standard implementation alternatives. The plan library, with its taxonomy of standard input-output specifications, data plans, temporal plans, and overlays between them is designed to support this synthesis method.

In our initial study of synthesis by inspection, we are separating the task of recognizing possible implementations from the task of choosing the appropriate one, because we believe the first task is easier. Thus our initial synthesis module will be interactive. In response to a request from the programmer it will produce a *menu* of implementation options for the part of the program currently in focus, as illustrated in the construction of UPDATE-USER-FILE in the scenario. Once the programmer makes a choice, the details of the implementation can be carried out automatically. The final result of this process is a detailed implementation plan which can be translated by the coder module into program text.

The selection of implementation alternatives takes advantage of the fact that the program under construction always has a plan, parts of which are more and less abstract. Overlays in the plan library give possible implementations of abstract plans and specifications in terms of more concrete ones (i.e. closer to the abstract machine provided by the target programming language). Finding implementation alternatives in this framework boils down to looking up parts of the current plan for a program in the library and retrieving possible implementations via overlays. One use of this module will be to "shake down" the contents of the plan library.

There are several directions in which this first version of synthesis by inspection will be further improved. It can be modified to take more initiative and suggest what part of the program might be implemented next. This will require the integration of the synthesis module with the improved analysis module described below. Another extension will be to reduce the number of options presented at a given choice point by eliminating those which are clearly inappropriate. In many situations the number of choices can be narrowed to one so that the system can proceed automatically. This improvement will require the addition of some simple deductive capabilities (discussed further in the next section) so that the choice of a particular option at one point in the program constrains the options available at other points.

Analysis by Inspection

Brotsky [3] is currently working on a module which is the inverse of the synthesis module described above. Given a fully implemented program, this module constructs a hierarchical plan for it which identifies instances of plans from the library at all levels of abstraction and the implementation relationships between them. We call this process *analysis by inspection* because it derives new knowledge about a program by recognizing standard forms rather than by applying general reasoning techniques, such as symbolic evaluation [5].

Analysis by inspection is an important part of our research for several reasons. First, we are interested in this kind of analysis from a theoretical point of view because it is a characteristic behavior of expert human programmers. For example, expert programmers presented with a program they have not seen before are able to identify the use of standard programming cliché's in it, which seems to help them understand how the program works as a whole. Also from a theoretical point of view, we have found that using the same plan library to drive both synthesis and analysis has helped us discover better factorizations of knowledge in the library as compared to other approaches, such program transformations, which are used for synthesis only.

Analysis by inspection also has several important practical applications. The most obvious application is analyzing programs which have already been written by other means so that they can be further modified and maintained using the PA. Even in programming with the PA from the start, it is possible for a program to evolve to a point when it is profitable to re-analyze it differently from the way it was originally built. For example, the new analysis may decompose it in a way which better reflects its current logical structure. Re-analysis may also reveal the presence of a standard pattern for which some optimizations are known.

The operation of Brotsky's analysis module is based on treating the plan library as a *web grammar* [29], which is the natural extension of the formal string grammar idea to more general graph structures, such as plans. Each plan and overlay in the library becomes a grammar rule whose left-hand side is the name of the plan or overlay. The right-hand side is either the set of roles and constraints which define the plan (represented as nodes and arcs), or in the case of an overlay rule, the name of another plan which implements it or which it can be viewed as. Thus the plans in the library become non-terminal symbols in the grammar and the hierarchical analysis which identifies instances of library plans in a program is formally the parse tree of the program according to the library grammar. Note however that due to the phenomena of multiple

points of view and overlapping implementations discussed earlier in this section, we allow for multiple parses and joined nodes in the parse trees.

Brotsky's algorithm for parsing such grammars is also of interest. The basic strategy is to use standard bottom-up parsing techniques on each of the chains (linear subgraphs) in a plan, and then to assemble the results of these parses into a parse of the plan as a whole. It is fairly clear, however, that this approach by itself is combinatorically infeasible for the size of program and grammar we have in mind. An important part of this research is therefore to discover constraints on the implementation structure of a program which make efficient analysis by inspection possible. In other words, we do not expect to be able to analyze all programs by this method, only those which are "well-structured". It will be interesting to see the extent to which the constraints developed for this purpose agree with our intuitive notions of clear program structure.

An Improved Knowledge Based Editor

The analysis and synthesis by inspection modules described above, combined with the already completed knowledge based editing facilities described in Section 2, will constitute a second level feasibility demonstration for parts of the PA. This system will be an improvement over the first system in two principal ways.

First, the ability to program with cliché's will be greatly enhanced. The new plan library contains many more plans than the old system, especially plans for standard data structures. Furthermore, the use of menus by the interactive synthesis module will help make these plans easier for the programmer to access.

Second, escape to the surrounding environment will be made more effective. In the current system, the analysis module is not strong enough to completely re-analyze code returning from an editing escape with no loss of information. With the new analysis module the cost of escaping will be significantly reduced and in many cases eliminated entirely.

Other Applications of Plans

Duffey [10] is currently working on a Master's thesis to develop a novel compiler strategy for generating high quality machine code. This research is a first step towards formalizing the expertise of the assembly language programmer in much the same way as we have been formalizing the expertise of the expert high-level language programmer. He has found some of the same fundamental principles to be applicable. In particular, he is using the plan representation and a library of standard implementation techniques.

Chapman is currently working on an undergraduate project to develop an incremental program testing assistant, similar in some ways to Lieberman's Tinker system [22]. Chapman's system will keep track of test cases supplied by the programmer and try to decide which ones to run whenever a modification is made to the program. The system will also have a rudimentary capability to modify test cases automatically when certain classes of program modifications occur. In order to do these things, the system maintains some simple plans for the user's program including information about calling relationships, side-effects, and patterns of variable usage. The system will also include a library of standard test cases which can be tailored to a programmer's needs. We eventually plan to integrate the test case assistant into the rest of the PA. In particular, we think that bug detection might be greatly enhanced by a synergy between simple deduction and direct testing of parts of the program.

Section 4 - Proposed Future Research

This final section of the proposal describes research which remains to be done in order to further formalize the AID paradigm of programming and develop the PA system to support it. Three major theoretical issues in this research are: the use of reasoning and dependencies in program evolution, extensions to the plan calculus and plan library to represent more knowledge, and automating plan acquisition and learning.

On the implementation side, we propose a third demonstration system showing the feasibility of all of the capabilities of the PA illustrated in the scenario. In addition, as soon as the improved knowledge based editor described in Section 3 is completed, we propose to begin implementation of the first well-engineered prototype of the programmer's apprentice, based on this editor. The prototype PA will not have the full capabilities illustrated in the scenario, but it will give us the ability to explore important human interface issues.

Reasoning and Dependencies

Disciplined use of the AID paradigm in programming requires, among other things, the ability to reason about dependencies between the parts of a program (for example to change design decisions) and about program behavior under various conditions (for example to predict the effects of a modification). Recent work in artificial intelligence [9] provides a model for this kind of reasoning in the form of *truth maintenance systems.*

A truth maintenance system (TMS) is essentially a data base in which each assertion is accompanied by a *justification* stating which other assertions form the logical support for believing it to be true. The procedural part of the TMS automatically maintains consistency in this data base. For example, if an assertion is retracted (i.e. its truth value changed from true to false or unknown), then all of the assertions which depend on it are automatically retracted (unless there are independent justifications for believing them to be true). Similar propagation through the dependency network takes place when an assertion becomes true (where it was previously false or unknown). Also, as part of consistency checking, the TMS automatically detects contradictions, i.e. chains of dependencies which would lead to an assertion being both true and false at the same time.

Dependency directed reasoning makes it possible for incremental modifications to a program to cause only incremental changes in one's understanding of the program. For example, suppose that a programmer decides to change the implementation of an abstract set from an array to a binary tree. This entails replacing all the loops which enumerate the elements of the array to tree traversals which enumerate the nodes of the tree. Although the new code might appear superficially very different from the old code, Shrobe [43] shows how reasoning using plans and a TMS can handle this modification by an incremental rather than a complete re-analysis.

Dependency directed reasoning would also be useful for keeping track of the relationship between multiple versions of a program (which might co-exist) [7] and modifications made by multiple programmers (perhaps in parallel). For both of these applications, the ability, provided by the plan representation, to relate features of a program at varying grain size would be an improvement over existing practices.

We propose to develop a reasoning module for the PA which is similar in some ways to Shrobe's system. However, our module will be based on a more recent TMS developed by McAllester [25]. McAllester's TMS provides, in addition to the basic consistency checking and propagation facilities described above, the automatic application of some simple rules of inference, namely substitution and propositional resolution. In

McAllester's TMS, if two terms are asserted to be equal, then an automatic mechanism operates to get the effect of substituting one term for the other throughout the data base. Furthermore, this is done in such a way that the substitutions are undone if the assertion of equality is later retracted. Propositional resolution means, for example, that if the assertion A and the assertion (A implies B) are both believed true, then B will be believed true. Furthermore, the justification for B will be A and (A implies B) so that if either is later retracted, B will be retracted also.

It is important to note that we are not proposing to build a reasoning module which is a powerful theorem prover. For example, the TMS described above does not apply any rules of inference involving quantification or induction. Rather, the reasoning module is intended to support the flexible use of *straightforward* reasoning. This is consistent with our knowledge based approach. Wherever possible we try to include enough knowledge in plans and the plan library so that the reasoning the PA needs to do is straightforward.

Extending the Plan Calculus and the Plan Library

Accounting for all of the aspects of the AID paradigm illustrated in the scenario will require both quantitative and qualitative extensions to the programming knowledge we have been able to represent thus far in our research.

As discussed in Section 2 (see Fig. 1), the initial focus of our representation work has been on the fundamental programming cliché's which are shared between all different kinds of programming. We estimate that on the order of several thousand plans (we now have only several hundred) are needed in order to get reasonable coverage of this domain. In addition, we expect that the each of the more specific programming domains, such as I/O handling, basic mathematical algorithms, etc. also contains on the order of a thousand plans. Thus we feel that the construction of a comprehensive library is a formidable but not impossible task. To demonstrate the feasibility of this task we propose to extend the current plan library into at least one or two more areas of programming.

There are several kinds of knowledge illustrated in the scenario which require qualitative extensions to the plan calculus and library. For example, the concepts of control flow and data flow embedded in the current plan calculus give significant leverage in representing the fine-grain structure of programs. They are less useful, however, for representing the organization of large systems of programs. In order to write plans for systems, the plan calculus will be extended to include concepts such as interrupt-driven control structures and data communication via a globally shared data base.

The scenario also illustrated knowledge about standard types of modifications, such as the addition or deletion of a field from a record definition. One way of representing this knowledge is by specific procedures for each type of modification attached to the appropriate plans in the library. We would prefer, however, to represent at least some of this modification knowledge more explicitly so that it can be explained and reasoned about. One way to do this is to structure the plan library so that every modification can be expressed as a change to one or more design decisions.

A similar category of knowledge is that of standard bugs associated with different types of plans. Some bug detection can be performed by the general mechanism of detecting contradictions between different design decisions and tracing them back through the dependencies in the TMS. However, a knowledge based approach can often yield a more explanatory (to the programmer) diagnosis of the bug. For example, Shapiro [41] has identified some standard bugs associated with manipulating lists by side effect and shown how they can be represented as expert procedures.

A very important kind of knowledge exhibited by the PA in the scenario is knowledge of program efficiency considerations. For example, improved program efficiency is the underlying justification for optimizations that the PA performs. The current plan calculus is oriented towards representing what a program does, how it does it, and why it is correct; there is nothing to indicate why it does something one way as opposed to another. Some work has been done by Kant [21] in this area. We propose to investigate how to add this kind of information to the existing plan calculus and library. This knowledge will enable the synthesis module described above to make better decisions automatically.

Plan Acquisition and Learning

Adding a new plan to the plan library currently means writing out its formal definition in the plan calculus -- something we do not expect the average programmer to be able to do. We propose to explore two ways of automating this process.

A first tool which will make it practical for programmers to define new plans without having to know the details of the plan calculus is an extension of the knowledge based editing system described in Section 2. We envision allowing a programmer to define a new plan by starting with a well-formed program containing the cliché he has in mind, pruning away the parts of the plan for the program which are irrelevant to the cliché, and indicating where the logical gaps (roles) occur. This new tool will use the analyzer module of the knowledge based editing system to transform the initial program into the plan calculus, and use the plan editor to do the pruning.

Later, Brotsky intends to investigate how the PA might automatically learn new plans from experience. One idea is to look at programs which perform known functions with some parts which are unfamiliar. Working up from the bottom, Brotksy's analysis module described in Section 3 will be able to recognize those parts of the program which are doing standard things in standard ways. Working down from the specifications, it should also be possible to identify the portion of the overall function of the program which is not accounted for by the recognized parts of the program. A learning module could then reasonably hypothesize that the unrecognized part of the program is an instance of a new plan for implementing the leftover portion of the specifications.

For example, consider how the PA might be taught a new way of deleting an association from an *alist* (list of dotted pairs) in Lisp. The most abstract plan for implementing this operation in the current library has two steps, roughly, "find it" and "remove it". The "find it" step can be implemented by a loop which enumerates the list (with a trailing pointer) and searches for a pair which has the given key as its CAR. The standard plan in the library for implementing the remove step uses RPLACD to splice the found pair out of the list. Suppose that we now want to teach the PA a new technique, useful in some circumstances, of using RPLACA to change the key of the found pair to NIL, thereby causing the associated datum be ignored in all subsequent retrieval (assuming that NIL is not a valid key). We present the PA with the following program which uses this new technique.

```
(DEFUN FUNNY-DELETE (KEY ALIST)
  (PROG (L)
        (SETQ L ALIST)
     LP (COND ((NULL L)(RETURN NIL))
              ((EQUAL (CAAR L) KEY)
               (RPLACA (CAR L) NIL)
               (RETURN ALIST)))
        (SETQ L (CDR L))
        (GO LP)))
```

Brotsky's analysis module will be able to recognize all but the underlined portion of the program as an instance of the standard plan for finding a pair in an alist by enumeration and search. Being told that the overall function implements deletion from an alist, a learning module could then hypothesize that the underlined portion of the program is a new way to implement the missing "remove it" step. It might even be practical, now that the focus of attention has been narrowed to this small part of the program, to try to prove that the new plan is a correct implementation of the specifications of the removal step. Finally, the new plan would need to be abstracted somewhat from this example and installed at the appropriate place in the taxonomy of the library.

This work on plan learning may have interesting implications for computer science education, such as predicting an optimal order in which to teach various programming techniques.

Feasibility Demonstration of the Full Scenario

There are five new or improved facilities which have to be added to the knowledge based editor demonstration described above in order to create a feasibility demonstration of the full PA scenario: propagation of design decisions, bug detection, enhanced automatic modification, automatic optimization, and flexible display. This section briefly describes the work to be done in each of these areas.

The first four facilities listed above can all be supported by a TMS integrated with the plan library and the plan for the program being worked on. All plans, specifications, overlays, and constraints between them are encoded as assertions and dependencies in the TMS. Design decisions made by the programmer are also represented as assertions, with dependencies linking them to the reasons why they were made and the resulting changes in the plan. Further dependencies in the library specify the conditions under which a particular decision choice is forced to be true. Given this structure, propagation of design decisions happens as a consequence of the TMS's operations to guarantee consistency in the data base.

Bug detection is implemented in a similar way. The TMS signals a contradiction whenever an assertion is made which implies an incompatible truth value for an existing assertion. Starting from the new assertion and the contradicted assertion, the TMS traces back through dependencies to the programmer-originated assertions (such as design decisions) which underlie the contradiction. A bug report can then be generated which focuses on the underlying problem rather than on the superficial manifestation of the bug. For example, in the scenario the problem of duplicate commands in `EDIT-USER-RECORD` is traced to the programmer's addition of a new field name. Using this technique bugs can be detected much earlier than in present programming practice.

Automatic modification and optimization will initially be achieved by including special procedures for each kind of change. The TMS can support the operation of these procedures in several ways. For example, the TMS can be used to reason hypothetically about the effects of a modification before it is actually made. Also, as mentioned above, if a modification can be characterized as a change in design choice, the TMS can automatically propagate the consequences of the change.

Finally, to obtain the kind of flexible display at different levels of detail illustrated in the scenario will require some simple modifications to the existing coder module, as well as some additional annotation on plans indicating what can be ignored under what circumstances. The basic technology of this improvement is fairly straightforward; the interesting problem to be studied is just what kinds of abstraction are useful.

A Well-Engineered Prototype

There are a number of important issues about the programmer's apprentice which can only be addressed through experimentation, such as: what library plans are most natural and convenient to use, what is a good command language for the programmer to use, and to what extent should the PA take initiative in its interactions with the programmer. To investigate these issues we propose to develop a prototype system what can be used on a daily basis to construct and modify programs. This section describes some of the problems to be solved in order to construct such a prototype.

To start with, the PA has to be fully integrated with the rest of our local programming environment. The basis for this already exists because the PA can analyze and synthesize standard program text (in Lisp, which is the language of choice in our local environment). However, much is yet to be done to make the PA interact gracefully with our local file system, editor, debugger, compiler and interpreter. We propose to use the Lisp Machine programming environment [16] as the host for the PA since, among other factors, all of the Lisp Machine system programs are themselves written in Lisp.

Thus far in our research no effort has been expended towards making the PA work with reasonable speed. There are therefore many bottlenecks which need to be removed. For example, many of the existing modules of the PA demonstration systems need to be made more incremental. Currently, the PA does a great deal of unnecessary recomputation. By breaking up internal tasks into smaller units and making greater use of dependencies, it should be possible to improve performance in this area.

A related issue is the PA's use of space. The current demonstration systems represent all of the knowledge in the plan library and all of the knowledge about particular programs solely in terms of the plan representation. This is computationally very convenient, but takes up an unreasonable amount of space in memory and in long term storage. A possible solution is to use a scheme wherein only the knowledge that is currently being used is represented in full, while other parts of the knowledge base are represented more compactly. The key to doing this is segmenting the knowledge and determining a small subset of the information stored in a plan which can be used as the basis for rapidly recomputing the rest. Among other things, program text is a very compact representation for some of the information in a plan. Using annotations of the text to store additional critical information, we believe that in principle a compact plan representation need not be more than two or three times the size of the program text.

Finally, the modules developed for the feasibility demonstrations need to be made considerably more robust to be suitable for routine use. Some of the modules, such as the one which converts program text into a simple plan, have seen a good deal of use and are approaching reliability. Other modules need to be rewritten. For example, we now know that the algorithm used in the coder module is too simple-minded. In order to reliably produce aesthetic program text, it needs to be replaced by an algorithm which explicitly reasons about trade-offs between various readability criteria.

Bibliography

[1] R. Balzer, "Transformational Implementation: An Example", *IEEE Trans. on Software Eng.*, Vol. 7, No. 1, January, 1981.

[2] D.R. Barstow, "Automatic Construction of Algorithms and Data Structures Using A Knowledge Base of Programming Rules", Stanford AIM-308, Nov. 1977.

[3] D. Brotsky, "Program Understanding Through Cliche Recognition", (M.S. Proposal), M.I.T. Dept. of Elect. Eng. and Computer Sci., May, 1981.

[4] R.M. Burstall and J.L. Darlington, "A Transformation System for Developing Recursive Programs", *J. of the ACM*, Vol. 24, No. 1, January, 1977.

[5] T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 4, July 1979, pp. 402-417.

[6] T.E. Cheatham, "Program Refinement by Transformation", *5th Int. Conf. on Software Eng.*, San Diego, Cal., March, 1981.

[7] T.E. Cheatham, "An Overview of the Harvard Program Development System", pp. 253-266 in "Software Engineering Environments" H. Hunke ed., North-Holland, 1981.

[8] V. Donzeau-Gouge, *et al.*, "A Structure-Oriented Program Editor: A First Step Towards Computer Assisted Programming", *Proc. Int. Computing Symp.*, Antibes, 1975.

[9] J. Doyle, "Truth Maintenance Systems for Problem Solving", MIT/AI/TR-419, January, 1978.

[10] R. Duffey, II, "Formalizing the Expertise of the Assembler Language Programmer", (M.S. proposal), MIT/AI/WP-203, September, 1980.

[11] G. Faust, "Semiautomatic Translation of COBOL into HIBOL", (M.S. Thesis), MIT/LCS/TR-256, March, 1981.

[12] P. Feiler and R. Medina-Mora, "An Incremental Programming Environment", *5th Int. Conf. on Software Eng.*, San Diego, Cal., March, 1981.

[13] C. Frank, "A Step Towards Automatic Documentation", MIT/AI/WP-213, December, 1980.

[14] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," *Current Trends in Programming Methodology, Vol. IV*, (ed. Raymond Yeh), Prentice-Hall, 1978.

[15] C. Green, "Theorem Proving by Resolution as a Basis for Question-Answering Systems, *Machine Intelligence 4*, D. Michie and B. Meltzer, Eds., Edinburgh University Press, Edinburgh, Scotland, 1969.

[16] R. Greenblatt, T. Knight, J. Holloway and D. Moon, "A Lisp Machine", *Fifth Workshop on Computer Architecture for Non-numeric Processing*, Pacific Grove, CA, *ACM SIGIR Notices* Vol. 15, No. 2, *ACM SIGMOD Record* Vol. 10, No. 4, March, 1980.

[17] J. Guttag, "Abstract Data Types and the Development of Data Structures", *Comm. of the ACM*, Vol. 20, No. 6, June 1977, pp 396-404.

[18] D.J. Hamilton, W.G. Howard, *Basic Integrated Circuit Engineering*, McGraw-Hill, 1975.

[19] S. Hardy, "Synthesis of LISP Functions from Examples", *Advance Papers of the 4th Int. Joint Conf. on Artificial Intelligence*, Tblisi, Georgia, USSR, September 1975, pp. 240-245.

[20] P. Henderson, "Is It Reasonable to Implement a Complete Programming System in a Purely Functional Style?", The University of Newcastle Upon Tyne, PMM/94, December, 1980.

[21] E. Kant, "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach", Stanford AIM-331, (Ph.D. Thesis), September 1979.

[22] H. Lieberman and C. Hewitt, "A Session with TINKER: Interleaving Program Testing with Program Design", *Proc. of the 1980 Lisp Conference*, Stanford University, August 1980, August, 1980.

[23] B. Liskov *et. al.*, "Abstraction Mechanisms in CLU", *Comm. of the ACM*, Vol. 20, No. 8, August 1977, pp. 564-576.

[24] Z. Manna and R. Waldinger, "Synthesis: Dreams => Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 4, July 1979, pp. 294-327.

[25] D.A. McAllester, "An Outlook on Truth Maintenance", MIT/AIM-551, August, 1980.

[26] J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *Machine Intelligence 4*, D. Michie and B. Meltzer, Eds., Edinburgh University Press, Edinburgh, Scotland, 1969.

[27] J.M. Neighbors, "Software Construction Using Components", Technical Report 160, (Ph.D. Thesis), Dept. of Info. and Computer Science, U. of Cal., Irvine, Cal., 1981.

[28] A.J. Perlis and S. Rugaber, "Programming with Idioms in APL", *APL79 Conf. Proc.*, Rochester, N.Y., June, 1979.

[29] J.L. Pfaltz and A. Rosenfeld, "Web Grammars", *Proc. Int. Joint Conf. on Artificial Intelligence*, Washington, D.C., 1969, pp. 609-619.

[30] C. Rich and H.E. Shrobe, "Initial Report On A LISP Programmer's Apprentice", (M.S. Thesis), MIT/AI/TR-354, December 1976.

[31] C. Rich and H. Shrobe, "Initial Report on A Lisp Programmer's Apprentice", *IEEE Trans. on Software Eng.*, Vol. 4, No. 5, November, 1978.

[32] C. Rich, H. Shrobe and R. Waters, "Computer Aided Evolutionary Design for Software Engineering", (NSF Proposal), MIT/AIM-506, January, 1979.

[33] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (Ph.D. thesis), December, 1980.

[34] C. Rich, "Multiple Points of View in Modeling Programs", *Proc. of Workshop on Data Abstraction, Data Bases and Conceptual Modeling, ACM SIGPLAN Notices*, Vol. 16, No. 1, January, 1981, pp. 177-179.

[35] C. Rich and R. Waters, "Computer Aided Evolutionary Design for Software Engineering", (Progress Report), *ACM SIGART Newsletter*, No. 76, April 1981, pp. 13-15.

[36] C. Rich and H. Shrobe, "Initial Report on a LISP Programmer's Apprentice", in *Interactive Programming Environments*, D. Barstow, E. Sandewall, H. Shrobe editors, McGraw-Hill, December, 1981.

[37] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, August, 1981.

[38] G.R. Ruth, "Protosystem I: An Automatic Programming System Prototype", MIT/LCS/TM-72, July, 1976.

[39] E.D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence*, Vol. 5, No. 2, 1974. pp. 115-135.

[40] D. Shapiro, "Sniffer: a System that Understands Bugs", (M.S. proposal), MIT/AI/WP-202, July, 1980.

[41] D. Shapiro, "Sniffer: a System that Understands Bugs", (M.S. Thesis), M.I.T. Dept. of Elec. Eng. and Computer Science, May, 1981.

[42] D. Shaw, W. Swartout, and C. Green, "Inferring LISP Programs from Examples", *Advance Papers of the 4th Int. Joint Conf. on Artificial Intelligence*, Tblisi, Georgia, USSR, September 1975, pp. 260-267.

[43] H.E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding", (Ph.D. Thesis), MIT/AI/TR-503, April 1979.

[44] L. Siklossy and D. Sykes, "Automatic Program Synthesis from Example Problems", *Advance Papers of the 4th Int. Joint Conf. on Artificial Intelligence*, Tblisi, Georgia, USSR, September 1975, pp. 268-273.

[45] B.K. Steele, "An Accountable Source-to-Source Transformation System", (M.S. Thesis), M.I.T. Dept. of Elec. Eng. and Computer Sci., July, 1980.

[46] G.J. Sussman, "The Virtuous Nature of Bugs", *Proc. of Conf. on Artificial Intelligence and the Simulation of Behavior*, U. of Sussex, July 1974..

[47] M.E. Warren, "Program Generation by Questionnaire", *Information Processing 68*, North Holland, 1969.

[48] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, (Ph.D. Thesis), December, 1978.

[49] R.C. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 237-247.

[50] R.C. Waters, "Programmer's Apprentice", in *Handbook of Artificial Intelligence*, A. Barr and E. Feigenbaum, (to appear).

[51] R.C. Waters, "A Knowledge Based Program Editor", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, August, 1981.

[52] R.C. Waters, "A Knowledge Based Program Editor", in *Interactive Programming Environments*, D. Barstow, E. Sandewall, and H. Shrobe editors, McGraw-Hill, December, 1981.

[53] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", (*to appear*), *IEEE Trans. on Software Eng.*, (submitted for publication).


DA
FILM