A Method for Analyzing Loop Programs

RICHARD C. WATERS, MEMBER IEEE

Abstract—This paper presents a method for automatically analyzing loops, and discusses why it is a useful way to look at loops. The method is based on the idea that there are four basic ways in which the logical structure of a loop is built up. An experiment is presented which shows that this accounts for the structure of a large class of loops. The paper discusses how the method can be used to automatically analyze the structure of a loop, and how the resulting analysis can be used to guide a proof of correctness for the loop. An automatic system is described which performs this type of analysis. The paper discusses the relationship between the structure building methods presented and programming language constructs. A system is described which is designed to assist a person who is writing a program. The intent is that the system will cooperate with a programmer throughout all phases of work on a program and be able to communicate with the programmer about it.

Index Terms—Loops, plans, program analysis, program verification, program understanding.

I. Introduction

THIS paper presents one part of a general method (described in full in [28]) for analyzing programs. The result of an analysis is a "plan" which represents the underlying logical structure of a program. The plan directly specifies how the parts of the program interact in order to produce the behavior of the program. It abstracts away from the surface syntactic details of the code for a program and is substantially programming language independent. The analysis method is based on the observation that plans are built up in a small number of stereotyped ways referred to as plan building methods (PBM's). This paper discusses the four PBM's which build up plans for loops.

An important application of PBM's is that they can be used to analyze a loop in a way that makes it easier to understand what the loop does and why. Section II shows why this is a more useful analysis than one based on basic structured programming constructs. Section III describes the PBM's in detail from the point of view of their use in analysis. Section IV discusses how an analysis of a loop in terms of the PBM's can be used to guide a proof of correctness for the loop. It also shows why the resulting proof is more useful than a proof based on a single loop invariant.

An experiment is discussed in Section V which shows that the PBM's can be used to analyze the loops in a representative sample of programs from the IBM Scientific Subroutine Package (SSP) [12]. The SSP was chosen as an object of study because it is a large group of clearly written programs which is an actual commercial product. The experiment also shows that the pieces which result from the analysis are largely simple and easy to understand. A system (described in Section VI) has been implemented which performs this type of analysis automatically. Section VII describes the relationship between the PBM's and current programming language constructs. It also discusses how a language could be extended in order to include constructs based on the PBM's.

The method for analyzing the logical structure of loops presented in this paper was developed as part of a larger research project. The goal of this project is to develop a system which can assist a person who is writing a program. Research on this system [20]-[23], [25], [27], [28] is being carried out by a group consisting of C. Rich, H. Shrobe, and the author. The intent is that the system will cooperate with a programmer throughout all phases of work on a program and be able to communicate with him about it. The system is described in Section VIII.

II. Desiderata For an Analysis Method

An analysis method views a program as built up out of parts, and makes it possible to understand the relationship between the operation of the program as a whole and the operation of its parts. From the point of view of gaining an understanding of a program, the parts are subproblems. Once the parts have been understood, their understandings can be combined in order to obtain an understanding of the whole.

There are several criteria which can be used to evaluate the usefulness of an analysis method. First, given a program, it should be straightforward to identify the parts. Second, the parts should be easier to understand than the whole. Third, the process of developing an understanding of the whole based on understandings of its parts should be as easy as possible.

The development which is most similar to PBM's has been the development of structured programming constructs. Consider the loop program in Fig. 1. A naive approach to looking at how this program is logically built up is based on the idea that it is constructed on a line by line basis. Given this approach, it is easy to analyze the program in order to break it up into its component parts (the six lines of the program).

Manuscript received January 6, 1977; revised January 5, 1979. This paper 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 N00014-75-C-0643.

The author is with the Artificial Intelligence Laboratory and the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

```
       I = 1;
       Z = 0;
LOOP:  IF NOT(A(I)>0) THEN GOTO SKIP;
       Z = Z+A(I);
SKIP:  I = I+1;
       IF I<N THEN GOTO LOOP;
```

Fig. 1. An example program.

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

Fig. 2. The example program from Fig. 1 in structured form.

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

Fig. 3. The example program from Fig. 1 analyzed by PBM's.

Further, the parts are indeed much easier to understand than the program as a whole. However, the problem with this approach is that it is not at all easy to discover what the program does once the parts are understood, because the effects of the lines upon each other are complex.

The basic structured programming constructs (composition, if-then-else, and do-while) suggest a much better analysis method. They indicate that the program should be viewed logically as being built up hierarchically using these structured programming constructs. Fig. 2 depicts the program in Fig. 1 analyzed in terms of the three basic structured programming constructs. The program is analyzed as being a composition of "Z=0" with an extended form of do-while which consists of counting from 1 to N in I and a body which is an if-then-else consisting of a predicate "A(I)>0" and a then clause "Z=Z+A(I)."

When it is possible without transforming a program, this is an easy analysis to perform. This is true whether or not the program is written in a syntactically structured way. It is possible to write a program which cannot be analyzed in terms of the basic structured programming constructs, unless it is first transformed to change the topology of its control flow (for example, a program which contains a loop with more than one entry point). However, a large number of programs can be directly analyzed in this way. In any case, the parts are easy to understand once they are isolated.

Let us now look at the structured programming oriented approach in the light of how easy it is to develop an understanding of the result based on understandings of its parts. First consider if-then-else. If the parts are understood, then it is easy to get an understanding of the whole. Namely, in a given situation an if-then-else either acts like the then clause, or like the else clause, depending on the value of the predicate. In Fig. 2 the if-then-else adds A(I) to Z if A(I)>0. Composition is also an easy operation to understand. In general, these two structured programming constructs, which describe nonlooping programs, meet the criteria set forth above very well. The general method described in [28] uses five PBM's closely related to these two structured programming constructs in order to analyze straight-line sections of programs.

Consider the construct do-while in the light of how easy it is to develop an understanding of the resulting program based on understandings of its parts. The body of the loop in Fig. 2 can be understood as a conditional which adds A(I) to Z if A(I)>0. Unfortunately, it is not easy to go from this to an understanding that the loop adds the sum of the positive members of the first N elements of A to the initial value of Z. It is easy to conclude this if an appropriate loop invariant can be found. However, in general, it is not easy to find such an invariant.

Another problem with the analysis in the figure is that the close relationship between the statements "Z=0" and "Z=Z+A(I)" is not made clear. In order to meet the goal of analyzing a program in such a way that things which are intimately related are closely linked together, these two statements should be put together in a single locality distinct from the rest of the loop. Having the statements spread through the loop makes it harder to understand that the program as a whole computes the sum of the positive members of the first N elements of A.

The difficulties with do-while stem from the way it looks at a loop. The body of the loop is first analyzed like any other straight-line program. This understanding of the body is then bootstrapped up to an understanding of the loop as a whole. The problem is that this bootstrapping process is far from automatic. The PBM's for loops take a different approach. They are based on the idea that the body of a loop should not be analyzed in the same way as a straight-line program. Instead, they break the loop up in order to analyze it as built up out of stereotyped loop fragments. The lines "Z=0" and "Z=Z+A(I)" are an example of just such a fragment.

The PBM's for loops break the loop in Fig. 2 up into three fragments, as shown in Fig. 3. The first fragment (A) counts up by one from one to N. It enumerates the sequence of integers $\{1,2,\cdots N\}$. (Note that this fragment corresponds to the DO construct itself.) The second fragment (B) operates on the sequence of integers produced by the first. It restricts the sequence by selecting only those integers which correspond to positive elements of the vector A. The last fragment (C) computes the sum of the elements of A corresponding to the integers in the restricted sequence produced by B. In the loop as a whole, the three fragments are cascaded together so that the loop computes the sum of the positive members of the first N elements of A.

The key feature of the analysis above is that it breaks the loop apart along a different dimension from the one used by an analysis in terms of do-while. There are two principle advantages to looking at a loop in this new way. First, the loop is broken up into pieces which correspond to easily understood stereotyped fragments of looping behavior. Second, the way the pieces are combined is logically equivalent to composition, which makes it easy to understand.

In order to make the idea that the fragments of the loop are composed together precise, the notion of a temporal sequence of values has been developed jointly by H. Shrobe [25] and the author [28]. Given a program, such as loop, which is repetitively executed, it can be useful to talk about the sequence of states in which some part of the program is executed, and about the sequences of values available in those states. For example, consider the statement "Z=Z+A(I)" in the loop in Fig. 2. This statement is executed in a sequence of states. There are sequences of values of I and Z which are available in those states. These sequences are referred to as temporal sequences of values. The insight is the realization that, logically, a temporal sequence of values can be treated in the same way as any aggregate data object. The concept of lazy evaluation [5], [8] uses the same insight going in the other direction. If an aggregate data object (such as sequence of numbers) is desired, then it can be created temporally, rather than all at once, so that each piece of it is not actually created until it is needed.

Looking back at Fig. 3, fragment A produces a temporal sequence of values of I. The elements of this sequence are tested by fragment B. Logically, the key relationship is that A creates data used by B. A is composed with B by passing this data from A to B. Viewed abstractly, it makes no difference whether these data are put into a vector which is passed all at once to B, or, as in the example, A and B are intermingled so that B can use each individual value created by A as soon as it is produced. Intermingling A and B is just an efficient way of implementing the data flow from A to B.

The key property of composition which makes it an easy process to understand is that the only interaction between two things which are composed together is that one passes data to the other. They have no other effect on each other. The loop fragments above have this vital property. Fragment A will produce a sequence of values of I counting up from one no matter what is happening in the rest of the loop. Fragment B tests the values of I it sees no matter what else is going on in the loop. As a result, each of the fragments can be understood completely in isolation from whatever loop they are being used in.

III. The Loop Plan Building Methods

This section is divided into four subsections which describe four PBM's which create loop fragments and combine them together. Each subsection first describes how a loop can be built up in accordance with the PBM being described. It then discusses how this process can be reversed in order to analyze a loop in terms of the PBM. Section VI describes a system which can perform this analysis automatically. The remainder of each subsection discusses how the PBM leads to an understanding of a loop.

A. The Augmentation PBM

Given a loop, it may be extended to perform additional calculations by augmenting it by adding an additional fragment into it. Abstractly, the augmentation is composed with the loop by putting it into the loop so that it can use temporal sequences of values generated by the loop. An augmentation consists of two parts: a body and an initialization. The body is a piece of code with one entry point and one exit point which is placed somewhere in the control flow of the loop. In Fig. 4 the augmentation body "X=X+Z*A(I)" may be added to the loop (the same loop as in Fig. 2) at positions C, D, E, or F. The augmentation body may use data values from outside the loop (for example, A). It may use values computed by the rest of the loop (for example, Z and I). It may use values computed by earlier executions of itself (for example, X). It may provide values which will be used outside of the loop (for example, X). However, it is not allowed to affect any values which are used by the rest of the loop (for example, I, Z, or A).

The augmentation initialization is a piece of code which is placed somewhere before the loop. In Fig. 4 the augmentation initialization "X=0" may be put at positions A or B. The augmentation initialization provides values which are used by the augmentation body. It may not affect any values which are used by the rest of the loop. When an augmentation body uses values computed by earlier executions of itself, then an augmentation initialization will usually provide the values to be used by the first execution of the augmentation body, as in the example.

When an augmentation is added to a loop, a more complex loop results. It is then possible to add another augmentation to the new loop. This second augmentation can use values computed by the first augmentation. In Fig. 4 the augmentation is being added to a loop which already has an augmentation (i.e., fragment C in Fig. 3).

The key restriction that there may be no data flow from an

```
        a loop                  an augmentation
A:                                X = 0;
  Z = 0;                          X = X+Z*A(I);
B:
  DO I=1 TO N;
C:
    IF A(I)>0 THEN DO;
D:
      Z = Z+A(I);
E:
    END;
F:
  END;
```

the result of adding the augmentation to the loop at points A and D

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

Fig. 4. An example of an augmentation.

augmentation to the rest of the loop is the basis for the logical properties of the augmentation PBM. This restriction can also be used to locate augmentations when a loop is being analyzed. Once the data flow in a loop has been determined, augmentation bodies can be recognized by locating minimal subsegments of the code for the loop which do not have data flow going to other parts of the loop. The corresponding augmentation initializations can be recognized by seeing what parts of the initialization of the loop have data flow to the augmentation body. Consider the loop at the bottom of Fig. 4. The line "X=X+Z*A(I)" can be identified as a subsection of the loop which does not have any data flow to other parts of the loop; the line "X=0" can then be identified as the corresponding augmentation initialization because it is the only part of the initialization for the loop which has data flow to the augmentation body. After this augmentation has been removed from the loop, a similar analysis reveals that "Z=0; Z=Z+A(I);" is also an augmentation.

Suppose a complex loop is analyzed by the augmentation PBM into a simpler loop, and an augmentation. The complex loop can then be understood in basically the same way as a composition. The simpler loop can be understood in isolation by recursively analyzing it using the methods being described here. Continued decomposition of the simpler loop will eventually lead to a basic loop which can be understood as discussed in Section III-C. The fact that there is no data flow from the augmentation to the simpler loop means that the addition of the augmentation does not alter the behavior of the simpler loop. The complex loop does everything that the simpler loop does. For example, it terminates if and when the simpler loop terminates.

What the augmentation does can also be understood in isolation. The augmentation can be looked at as taking temporal sequences of inputs (one for each variable it uses) and producing temporal sequences of outputs (one for each variable it assigns to). It is easy to develop recurrence relations which specify its behavior. For example, consider the augmentation being added in Fig. 4 "X=0; X=X+Z*A(I);". It is easy to develop the recurrence relation "$X_0=0 \wedge X_{i+1}=X_i+Z_i*A(I_i)$" which describes its behavior. Further, it is easy to recognize that this recurrence relation computes a sum and therefore "$X_{i+1}=\Sigma_{i=0,N}Z_i*A(I_i)$." The experiment in Section V shows that 89 percent of the augmentations in the loops studied could be recognized as either a product, a sum, a count, a max, a min, or as a trivial recurrence relation in which prior values of X_i do not appear (for example, "$X_i=2*Z_i$").

In order to put the understandings of the simpler loop and the augmentation together to gain a complete understanding of what the complex loop does, it is necessary to consider where the augmentation body is placed in the simpler loop in order to know what temporal sequences of values it receives. In the example in Fig. 4 it is necessary to know what values of I are received by the augmentation body in order to know which elements of A are selected. If the augmentation is placed at points C or F, it will receive all the first N elements of A. If it is placed at points D or E, it will receive only the positive members of the first N elements of A.

Once the simpler loop, the augmentation, and the temporal sequences of values it receives have been understood, the complex loop can be understood as simply the composition of what the simpler loop does, with what the augmentation does with the particular sequences of values it receives.

```
            a loop                    a filter
A:
  Z = 0;                         IF A(I)<100 THEN
B:
  DO I=1 TO N;
C:
     IF A(I)>0 THEN DO;
D:
          Z = Z+A(I);
E:
     END;
F:
  END;
```

the result of adding the filter to the loop at point E

```
Z = 0;
DO I=1 TO N;
   IF A(I)>0 THEN DO;
      Z = Z+A(I);
      IF A(I)<100 THEN;
   END;
END;
```

Fig. 5. An example of a filter.

B. The Filtering PBM

The filtering PBM is a special case of the augmentation PBM. A filter is an augmentation whose body is an if-then-else with a null then clause and a null else clause. As an augmentation, a filter does not affect any of the values used by the rest of the loop. The filter takes temporal sequences of values as inputs and produces restricted sequences of values as outputs. These restricted sequences of values can be used by putting augmentation bodies into the then clause and the else clause of the filter. The purpose of the filter is to create the execution environments where these restricted sequences are available. Fig. 5 gives an example of a filter being added to a loop. Filtering can be recognized as a special case during the process of recognizing augmentations. All augmentations which are if-then-elses containing only a predicate are filters.

Suppose that a complex loop is analyzed by the filtering PBM into a simpler loop and a filter. The simpler loop can be understood in isolation. The filter can also be understood in isolation as restricting temporal sequences of values. The filter will be executed in some sequence of computation states. It sets up control environments which are executed in subsets of those states. In so doing, it restricts all of the temporal sequences of values available. As viewed from the then clause, the filter being added in Fig. 5 restricts the sequence of computation states, and hence the available temporal sequences of values, to those which correspond to values of A(I) which are less than 100.

As with an augmentation, in order to gain a complete understanding of what a filter does, it is necessary to know where in the simpler loop the filter is placed. It is necessary to know what the available sequences of values are, in order to know what they are restricted to. In Fig. 5, if the filter is put at positions C or F, then the sequence of values of I is restricted to those corresponding to members of the first N elements of A which are less than 100. If the filter is put at positions D or E, then the sequence of values of I is restricted to those corresponding to members of the first N elements of A which are greater than zero and less than 100. The behavior of the complex loop can be understood as the composition of the behaviors of the simpler loop and the filter.

C. The Basic Loop PBM

Once all of the augmentations and filters have been removed from a loop, a residual loop remains. This residue corresponds to what T. Pratt [19] refers to as the "control computation" for the loop. A basic loop can be characterized by the fact that all of the computation in the body of the loop can potentially affect the termination of the loop. In his article, Pratt calls for an improvement in the design of loop control structures. His basic observation is that a small number of common control computations appear in a large percentage of loops which are written, and that unfortunately, the looping constructs currently in use largely obscure rather than highlight this fact. He suggests that the parts of the control computation, including any initialization, should be grouped together, rather than spread through a loop, and that stereotyped control computations should be clearly identified as such. In the analysis procedure, the basic loop PBM fills just this function.

The residual loop is analyzed as having three parts (an initialization, a test, and a body) by the basic loop PBM. This is essentially the same as the do-while construct described above. The only difference is that the initialization is closely associated with the loop.

The basic loop PBM has the same basic defect that do-while has. It can be arbitrarily difficult to determine what a basic loop does given the behavior of its parts. However, it is possible to recognize special cases such as the basic loop "DO I=N TO M; END;". The experiment in Section V indicates that in the loops studied, most of the complexity of most loops is embodied in their augmentations and filters rather than in their basic loops. More than 90 percent of the time, the basic loop which remains after PBM analysis merely enumerates a simple sequence of values and can easily be recognized as a special case.

D. The Interleaving PBM

In the interleaving PBM, two loops (A and B) are intermingled so that they are executed in synchrony. They are put together in such a way that there is no data flow from parts of loop A to parts of loop B or vice versa. Fig. 6 gives an example of two loops being interleaved. The figure shows one of the many different ways in which the two loops could have been interleaved. Another loop can be interleaved with the loop which results from interleaving two loops.

Interleaving can be recognized as follows. The primary clue is the existence of two separate exit tests. Given that, interleaving can be detected by looking at the data flow in order to see whether two separate bodies and initializations can be found. This distinguishes interleaved loops from basic loops which have compound exit tests.

```
  loop A                  loop B
DO I=1 TO 10;           X = 1.0;
END;              LOOP: X = (X+Z/X)/2.0;
                        IF ABS(X*X-Z)<.00001
                           THEN GOTO EXIT;
                        GOTO LOOP;
                  EXIT: ...

    an interleaving of the two loops
   X = 1.0;
   DO I=1 TO 10;
      X = (X+Z/X)/2.0;
      IF ABS(X*X-Z)<.00001 THEN GOTO EXIT;
   END;
EXIT: ...
```

Fig. 6. An example of interleaving.

The two loops (A and B) can be understood separately. Since there is no data flow between them, their effect on each other is limited. Their only interaction is that when one terminates, the other is forced to terminate. Thus if either of the loops can be shown to terminate, then the combination can be shown to terminate. This property of interleaving leads to one of its primary uses: guaranteeing that a loop will terminate. This is the purpose of the interleaving in Fig. 6.

The behavior of the interleaved loop is the union of everything that the subloop which terminates first does, with a subset of what the other subloop does. The interleaved loop in the example terminates after at most 10 iterations. Further, it can be seen that after it terminates "I=11 ∨ ABS(X*X-Z)<.00001."

A common error associated with using interleaving follows from the misconception that interleaving guarantees termination without affecting the primary loop in any other way. If this were the case then it would be possible to say that after the interleaved loop in the example terminates "I<11 ∧ ABS(X*X-Z)<.00001." This misconception would lead a programmer to write the rest of his program assuming that the loop always computed the square root of Z to within .00001. However, this is not true. The interleaved loop either computes the square root, or just gives up trying.

IV. Using PBM Analysis to Guide Verification

The purpose of this section is to show how PBM analysis reveals the logical structure of a program and to show how this can be used to guide a proof of correctness. In addition, this method for verifying a loop is contrasted with the standard approach of using a single loop invariant.

Consider the program in Fig. 7 and how it would be verified by using a single loop invariant as originally introduced by Floyd and Hoare [4], [10]. Assertions are passed over the body of the loop in order to develop a statement of what the body does. Then an appropriate loop invariant is developed.

```
X = 0;
Y = 0;
L = 11;
DO K=1,10;
   IF C(K)>0 THEN DO;
      X = X+A(K);
      Y = Y+A(K)*A(L);
   END;
   L = L+1;
END;
```

the program is claimed to compute

$X=\Sigma_{\{i \mid i \in \{1,\ldots,10\} \wedge C(i)>0\}} A(i) \wedge$
$Y=\Sigma_{\{i \mid i \in \{1,\ldots,10\} \wedge C(i)>0\}} A(i)*A(i+10)$

assertions summarizing the actions of the loop body

$L=L'+1 \wedge (C(K')>0 \rightarrow (X=X'+A(K') \wedge Y=Y'+A(K')*A(L'))) \wedge$
$(C(K')\leq 0 \rightarrow (X=X' \wedge Y=Y')) \wedge K=K'+1$

a loop invariant

$0\leq K\leq 11 \wedge L=K+10 \wedge$
$X=\Sigma_{\{i \mid i \in \{1,\ldots,K\} \wedge C(i)>0\}} A(i) \wedge$
$Y=\Sigma_{\{i \mid i \in \{1,\ldots,K\} \wedge C(i)>0\}} A(i)*A(i+10)$

Fig. 7. Steps in a proof by the standard single invariant method.

PBM	section of the program	the sequence of values it produces
basic	DO K=1,10;	$1\leq i\leq 10$
loop	END;	$K_i=i$
aug.	L = 11;	$1\leq i\leq 10$
	L = L+1;	$L_i=i+10$
filter	IF C(K)>0 THEN	the above sequences are restricted to the elements which correspond with positive values of C(K)
aug.	X = 0;	$1\leq i\leq 10$
	X = X+A(K);	$X_i=\Sigma_{\{j \mid j \in \{1,\ldots,i\} \wedge C(j)>0\}} A(j)$
aug.	Y = 0;	$1\leq i\leq 10$
	Y = Y+A(K)*A(L);	$Y_i=\Sigma_{\{j \mid j \in \{1,\ldots,i\} \wedge C(j)>0\}} A(j)*A(j+10)$

Fig. 8. Steps in a proof based on PBM analysis.

Finally, the statement of what the body does is used to prove the invariance of the loop invariant, and the loop invariant is used to verify the specifications of the loop.

One of the most difficult steps in a proof of this form is the determination of an appropriate loop invariant. Considerable research has been done on ways to automatically develop invariants. Much of this work centers around heuristic methods which can be used to guide a search for an invariant [14], [29]. Some of it is oriented toward directly deriving invariants for specific classes of loops [2], [3], [17]. The work of Basu and Misra [2], [3] is particularly interesting. They analyze the mathematical properties of a loop in order to directly derive an appropriate loop invariant for certain classes of loops.

Fig. 8 shows how the program in Fig. 7 would be analyzed by PBM's, and how this leads to a proof of correctness. The program is divided into five parts: a basic loop which enumerates the integers from 1 to 10 in K, an augmentation which enumerates the integers from 11 to 20 in L, a filter which restricts these temporal sequences of values by selecting only those elements which correspond to positive values of C(K), an augmentation which computes the sum of the indicated elements of A, and an augmentation which computes the sum of the products of the indicated elements of A. This decomposition leads to a style of proof based on composition which is very different from the proof in Fig. 7. First, five lemmas are proved. Each lemma summarizes the actions of one of the five parts of the program. Second, these lemmas are combined to yield the desired result.

The problem of finding the loop invariant is solved by breaking it up into five pieces. No invariant is needed in order to verify the program as a whole. Rather, each of the proofs of the five lemmas requires an invariant. However, each of these proofs is so simple that it is easy to determine what the invariant should be by the methods of Basu and Misra, if not by simple recognition. It should be noted that not all programs can be decomposed by PBM's as nicely as the one in the example. There is no limit to the complexity of the pieces which result. Therefore, it may be very difficult to determine the invariant needed to prove one of the lemmas needed. Even in this situation, the PBM analysis is useful because it determines the parts of the invariant as a whole which are easy and separates them from the parts which are difficult. PBM analysis does not claim to be a uniform procedure which will determine the invariant for any loop; rather, it claims to greatly simplify the problems involved with finding most of the invariant for most loops (see Section V).

The fundamental difference between the form of the proof engendered by PBM analysis and the form of the proof resulting from the single invariant method becomes apparent when the proof is used for something other than giving a yes/no answer to the question of whether or not the program is correct. For example, suppose that the program were incorrect and that therefore the proof failed. The PBM proof is broken up into a sequence of steps which are directly linked to parts of the program. If the failure of the proof can be localized to one of the steps of the proof, then the bug in the program can be localized to the corresponding part of the program. The failure of a proof based on a single loop invariant does not lend itself to this kind of analysis.

The same kind of difference appears in a variety of other tasks. For example, the PBM proof can be used to help explain how the program works because it indicates what parts of the program contribute to what parts of the specifications. The analysis according to PBM's, and the resulting proof of correctness, are specifically designed to reveal the logical structure of a program.

V. An Experiment

This section addresses the following two questions. Given that it is possible to write loops which cannot be directly analyzed in terms of the four loop PBM's discussed above, how often is this a problem? Given that there is no limit to the complexity of the pieces which result from the analysis of a loop by PBM's, how often are the pieces simple?

In order to investigate these questions, an experiment was performed. Twenty percent of the 220 programs in the IBM SSP [12] were chosen at random and analyzed in terms of PBM's by hand. The SSP was chosen because it is a large corpus of reasonably written programs.

The 44 programs chosen contained 164 loops. The interleaving PBM was used 23 times. This yields a total of 187 underlying loops. These were analyzed as being built up out of 187 basic loops, 273 augmentations, and 3 filters.

It was possible to analyze all of the loops with the PBM's discussed in Section III without having to transform them in any way. Configurations which cannot be directly analyzed (such as a loop with more than one entry point) did not occur in these loops. The only difficulty was that one loop had multiple level error exits which branched outside of the loop from inside an inner loop. In order to analyze the error exits in terms of the PBM's above, they had to be looked at as being explicit exits from the inner loop and from the outer loop. This may or may not be the best way to look at them.

It turned out that nearly 90 percent of the time, the pieces which resulted from PBM analysis were very simple. Of the 187 underlying loops, 166 (88 percent) were semantically of the form "DO I=L TO M BY N;" though some of them were not syntactically of that form. Of the 273 augmentations, 243 (89 percent) were easy to recognize as either a product, a sum, a max, a min, a count, or a trivial augmentation whose recurrence relation did not reference prior values of the variable being defined (such as "x_i=2*z_i"). All three filters were simple comparisons with zero.

It is interesting to note that because of the interleaving PBM, 162 (99 percent) of the 164 loops could be shown to terminate based simply on the fact that "DO I=L TO M BY N;" terminates. Although, there were 21 underlying loops which were not of the form "DO I=L TO M BY N;" interleaving combined most of these loops with loops which were of that form. As a result, all but 2 of the 164 loops examined contained at least one underlying loop of the form "DO I=L TO M BY N;" and therefore clearly terminate.

The four loop PBM's together with five additional PBM's for straight-line programs were able to account for all of the structure in the 44 programs studied, not just the looping structure. However, there are other ways in which programs are organized, such as, interrupt processing and recursion. Extending the set of PBM's in order to cover a wider class of programs is an important direction for future research on PBM's. For example, [28] shows how the loop PBM's can be improved and extended so that they cover recursive as well as iterative programs.

VI. An Automatic Analysis System

The author has implemented a system (described fully in [28]) which automatically analyzes programs in terms of PBM's. The system operates in two phases. A translation phase reads in a program and converts it to a language-independent internal form. An analysis phase then analyzes the program by looking at the internal form.

The internal form makes the control flow and data flow explicit independent of the syntactic constructs of any particular language. The internal form is essentially a graph with two kinds of arcs. The nodes of the graph correspond to primitive operations such as: plus, times, and less than. One type of arc specifies the flow of a data object from one node to another. The other type of arc specifies the flow of control from one node to another. Constructs which languages use to implement data flow (such as: variables, assignment, parameters, and nesting of expressions) are not present in the internal form. All information about data flow is contained in the data flow arcs. Similarly, constructs which languages use to implement control flow (such as GOTO's and sequential placement of statements) are eliminated in favor of the control flow arcs.

A translator has been written which converts Fortran [11] programs, like those in the IBM SSP [12], into the internal form. C. Rich [21] has written a translator which converts Lisp [16] programs into the internal form. After this conversion, the analysis phase of the system works on Lisp programs just as easily as on Fortran programs.

The translator works by running over the program like an evaluator, creating control flow arcs, data flow arcs, and nodes as it goes. The translator has detailed knowledge of the constructs which implement data flow and control flow. It does not have any knowledge of what the primitive functions do except how many inputs and outputs they have.

The principle difficulty in translation comes from the fact that the internal form requires that all of the data flow be explicit. In a program it can be very difficult to determine what the data flow actually is. One source of trouble is the use of side effects. The current translator can deal with assignment to variables and to array elements, but assumes that there are no other side effects.

Predicates which create multiple control flow paths lead to multiple data flow paths which signify that a given operation will receive data from one of several different sources depending on which of several control flow paths is taken. It is possible for predicates to interact in such a way that not all control flow paths are accessible. This reduces the number of possible data flow paths. Unfortunately this is difficult to detect without knowledge of what the primitive operations do. The current translator simply makes the pessimistic assumption that all control flow paths are accessible.

Once a program has been translated into the internal form, it is then analyzed in terms of PBM's. This is done in several steps. The first step looks primarily at the control flow arcs and analyzes the straight-line sections of the program while merely identifying the loops in it. This is essentially an analysis in terms of basic structured programming constructs as discussed in Section II. The analysis is done bottom up by locating minimal configurations which can be grouped together in accordance with a PBM. Whenever such a configuration is located, it is grouped together into a single node, and the analysis continues. This process terminates when the entire program is grouped into a single node.

The principle problem which might occur during this process is an explosion of competing hypotheses which would send the system into a lengthy search for the correct analysis. However, due to the fact that only a very small number of PBM's are being considered, and that they are all very different from each other, this problem does not arise. In fact, the system is never forced to change its mind and undo a grouping it has previously made.

The first step of analysis can be compared with the system B. Baker [1]. Her system uses graph theoretic methods in order to analyze Fortran programs based on their control flow in terms of basic structured programming constructs. Her system then outputs the program in a structured form. GOTO's are used in situations where an analysis in terms of the structured programming constructs is not possible.

After the first step of analysis, which is based largely on control flow, the system described here performs a second step of analysis which looks at the data flow and rearranges some of the groupings. The major goal of this step is to locate the initializations for loops and explicitly associate them with the loops. The initialization of a loop is located by finding those pieces of code which have data flow to and only to the loop. This step is necessitated by the fact that the syntax of Fortran does not require the initialization for a loop to immediately precede the loop. Large parts of the program may intervene in the flow of control between the initialization and the rest of a loop.

The third and final analysis step analyzes the loops discovered by the earlier steps according to the four PBM's: basic loop, augmentation, filtering, and interleaving. The analysis is based primarily on data flow connectivity and proceeds as described in Section III. For example, augmentation bodies are located by finding minimal subsets of the body of a loop which do not have data flow to any other part of the loop. Once an augmentation body is discovered, it is removed from the loop and the remaining loop is analyzed further.

The structure which results from the three steps of analysis is essentially a parsing of the program. It shows how the entire program is built up from simple pieces by means of the PBM's.

VII. The Relationship Between PBM's and Programming Language Constructs

The PBM's which are used in the first step of analysis (conjunction, composition, conditional, and basic loop) correspond to basic structured programming constructs. If the analysis system operated on programs written in a language which required the use of these basic structured programming constructs, the first step of analysis would not be necessary. However, the loop PBM's (augmentation, filtering, and interleaving) do not correspond to any existing structured programming constructs.

However, the loop PBM's do embody natural ideas, and many programming constructs have features in common with them. For example, there are a variety of constructs which have the feature that they separate out a basic loop which enumerates a sequence of values from the rest of the loop which does something with them. The most common example of this is the DO statement. It makes a clear syntactic distinction between the enumeration of a sequence of integers and their use. However, in most languages, the DO statement does not correspond semantically to a basic loop because the programmer is not prohibited from assigning to the DO variable in the body of the loop. If the DO variable is modified in the body of a loop then the logical separation between the DO loop and its body is destroyed. It is no longer true that the behavior of the basic loop is completely specified by the DO statement itself.

The MAPC function in Lisp [16] is similar to the DO statement except that it enumerates the elements of a list instead of a sequence of integers. GENERATORS in Alphard [24] and ITERATORS in CLU [15] extend this concept to the enumeration of elements of arbitrary data types. The MAPCAR function in Lisp is interesting because as well as enumerating the elements in a list, it contains a standard augmentation

"(SETQ X NIL) (SETQ X (APPEND X result))" which forms a list of the results.

The language APL [18] has several operators which operate directly on vectors in a fashion very similar to the way PBM's operate on temporal sequences of values created by loops. For example, the index generator function generates a sequence of integers in a vector. The compression function filters out elements in a vector based on a bit vector. The reduction operator is similar to augmentation in that it applies an operator such as plus or times to the elements of a vector in order to compute the sum or product of all the elements in the vector.

Current programming languages lack structured looping notations for multiple augmentations and interleaving. It is interesting to consider the possibility of extending a language to add these features. Each structured construct in a language plays two roles. It provides a tool for the programmer to use when he is constructing the program in the first place. More importantly, the use of the construct makes the program more readable, because it makes the logical structure of the program more explicit. In this role, it enhances the value of the code for the program as documentation.

One important aspect of any construct is how it will look when a program is printed on paper. This is particularly important in its role as documentation. Virtually all structured constructs are expressed using syntactically nested notations. An example of a syntactically nested notation is the if-then-else statement: "IF pred THEN statement1; ELSE statement2;". This specifies how pred, statement1, and statement2 are to be combined without referring to the internal structure of any of them. The construct treats them like black boxes. Syntactically nested constructs have several very nice features. In particular, they give the program a tree-like structure and provide a lot of locality. All of the information about how the pieces will fit together is contained solely in the construct itself, and each piece can be understood in isolation.

One reason that augmentation, filtering, and interleaving have not appeared in full form in languages may be the fact that they do not lend themselves to being expressed in syntactically nested notations. For example, consider augmentation. A nice syntactically nested construct such as "(AUGMENT loop augmentation)" will not work because it does not specify where the augmentation body will be put in the loop. Viewed more abstractly, it does not specify what temporal sequences of values the augmentation will receive. The problem is that the loop cannot be looked at as simply a black box during the process of augmentation. Using current programming languages, a programmer specifies where in a loop an augmentation body should be put, by putting it there. This does an effective job of specifying the necessary information, but it does not make the logical structure of the resulting loop explicit in the code for the program.

Constructs such as DO and MAPCAR make a compromise between the desire to have a syntactically nested construct, and the fact that the basic loop to be augmented cannot be looked at as a black box. With both constructs, the basic loop, and the place to put the augmentation body, are specified by the underlying semantics of the construct, and are not intended to be specified by the programmer. It is not clear whether this approach of using conventions to specify the placement of augmentations could be extended in order to handle multiple augmentations with a nested construction which would serve as good documentation.

One way to introduce augmentation and filtering into a programming language would be to move up to a more abstract level which talks directly in terms of temporal sequences of values. At this level, augmentation and filtering are just composition of fragments and therefore can be expressed easily with syntactically nested constructs. This would lead to a language which had constructs comparable to the APL operators described above. This is the approach taken by Kahn and MacQueen [13]. Their language is capable of expressing a program directly as a set of coroutines operating on vectors which may be spread out in time. In order to get the significant savings in time and space which result from implementing augmentations and filters with loops operating on temporal sequences of values rather than by coroutines or by functions operating on actual vectors of values, the language would then have to have a smart complier. The compiler would have to know how to combine the pieces together into a loop. The work which has been done on developing a compiler for APL (see for example, [7]) indicates that this is not an unreasonable goal.

A different approach to introducing the loop PBM's into a programming language could be based on an interactive program editing environment. In such an environment, the programmer could develop a program in a structured way by issuing commands in terms of PBM's. In response to a command such as "add this augmentation to that loop at that position" the editor would modify the loop to include the augmentation. The programmer would have the added advantage of being able to edit the resulting loop in order to increase its efficiency by promoting sharing and the like if necessary.

The main problem with this approach is that it does not contribute a construct which aids in documenting the code. However, the fact that the logical structure of the program cannot be printed out in a nice tree-like manner does not prevent the editing system itself from remembering what the logical structure of the program is. This could be used to the programmer's advantage if the editing system itself could use this knowledge of the structure of the program in order to aid the programmer. This is the approach taken by the system described in the next section.

VIII. A PROGRAMMER'S APPRENTICE

The PBM's discussed in this paper were developed in the context of the design of a system which can assist a person who is working on a program. Research on this system [20]-[23], [25], [27], [28], which has been called a programmer's apprentice (PA), is being carried out by a group consisting of C. Rich, H. Shrobe, and the author. This research has its roots in the work of Hewitt and Smith [9], Goldstein [6], and Sussman [26].

The PA is intended to be midway between an improved programming methodology which facilitates good programming style, and an automatic programming system. The intention is that the PA and a programmer will work together throughout all phases of the development and maintenance of a program. The programmer will do the hard parts of design and implementation while the PA will act as a junior partner keeping track of all the details and assisting the programmer whenever possible.

The PA is designed to perform a variety of tasks. Underlying all of these tasks is the ability to understand the program which is being worked on in a way which allows it to effectively communicate with the programmer about the program. One of the primary roles of the PA is to serve as continuing in-depth documentation for the program. It can describe the structure of the program and answer questions about it. The PA can aid in verifying a program. As part of understanding a program, the PA knows why it works. This is the backbone of a proof of correctness. The PA can aid in debugging a program. While the program is being developed, the PA is developing an understanding of why it works. The PA detects bugs in the program by detecting inconsistencies in this understanding. It can then aid in localizing the bug by using its understanding of what parts of the program are causing the inconsistency. The PA can help assess the effects of a modification. Since it knows the logical dependencies in the program, it can determine what parts of the program can be affected by a proposed change, and can draw some conclusions about these effects.

The PA is designed so that it has so much explicit information about the program being worked on that it can perform each of the above tasks using only small steps of deduction. This being the case, the primary effort in designing the PA has been directed toward determining what kinds of information it should have about a program, and how it can acquire this information. The PA has knowledge of the structure of the data objects used by a program organized in a manner similar to data abstractions. In addition, it has knowledge of the logical structure of the program itself. This information is embodied in a plan for the program.

The plan for a program records the basic structure of the program by specifying the control flow and data flow in the program. This part of the plan corresponds exactly to the internal representation for a program discussed in Section VI. The plan also records the logical structure of the program. It does this through several mechanisms. The plan is broken up into a hierarchy of segments and subsegments. Each segment and subsegment has input/output specifications which describe its behavior. In addition, the plan is annotated with dependency links which form an outline of a proof of correctness for each segment. These links specify how the behavior of a segment is derived from the behavior of its subsegments.

Consider how the PA can perform the tasks described above given that it can develop a plan for a program. In its role as documentation, the PA just reports out information stored in the plan. The dependency links directly record the backbone of a proof of correctness and can be used to guide verification. Bugs are detected as contradictions among the dependency links. Localizing bugs and assessing modifications are both accomplished by looking at the dependency links in order to determine what depends on what.

Looking at the plans for particular programs, it was discovered that plans are built up in stereotyped ways. This observation led directly to the PBM's discussed above. Each PBM specifies a method by which subsegments can be combined to produce a new segment. It specifies what the resulting data flow and control flow will be. More importantly, it specifies what the dependency links will be. It gives a description of what the behavior of the result will be, and describes how that behavior is derived from the behavior of the subsegments.

The pivotal activity of the PA is the process of acquiring the knowledge which is represented by the plan for a program. PBM's play an important role in this process. All of the information in the plan must be either known to the PA in advance, supplied by the programmer, or derived by the PA. The PA knows about the primitive operations, and about a variety of common data structures and algorithms. The programmer supplies the specifications and design for the program to be written. The PA and the programmer cooperate when working in the areas between these two extremes. The programmer can write code and, as discussed in Sections III and VI, the PA can use the PBM's to analyze the code in order to break it down into pieces which it already knows about. It can then use the knowledge associated with each PBM in order to build up a complete plan. Alternatively, the programmer can talk with the PA in terms of PBM's in order to directly specify how to build up the plan. Given a complete plan, it is easy for the PA to produce code. The PBM's are also useful as a vocabulary which the PA can use when talking about the logical structure of a program.

As mentioned above, the research on the PA is a three person project. Rich and Shrobe laid out the basic goals for the PA and the basic description of what constitutes a plan [21], [22]. The author outlined a PA system which would operate in the restricted domain of Fortran programs [27]. Since that time, research has continued in parallel in three main areas. H. Shrobe has implemented a prototype reasoning system which operates in the context of plans for programs [25]. It is designed to be able to perform the many deductions needed by the PA, and to aid in deriving the dependency links in the plan for a program. C. Rich is working on the problem of how knowledge of common data structures and algorithms should be represented in the PA, and what things it should know about a particular programming domain [20]. He is also investigating how the PA can recognize instances in which a programmer has used data structures and algorithms it knows about. The author has concentrated on studying how the logical structure of a program is built up. This has led to the analysis method based on PBM's [28]. The method is important in the context of the PA because it enables the PA to understand a program by breaking the program up into pieces which it knows about and then combining its understandings of the pieces into an understanding of the whole program.

IX. Conclusion

The basic idea behind the analysis method presented here is that a typical loop can be looked at as a composition of stereotyped fragments of looping behavior. This point of view reflects the way many programmers analyze, understand, and reason about loops. The PBM's, and the idea of temporal sequences of values are important because they make it possible for an automatic system to analyze, understand, and reason about loops in this same straightforward way.

PBM's are a particularly good basis for analyzing a loop because they break up a loop in a way which makes it easy to combine understandings of the pieces into an understanding of the loop as a whole. A PBM analysis can be used to help construct a proof of correctness for a loop because it makes it easier to determine appropriate loop invariants. More importantly, it leads to a style of proof which is particularly useful because it is linked to the program. The PBM analysis reveals what parts of a loop are contributing to each part of the behavior of the loop as a whole. The most important application of PBM's is in the PA which is designed to assist a programmer, based on its ability to understand the program being worked on. PBM's play an important role in the process by which the PA develops an understanding of and reasons about a program.

Acknowledgment

The author would like to acknowledge the valuable assistance of C. Rich, G. Roylance, H. Shrobe, and G. Sussman.

References

[1] B. S. Baker, "An algorithm for structuring flowgraphs," *J. Ass. Comput. Mach.*, vol. 24, pp. 98-120, Jan. 1977.

[2] S. Basu and J. Misra, "Proving loop programs," *IEEE Trans. Software Eng.*, vol. 1, pp. 76-86, Mar. 1975.

[3] —, "Some classes of naturally provable programs," in *Proc. 2nd Int. Conf. Software Eng.*, Oct. 1976, pp. 400-406.

[4] R. W. Floyd, "Assigning meaning to programs," in *Proc. Symp. Applied Math.*, vol. 19, 1967, pp. 19-32.

[5] D. P. Friedman and D. S. Wise, "CONS should not evaluate its arguments," Indiana Tech. Rep. 44, Nov. 1975.

[6] I. P. Goldstein, "Understanding simple picture programs," MIT, Cambridge, MA, MIT/AI/TR-294, Sept. 1974.

[7] L. J. Guibas and D. K. Wyatt, "Compilation and delayed evaluation in APL," in *Proc. 5th ACM POPL Conf.*, 1978.

[8] P. Henderson and J. H. Morris, "A lazy evaluator," presented at the SIGPLAN-SIGACT Symp. Principles of Programming Languages, Atlanta, Jan. 1976.

[9] C. Hewitt and B. Smith, "Towards a programming apprentice," *IEEE Trans. Software Eng.*, vol. 1, pp. 26-46, Mar. 1975.

[10] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576-583, Oct. 1969.

[11] IBM, "FORTRAN IV language," IBM, White Plains, NY, C28-6515-6, 1966.

[12] IBM, *Scientific Subroutine Package Version III Programmer's Manual*, IBM, White Plains, NY, GH20-0205-4, 1970.

[13] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," in *1977 Proc. IFIP Congr.* Amsterdam, The Netherlands: North-Holland, 1977.

[14] S. Katz and Z. Manna, "Logical analysis of programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 188-206, Apr. 1976.

[15] B. H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 564-576, Aug. 1977.

[16] D. A. Moon, *MACLISP Reference Manual*, MIT, Cambridge, MA, 1974.

[17] J. H. Morris, Jr. and B. Wegbreit, "Subgoal induction," *Commun. Ass. Comput. Mach.*, pp. 209-222, Apr. 1977.

[18] R. P. Polivka and S. Pakin, *APL: The Language and Its Usage*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[19] T. W. Pratt, "Control computations and the design of loop control structures," *IEEE Trans. Software Eng.*, vol. 4, pp. 81-89, Mar. 1978.

[20] C. Rich, "A representation for programming knowledge and applications to recognition, generation, and cataloguing of programs," forthcoming Ph.D. dissertation, MIT.

[21] C. Rich and H. Shrobe, "Initial report on a LISP programmer's apprentice," MIT, Cambridge, MA, MIT/AI/TR-354, Dec. 1976.

[22] —, "Initial report on a LISP programmer's apprentice," *IEEE Trans. Software Eng.*, vol. 4, pp. 456-466, Nov. 1978.

[23] C. Rich, H. E. Shrobe, R. C. Waters, G. J. Sussman, and C. E. Hewitt, "Programming viewed as an engineering activity," MIT, Cambridge, MA, MIT/AIM-459, Jan. 1978.

[24] M. Shaw and W. A. Wulf, "Abstraction and verification in ALPHARD: Defining and specifying iteration and generators," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 553-564, Aug. 1977.

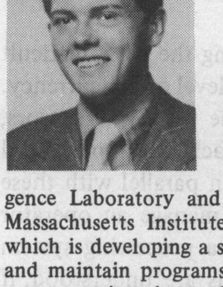
[25] H. E. Shrobe, "Reasoning and logic for complex program understanding," Ph.D. dissertation, MIT, Aug. 1978.

[26] G. Sussman, "A computational model of skill acquisition," MIT, Cambridge, MA, MIT/AI/TR-297, Aug. 1973.

[27] R. C. Waters, "A system for understanding mathematical FORTRAN programs," MIT, Cambridge, MA, MIT/AIM-368, Aug. 1976.

[28] —, "Automatic analysis of the logical structure of programs," (based on an MIT Ph.D. dissertation, Aug. 1978), MIT/AI/TR-492, Dec. 1978.

[29] B. Wegbreit, "The synthesis of loop predicates," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 102-112, Feb. 1974.



Richard C. Waters (M'78) received the B.S. degree magna cum laude in applied mathematics (computer science) from Brown University, Providence, RI, in 1972, the M.S. degree in computer science from Harvard University, Cambridge, MA, in 1973, and the Ph.D. degree in computer science with a minor in linguistics from the Massachusetts Institute of Technology, Cambridge, in 1978.

He currently has a joint appointment as a Research Associate in the Artificial Intelligence Laboratory and the Laboratory for Computer Science at the Massachusetts Institute of Technology. He is working on a project which is developing a system which can assist programmers to develop and maintain programs. His other interests include programming languages, engineering problem solving, and natural language processing.

Dr. Waters is a member of the Association for Computing Machinery, SIGART, SIGPLAN, and the IEEE Computer Society.