A Knowledge Based Program Editor

Richard C. Wafers

The Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

This paper describes an initial implementation of an interactive programming assistant system called the Programmer's Apprentice (PA). The PA. is designed to be midway between an improved programming methodology and *an* automatic programming system. The intention is that the programmer will do the hard parts of design and implementation while the PA will assist him wherever possible. One of The major underpinnings of the PA is a representation (called a *plan)* for programs which abstracts away from the inessential features of a program, and represents the basic logical properties of the algorithm explicitly.

The current system is composed of four parts: an analyzer that can construct the *plan* for a program; a coder that can create program text corresponding to a *plan,* a library of *plans* for common algorithmic fragments; and a plan editor which makes it possible for a programmer to modify a program by modifying its *plan.* The greatest leverage provided by the system comes from the fact that a programmer can rapidly and accurately build up a program by referring to the fragments in the library and from the fact that the editor provides commands specifically designed to facilitate program modification.

I. Introduction

The Programmer's Apprentice (PA) is an interactive system tor assisting programmers with the task of programming. It is intended to act as a junior partner and critic, keeping track of details and assisting in the documentation, verification, debugging, and modification of a program while the programmer does the hard parts of design and implementation. In order to cooperate with a programmer, the PA must be able to *understand* what is going on. From the point of view of artificial intelligence, one of the central developments of the Programmer's Apprentice project has been the design of a representation (called a *plan)* for programs and for knowledge about programming which serves as the basis for this understanding.

Rich and Shrobe [10,11] laid out the initial design for the PA and for plans (see Section III). The author [13] designed a limited system intended to operate in the domain of mathematical FORTRAN programs and extended the idea of a plan by developing a theory of how plans could be segmented [14,15] (see Section IV). More recently Rich [8,9] has further developed the plan formalism (see Section IX) and designed a library of plans representing many of the standard cliches in non-numerical programming.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Navel Research contracts N00014-75-C-0643 and 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 author, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, the National Science Foundation, or the United States Government.

This paper describes an initial implementation of the PA recently completed by the author. This system is an editor which allows a programmer to build up a program from prototypical fragments, and to modify a program in terms of its logical structure. In both cases the system gains most of its power from the fact that all of these actions are performed on the plan for the program rather than on the program text. It should be noted that the current system is intended as a pre-prototype proof of principle. It is neither efficient nor robust. However, it demonstrates the feasibility of some of the basic ideas behind the PA.

Section II outlines the architecture of the system. The plan formalism which underlies the system is briefly discussed in Section III Sections IV-VII describe the four components of the system Section VIII gives a scenario of the use of the system Section IX discusses the weaknesses of the current system, and how these problems will be addressed in the second implementation of the PA which is now unde^r way.

II. Outline of the Current System

The current system is composed of four modules as shown in Figure 1. Given the text for a piece of a program, the analyzer module can construct a plan corresponding to it The coder module performs the reverse transformation, creating program text corresponding to a plan. The plan library contains common program fragments represented as plans The plan editor makes it possible for the user to modify a program by modifying its plan.

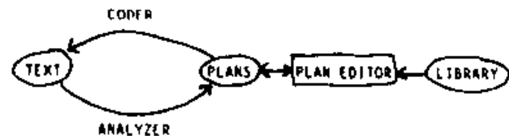

Figure 1: Architecture of the current implementation.

The system maintains two representations (program text and a plan) for the program being worked on. If the programmer uses the plan editor to change the plan, then the system uses the coder to determine what the new program text should be If the programmer uses an ordinary text editor to change the program text, then the analyzer is used to determine what the resulting plan should be. It should be noted that the existence of the analyzer means that the system can be used to modify programs that were not originally constructed using the system.

The leverage provided by the system comes primarily from two things. First, the programmer can rapidly and accurately build up a program be referring to the fragments in the plan library. Second, the editor provides a variety of commands specifically designed to facilitate program modification.

There is a similarity between this system and a programming language syntax editor (such as Mentor [5], the Cornell program synthesizer [12], or Gandalf [7]) in that the user operates on an underlying representation for the program rather than on its textual representation. In both cases many simple errors are prevented because it is no longer possible to even state them.

There are however three Key differences between this system and a programming language syntax editor. First, this system does not depend on the syntax of any particular programming language. All of the operations of the system are performed on plans. Except

for parts of the coder and analyzer modules, the whole system is essentially programming language independent. Second, the existence of the plan library changes the character of the interaction. The only prototypes known to the typical syntax editor are ones related to the basic syntactic constructs of the language. Third, this system specifically supports non-local modifications such as sharing. The typical syntax editor only supports the addition and deletion of individual syntactic units.

There is also a similarity between the current system and a transformational program development system such as TI [1], PSI [2] or PDS [4]. The key similarity is that both approaches rely heavily on libraries of standard programming cliches. In a transformational system these are represented as transformations. In the PA these are represented as plans. However, there is a considerable difference in the approach to the way this knowledge is used.

In the transformational approach a program is first completely specified at an abstract level and then a transformational component selects (more or less automatically) transformations (which are intended to be correctness preserving) in order to implement the program in an efficient way. There is, however, no support for creating or modifying the abstract specification itself. The current implementation of the PA is both more modest in its goals, and more flexible. A program does not have to be specified at an abstract level; there is no automatic selection of cliches to use; and the addition of a library cliche into a program is not expected to be a correctness preserving process. The PA is intended to support the programming process at all levels. Future implementations of the PA will include some automatic cliche selection and are intended to support the kind programming style pioneered by research on transformational systems, in addition to the interactive programming style presented here.

III. Plans

The importance of plans as the underlying representation used by the system cannot be overemphasized. The utility of the plan representation stems from the fact that it is specifically designed to make the kind of tasks performed by the system easy to do. In order to do this, the plan representation is designed to make all of the information the system needs to know explicit and local in a plan.

A plan is like a flow chart except that the data flow as well as the control flow is represented by explicit arcs. The basic unit of a plan is a *segment*. A segment corresponds to a unit of computation. It has a number of *input ports* and *output ports* which specify the input values it receives and the output values it produces. It has a set of specifications which give preconditions which must be true of the inputs in order for the segment to execute properly, and postconditions which describe what will be true after the segment is executed.

Control flow from one segment to another is represented by an explicit arc from the output side of the first segment to the input side of the second segment. Similarly, data flow from the output of one segment to the input of another is represented by an explicit arc from the appropriate output port of the first segment to the appropriate input port of the second segment.

In addition to segments corresponding to primitive computations there can be intermediate segments containing groups of inner segments. All of the data flow between segments outside of an intermediate segment and segments inside an intermediate segment is channeled through input and output ports attached to the intermediate segment. All of the computation corresponding to a single subroutine will be grouped together into one outermost segment.

An important feature of the plan representation is that it abstracts away from inessential features of a program. Whenever possible it tries to eliminate features which stem from the way things must be expressed in a particular programming language, and keep only those features which are essential to the actual algorithm. For example, a plan does not represent a data flow in terms of the way it could be implemented in any particular programming language (e.g. with variables, or nesting of expressions, or parameter passing). Rather, it just records what the net data flow is. (For the convenience of the coder, a plan can have annotations which suggest what variable names are appropriate for a particular data flow.) Similarly, no information is represented about how control flow is implemented. One result of this abstraction is that plans are much more canonical than program text. Programs which vary only in the way their control flow and data flow is implemented all correspond to the same plan.

Another important feature of the plan representation is that it tries to make things as local as possible. For example, each data flow arc represents a specific communication of data from one place to another and, by the definition of what a data flow arc is, the other data flow arcs in the plan cannot have any effect on this. The same is true for control flow arcs. This leads to the property of *additivity*. It is always permissible to put two sections of a plan side by side without their disturbing each other. If there is no data flow or control flow between the sections, then they can have no effect upon each other.

Additionally, intermediate segmentation breaks a plan up into regions which can be manipulated separately due to the fact that the plan representation is designed so that nothing outside of a segment can depend on anything inside that segment. This leads to the property of *substitutability*. It is always permissible to replace any segment with another segment as long as the two are *externally* indistinguishable.

IV. The Analyzer

The analyzer (described in [15] and more fully in [14]) automatically produces a plan corresponding to a program. This is done in three steps: source translation, surface analysis, and segmentation. In the source translation step, a language specific module parses the program text and converts it into a Lisp-like intermediate form. Source translators currently exist for Fortran and for Cobol. None is needed for Lisp.

The surface analyzer runs over the intermediate form like an evaluator creating a *surface* plan as it goes. A *surface* plan is composed solely of a large number of terminal segments corresponding to primitive operations (such as '+' and "O grouped into one large segment and connected by data flow and control flow. The surface analyzer has detailed knowledge of the constructs which implement data flow and control flow. However, it does not have any knowledge of what the primitive functions do except how many inputs and outputs they have.

A surface plan is *flat* in that it has no hierarchical structure. The segmentation step introduces intermediate segmentation reflecting the logical structure of the plan. The primary goal of this process is to group segments which interact heavily close together, and keep segments which interact little, if at all, far apart. The segmentation is done in terms of six basic configurations called plan building methods (PBMs). It proceeds bottom up based on the topology of the control flow and data flow by locating minimal configurations which can be grouped together inside intermediate segments in accordance with the PBMs.

In general, the PBMs correspond to basic structured programming constructs: expressions, complex predicates, conditional expressions, and loops. The primary innovative feature is that loops are analyzed as compositions of fragments of looping behavior communicating by means of temporal sequences (streams) of values, rather than as repetitively executed pieces of straight-line code. For example, the loop in Figure 2 would be analyzed as a composition of four loop fragments as shown in Figure 3.

```
Z = 0;
DO I=1 TO N;
   IF A(I)>0
      THEN Z = Z+A(I);
END;
```

Figure 2: An example loop.

```
generator   terminator       filter            accumulator
 I=1;       IF I>N THEN                         Z = 0;
 I=I+1;       GOTO EXIT;   IF A(I)>0 THEN       Z = Z+A(I);
```

Figure 3: The example analyzed by PBMs.

The first fragment (a generator) counts up by 1 from 1 creating an unbounded temporal sequence of values {1,2,...}. The second fragment (a terminator) tests the sequence of integers produced by the generator and stops the loop when an integer greater than N is found. This has the effect of truncating the sequence of integers to the sequence {1,2,... N}. (Both of these fragments are part of the 00 construct.) The third fragment (a filter) restricts the truncated sequence of integers by selecting only those integers which correspond to positive elements of the vector A. The last fragment (an accumulator) computes the sum of the elements of A corresponding to the integers in the restricted sequence produced by the filter. In the loop as a whole, the four fragments are cascaded together so that the loop computes the sum of the positive members of the first N elements of A.

From the point of view of the system being described here, this analysis is important because it provides very convenient intermediate segmentation. For example, if the programmer wanted to change the loop above into one which computed the product of all of the elements in the vector, he would only have to remove the filter, and replace the summation accumulator with a product accumulator.

V. The Coder

The coder takes in a segmented plan and creates the code for a Lisp program corresponding to it. This process is relatively straightforward. The structure of the program produced closely follows the structure of the segmented plan. In general, each terminal segment in the plan is implemented as a function call, and etch non-terminal segment in the plan is Implemented using a syntactic construct corresponding to its PBM. For example, * segment whose PBM is conditional will usually be coded using a COND. There are only two areas of real difficulty: determining how to implement the data flow in the plan, and dealing with loops.

The data flow is a problem because the plan formalism deliberately does not represent much information about how data flow should be implemented. The coder is forced to discover how to use variables and nesting of function calls in order to implement the required data flow. In order to get mnemonic variable names, the plan contains explicit suggestions for what variable names to use when variables are required.

When deciding how to implement the data flow in a plan, the coder first identifies *every* place where the nesting of function calls can be used. It then identifies related groups of data flow arcs that can be implemented using the same variable. Its basic goal is to use the smallest number of variables possible. When picking names for these variables, it uses any suggestions recorded in the plan. It has to be particularly careful to make sure that there is no conflict when the same name is suggested for two different groups of data flow arcs.

In order to produce lisp code for a loop analyzed as a composition of loop fragments, the fragments must first be combined together into a single loop. This process (see [14]) is designed so that the validity of the simple logical analysis in terms of composition will be preserved. The fragments *ere* taken apart, and their pieces *ere* reassembled into the resulting loop. The arrangement of the pieces in the loop is dictated by the data flow communicating temporal sequences of values between the fragments. The key idea is that if a piece of one fragment uses a temporal sequence created by a piece of another fragment, then the consuming piece should be placed after the producing piece in the resulting loop. The action of • filter is produced by placing the pieces which receive its outputs within the scope of a conditional predicated on the filter's test. Similar problems have been attacked by the people working on compilers for API (see for example, [6]).

Vt. The Plan Library

The plan library is a collection of plans for common algorithmic fragments. There are fragments corresponding to the six PBMs (eg conditional, predicate, loop, etc.). In addition, there are a variety of more specific programming cliches such as those in Figure 3 (i.e counting up, stopping at a given number, selecting out the positive elements, and summing up). It should be noted that this is more usefut than a collection of fragments represented in a programming language for three reasons. First, the plan fragments are more canonical. A given plan can be realized as code in many different ways. Second, many fragments *ere* so fragmentary that they do not correspond to any syntactically reasonable section of code. Third, due to the additivity property of plans, having the fragments represented as plans makes it easier to combine them. The system does not have to worry about issues like variable name conflicts, because there are no variables.

The library makes two basic things available to the user. First, he can include a fragment in the program he is constructing by referring to it by *name* (e.g. "conditional*, "count", "summation", etc.). Second, etch fragment has named *parts* some of which may be unspecified. For example, a conditional has some unspecified test, a summation has tn "accumulation" part which is specified to be +, and it has a sequence part (the temporal sequence to oe summed; which is unspecified. When a fragment is instantiated its part names are carried over as well. The programmer can then use them when referring to the program he is constructing (e.g. "test of *conditional,* "accumulation of *summation",* "sequence of *summation").*

VII The Plan Editor

The plan editor provides a set of commands which a programmer can use to modify the program he is working on, by modifying its plan. He communicates with the plan editor using an LL1 pseudo English. The central feature of the system is the vocabulary that the programmer uses. The semantics of this vocabulary is based on the plan notation and the plan fragments in the library.

The basic objects known to the system are the objects in a plan: segments, input and output ports of segments, data flow, and control flow. Various adjectives are available for referring to particular kinds of objects. For example, "constant *segment",* "function call *segment",* "first *segment", "tree* variable *input",* or "side-effect *output".* Part name expressions refer to the relationships between objects. For example, "subsegment of *segment",* "input of *segment",* "use of *output",* or "source of *data flow".* As discussed above, the fragments in the plan library and their parts can be referred to by name.

A programmer can refer to a specific part of the plan for the program he is working on by using the word "the" (e.g. "The test of the program FOO"). Complex references can be built up by nesting simple ones (e.g. "the use of the side-effect output of the action of the *program* foo"). These phrases make it possible for the programmer to treat his program as an algorithmic structure, rather than a syntactic one He can identify parts of the program based on their role in the algorithm, rather than based on their location in the text, or by their relationship to the parse tree for the program, as in a programming language structure editor. As a convenient method of reference, the pronoun "it" is used to refer to the object which is the current focus of the system's attention. In addition, program names and variable names can be used to refer to the associated objects.

The word "a" is used to create an instance of something. For example, "a summation", or "a side-effected free variable output TBI". The programmer can also use literal pieces of program text (such as (eg (car current) symbol)) in order to create a new fragment of plan. (The analyzer is used to convert the text into a plan.)

The editor provides a variety of verbs which specify actions to be performed. Two of these: "define" (a new program) and "implement" (an unspecified part a plan fragment), are used to build up a program by combining prototypes from the library As this is being done, the system annotates the resulting plan so that it embodies an analysis of the program in terms of the fragments used to create it.

Another group of commands (such as "add" (some object to the plan), "remove" (some object from the plan), "replace" (some object with another), "share" (two segments together as a single instance), etc.) are used to modify a plan. The primary leverage here is that it is much easier to make changes to a plan without causing unwanted side-effects, than it is to make changes to program text without causing unwanted side-effects. This is due to the fact that in a plan things are represented explicitly, locally, and irredundantly so that in general if some feature of the plan is changed, it doesn't effect any other feature of the plan. In contrast consider the use of variables in a programming language. Using a variable to implement a data flow in one place can have arbitrary effects on other data flows that happen to be implemented using the same variable.

Note that the process of modification interacts strongly with the process of constructing a plan from prototypical fragments. As mentioned above, the plan embodies an analysis of the program in terms of the fragments used to create it. Once an orthogonal modification is made to the program, it must be reanalyzed in order to determine what fragments correspond to its new structure. Whenever possible this reanalysis is done incrementally so that only the immediate neighborhood of the change is effected. If complete reanalysis is required, then information about the specific library cliches used to build up the plan is lost due to the fact that the current analyzer can only analyze a program in terms of the PBMs.

The plan editor also makes available several commands for displaying information to the programmer. Tor example, he can ask what parts of the plan are still incomplete, he can ask the system to describe parts of the plan, or he can ask the system to display the parts of the program which correspond to a part of the plan.

VIII. A Scenario

This section presents a brief scenario showing an actual interaction with the current system. The programmer is constructing a function DELETE which will delete (by side-effect) an occurrence of a symbol from a set implemented as a hash table. As shown in Figure 4, the hash table is a vector of lists (called buckets). Each symbol in the set is placed in one of the buckets. In order to simplify the deletion process, the hash table is implemented using buckets which have special header cells of NIL.

```
The set (A B C D E F G H I J L M N) is stored as:

        [(NIL A E I M)
         (NIL B F J N)
         (NIL C G)
         (NIL D H L)]
```

Figure 4: An example of the hash table.

The programmer has already coded the function in Figure 5 which takes in a symbol, and returns the appropriate bucket of the hash table. The function works by generating a unique number corresponding to the symbol, taking the remainder of this number after dividing by the size of the table, and then using this remainder to index into the table. Note that the hash table itself (TBL) as well as its size (TBLsizO are free variables.

```
(DEFINE HASH (SYMBOL)
  (ARRAYFETCH TBL (REMAINDER (MAKNUM SYMBOL) TBLSIZE)))
```

Figure 5: The hashing program.

The programmer communicates with the system by typing at a video terminal The screen of the terminal is divided into two regions (see Frame 1). The top region shows what the programmer types. The lower region is used by the system to display code for the program being worked on The part of the program that the system's attention is focused on (it) is highlighted on the display (this is indicated by underlining in the figures below). The last line of the lower region is a part expression identifying the highlighted section Each frame in this scenario corresponds to a screen image seen by the programmer.

The first frame shows the programmer starting to build up his program by combining fragments from the library. First, he begins the definition of the function, specifying its parameter. He then implements the program as a trailing-pointer-list-enumeration.

```
>Define a program DELETE with a parameter SYMBOL.
>Implement it as a trailing-pointer-list-enumeration.

(DEFINE DELETE (SYMBOL)
  (PROG (CURRENT PREV)
        (SETQ PREV ..initial-list.. )
        (SETQ CURRENT (CDR PREV))
     LP (COND ((NULL CURRENT) (RETURN NIL)))
        (PSETQ PREV CURRENT
               CURRENT (CDR CURRENT))
        (GO LP)))

The trailing-pointer-list-enumeration program DELETE.
```

Frame 1

The reader might well complain at this point that "trailing-pointer-list-enumeration" is a *Very* cumbersome name and that no one would want to have to remember it or use it. There are several points to be made. First, the actual name itself is not important. What is important is that the process of moving a pointer (here CURRENT) down a list starting at the second element while maintaining a second (trailing) pointer (here pREV) pointing to the preceding list element (see Figure 6) is a common algorithm that is known both to programmers and to the system. Second, there has to be some way to refer to it so that the programmer can pull it into his program when he wants it. Just as in Interlisp a programmer has to know the names of the functions in the manual in order to use them, the user of this system has to know the names of the prototypes in the library. Third, Section IX shows how the next implementation of the PA will reduce the need for the programmer to know idiosyncratic names like this.

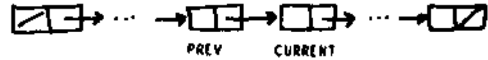

Figure 6: Trailing-poinler-list-enumeration.

The lower part of Frame 1 shows the code produced by the coder based on the plan constructed by the user's commands. The coder uses a PSETQ (parallel assignment) in the code it produces because that is easier than trying to figure out what order the assignments should be done in. The plan, and therefore the code, is fragmentary because the programmer's description of the program so far is fragmentary. The coder is specifically designed so that it will produce legible (though perhaps not runnable) code for a plan no matter how fragmentary the plan is. The code in the display serves as a summary of what has gone on so far in the editing session. The symbol ". .initial-list .." is used to represent the, so far unspecified, initial-list part of the trailing-pointer-list-enumeration fragment.

In Frame 2, the programmer fills in the initial-list slot by implementing it as the expression "(HASH SYMBOL)". Through the ability to pull in fragments from the library, and then fill in their unspecified parts with pieces of code, or other fragments from the library, the system supports the top down construction of a program.

```
>Implement the initial-list as '(HASH SYMBOL).

(DEFINE DELETE (SYMBOL)
  (PROG (CURRENT PREV)
        (SETQ PREV (HASH SYMBOL))
        (SETQ CURRENT (CDR PREV))
     LP (COND ((NULL CURRENT) (RETURN NIL)))
        (PSETQ PREV CURRENT
               CURRENT (CDR CURRENT))
        (GO LP)))

The initial-list of the generator of DELETE.
```

Frame Z

The next two frames illustrate a completely different mode of interaction with the system: modification. Typically program construction (and certainly debugging) involves a lot of modification as the program evolves into an acceptable implementation. In Frame 3 the user has decided that he doesn't like the algorithm for trailing pointer list enumeration used by the system because it contains an unnecessary instance of CDR He asks the system to get rid of this by sharing the two instances of CDR This is a good example of a modification which is easy to state, but which is not so easy to perform on program text. The main problem is that looking simply at the text, it is not easy to tell whether or not the sharing is possible. The whole program must be checked to see that there are no side-effects or modifications of the variables involved which prevent the sharing. The system gets considerable leverage on this because these questions are easy to answer In the plan representation due to the fact that all of the data flow is represented as explicit arcs.

Once it has been determined that the sharing is permissible, relatively global changes may have to be made in the code. For

example, a temporary variable may have to be introduced. Here things are particularly complicated due to the fact that the CCR in the loop has to be shifted around to the top of the loop before it can be shared with the COf in the initialization. Here again the required tasks *are* easier to *perform* on a plan than on program text. The coder is used to create the new text after the plan is modified.

```
>Share the COff in the initialization with the CCR in
 the bump of the generator.


(DEFINE DELETE (SYMBOL)
  (PROG (CURRENT PREV)
        (SETQ PREV (HASH SYMBOL))
     LP (SETQ CURRENT (COR PREV))
        (CONO ((NULL CURRENT) (RETURN NIL)))
        (SETQ PREV CURRENT)
        (GO LP)))


The bump of the generetor of DELETE.
```

Frame 3

In Frame 4 the programmer adds a second terminator into the loop. This causes the loop to have two exits. The system responds to this by embedding the loop as the test of a conditional. This reflects the assumption that the programmer intends to perform separate (unspecified) actions corresponding to the two loop terminations (otherwise he would have combined them into one termination with a compound test), and that he intends to join the control flow after these two distinct actions, and simply return from the function in either case. The system notifies the user of this change because it is a somewhat non-obvious one, and because it changes the way things can be referred to by adding an additional level of nesting into the plan.

The second programmer request *in Frame* 4 specifies the test of the additional terminator. The test stops the loop if *an* element identical to the input symbol is found. Note that the programmer simply gave a little piece of code because that was easier than specifying what he wanted any other way.

```
>Add a terminator into the loop after the first
 terminator of the loop

Since the loop now hes two exits. I assume that
DELETE is a conditional with the loop as its test.

>Implement the test of the terminator
 as '(EO (CAR CURRENT) SYMBOL).


(DEFINE DELETE (SYMBOL)
  (PROG (CURRENT PREV)
        (SETQ PREV (HASH SYMBOL))
     LP (SETQ CURRENT (COR PREV))
        (CONO ((NULL CURRENT) (RETURN  .action..)))
        (CONO ((EQ (CAR CURRENT) SYMBOL)
               (RETURN   action,.)))
        (SETQ PREV CURRENT)
        (GO LP)))


The test of the second terminator of the test of ...
```

Frame 4

At this point, in order to demonstrate another capability of the current system, the programmer asks what needs to be done (see Frame 5). The system keeps track of all of the aspects of the plan which *are* incomplete. It replies that the two actions still need to be implemented. The programmer then specifies these actions. Note that he refers to the simple programming cliche "splice-out".

```
>What needs to be done?
Implement the first ection of DELETE
Implement the second ection of DELETE.

>implement the second action of the program as a
 splice-out of 'PREV

>implement the first action of the program as  Nil


(DEFINE DEIETE (SYMBOL)
  (PROG (CURRENT PREV)
        (SETQ PREV (HASH SYMBOL))
     LP (SETQ CURRENT (CUR PREV))
        (CONO ((NULL CURRENT) (RETURN NIL)))
        (COND ((EQ (CAR CURRENT) SYMBOL)
               (RPLACD PREV (COOR PREV))
               (RETURN NIL )))
        (SETO PREV CURRENT)
        (GO LP)))

The NIL action of DELETE
```

Frame 5

The last frame demonstrates some other capabilities of *he current system. In response to the request "describe", the system can report out simple features of the plan. Here, the programmer asks for a description of the inputs and outputs of the program. Note that the system has figured out that TBL and TBLSIZE *are* free variable inputs to DELETE since they are free variable inputs to HASH.

```
>describe the ports of DEIETE
The SYMBOL parameter
The TBL free variable input
The TBLSIZE free variable input
The return value
The TBL side-effect output

>Dissplay the source of the TBL output


(DEFINE DELETE (SYMBOL)
  (PROG (CURRENT PREV)
        (SETQ PREV (HASH SYMBOL))
     LP (SETQ CURRENT (COR PREV))
        (CONO ((NULL CURRENT) (RETURN NIL)))
        (CONO ((EO (CAR CURRENT) SYMBOL)
               (RPLACP PREV (COPR PREV))
               (RETURN NIL)))
        (SETQ PREV CURRENT)
        (GO LP)))

The TBL side-effect output of the RPLACO action of
```

More interestingly, the system realizes that TBl is also *an* output of DELETE The system deduces this, even though it is not at all obvious from the way the programmer built the program up, because it has a simple model of side-effects which allows it to understand many common situations where side-effects are used. It knows that if a program side-effects some part of a structure, then the structure as a whole is also side-effected. Here, the RPLACO potentially changes PREV which through a chain of CDRs is part of the list returned by (MASH SYMBOL). Because HASH has already been defined the system knows that this list is part of the vector TBL and that therefore this, input to DELETE is potentially modified. It is *very* important that the system be able to make this kind of deduction, because much of what goes on in the system depends on the fact that the data flow is correct. If some more complex side-effect were going on the user would have to tell the system about it so that the effect would not be overlooked. As his last request, the programmer asks the system to show him where the side-effect output comes from, and the system highlights the call on RPLACO indicating the source

IX. The Next Implementation of tht PA

Figure 7 shows the commands from the scenario above which were necessary in order to construct the program DELETE One important way to evaluate these commands is to consider the level of understanding exhibited by the current system. There are four classes of things that it understands. It understands the basic plan notions (segments, inputs, outputs data flows etc.) and how they can be referred to. It understands the PBMs (expression, conditional, loop, etc.). It understands the verbs in the commands (define, implement, add, etc.). Finally, it understands the fragments in the plan library (trailing-pointer-list-enumeration, splice-out, etc.), but only in that it can recall them by name, and interpret references to their parts.

```
>Define a program DELETE with a parameter SYMBOL
>Implement it as a trailing-pointer -list-emimeration
>Implement the initial-list as '(HASH SYMBOL)
>Add a terminator into the loop after the first
 terminator of the loop.
>]mplement the test of thr terminator
 as '(CO (CAR CURRENT) SYMBOI ) .
>Implemcnt the second action of the program as a
 splice out of 'PREV
>Implemenl the first action of the program as 'NIL
```

Figure, 7: The commands *used* to construct DELETE.

However, there *are* three very important things that the current system does not understand. First, it does not understand anything about data structures. Prior to the scenario, the hash table was described for the benefit of the reader, but the programmer said nothing about it to the system. He just chose the correct algorithmic fragments for working on the data structure he had in mind. Second, the current system understands nothing about specifications. Again, the programmer just chose the correct fragments to implement the specifications he had in mind. Third, the current system understands nothing about interrelationships between the fragments in the library.

The original design of plans included the notion of specifications. However, this feature was not used in the current system. Rich [8,9] has extended the plan formalism so that it can be used to represent Knowledge about data structures and about interrelations between plans for data structures, plans for algorithmic structures, and specifications. He has also designed and is implementing a comprehensive plan library of common programming cliches in the domain of non-numerical programming using the improved plan formalism Together with the components described here, this new library will serve as the basis for the next implementation of the PA.

Using the next implementation of the PA, the programmer will be able to create the program DELETE using a sequence of commands like the ones shown in Figure 8. First, since the system understands data structures and specifications, the user tells it about them. Before constructing the program HASH the programmer tells it about the data structure being used by saying "TBI implements • set as a hash table whose buckets are lists with header cells of Nil". This command refers to several data structure terms (i.e. set, hash table, bucket (part of hash table), list, header cell (part of list), NIL). The system understands these terms due to the presence of plans for common data structures in the new plan library. Based on this description the system builds up a plan for the data structure.

```
        Before creating HASH.
>TBL implements a set as a hash table whose buckets
 are lists with header cells of NIL.
>The function HASH mops SYMBOL into the corresponding
 bucket in TBL
        In order to create DELETE:
>Define a program DELETE with a parameter SYMBOL.
>DELEU removes SYMBOL from TBL .
>Implement DEIEU so that it searches the bucket in
 TBL corresponding to SYMBOL and splices out the
 occurrence of SYMBOL, if there is one.
```

Figure 8. Commands needed in the next implementation

The programmer then gives a specification for HASH "The function HASH maps SYMBOL into the corresponding bucket 1n TBL*. This refers to a number of specification terms understood by the system (i.e. map, corresponding). Based on this, the system constructs specifications for the function HASH.

Once HASH is constructed, the programmer starts to construct DELETE by setting up the Outer segment "Define • program DELETE with • parameter SYMBOL", and giving its specifications "DELETE removes SYMBOL from TBL"

He then describes the algorithm to be used by using the command "Implement DELETE so that 1t searches the bucket In TBL corresponding to SYMBOL and splices out the occurrence of SYMBOL, If there is one". This final command corresponds to the last six commands in Figure 7. He is able to be more concise here because, due to the interrelationships in the new library, the system will be able to reason about how the choice of one fragment determines the choice of other fragments.

The programmer specifies that he wants to search "the bucket in TBL corresponding to SYMBOL". Because of the specifications the programmer gave for HASH, the system knows that this bucket can be calculated by calling (HASH SYMBOL).

"Search" is an abstract fragment that specifies that in order to search something a program must enumerate the things in the structure and check each item to see if it is the one desired. There are two exits from the searching loop depending on whether the desired item is found or not. Due to the fact that the programmer said that the buckets in the hash table were lists, the system knows that it must do a list enumeration in order to search it. The programmer specifies that the predicate he wants to use in the search is one which tests for an "occurrence of SYMBOL".

Finally, the programmer specifies that what he wants to do when he finds the occurrence of SYMBOL is "splice (it) out". The system has the same fragment for splice out that is used »n Figure 7, but additionally it knows that in order for a splice out to work, there has to be a trailing pointer. As a result, it reasons that it needs to do a trailing pointer list enumeration (a special kind of list enumeration) in the search.

Note that because the new system will be able to do the above kind of simple reasoning, the user will not have to use specific fragment names such as "trailing-pointer-list-enumeration", but rather can, in general, rely on more abstract terms such as "search". Another important benefit to be gained from this kind of reasoning is that the system can do simple consistency checking, and complain whenever the user tries to combine incompatible fragments. The current system only complains when the user violates the basic semantics of a plan.

Another major improvement which is being made to the system, is the strengthening of the analyzer. Brotsky [3] is working on adding a fourth stage to the analysis component of the system so that it can analyze a program in terms of all of the fragments in the library, not just in terms of the six PBMs. The fact that analysis in terms of PBMs is done first should simplify this task because it breaks the program up into a number of small pieces. The recognition process should be able to avoid combinatorial explosion by looking at each piece separately, rather than searching through the entire program at once. Once complete analysis is possible, the system will be able to apply the full force of the knowledge in the plan library to bear on a program whether or not it was originally built up using the system, and whether or not it has been modified.

Looking further in the future there are several other kinds of knowledge which should be added to the PA. For example, both the current system and the next one *are* oriented toward understanding how a program does what it does. There is little or no representation of why it does it the way it does as opposed to some other way. This problem can be addressed by adding knowledge of efficiency issues into the system. In addition, the current plan representation is oriented toward representing single programs and small clusters of programs. It needs to be extended so that it can represent large scale system organizations. Finally, the knowledge now being represented is limited to general knowledge about programming. In order to be more useful to a programmer in a particular domain, the library will have to be extended so that it contains a variety of fragments pertaining to the particular domain of use. In the long run, as the PA gains more and more knowledge in more and more areas, it will be able to take on Increasingly large portions of the programming task.

References

[1] R. Balzer, Transformational Implementation: An Example", IEEE Trans, on Software Eng., V7 e1, January, 1981.
[2] OR. Barstow, "Automatic Construction of Algorithms and Data Structures Using A Knowledge Base of Programming Rules", Stanford AIM-308, Nov. 1977.
[3] D.C. Brotsky, "Program Understanding Through Cliche Recognition", MIT masters thesis proposal, March 1981.
[4] T.E. Cheatham, "An Overview of the Harvard Program Development System", pp. 253-266 in "Software Engineering Environments" H. Hunke ed., North-Holland, 1981.
[5] V. Donzeau-Gouge, et al, "A Structure-Oriented Program Editor; A First Step Towards Computer Assisted Programming", Proc. Int. Computing Symp., Antibes, 1975.
[6] L.J. Guibas and D.K. Wyatt, "Compilation and Delayed Evaluation in API", Proceedings of 5th ACM POPL Conference, 1978.
[7] P. Feiler and R. Medina-Mora, "An Incremental Programming Environment", proc. of the 5th Int. Conf. on Soft. Eng., San Diego CA, March 1981.
[8] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (based on an MIT PhD. thesis June 1980), to eppear.
[9] C. Rich, "A Language-independent Representation for Standard Program Forms", IJCA1-81, August 1981.
[10] C. Rich and H Shrobe, "Initial Report on a Lisp Programmer's Apprentice", MIT/AI/TR-354, MIT Cambridge MA, December 1976.
[11] C. Rich and H Shrobe, "Initial Report on a Lisp Programmer's Apprentice", IEEE Trans, on Soft. Eng., V4 e6, November 1978, pp. 456-467.
[12] T. Teitlebaum, "The Cornell Program Synthesizer", Tech. Rep. 79-370, Dept. of Computer Science, Cornell Univ., 1979.
[13] R.C. Waters, "A System for Understanding Mathematical FORTRAN Programs", MIT/AIM-368, MIT Cambridge MA, August 1976.
[14] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, (Based on an MIT PhD. Thesis August 1978), December 1978.
[15] R.C. Waters, "A Method for Analyzing Loop Programs", IEEE Trans, on Software Eng. V5 e3, May 1979.