Massachusetts Institute of Technology
Artificial Intelligence Laboratory

AI Working Paper No. 150 July 1977

A Method, Based on Plans, for Understanding How a Loop Implements a Computation

by Richard C. Waters

ABSTRACT

The plan method analyzes the structure of a program. The plan which results from applying the method represents this structure by specifying how the parts of the program interact. This paper demonstrates the utility of the plan method by showing how a plan for a loop can be used to help prove the correctness of a loop. The plan does this by providing a convenient description of what the loop does. This paper also shows how a plan for a loop can be developed based on the code for the loop without the assistance of any commentary. This is possible primarily because most loops are built up in stereotyped ways according to a few fundamental plan types. An experiment is presented which supports the claim that a small number of plan types cover a large percentage of actual cases.

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 Reasearch Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

I. Introduction

This paper presents a method, based on plans, for understanding how a loop can be used to implement a computation. In order to demonstrate the utility of the method, the paper discusses the verification of loops. Suppose an automatic verification system wishes to verify a claim C, expressed in the notation of Floyd and Hoare [Floyd 67; Hoare 69] as P{A}Q, about a program A. Unless the verification system has access to explicit theorems about the behavior of A, the system must look at the internal structure of A and develop a set of theorems about parts of A which can be used to prove C. Developing appropriate theorems is particularly difficult when A is a loop.

The plan method analyzes the structure of a program. It uses one of a small number of plan types to decompose a complex program into a set of parts which are in turn analyzed. The method then infers a description of the behavior of the program from descriptions of the behavior of its parts.

The plan method differs significantly from methods, such as those described in [Wegbreit 74; German & Wegbreit 75; Katz & Manna 76], which attempt to develop appropriate loop invariants starting from the output assertion Q. In a majority of cases Q does not contain enough information to generate correct invariants. In these cases, these methods use a heuristically guided search in order to find invariants. The work described in [Basu & Misra 75, 76; Morris & Wegbreit 77] is notable in that it identifies subcases where an invariant can be directly inferred from Q.

The next section describes a programming assistant system based on plans. Section III describes how a plan is represented. Sections IV-VII describe specific plan types which can be used to analyze loops. Section VIII discusses an experiment which shows that these plan types are useful in a large class of typical programming situations.

II. A Programming Assistant System Based on Plans

The research reported here is one facet of an attempt to develop a system to assist a person who is writing FORTRAN programs. This FORTRAN programming assistant system (FORPAS) is described more fully in [Waters 76]. FORPAS is designed to be intermediate between current programming systems, which are of very little assistance to a programmer, and an automatic programming system.

The intention is that FORPAS and a programmer cooperate in the task of developing a program. The programmer writes a program based on his knowledge of the problem domain (i.e., knowledge of the particular problem to be solved, and knowledge of the particular algorithm to use to solve it). By looking at the program and annotation supplied by the programmer, FORPAS learns what the algorithm is and develops an understanding of how the program implements the algorithm. FORPAS uses this understanding in order to assist the programmer in debugging and verifying the program.

Consider, for example, a program for finding the inverse of a matrix by using Gaussian elimination. The programmer assumedly understands both inversion and why Gaussian elimination yields the inverse. FORPAS understands how the program implements Gaussian elimination. From the program, FORPAS learns what Gaussian elimination is, but it does not understand how or why it works.

This particular division of responsibilities was chosen for pragmatic reasons. The amount of knowledge a system must have in order to understand mathematical programs increases with the depth of its understanding. In order to understand the mathematics involved, a system would have to know many algorithms and theorems and be able to apply them. However, it appears that in order to understand how the programs are implemented, FORPAS only needs to be able to work with approximately 25 basic plan types.

FORPAS' understanding of a program is embodied in a plan for the program. The plan gives the teleological structure of the program. It specifies the purpose of each feature of the program, and how these features interact in order to produce the behavior exhibited by the program. The plan views a program as logically consisting of a hierarchical collection of segments, where each non-terminal segment is implemented by a small number of subsegments according to one of the basic plan types.

FORPAS can use the plan for a program in a variety of ways. FORPAS can answer questions about the program, describing, explaining, and justifying parts of the program. Further, FORPAS can develop an

outline of a proof of correctness for the program. It cannot automatically verify the program because it lacks a deductive system powerful enough to prove all of the theorems involved; however, it can prove the easy ones.

FORPAS can assist in the debugging process in three ways. First, it can detect a bug in a program by detecting that one of the theorems required by the proof of correctness cannot be valid. Bug detection is facilitated by the fact that most bugs invalidate several theorems. Second, once a bug has been found, by FORPAS or by the programmer, FORPAS can use its understanding of the program to identify what parts of the program could be responsible for the bug. Third, when a program is modified in order to fix a bug, FORPAS can assess some of the consequences of the change.

The current effort (see [Waters 76]) focuses on one particular corpus of programs, the IBM Scientific Subroutine Package [IBM 70], and on the question of how FORPAS can develop a plan for a program. This paper extends the work in [Waters 76] with regard to plans for loops, and how they can be used to develop an outline of a proof of correctness. Other types of plans, and other uses of plans are discussed in [Waters 76].

The attempt to develop FORPAS extends the ideas of Sussman [Sussman 73], Goldstein [Goldstein 74], and Hewitt and Smith [Hewitt & Smith 75]. The research of Rich and Shrobe [Rich & Shrobe 76] focuses on different aspects of a programming assistant system in a different programming domain.

III. Plans

The plan for a segment shows how the behavior of the segment is produced through the combined effort of a set of subsegments. The behavior of each segment and subsegment is described by a "behavioral description". The behavioral description for a segment S specifies the inputs (I) and the outputs (O), which must all be distinct from each other. It also specifies a set of prerequisites (P), stated in terms of the input values, and a set of assertions (A), which restrict the possible output values. The central claim of the behavioral description is that, in the notation of Floyd and Hoare, P{S}A.

The plan for a segment contains a description of the teleological structure of the segment together with behavioral descriptions for the segment and each subsegment. The teleological structure, which shows how the subsegments interact in order to produce the behavior of the segment, is described through a combination of several mechanisms.

Each plan is an instantiation of one of a small number of plan types. General information about the plan type indicates a lot about how the subsegments interact. This information indicates how to go about verifying that the segment as a whole works correctly, and what some of the common bugs are in programs using the plan type. Specific teleological information about a given segment is represented by a network of "reason" links.

Taken together, the reason links are an outline of a proof of correctness for a segment. In order to verify that a segment operates in the way that its behavioral description specifies, it must be shown that if the inputs satisfy the prerequisites, then the outputs will appear and will satisfy the assertions. The reason links terminating on a particular assertion indicate the set of assertions of subsegments from which the target assertion can be inferred. This proof goes through under the assumption that the subsegments are being used correctly, i.e. that their prerequisites are satisfied. Additional reason links indicate how this can be verified.

III.1 An Example Plan

The figure below shows a plan for the simple FORTRAN program SINCOS. The behavioral description for SINCOS states that it has one input (x) and two outputs (sx and cx). It states, as a prerequisite, that the input x must be a real number. It states, as assertions, that after executing SINCOS, the two outputs produced will be real numbers and that sx will be the sin of x and cx will be the cos of x.

The plan (which is of plan type "and", discussed in the next section) analyzes SINCOS as being composed of two subsegments, one a call on the function SIN and the other a call on COS. Behavioral descriptions describe these subsegments as computing the sin and cos, respectively.

The plan shows how these subsegments combine to produce the behavior of the program as a whole. the relationship between the data items in the various behavioral descriptions is indicated by using identical names. For example, the outputs sx and cx of the two subsegments each become outputs of SINCOS.

In the figure, reason links are represented by putting the line number of the start of a reason link in the reason column of the line where the reason link terminates. For example, there is a reason link from line 7 to line 3 and from line 1 to line 5. The reason links summarize a proof of correctness for the program SINCOS based on the assumption that the descriptions of SIN and COS are correct. For example, it can be concluded that sx=(sin x) (in line 4) is a valid assertion for SINCOS because it is directly supported by an assertion of SIN (in line 8). Theorems like the above are summarized by reason links from parts of lines 8 and 12 to parts of line 4. Other reason links show that the prerequisites of the subsegments are satisfied by the prerequisites of SINCOS. Reason links also show that the claim that SINCOS has two outputs is supported by the fact that SIN and COS each have an output.

For simplicity, the figure does not show how an abstract plan, like the one in the figure, is actually brought into correspondence with a specific program such as SINCOS. This involves issues such as showing that the data flow constructs in the program actually implement the data flow required by the plan.

```
SUBROUTINE SINCOS (X, SX, CX)
SX = SIN(X)
CX = COS(X)
RETURN
END
```

#	REASON	Plan of type "and" for the program SINCOS
		Behavioral Description of the Subroutine SINCOS
1		Inputs: x
2		Prerequisites: (real x)
3	7,11	Outputs: sx,cx
4	8,12	Assertions: (real sx) (real cx) sx=(sin x) cx=(cos x)
		Behavioral description of the call on SIN
5	1	Inputs: x
6	2	Prerequisites: (real x)
7		Outputs: sx
8		Assertions: (real sx) sx=(sin x)
		Behavioral description of the call on COS
9	1	Inputs: x
10	2	Prerequisites: (real x)
11		Outputs: cx
12		Assertions: (real cx) cx=(cos x)

Fig. 1: An example of a plan for the program SINCOS.

III.2 An Example Plan Type: "and"

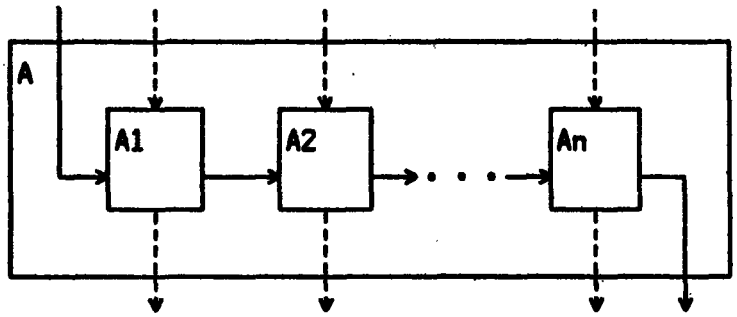
The plan type "and" abstracts the common features from plans such as the one in the previous section. The key feature of that plan, and ones like it, is that several independent subsegments are combined to produce a supersegment in such a way that they do not interact in any way. The behavior of the supersegment is simply the union of the behavior of the subsegments.

The figure below summarizes the structure of the plan type "and". N subsegments (represented by boxes and named A1 ... An) are combined to produce an outer segment A. Solid lines represent control flow, and dashed lines represent data flow. These indicate the fact that each subsegment is executed once, and that an input of a subsegment cannot come from an output of another subsegment.

Equations in the figure summarize how the behavioral description of the outer segment is related to the behavioral descriptions of the inner segments. The operators *I*, *O*, *P*, and *A* select the inputs, outputs, prerequisites, and assertions, respectively, of the behavioral description of a segment. The first equation above the figure shows that the inputs of A are a union of the inputs to the subsegments. The second equation shows that the prerequisites of A are a conjunction of the prerequisites of the subsegments. The first equation below the figure shows that the outputs of A are a union of the outputs of the subsegments. The second equation shows that the assertions of A are the conjunction of the assertions of the subsegments.

It is a prerequisite of the applicability of this plan type that the equation for *A*(A) not be contradictory. If the assertions of two subsegments contradict each other, then *A*(A) becomes vacuous. However, if this is the case, then the two subsegments are interacting and the plan type "and" is not applicable.

$$I(A) = U_{i=1,n}\ I(Ai)$$
$$P(A) = \wedge_{i=1,n}\ P(Ai)$$



$$O(A) = U_{i=1,n}\ O(Ai)$$
$$A(A) = \wedge_{i=1,n}\ A(Ai)$$

PR(Ai) = contributes *O*(Ai) to *O*(A) and *A*(Ai) to *A*(A).

Fig. 2: Schematic for the plan type "and".

The reason links for a plan of this type embody the dependency structure of the equations. For example, there would be a reason link from each output assertion of a subsegment to an output assertion of the outer segment. The final line highlights some of the teleological information by stating that the purpose of each subsegment is to contribute part of the outputs and assertions of A.

It can be seen that the example plan in the last section is of plan type "and". Based on the plan type "and" and the behavioral descriptions of SIN and COS given in the example, FORPAS would derive the behavioral description for SINCOS given in the example.

IV. Basic Loops

this paper looks at plans for loops. Plans for non-looping structures are discussed more fully in [Waters 76]. Consider the loop below which calculates the square root.

```
      SUBROUTINE SQRT (X,Z)
    5 X = 1.0
   10 IF (ABS(X*X-Z)-1.0E-10) 20,20,15
   15 X = (X+X*Z)/2.0
      GOTO 10
   20 RETURN
      END
```

The basic loop plan type analyzes a loop by unrolling it into an unbounded linear calculation.

```
      SUBROUTINE SQRT (X,Z)
    5 X = 1.0
  100 IF (ABS(X*X-Z)-1.0E-10) 200,200,150
  150 X = (X+X*Z)/2.0
  101 IF (ABS(X*X-Z)-1.0E-10) 200,200,151
  151 X = (X+X*Z)/2.0
  102 IF (ABS(X*X-Z)-1.0E-10) 200,200,152
  152 X = (X+X*Z)/2.0
                :
                :
  200 RETURN
      END
```

This structure can be schematically represented as follows.

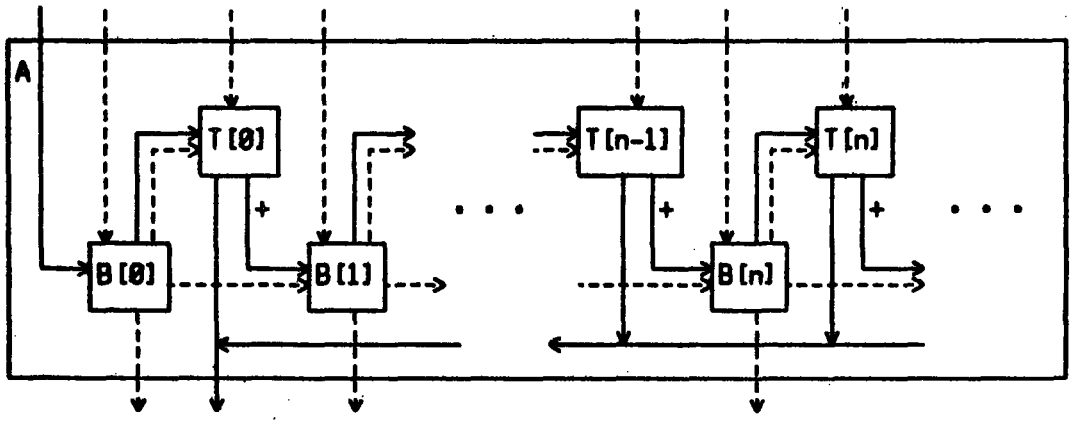


Fig. 3: Schematic for the unbounded computation corresponding to a loop.

In the figure, solid lines represent control flow and dashed lines represent data flow. A plus (+) indicates the true exit from a test. In the FORPAS system, a test is a segment which has no data outputs, and two control flow exits. The assertions in the behavioral description for a test apply to the true exit. On the other exit, the assertions are the negation of the assertions in the behavioral description. Square brackets are used to generate names for the unbounded number of subsegments created by the unrolling. The segments T[i] refer to the segments generated by the original loop's test (line 10). For example,

T[0] is line 100 and T[1] would be line 101. The segments B[i] refer to segments created by the body of the original loop (line 15). For example, B[1] is line 150 and B[2] would be line 151. In order to enhance the repetitive structure of the schematic, the initialization of the original loop (line 5) is represented by B[0]. In a loop where there is no test between the initialization and the first execution of the body, T[0] is vacuous.

The next figure shows a plan for the program SQRT. It lists the behavioral descriptions of the subsegments and the segment as a whole. In these behavioral descriptions, the square bracket notation is used to refer to the outputs of the various subsegments. For example, x[2] is the output of B[2] and an input to T[2] and B[3].

The behavioral description for the loop as a whole records all of the assertions made up to the execution of T[n]. It is assumed that this test has failed (as reflected in line 5 of the plan) causing the loop to terminate. This plan, and the basic loop plan type in general, are only applicable to loops which halt. One obvious problem with the plan is that the value of n is not given explicitly. It is only given implicity in the behavioral description.

```
#  REASON   Plan of type basic loop for SQRT

                Behavioral description for A
1  6,8            Inputs: z        Prerequisites: (real z)        Outputs: x[n]
2  7              Assertions: x[0]=1.0 (real x[0])
3  9                  (∧i=1,n x[i]=(x[i-1]+x[i-1]*z)/2.0 ∧ (real x[i]))
4  11                 (∧i=0,n-1 |x[i]^2-z|>1.0E-10)
5  11                 |x[n]^2-z|≤1.0E-10

                Behavioral description for B[0]
6                  Inputs: _        Prerequisites: _               Outputs: x[0]
7                  Assertions: x[0]=1.0 (real x[0])

                Behavioral description for B[i] i=1,∞
8  1,6,7,8,9       Inputs: x[i-1],z  Prerequisites: (real x[i-1],z)  Outputs: x[i]
9                  Assertions: x[i]=(x[i-1]+x[i-1]*z)/2.0 (real x[i])


10 1,6,7,8,9       Inputs: x[i],z    Prerequisites: (real x[i],z)    Outputs: _
11                 Assertions: |x[i]^2-z|>1.0E-10
```

Fig. 4: An example of a basic loop plan for SQRT.

In order to understand this plan type more fully, it is necessary to look at the basic loop plan type. this plan type is an abstraction of the features which are common to all terminating loops. The reason links in the plan above can only be understood in the context of this plan type. For example, the claim that line 11 inplies lines 4 and 5 is based on the fact that the subsegments interact according to the basic loop plan type.

The next figure summarizes the basic loop plan type. The graphic portion of the figure is a condensed form of the unbounded schematic above. The box labeled T stands for the subsegments T[i]. The box B stands for the subsegments B[i]. The box I stands for B[0] and optionally T[0] depending on how the loop is initialized and started.

As represented in the figure, the structure of a basic loop is as follows. There is a test (T) and a body (B) which are repetitively executed one after the other. The loop terminates as soon as the test fails. the loop is started by an initialization (I) which performs B[0] and then starts the loop at either point X (as in SQRT) or point Y in which case T[0] is vacuous.

T and B can have inputs both from outside the loop (for example, z in SQRT) and from previous

executions of B (for example, x[i] in SQRT). Two new operators IX and II are used to refer to the inputs which are external to the loop and internal to the loop respectively. Two other operators PX and PI are used in recognition of the fact that the prerequisites of T and B can also be divided into two sets: those which are externally satisfied and those which are internally statisfied by earlier executions of T and B.

As in the schematic for the "and" plan type, equations above and below the figure show how the behavioral description of the loop as a whole is related to the behavioral descriptions of the subsegments. The equations beside the figure give auxiliary information.

In the figure, n represents the iteration at which the loop will terminate. The first equation states that n is the first iteration where the test fails. The second equation states that the inputs to the loop are the union of the external inputs to I, B, and T. The third equation states that the prerequisites of the loop are the conjunction of the external prerequisites of I, B, and T.

The equations beside the figure introduce the new operators *IX*, *II*, *PX*, and *PI*. Then they state that the internal inputs are provided for by internally generated outputs. They also state that the internal prerequisites are satisfied by internally generated assertions. The notation R[i] is introduced as an abbreviation for the conjunction of the assertions of all of the subsegments executed up to and including B[i] (i.e., $R[i] = A(B[0]) \wedge A(T[0]) \wedge A(B[1]) \wedge \ldots \wedge A(T[i-1]) \wedge A(B[i])$).

The first equation below the figure states that the set of outputs of the loop is a union of all of the outputs of I and B. The second equation states that the assertions of the loop are a conjunction of the assertions of the subsegments executed up to the time it terminates.

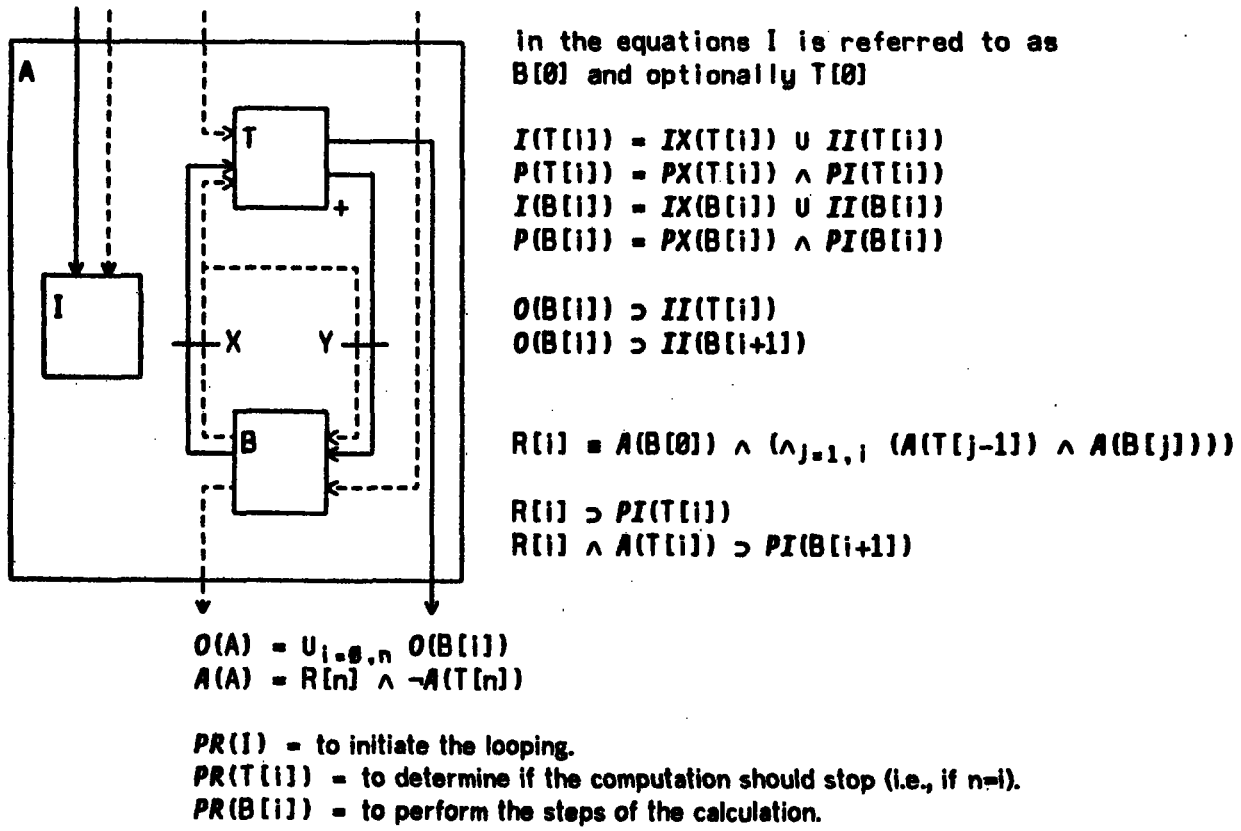


Fig. 5: Schematic for the basic loop plan type.

It should be noted that the equations above try to capture the full complexity that a loop could

have. Most actual loops do not make use of all the complexity possible. As a result, simplifications usually occur in the equations. For example, the expressions for the inputs and prerequisites are often simplified. B and/or T may not have any external inputs or prerequisites. Failing this, the expressions for I(A) and/or P(A) may not depend on n (as in SQRT). In addition, outputs often come only from the last step (as in SQRT). Unfortunately, A(A) seldom simplifies very much.

IV.1 Recognizing a Basic Loop

Consider how a piece of a program can be recognized as being of the basic loop plan type. When a cycle is discovered in the flow of control for a program, all of the code in the cycle is considered to be part of the loop. Code which does a test and which can branch out of the cycle is considered part of T. The rest is taken as B. Any code outside the cycle which has data flow only into T and B is considered potentially part of I. Additional clues to the grouping can be gained from comments, and the syntactic realization of the program. For example, the code for a loop is often surrounded by comments. Also the use of a syntactic statement such as "DO" gives information about the extent of the loop. One complication is that the program may have been transformed (for example, to promote efficiency) in such a way that the basic structure is obscured. Common transformations and methods for dealing with them are discussed in [Waters 76].

IV.2 Verifying a Claim About a Basic Loop

The basic loop plan type gives $P(A)\{A\}A(A)$ as a description of the behavior of A. Furthermore it makes the claim that this is a complete specification of the actions of the loop. As a result, in order to prove a claim $P\{A\}Q$ about the loop one need only show that $P \supset P(A)$ and $P \wedge A(A) \supset Q$.

Consider this from the point of view of verifying $P\{A\}Q$ by finding an appropriate loop invariant V at point X in the figure above. An invariant V is appropriate if and only if $P \supset V$ on entrance to the loop, $V \wedge A(T)\{T+B\}V$ each time around the loop, and $V \wedge \neg A(T) \supset Q$ on exit from the loop. The basic loop plan type claims that if there is any such V then a sufficient V is $V' = P \wedge R[i]$. In V', "i" refers to the number of times around the loop so far. Comparison of this equation with the ones in the figure above shows that $P \supset V'[0]$ if $P \supset P(A)$, $V'[i] \wedge A(T[i])\{T+B\}V'[i+1]$ for $0 \leq i < n$ if and only if $P \supset P(A)$, and that $V'[n] \wedge \neg A(T[n]) \supset Q$ is equivalent to $P \wedge A(A) \supset Q$.

Given a loop and a claim, the basic loop plan type specifies what theorems must be proved in order to verify the claim. Unfortunately, this usually isn't very helpful because $P \wedge A(A) \supset Q$ is generally difficult to prove. The basic problem comes from the fact that A(A) is in a clumsy form and is powerful enough to prove anything which can be proved about the loop. For a given Q, A(A) usually has excess information which gets in the way of proving $P \wedge A(A) \supset Q$. Methods which try to develop a more reasonable V by starting from Q fall into the reverse difficulty that Q usually contains too little information to yield a sufficient V.

There are several aspects of A(A) which lead to particular difficulties. First, it depends on n which is not explicitly known. Second, it refers to intermediate values created during the loop's execution which may not be outputs of the loop as a whole. A typical Q will not mention any intermediate values. Third, Q often makes some summary statement about the outputs whereas A(A) only gives what is essentially a recurrence relation relating the inputs to the outputs. These features usually combine to assure that the conceptual distance between $P \wedge A(A)$ and Q is large, thereby complicating the proof of $P \wedge A(A) \supset Q$. The plan method attacks this problem by decomposing a loop into subparts in order to develop a statement of A(A) which is conceptually closer to Q.

V. Enumeration Loops

If everything is removed from a loop except for the computation of n, then what remains is an enumeration loop. An enumeration loop generates a sequence of values of its variables. This type of loop is important as a basis for building up more complex loops. The example below shows the

prototypical enumeration loop in the mathematical domain. There are other types of enumeration loops, such as ones that count down, instead of up.

```
   J = INIT
10 J = J+INC
   IF (J-END) 10,10,20
20 ...
```

Behavioral Description According to the Basic Loop Plan Type

$I(A)$ = {INIT, INC, END}
$P(A)$ = TRUE
$O(A)$ = {J}
$A(A)$ = $J[0]=INIT \wedge (\wedge_{i=1,n}\ J[i]=J[i-1]+INC) \wedge (\wedge_{i=1,n-1}\ J[i]\leq END) \wedge J[n]>END$
n such that $((\wedge_{i=1,n-1}\ J[i]\leq END) \wedge J[n]>END)$

Behavioral Description According to the Enumeration Loop Plan Type

$I(A)$ = {INIT, INC, END}
$P(A)$ = $INIT\leq END \wedge INC>0$
$O(A)$ = {}
$A(A)$ = $\wedge_{i=0,n}\ J[i]=INIT+i*INC$
n = $1+\lfloor(END-INIT)/INC\rfloor$

Fig. 6: The prototypical enumeration loop in the mathematical domain.

The behavioral description generated by the enumeration loop plan type is superior to the behavioral description generated by the basic loop plan type in several ways. The value of n is given by an explicit equation. As a result, there is no doubt about the loop's termination. the values of J are also given by an explicit equation. Due to the changes in $P(A)$ and $O(A)$ the enumeration loop plan type's behavioral description is weaker, however, it is more understandable and to the point. For example, the new form of $P(A)$ highlights the fact that INIT>END and INC≤0 are commonly associated with bugs in an enumeration loop of this kind.

V.1 Recognizing an Enumeration Loop

The key recognizable features of an enumeration loop are the fact that its output is not used anywhere, the fact that all of the body contributes to the calculation of n, and the simple form of the recursion relation in $A(A)$ (as produced by the basic loop plan type). These loops can also be recognized by the fact that they are often written in stereotypical syntactic forms such as:

```
   DO 10 J=INIT,END,INC
10 CONTINUE
```

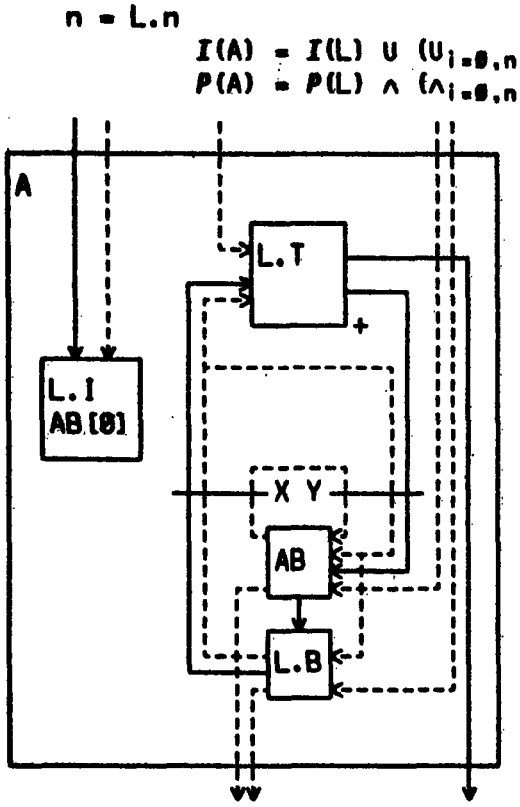
V.2 Verifying a Claim About an Enumeration Loop

There are not very many things one would like to prove about an enumeration loop. Its usefulness comes from the fact that it can be used to prove things about more complex loops (see below). The enumeration loop plan gives a clear statement of what each value of J is and what n is.

VI. Augmented Loops

The plan method is able to understand a complex loop by decomposing it into a set of parts which can be understood totally or partially in isolation from each other. One method of decomposition is based on locating parts of the body of a loop which do not affect the termination of the loop.

The augmented loop plan type looks at a loop as being composed of a basic loop "L" (often an enumeration loop) to which an augmented body "AB" has been added. The only restriction on the way AB and L.B (the body of the loop L) interact is that no data can flow from AB to L.B or T. The subsegment I may be modified by the addition of AB[0] in order to initialize the actions of AB.



PR(I) = to initiate the looping.
PR(T[i]) = to determine if the computation should stop i.e. if n=i.
PR(L.B[i]) = to perform the calculation for loop L.
PR(AB[i]) = to perform the added steps of calculation.

```
      DO 10 I=1,10                    Z = 1.0
                                      DO 10 I=1,10
10    X(I) = Y(I)*Z(I)          10    Z = Z*X
```

Fig. 7: Schematic for, and examples of, the plan for an augmented loop.

There are two basic subclasses of the augmented loop plan type. The distinction is based on whether or not there is data flow from AB to itself. If there is no feedback, then AB is referred to as an AND augmentation, otherwise, it is referred to as a COMP(osition) augmentation. A loop can often be analyzed as having several separate augmentations.

VI.1 Recognizing Augmentations

Augmentation is recognized by analyzing the data flow in a loop. The parts of a loop which can affect termination are just those parts which can have direct, or indirect, data flow to the test. The rest of the body is one or more augmentations. It is decomposed into as many augmented bodies as possible without violating the following restriction. It must be possible to arrange the bodies in a tree with the body of the basic loop at the root in such a way that each body has data flow only to its inferiors in the tree and optionally to itself. This restriction allows each body to be understood with reference only to its superiors in the tree. Consider the following loop as an example.

```
1 J = 10
2 DO 5 K=1,10
3 J = J+1
4 A(K) = B(K)
5 C(K) = D(J)
```

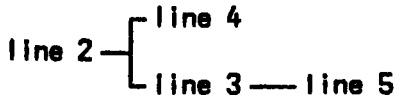


Fig. 8: An example loop with three augmentations.

The example above can be analyzed as being an enumeration loop (line 2) with three additional bodies (lines 3, 4 and 5). The tree on the right shows that the restriction is satisfied. For example, it shows that line 4 can be understood without reference to lines 3 or 5. Line 1 is an initialization related to the additional body in line 3.

VI.2 Verifying a Claim About an Augmented Loop

An important feature of the augmented loop plan type is that any claim about the termination of the loop can be reduced to a question about the termination of the basic loop L (which is usually an enumeration loop).

Consider AND augmentations such as line 4 in the example above. If the basic loop plan type was able to single out the actions of line 4 from the actions of the loop as a whole (which it can't) it would summarize the actions of this line as: $(\wedge_{i=1,n}$ A[i](K[i-1])=B(K[i-1]) other elements of A[i] unchanged). The augmented loop plan type, in conjuction with the enumeration loop plan type, would produce the following summary: $(\wedge_{i=1,n}$ A(i)=B(i) other elements of A unchanged). If A and B were both of dimension 10, then FORPAS could conclude that A was not an input to the loop and summarize the actions of line 4 as: A=B.

The above would be achieved through a sequence of steps. The augmented loop plan type would separate out the augmented body in line 4 and the enumeration loop in line 2. The enumeration loop plan type would conclude that K[i]=i+1 for $0 \le i \le 10$. FORPAS' knowledge of vectors would tell it that line 4 is an AND augmentation since B is distinct from A and K can never have the same value twice. This leads to the second summary above. Further reasoning about vectors would lead to the third summary. Knowledge of vectors and matrices is an important part of FORPAS' ability to understand programs in general and loops in particular. This is further discussed in [Waters 76].

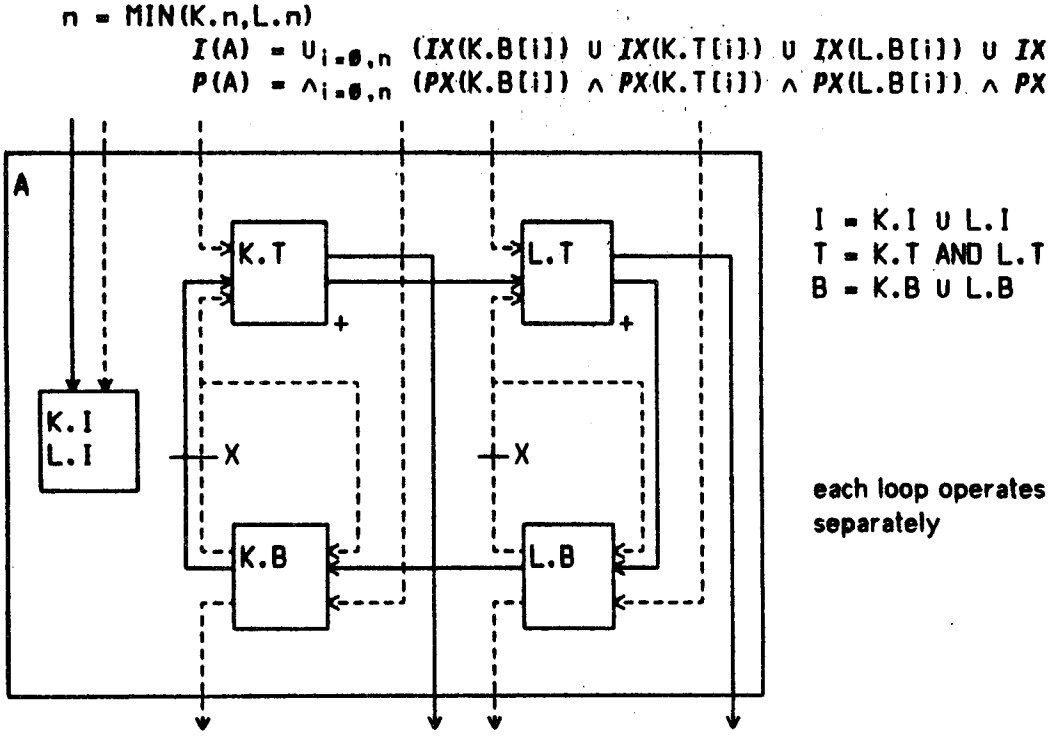
In general, analysis of an AND augmentation leads to a convenient description of its behavior. As a result, a reasonable Q will usually be easy to prove. COMP augmentation, such as line 3 in the example above, is similar, but somewhat more complex. The basic loop plan type would summarize the actions of line 3 as: $J[0]=10 \wedge (\wedge_{i=1,n}$ J[i]=J[i-1]+1) (if it could single the actions out). The augmented loop plan type would produce the following summary: $(\wedge_{i=0,10}$ J[i]=10+i).

This is possible because the augmented loop plan type is able to isolate the computation of J and because FORPAS is able to recognize a few of the most common recurrence relations. In a situation where FORPAS is not able to simplify the recurrence relation involved, it is still valuable to be able to isolate the difficult part of the loop from the other parts of the loop. Further, the requirements for a loop sometimes consist of a recurrence relation, in which case a recurrence relation is a reasonable form

for *A*(A).

VII. Interleaved loops

This plan type combines two loops, K and L, so that they are computed in synchrony. The combination terminates as soon as either one terminates. The segments K.T, L.T, K.B, and L.B are executed in any order in a ring. Initialization starts the loop at any of four points. The figure below shows one possible arrangement. The key requirement is that there is no data flow between the two subloops. Their only interaction is that when one terminates, the other is artificially stopped. It should be noted that when one of the subloops (say K) terminates, then the whole loop A acts in all respects just like loop K with some additional computation from the partial execution of the other subloop (L). In particular, the outputs are the outputs of K with the addition of a subset of the outputs of L.



PR(I) = to initiate the looping.
PR(L.T[i]) = to see if the computation should stop due to loop L.
PR(K.T[i]) = to see if the computation should stop due to loop K.
PR(L.B[i]) = to perform the steps of the calculation for loop L.
PR(K.B[i]) = to perform the steps of the calculation for loop K.

```
    X = 20.0
    DO I=1,10
    X = F(X)
    IF (X) 10,20,10
10  CONTINUE
20  ...
```

Fig. 9: Schematic for, and example of, the plan type interleaved loop.

One of the most important attributes of this plan type is that it can be shown to terminate even if only one of the two loops K and L can be shown to terminate. As a result of this, the interleaved loop plan type is often used to bound the execution of a possibly non-terminating loop with a simple enumeration loop (see the example).

The other major use of this plan type is based on considering that it computes either K or L, which ever completes first. Here, "first" is defined in terms of the sequence of states produced by the loops. This plan type is used in a situation where, for example, the results of L are not desired (or perhaps are not computable) if K terminates first.

VII.1 Recognizing an Interleaved Loop

The key features of an interleaved loop are that there are multiple exits from the loop, and that each exit can be associated with a distinct body. The plan type can be used even when the internal bodies are not distinct by logically duplicating part or all of the body of the loop. Similarly the basic loop plan type can be applied to a multiple exit loop by converting the loop into a single exit loop with a complex predicate in the test.

VII.2 Verifying a Claim about an Interleaved Loop

From the point of view of verification, there are two primary aspects of this plan type. First, it provides a decomposition of a loop into two subloops which can be understood separately. Second, it simplifies the problem of proving termination to the problem of proving that either one of the subloops terminates.

In the example, the interleaved loop plan type, in conjunction with the other plan types dicussed above, allows FORPAS to conclude that: $n \leq 10.0$, $X = F^n(20.)$, $(\wedge_{i=1,n-1} F^i(20.) \neq 0.0)$, and $X = 0.0 \vee n = 10.0$. This is a considerable improvement over what the basic loop plan type would conclude.

VIII. An Experiment

The basic loop plan type reduces the problem of verifying $P\{A\}Q$ to the problem of proving $P \supset \mathcal{P}(A)$ and $P \wedge \mathcal{A}(A) \supset Q$. The other plan types look deeper into the internal structure of A in order to develop a more convenient form for $\mathcal{A}(A)$. The claim is that $P \wedge \mathcal{A}(A) \supset Q$ will then be relatively easy to prove because the conceptual distance between $\mathcal{A}(A)$ and Q will be small. Introspection suggests that the form of a typical Q is compatable with the extended plan types, when they apply. An experiment was undertaken to determine how often the extended plan types apply.

A manual analysis of a random sample consisting of 44 (20%) of the 221 programs in the IBM SSP subroutine package [IBM 70] was performed. This analysis yielded the following results. The 44 programs contained 164 syntactic loops (3 had no loops and 3 had 14 or more loops). It was possible to apply the loop plan types 476 times in order to decomposed the 164 loops into 476 subparts. The plan type enumeration loop was used 160 times (34%). The plan type basic loop was used only 21 times (4%). The plan type interleaved loop was used 22 times (5%). There were 72 AND augmentations (15%) and 201 COMP augmentations (42%). Of the 201 COMP augmentations, 127 (63%) were simple sums, products, counts, maximums, and minimums, 30 (15%) presented definite difficulties, and 44 (22%) were intermediate, yielding straightforward recurrence relations which did not have obvious closed form solutions.

Of the 181 (160+21) subparts which were loops, 88% were enumeration loops and 12% were more general loops. All but 1 of the 21 general loops were combined with enumeration loops through the use of the interleaved loop plan type. As a result, dealing with termination presented a problem with only 1 of the 164 syntactic loops. Of the 273 (72+201) augmentations, 73% (72+127) were cleanly handled, 11% (30) presented significant difficulties, and 16% (44) were of intermediate difficulty.

In summary, 164 loops were decomposed into 476 subparts, 80% (381) of which were cinveniently dealt with as AND augmentations, simple COMP augmentations, enumeration loops, and interleaved loops. A residue of 11% (51) of the subparts was composed of general loops and complex COMP augmentations.

Some program annotation would probably be required in order for FORPAS to develop a full understanding of this residue.

IX. Conclusion

The plan method analyzes the structure of a program. The plan which results from applying the method represents this structure by specifying how the parts of the program interact. The plan can be used to explain the program, find bugs in the program, aid in modifying the program without introducing new bugs, and aid in verifying the program.

This paper demonstrated the utility of the plan method by showing how a plan for a loop can be used to help prove the correctness of a loop. The plan does this by providing a convenient description of what the loop does. This paper also showed how a plan for a loop can be developed based on the code for the loop without the assistance of any commentary. This is possible primarily because most loops are built up in stereotyped ways according to a few fundamental plan types. An experiment was presented which supports the claim that a small number of plan types cover a large percentage of actual cases.

The long range goal of this research is to develop a programming assistant system. Plans will be the basis of that system. In order for the system to aid a programmer, it will have to have a plan for the program being worked on. Often, the easiest way for the programmer to communicate part of the plan to the system will be for him to write part of the program. However, commentary by, and interaction with, the programmer will also be used to aid the system in developing a complete plan for the program.

When using the programming assistant system, the major contribution of the programmer will be to design the plan. The system will understand the plan, and how the program implements the plan. The system's major contribution will be to be a devil's advocate with regard to the plan and the program. It will attempt to find any flaws in the plan, and in the way the program implements the plan.

BIBLIOGRAPHY

Basu, S. & Misra, J. [75] "Proving Loop Programs", IEEE Trans. on Software Eng. V1 #1, March 1975.

Basu, S. & Misra, J. [76] "Some Classes of Naturally Provable Programs" Proc. 2nd Int. Conf. on Software Engineering, pp. 400-406, Oct 1976.

Floyd, R.W. [67] "Assigning Meaning to Programs", proc. symp. in Appl. Math. V19, pp. 19-32, 1967.

German, S. & Wegbreit, B. [75] "A Synthesiser of Inductive Assertions" IEEE Trans. on Software Eng. V1 #1, March 1975.

Goldstein, I.P. [74] "Understanding Simple Picture Programs", MIT AI-TR-294, Sept. 1974.

Hewitt, C. & Smith, B. [75] "Towards a Programming Apprentice", IEEE Trans. on Software Eng. V1 #1, pp. 26-46, March 1975.

Hoare, C.A.R. [69] "An Axiomatic basis for Computer Programming" CACM V12 #10, pp. 576-583, Oct 1969.

IBM GH20-0205-4 [70] "Scientific Subroutine Package Version III Programmer's Manual", White Plains N.Y., 1970.

Katz, S. & Manna, Z. [76] "Logical Analysis of Programs", CACM V19 #4, pp. 188-206, April 1976.

Morris, J.H. jr & Wegbreit, B. [77] "Subgoal Induction", CACM v20 #4, pp. 209-222, April 1977.

Rich, C. & Shrobe, H. [76] "Initial Report on a LISP Programmer's Apprentice", MIT AI-TR-354, Dec. 1976.

Sussman, G. [73] "A Computational Model of Skill Acquisition", MIT AI-TR-297, Aug. 1973.

Waters, R. [76] "A System for Understanding Mathematical FORTRAN Programs", MIT AIM-368, Aug. 1976.

Wegbreit, B. [74] "The Synthesis of Loop Predicates", CACM V17, pp. 102-112, Feb. 1974.