

A Method for Automatically Analyzing Programs

Richard C. Waters

Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Mass. 02139

ABSTRACT - This paper presents a method for automatically analyzing programs and discusses why it is a useful way to look at programs. The method is based on the idea that there are only a few basic ways in which the logical structure of programs is built up. An experiment is presented which shows that this accounts for the structure of a large class of programs. The paper discusses how the method can be used to automatically analyze the structure of a program, and how the resulting analysis can be used to guide a proof of correctness for the program. 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.

1. Introduction

This paper presents a method (described fully in [21]) for analyzing programs. This analysis of a program results in a "plan" for the program which represents the underlying logical structure of the program. The analysis method is based on the observation that programs are built up in a small number of stereotyped ways referred to as plan building methods (PBMs).

An important application of PBMs is that they can be used to analyze a loop in a way which makes it easier to understand what the loop does and why. Section 2 shows why this is a more useful analysis than one based on basic structured programming constructs. Section 3 describes the PBMs in detail. Section 4 discusses how an analysis of a loop in terms of the PBMs 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 5 which shows that the PBMs can be used to analyze a large class of programs. The experiment also shows that the pieces which result from the analysis are largely simple and easy to understand. A system (described in Section 6) has been implemented which performs PBM analysis automatically. Section 7 discusses the relationship between the PBMs and current programming language constructs.

2. 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

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

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 PBMs has been the development of structured programming constructs. Consider the loop program in Figure 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). 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 interactions of the lines with each other are complex.

```
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;
```

Figure 1: An example program.

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. Figure 2 depicts the program in Figure 1 analyzed in terms of these 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)".

```

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

```

Figure 2: The example program in structured form.

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. (An example of this is 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 Figure 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 non-looping programs, meet the criteria set forth above very well.

In contrast, 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 Figure 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 analyze 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 PBMs 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 such a fragment.)

The PBMs for loops break the loop in Figure 2 up into four fragments as shown in Figure 3. The first fragment (A) counts

A	B	C	D
I=1;	IF I>N THEN		Z = 0;
I=I+1;	GOTO EXIT;	IF A(I)>0 THEN	Z = Z+A(I);

Figure 3: The example program analyzed by PBMs.

up by one from one. It enumerates the sequence of integers {1,2,...}. The second fragment (B) tests the sequence of integers produced by A and stops the loop when an integer greater than N is found. This has the effect of truncating the sequence of integers to the sequence {1,2,...N}. The third fragment (C) operates on the truncated sequence of integers. It restricts the sequence by selecting only those integers which correspond to positive elements of the vector A. The last fragment (D) computes the sum of the elements of A corresponding to the integers in the restricted sequence produced by C. In the loop as a whole, the four fragments are cascaded together so that the loop computes the sum of the positive members of the first N elements of A.

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 Howard Shrobe [19] and the author [21]. Given a program, such as a 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 Figure 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 a 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 Figure 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 C. A is composed with C by passing this data from A to C. Viewed abstractly, it makes no difference whether this data is put into a vector which is passed all at once to C, or, as in the example, A and C are intermingled so that C can use each individual value created by A as soon as it is produced. Intermingling A and C is just an efficient way of implementing the data flow from A to C.

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 key 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. (Fragment B presents a problem because it controls the number of times the other fragments are executed. In order to maintain the independence of the fragments in the face of this interaction, the descriptions of the fragments are prohibited from making any statements about the absolute number of elements in any temporal sequence of values. As a result, from the point of view of the descriptions, the fragments do not interact because their interaction cannot affect anything in the descriptions.)

3. The Plan Building Methods

This section describes the ten PBMs which have been developed. Figure 4 shows how the PBMs are related. They are divided into two main groups. One group can be used to analyze straight-line (non-looping) programs while the other group is used to analyze recursive programs and loops.

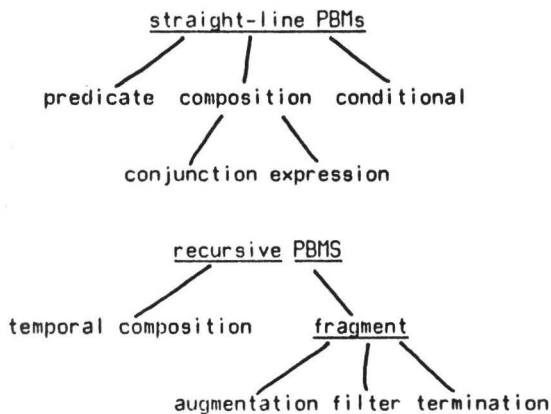


Figure 4: A taxonomy of the PBMs.

There are three basic straight-line PBMs: predicate, composition, and conditional. In addition, two special cases of composition are recognized: conjunction and expression. Figure 5 shows examples of segments of code which would be analyzed in terms of straight-line PBMs.

predicate	
IF X<Y THEN GOTO L1;	
ELSE IF Z<Y THEN GOTO L2;	
ELSE GOTO L3;	
conditional	composition
IF X<0 THEN X=-X;	Z=FIRST(SORT(A));
conjunction	expression
Z=COS(X); W=SIN(X);	Z=SIN(COS(X));

Figure 5: Examples of straight-line PBMs.

The PBM predicate is used to build up complex tests out of simpler ones. In the example, two two-way branching tests are combined into a three-way branching test. The PBM conditional combines a test with some actions in order to construct a conditional expression. It is exactly analogous to the structured programming construct if-then-else.

The PBM composition is used to construct arbitrary non-branching sections of straight-line code. Two special cases of composition are recognized because they have particularly useful logical properties. The PBM conjunction combines segments in such a way that there is no data flow between the segments. This means that the pieces are totally independent and can be understood completely in isolation from each other. It is particularly easy to understand what a conjunction does, because the specifications for the conjunction as a whole are simply the conjunction of the specifications for the segments which have been combined.

With the PBM expression, as with composition in general, there is data flow carrying the outputs of segments to the inputs of other segments. The PBM expression is more restricted than general composition in that the segments combined are required to have "substitutable" specifications. A specification is said to be substitutable if each output is described by a single assertion of the form "output=expression-involving-the-inputs". For example, the segment "COS" is described by the assertion "output=cos(input)". It is easy to understand what an expression does, because substitution can be used in order to develop simple specifications for the expression as a whole.

In contrast to an expression, a general composition can combine segments with non-substitutable specifications. This makes it cumbersome to develop a description of what the result does. For example, in the composition in Figure 5, SORT might well be described as producing an output which contains the same elements as the input, and in which the elements appear in increasing order. This being the case, simple substitution will not suffice to conclude that the composition as a whole computes the minimum element in the input. Some actual deduction is required.

It is very easy to analyze a straight-line segment of a program in terms of the straight-line PBMs because it is not possible to confuse an instance of one with an instance of another. This is due to the fact that there are only three basic PBMs (predicate, composition, and conditional) and the fact that they are completely different from each other. It should be noted that it is possible to write a straight-line program which is unstructured in that it cannot be analyzed in terms of these PBMs, or in terms of structured programming constructs, without first transforming the code.

The four PBMs which build up recursive structures embody the idea that recursive structures are built up by composing together stereotyped fragments of recursive behavior in a way which is exactly analogous to the way non-branching straight-line segments of code are built up by composing simple segments together. For simplicity, the discussion below describes the recursive PBMs solely in terms of loops. The discussion in [21] shows how they apply to arbitrary singly self-recursive programs. Three of the recursive PBMs (augmentation, filter, and termination) construct loop fragments. The fourth one (temporal composition) combines fragments together.

The PBM augmentation combines an initialization which is executed once with a body which is executed repetitively in order to create a loop fragment. The resulting fragment takes in temporal sequences of values and produces temporal sequences of values. The name "augmentation" is derived from the fact that an augmentation can be added into a loop in order

to augment its activities without affecting anything else the loop does.

	count	sum
init:	$I=1;$	$Z=0;$
body:	$I=I+1;$	$Z=Z+\underline{X};$

	max	feedback free
init:	$Z=\text{very-small};$	
body:	$\text{IF } \underline{X}>Z \text{ THEN } Z=\underline{X};$	$Z=2*\underline{X};$

Figure 6: Example augmentations.

Figure 6 shows four different augmentations. Consider the second one " $Z=0; Z=Z+\underline{X};$ " which computes a sum. The variable X is underlined in order to indicate that it stands for the place where the fragment receives a temporal sequence of values. It creates a corresponding sequence of partial sums in the variable Z . The behavior of an augmentation can be straightforwardly described in terms of a recurrence relation derived from the initialization and the body. For the summation fragment this recurrence relation is " $Z_1=0 \wedge Z_{i+1}=Z_i+X_i$ ". In situations where the recurrence relation has a simple solution, better specifications can be derived (such as " $Z_i=\sum_{j=1,i-1} X_j$ " for the summation augmentation). A large number of the augmentations seen in programs correspond to simple actions such as counting, summing, or calculating a maximum. The count augmentation is interesting in that it does not take any temporal sequence of values as an input but rather simply generates a sequence of values. The feedback free augmentation is particularly easy to understand because its recurrence relation (" $Z_i=2*X_i$ ") does not refer to prior values of Z and therefore is trivially solvable.

```
IF  $\underline{X}<0$  THEN ...;
ELSE ...;
```

Figure 7: An example filter.

The PBM filter constructs a loop fragment from a test which is repeatedly executed. The fragment takes in a temporal sequence of values and restricts the sequence by filtering out some of the values. In a loop, this action is realized by creating a restricted execution environment which is only executed on some of the iterations of the loop. The filter in the example takes in a sequence of values of X and creates two outputs: the sequence of negative values of X and the sequence of non-negative values of X .

```
IF  $\underline{I}>N$  THEN GOTO EXIT;
```

Figure 8: An example termination.

Like a filter, a termination creates a fragment out of a test which is repetitively executed. Also like a filter, it takes in a temporal sequence of values and outputs a restricted sequence of values. The restricted sequence corresponds to the initial portion of the input up to the first value which satisfies the test. However, unlike a filter, a termination has the far reaching effect that it causes the loop as whole to terminate.

The PBM temporal composition builds complex loops by combining loop fragments together. The process of combining the fragments is logically equivalent to composition. This makes it easy to understand what the resulting loop does. Figure 9 shows an example of this. The four fragments in the figure are composed together in order to yield the loop shown (which is the same as the one in Figure 1). Logically this is just a composition of $D(C(B(A))))$ and therefore it is clear that the

$Z=D(C(B(A))))$ where

A is	B is	C is	D is
$I=1;$	$\text{IF } \underline{I}>N \text{ THEN}$		$Z = 0;$
$I=I+1;$	GOTO EXIT;	$\text{IF } A(\underline{I})>0 \text{ THEN}$	$Z = Z+A(\underline{I});$

the loop which results from the temporal composition

```

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

Figure 9: An example temporal composition.

result computes the sum of the positive members of the first N elements of the array A since the augmentation A counts up by 1 from 1, the termination B truncates this sequence at N , the filter C selects the elements of the sequence which correspond to positive members of A , and the augmentation D computes a sum of these elements.

The process by which the fragments are actually combined in order to produce the resulting loop is somewhat complicated. This process is designed so that the validity of the simple logical analysis in terms of composition will be preserved. The fragments are taken apart, and their pieces are reassembled into the resulting loop. The initializations of the fragments are placed before the resulting loop. The bodies are placed inside the loop. Their arrangement in the loop is dictated by the data flow communicating temporal sequences of values between the fragments. The key idea is that if the body of one fragment uses a temporal sequence created by the body of another fragment, then the first body should be placed after the second one in the resulting loop. The action of a filter is produced by placing the bodies which receive its outputs within the scope of a conditional predicated on the filter's test.

It turns out to be straightforward to analyze a loop in order to break it apart into fragments combined by temporal composition. The analysis process is based on locating sections of the loop which do not affect any other parts of the loop. One section affects another if it either has data flow to it, or if it controls when it will be executed. Once such a section is located, it is removed as a fragment and the analysis continues. This process is sufficient to locate augmentations and filters. Consider the resulting loop in Figure 9. If it were being analyzed, the first fragment to be located would be D . This would be done by noticing that the line " $Z=Z+A(I)$ " does not have data flow to any other part of the loop. Once this line is removed the filter C is no longer controlling the execution of anything, so it too can be isolated and removed.

Terminations are somewhat more complex because they perform affect everything else in the loop since they determine when the loop will terminate. To allow them to be located and removed, the definition of not affect is weakened in order to allow this particular kind of effect. In order to maintain the logical structure based on composition, the descriptions of the fragments have to be prohibited from mentioning anything which a termination can affect (namely the absolute number of times anything will be executed).

4. Using PBM Analysis to Guide Verification

The most fundamental impact of the PBMs on verification is that they break up a program into a useful hierarchy of segments within segments. This structure can be used as the underlying structure of a proof of correctness, by determining specifications for each segment and then verifying these specifications. With each subproof, the method which is used to generate a proof can benefit from the fact that it only has to work on the single subproof which has been singled out by the PBMs, rather than on the program as a whole.

The impact of the PBMs on the structure of a correctness proof is greatest with regard to loops. They lead to a proof which is fundamentally different in form from the standard approach of using a single loop invariant. Consider the program in Figure 10 and how it would be verified by using a single loop invariant as originally introduced by Floyd [4] and Hoare [9]. 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. 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.

```

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
 $X = \sum_{1 \leq i \leq 10} C(i) \cdot A(i) \wedge$
 $Y = \sum_{1 \leq i \leq 10} C(i) \cdot A(i) \cdot A(i+10)$

assertion summarizing the actions of the body
 $(C(K') > 0 \rightarrow (X = X' + A(K') \wedge Y = Y' + A(K') \cdot A(L'))) \wedge$
 $L = L' + 1 \wedge (C(K') \leq 0 \rightarrow (X = X' \wedge Y = Y')) \wedge K = K' + 1$

a loop invariant
 $0 \leq K \leq 11 \wedge X = \sum_{1 \leq i \leq 10} C(i) \cdot A(i) \wedge$
 $L = K + 10 \wedge Y = \sum_{1 \leq i \leq 10} C(i) \cdot A(i) \cdot A(i+10)$

Figure 10: Steps in a proof using a single invariant.

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 [6, 12, 22]. Some of it is oriented toward directly deriving invariants for specific classes of loops [2, 3, 14]. The work of Basu and Misra 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.

Figure 11 shows how the program in Figure 10 would be analyzed by PBMs, and how this leads to a proof of correctness. The program is divided into six parts: an augmentation which enumerates integers in K, a termination which truncates this to the sequence of 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 Figure 10. First several lemmas are proved. Each lemma summarizes the actions of one of the parts of the program. Second, the lemmas are combined by composition in order to yield the desired result.

loop fragment	values it produces
K = 1;	1 ≤ i
K = K+1;	K _i = i
IF K>10	1 ≤ i ≤ 10
THEN GOTO EXIT;	K _i = i
L = 11;	1 ≤ i ≤ 10
L = L+1;	L _j = i+10
IF C(K)>0 THEN	restricts K _i and L _j
X = 0;	1 ≤ i ≤ 10
X = X+A(K);	X _i = $\sum_{1 \leq j \leq i} C(j) \cdot A(j)$
Y = 0;	1 ≤ i ≤ 10
Y = Y+A(K)*A(L);	Y _i = $\sum_{1 \leq j \leq i} C(j) \cdot A(j) \cdot A(j+10)$

Figure 11: Steps in a proof based on PBM analysis.

The problem of finding the loop invariant is dealt with by breaking it up into pieces. No invariant is needed in order to verify the program as a whole. Rather, each of the proofs of the 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 PBMs 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 is not a uniform procedure which will determine the invariant for any loop; rather, it simplifies the problems involved with finding most of the invariant for most loops.

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.

5. An Experiment

The straight-line PBMs are similar to basic structured programming constructs, and can be used to analyze any straight-line program that does not have spaghetti-like control flow. The loop PBMs are more novel, and it is not immediately clear how wide their range of applicability is. An experiment was performed in order to estimate what percentage of the loops which programmers actually write can be analyzed in terms of the PBMs without anything more than trivial transformations being done. 20% of the 220 programs in the IBM FORTRAN Scientific Subroutine Package (SSP) [10] were chosen at random and analyzed in terms of the PBMs by hand. 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 44 programs chosen contained 164 loops. It was possible to analyze all of the loops with the PBMs. They were analyzed as being built up by means of the PBM temporal composition out of 442 augmentations, 6 filters, and 186 terminations. Configurations which cannot be analyzed (such as a loop with more than one entry point) did not occur in these loops. The sole 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 PBMs, they had to be looked at as being explicit exits from the inner loop and the outer loop. This may or may not be a good way to look at them.

Nearly 98% of the time, the pieces which resulted from PBM analysis were very simple. Of the 442 augmentations, 374 (85%) were either easy to understand because they did not have feedback to themselves, or easy to recognize as a product, a sum, a max, a min, or a count. Only 31 (7%) were really difficult to understand. The remainder were of intermediate complexity. All 6 filters were simple comparisons with zero and therefore easily understood.

Of the 186 terminations, 165 (89%) were very simple to understand because they were simple comparisons with a fixed number, and the input which they tested was a simple sequence of numbers, most often {1,2,3,...}. Due to the fact that their tests and inputs were so simple, it was trivial to determine the exact number of states associated with their inputs, and hence exactly when the loops they were in would terminate. Most of the 21 terminations that were more complex, were additional terminations in loops which had at least one simple termination. As a result of this, 162 (99%) of the 164 loops contained at least one simple termination, and therefore can be trivially shown to terminate.

6. An Automatic Analysis System

The author has implemented a system (described fully in [21]) which automatically analyzes programs in terms of PBMs. The system operates in two phases. A translation phase reads in a program and converts it to a language independent internal form called a "surface plan". An analysis phase then analyzes the program by looking at its surface plan.

A surface plan makes the control flow and data flow explicit independent of the syntactic constructs of any particular language. It 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 a plan. All information about data flow is contained in the data flow arcs. Similarly, constructs which languages use to implement control flow (such as GOTOs and sequential placement of statements) are eliminated in favor of the control flow arcs.

A translator has been written which converts FORTRAN programs, like those in the IBM SSP, into surface plans. Charles Rich [17] has written a translator which converts LISP programs into surface plans. 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 which comprise a surface plan 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.

Once a program has been translated into a surface plan, it is then analyzed in terms of PBMs. This is done in two main 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 2. 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 first step of analysis can be compared with the system of 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. GOTOs are used in situations where an analysis in terms of the structured programming constructs is not possible.

The second analysis step analyzes the loops discovered by the earlier steps according to the four PBMs temporal composition, augmentation, filter, and termination. This analysis is based primarily on data flow connectivity and proceeds as described in Section 3. For example, augmentations 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 is discovered, it is removed from the loop and the remaining loop is analyzed further.

7. PBMs and Programming Language Constructs

The PBMs which are used in the first step of analysis 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. On the other hand, the loop PBMs do not correspond to any existing structured programming constructs.

However, the loop PBMs 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 an augmentation and a termination, which together enumerate 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 an augmentation and a termination because the programmer is not prohibited from assigning to the DO variable in the body of the loop.

The MAPC function in LISP is similar to the DO statement except that it enumerates the elements of a list instead of a sequence of integers. GENERATORS in ALPHARD [18] and ITERATORS in CLU [13], 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 [15] has several operators which operate directly on vectors in a fashion very similar to the way PBMs operate on temporal sequences of values created by loops. For example, 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.

The Language developed by Kahn and MacQueen [11] makes it possible to construct coroutine processes operating on sequences of values which interact in the same way as augmentations and filters. The processes can be combined together into expressions which can be evaluated either sequentially, or in parallel.

With either APL or the language of Kahn and MacQueen, in order to get the significant savings in time and space which result from implementing augmentations and filters by loops operating on temporal sequences of values the language would have to have a smart compiler. The compiler would have to know how to combine the pieces together into loops. The work which has been done on developing a compiler for APL (see for example, [7]) indicates that this is not an unreasonable goal.

8. 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 PBMs, 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.

The PBMs were developed as part of a larger research project. The goal of this project is to develop a system (called the "Programmer's Apprentice") which can assist a person who is writing a program. Research on this system [16, 17, 19, 20, 21] is being carried out by a group consisting of Charles Rich, Howard Shrobe, and the author. The intent is that the apprentice system will cooperate with a programmer throughout all phases of work on a program and be able to communicate with him about it. The programmer will do the hard parts of

design and implementation while the apprentice will act as a junior partner and critic, keeping track of details and assisting the programmer whenever possible. PBMs play an important role in the processes by which the apprentice develops an understanding of and reasons about a program.

ACKNOWLEDGMENTS

The author would like to thank Charles Rich, Gerald Roylance, Howard Shrobe, and Gerald Sussman for their assistance.

REFERENCES

- [1] B. S. Baker, "An Algorithm for Structuring Flowgraphs", JACM V24 #1, pp. 98-120, January 1977.
- [2] S. Basu and J. Misra, "Proving Loop Programs", IEEE Trans. on Software Eng. V1 #1, pp. 76-86, March 1975.
- [3] S. Basu and J. Misra, "Some Classes of Naturally Provable Programs", Proc. 2nd Int. Conf. on Software Engineering, pp. 400-406, October 1976.
- [4] R. W. Floyd, "Assigning Meaning to Programs", proc. symp. in Applied Math. V19, pp. 19-32, 1967.
- [5] D.P. Friedman and D.S. Wise, "CONS Should Not Evaluate its Arguments", Indiana Technical Report 44, Nov. 1975.
- [6] S. German and B. Wegbreit, "A Synthesiser of Inductive Assertions", IEEE Trans. on Software Eng. V1 #1, pp. 68-75, March 1975.
- [7] L.J. Guibas and D.K. Wyatt, "Compilation and Delayed Evaluation in APL", Proc. 5th ACM POPL Conf. 1978.
- [8] P. Henderson and J.H. Morris, "A Lazy Evaluator", Proc. 3rd ACM POPL Conf., 1976.
- [9] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", CACM V12 #10, pp. 576-583, October 1969.
- [10] IBM GH20-0205-4, "Scientific Subroutine Package Version III Programmer's Manual", IBM White Plains NY, 1970.
- [11] G. Kahn and D.B. MacQueen, "Coroutines and Networks of Parallel Processes", Proceedings of IFIP Congress-77, North-Holland Publ. Co, 1977.
- [12] S. Katz and Z. Manna, "Logical Analysis of Programs", CACM V19 #4, pp. 188-206, April 1976.
- [13] B.H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU", CACM V20 #8, pp. 564-576, August 1977.
- [14] J.H. Morris jr. and B. Wegbreit, "Subgoal Induction", CACM V20 #4, pp. 209-222, April 1977.
- [15] R.P. Polivka and S. Pakin, "APL: The Language and its Usage", Prentice-Hall, Englewood cliffs NJ, 1975.
- [16] C. Rich, "A Representation for Programming Knowledge and Applications to Recognition, Generation, and Cataloguing of Programs", forthcoming PhD thesis, MIT, August 1979.
- [17] C. Rich and H. Shrobe, "Initial Report on a LISP Programmer's Apprentice", MIT/AI/TR-354, MIT Cambridge MA, December 1976.
- [18] M. Shaw and W. A. Wulf, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", CACM V20 #8, pp. 553-564, August 1977.
- [19] H.E. Shrobe, "Reasoning and Logic for Complex Program Understanding", PhD thesis, MIT, August 1978.
- [20] R.C. Waters, "A System for Understanding Mathematical FORTRAN Programs", MIT/AIM-368, MIT Cambridge MA, August 1976.
- [21] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, December 1978.
- [22] B. Wegbreit, "The Synthesis of Loop Predicates", CACM V17 #2, pp. 102-112, February 1974.