Comput. Lang. Vol. 13, No. 3/4, pp. 109–123, 1988
Printed in Great Britain.

0096-0551/88 $3.00 + 0.00

PROGRAM COMPLEXITY USING HIERARCHICAL ABSTRACT COMPUTERS

WILLIAM G. BAIL[1] and MARVIN V. ZELKOWITZ[2]
[1]Intermetrics Inc., Bethesda, MD, U.S.A.
[2]Computer Science Department, University of Maryland, College Park, MD 20742, U.S.A. and Institute for Computer Sciences and Technology, National Bureau of Standards, Gaithersburg, MD 20899, U.S.A.

(*Received* 18 *February* 1988; *revision received* 6 *July* 1988)

Abstract—A model of program complexity is introduced which combines structural control flow measures with data flow measures. This complexity measure is based upon the prime program decomposition of a program written for a Hierarchical Abstract Computer. It is shown that this measure is consistent with the ideas of information hiding and data abstraction. Because this measure is sensitive to the linear form of a program, it can be used to measure different concrete representations of the same algorithm, as in a structured and an unstructured version of the same program. Application of the measure as a model of system complexity is given for "upstream" processes (e.g. specification and design phases) where there is no source program to measure by other techniques.

CASE tools Complexity Environments Measurement Prime programs

1. INTRODUCTION

The ease of developing correct programs has been related to the structural complexity of the resulting source program. There have been several attempts to understand and measure this complexity by restricting the use of control structures in a program. For example, structured programming is one method by which programs are more easily developed by imposing such restraints. There are also methods which are directed at managing data structure complexities, including the concepts of data abstraction and information hiding. By limiting the scope of data to only that which is needed by a given section of the program, local data responsibilities are easier to isolate and to maintain.

It is generally assumed that a program is well-structured if clusters of data and control activity are small. The purpose of this paper is to develop measures for both control and data structure that agree with these intuitive notions and can be used to extend our knowledge of program measurement. We would like to develop measures that can be used to classify the degree of *structuredness* with a system, so that such ideas can move from the general guidelines concept within the software life cycle to a practical analytic tool available to the software designer, much like the current use of compilers and editors.

In Section 2 of this paper we briefly describe our model of computation and its relationship to programming principles. The measure is related to information theoretic issues of randomness and complexity. We make the assumption that the randomness of a program (i.e. how long its source code representation must be) is a measure of its complexity. In Section 3 we add the prime program decomposition of a program as a mechanism to measure the structured complexity of an algorithm. The complexity measure is actually a measure on the graph representation of the program; therefore, it has applications to other graph-related processes. In Section 4 we give several examples of using these measures, both in the context of programming languages and in other processes in the development cycle such as specification and design notations.

We do not consider this measure as the ultimate measure of program complexity; however, we do believe that we have achieved a prototype of a class of interesting measures. By merging the information theory concept of complexity with the notions of structured programming and prime program decomposition, we have captured some of the basic ideas needed for an effective measure.

2. HIERARCHICAL ABSTRACT COMPUTERS

We base our model of complexity on the size of the program needed to describe an algorithm. Like Chaitin [1] we are interested in measures that describe the information content of an algorithm by its size and then we desire transformations that reduce the size of that representation. An algorithm is more complex if it requires more bits to represent its description. By decomposing an algorithm into its hierarchical structure using regular components "understood" by the "hardware interpreter", we can remove much of the complexity of a given description.

In order to compare two algorithms we first define a notation called a *Hierarchical Abstract Computer* (HAC) [2]. It has instructions which depend upon the number of distinct operators present in the particular algorithm being described. Superficially, it is similar to the software science models [3]; however, a HAC differs significantly from the Halstead model in that control flow and data structures are also taken into account. In the software science model, complexity is solely a measure of the number of objects (e.g. size) of each module, while in our case the complexity of the source program is determined by both the number of variables referenced by the module and by the complexity of its control flow graph, something ignored by the software science model.

A HAC is modeled by a directed graph, each node of which represents a single instruction. If a program consists of several procedures, than each procedure is a separate graph and each procedure invocation is represented as a single node in the graph of the calling procedure. The complete model is called a HAC program, and each procedure is called a HAC module.

2.1 Basic HAC model

A HAC instruction contains a label, an operator, and a list of operands. Each complete HAC module defines a function (e.g. operator) which can be used as an operator within another module, hence its hierarchical nature. Using this notation we can define several alternative representations. The simplest is called the *Direct Graph Form* (DGF) which is the direct implementation of the flow graph of a program. A program is a sequence of instructions, each instruction (representing a node in the program graph) has the syntax:

$$\langle label \rangle: \langle opcode \rangle \ \langle argument\ list \rangle \ (\langle label\ list \rangle)$$

where:

$\langle label \rangle$ represents the label on that instruction (e.g., name of graph node),
$\langle opcode \rangle$ represents the instruction to be performed,
$\langle argument\ list \rangle$ is a sequence of data names, and
$\langle label\ list \rangle$ are the following instructions (i.e. the set of outgoing arcs from that node).

There is one unique label called *exit* signifying that execution is to halt.

Let D be the set of data objects in a program, L be the set of labels (nodes) in a program and I be the set of operation codes (unique functions performed by the program nodes). If execution is at $label_i$, then $opcode_i$ is executed on data items $\{argument\text{-}list_i\}$ and the next instruction is chosen from one of the labels in ($label\_list_i$) until the label named *exit* is reached.

Figure 1 is a simple example of this model and represents the flow graph of the Absolute Value Function ABS. Its DFG is:

```
L1:  test       x,0    (L2,L3)
L2:  negassign  x,abs  (exit)
L3:  assign     x,abs  (exit)
```

In this case,

$$D = \{0, X, ABS\}$$

$$L = \{L1, L2, L3, EXIT\}$$

$$I = \{Test, Assign, Negassign\}$$

2.2 HAC complexity

Consider a machine which has the exact instruction set to execute only a single DGF. We define the size of an instruction to be the number of bits required to encode the instruction on this

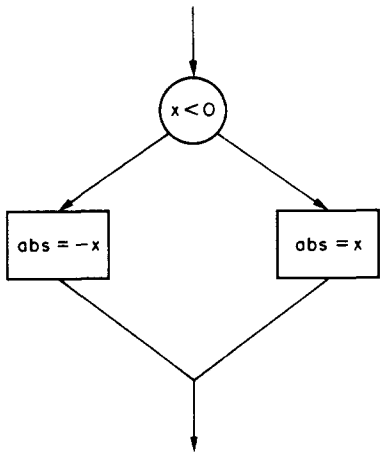


Fig. 1. Simple ABS function.

computer and the program size to be the number of bits necessary to contain the program. If there are n possibilities for any instruction field, the field must be $\log_2 n$ bits wide, and it can be viewed as the amount of information contained within the field [4].

We define the algorithm size (ASIZE) of a program P as the sum of the algorithm sizes of its constituent modules M:

$$\text{ASIZE} = \sum_{M \in P} \text{ASIZE}(M)$$

We define the algorithm size of a module M as the sum of the size of its constituent statements J:

$$\text{ASIZE}(M) = \sum_{J \in M} \text{ASIZE}(J)$$

For module M, let L be the set of labels, I be the set of opcodes, and D be the set of data objects. If J is an instruction in M with argument list of size k_j and label list of size m_j, we define the size of instruction J as:

$$\text{ASIZE}(J) = \log_2|L| + \log_2|I| + k_j{*}\log_2|D| + m_j{*}\log_2|L|.$$

From the DGF of Fig. 1, we can compute the size of the HAC program necessary to represent this program:

Each label field will be size $\log_2|L| = \log_2 4 = 2$
The instruction field will be size $\log_2|I| = \log_2 3 = 1.58$
Each argument field will be size $\log_2|D| = \log_2 3 = 1.58$

Since instructions have the format—*label: opcode, argument list (label list)*:

Instruction L1 has size = 2 + 1.58 + 2*1.58 + 2*2 = 10.74.
For node L2, the size is 2 + 1.58 + 2*1.58 + 2 = 8.74.
For node L3, the size is 2 + 1.58 + 2*1.58 + 2 = 8.74.
The size of the program is then 10.74 + 8.74 + 8.74 = 28.18 bits.

We believe that this model captures the actual implementation of any given program. In particular, the model captures:

- *The overall modular structure of a program in terms of the size of each module.* Operation size and label list fields are small if each module contains only a few instructions.
- *The data connectivity among program modules in terms of the number of data objects.* With few data items referenced in each module, data fields in each instruction are kept to only a few bits each.

2.3 Sequential form

As slight modification to an earlier presentation [5] we briefly describe the sequential form (SF) by assuming that the machine automatically sequences to the next instruction. This sequencing reduces the need for information in the instruction itself. Fall-through labels are no longer necessary, and the size of the label set L only needs to be the number of non-sequential labels actually referenced plus one *nil* label to cover sequential execution. The following represents the SF of the program of Fig. 1:

```
nil:  Test       x,0    (L3)
nil:  Negassign  x,abs  (exit)
L3:   Assign     x,abs
```

Only the third instruction needs a label (L3) since it is only instruction explicitly referenced (via instruction 1); the others are sequential execution. Only instruction 2 needs an explicit exit condition.

We compute a PSIZE (or program size) measure for this sequential form analogously to the ASIZE measure of the DGF. This yields a size of 22.12 bits while the ASIZE for this program was 28.18. The difference of approximately 6 bits represents information stored in the "machine" and represents the linearity of the source program. Hence the difference between these measures gives some indication of the "goto-ness" of the code.

It should be noted that we are actually defining a measure on a directed graph. When applied to a graph of a program, the value of the measure is sensitive to the granularity of each node—assembly language, programming language, specifications or requirements language. The measure is most useful in computing objects at the same level of abstraction, e.g. between 2 Ada modules or C modules and not, for example, between an Ada specification and its implementation.

3. PRIME PROGRAM COMPLEXITY

For the remainder of this paper, complexity will be defined in terms of a prime program decomposition of its flow graph. The prime program was developed by Maddux as a generalization of structured programming to define formally the unique hierarchical decomposition of a flow graph [6].

A *proper program* is a graph containing one input arc, one output arc and for each node in the graph, there is a path from the input arc through that node to the output arc. Figure 2 represents some proper programs, and Fig. 3 presents some examples of programs which are not proper.

A *prime program* is a proper program of more than one node that contains no proper subprograms of more than one node (i.e. no two arcs can be cut to extract another proper

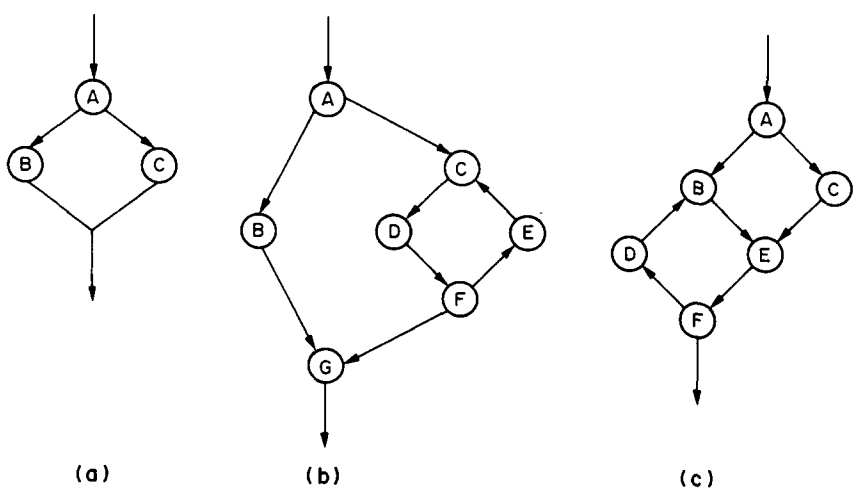


Fig. 2. Proper programs.

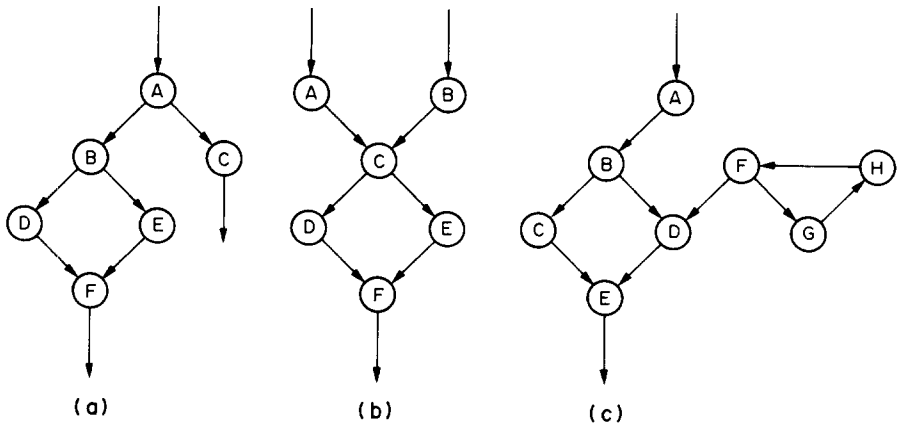


Fig. 3. Non-proper programs.

subprogram). For example, Fig. 2(a) is a prime while Fig. 2(b) is not. In Fig. 2(b), arcs A–C and F–G can be cut to extract proper program C–D–E–F.

The prime programs containing up to three nodes are the usual structured programming constructs such as *if*, *while*, and *repeat* statements (Fig. 4). But, the number of such primes is infinite [e.g. Fig. 2(c) is a prime of 6 nodes]. Replacing any subprogram in a graph by a single function node creates a unique hierachical decomposition of a graph into primes [7].

3.1 Prime program complexity

Consider the prime program decomposition of a graph so that each prime defines a unique HAC module, and the sum of the complexities of the modules is the complexity of the entire program. The prime sequential form (PSF) is the basis for our hierarchical complexity metric. Because of

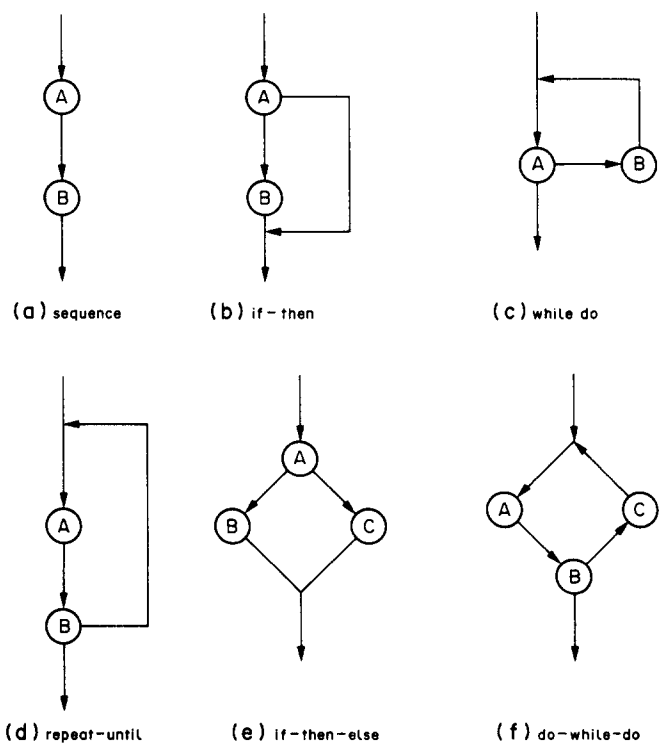


Fig. 4. Primes of up to three nodes.

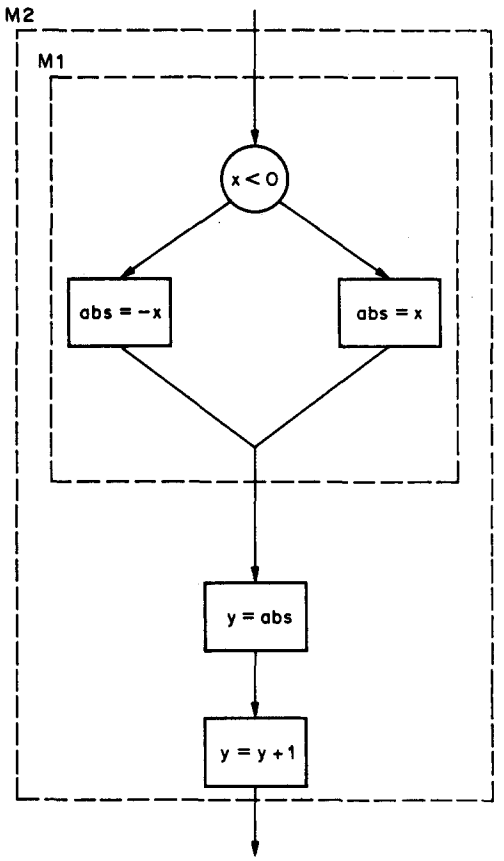


Fig. 5. Prime decomposition.

the unique prime factorization, we can eliminate labels altogether. The next location is uniquely determined by the particular instruction being executed—either a single operation if it is the function of some previously defined HAC or one particular instruction within the unique prime executed by this particular HAC. Thus instructions for the PSF are: ⟨*opcode*⟩ ⟨*argument list*⟩. Consider the prime decomposition of the program in Fig. 5 into the two prime PSF:

```
M1: test          x,0
    assign        x,abs
    else
    negassign     x,abs
M2: M1            x,abs
    assign abs,y
    + 1           y,y
```

The machine "knows" that following a false result from the *test* instruction execution continues with the *else* instruction, and a fall-through of the true path to the *else* means to exit the prime. With increased prime sizes, we simply need to define more special purpose instructions (opcodes) to handle the increased number of nodes. With the PSF, we capture the degree of locality evidenced by control flow operations and data object referencing.

For any prime program decomposition, we define the *Program Complexity* (CMPLX) measure. The Program Complexity metric is sensitive to the sizes of the prime programs present within the program modules since larger prime programs will result in larger complexity values. It is defined analogously to the ASIZE and PSIZE metrics.

Let P be a program with Modules M_p, each module decomposed into primes P_p. The program complexity (CMPLX) for a program P is:

$$\text{CMPLX}(P) = \sum_{M_p \in P} \text{CMPLX}(M_p)$$

The Program Complexity of a module M_p equals:

$$\text{CMPLX}(M_p) = \sum_{P_p \in M_p} \text{CMPLX}(P_p).$$

The Program Complexity of a prime program P_p equals:

$$\text{CMPLX}(P_p) = \sum_{J \in P_p} \text{CMPLX}(J)$$

for instructions J.

Let I = {opcodes in prime N}, D = {Data in prime N}, then the program compelexity of a HAC instruction $\text{opcode}_j a_{i_j} \ldots a_{k_j}$ equals:

$$\text{CMPLX}(J) = \text{Log}_2 |I| + k_j * \log_2 |D|$$

As an example, consider CMPLX for the program of Fig. 5:

M1: I = {Test, Assign, ELSE, negassign} $\log_2 |I| = 2$
D = {0, X, ABS} $\log_2 |D| = 1.58$

$$\text{CMPLX}(M1) = (2 + 2*1.58) + (2 + 2*1.58) + (2) + (2 + 2*1.58) = 5.16 + 5.16 + 2 + 5.16 = 17.48$$

M2: I = {M1, Assign, Assign + 1} $\log_2 |I| = 1.58$
D = {Y, ABS, X} $\log_2 |D| = 1.58$

$$\text{CMPLX}(M2) = (1.58 + 2*1.58) + (1.58 + 2*1.58) + (1.58 + 2*1.58) = 4.74 + 4.74 + 4.74 = 14.22$$

$$\text{CMPLX (program)} = 17.48 + 14.22 = 31.70$$

4. APPLICATIONS

Although defined on a sequence of linear instructions, the HAC represents a graph-based measure on a program. Therefore, our goal is to show the applicability of HAC complexity in graphically-based environments as a specifications or design aid. However, we first give several examples of the usefulness of the HAC measures using textural (e.g. source program) formats, and then describe the measure in terms of graph processes where other textual measures are not applicable.

4.1 Examples

In this section, we demonstrate the application of these measures in several situations that shows that they are sensitive to those features which affect complexity. In particular, we examine the behavior of the measures when presented with data abstractions and program modularizations with differing degrees of data coupling, and we conclude by contrasting their behavior with the McCabe metric to demonstrate that they are consistent with previous results [8].

4.1.1 Data abstractions. Current programming practice encourages the use of data abstractions as a means to enhance the quality of the program. The proper use of such abstractions allows the encapsulation of clusters of data activity into small defined objects and operations. While the practice has existed for many years, only recently have programming languages been available that provide sufficient facilities to create the abstractions conveniently and efficiently.

Figure 6(a) contains an Ada program which adds two rational numbers. Each number is implemented as pairs of integers. Figure 6(b) contains a modified version of the program which contains an abstraction of the data using a new data type *rational* and a new operator $\pm$ which sums data objects of this new type. Figure 7 contains the HAC PSF representations of these programs.

```
procedure add_rational(x1,x2,y1,y2: in integer; z1,z2: out integer) is
 begin
  z1 := x1*y2 + y1*x2 ;
  z2 := x2*y2 ;
 end add_rational;

procedure main is
  x1,x2,y1,y2,z1,z2 : integer;
 begin
  ---
  add_rational ( x1,x2,y1,y2,z1,z2 );
  ---
 end main;
```

a. Add Rational Numbers (no abstraction)

```
package rational_arith is
 type rational is record
                   numerator,denominator : integer
                 end record;
 function "+" (x,y : in rational ) return rational;
end rational_arith;

package body rational_arith is
 function "+" (x,y : in rational ) return rational is
  begin
     return ( x.numerator*y.denominator +
              y.numerator*x.denominator,
             x.denominator*y.denominator );
  end "+" ;
end rational_arith;

with rational_arith; use rational_arith;
procedure main is
  x,y,z : rational ;
 begin
  ---
  z := x + y
  ---
 end main;
```

b. Add Rational Numbers (data abstraction)

Fig. 6. Ada data abstraction example.

```
MAIN :
    ---
    ADD_RATIONAL x1,x2,y1,y2,z1,z2
    ---

ADD_RATIONAL:
    *  x1,y2,t1
    *  x2,y1,t2
    +  t1,t2,z1
    *  x2,y2,z2
```

a. HAC for non-abstraction program

```
MAIN :
    ---
    +  x,y,z
    ---

"+" :
    * x.num,y.den,t1
    * y.num,x.den,t2
    * t1,t2,z.num
    * x.den,y.den,t.den
```

b. HAC for abstraction program

Fig. 7. Prime sequential form for Ada example.

By applying the complexity measure, we observe that for the initial program, CMPLX = 15.48 + 40 = 55.48 bits, while for typed-rationals, CMPLX = 4.74 + 40 = 44.74 bits. The called procedures in this example have the same complexity (40 bits) since they perform the same transformation on the same data objects with the reduction in complexity due to the abstraction in one of the calling procedures (15.48 versus 4.74 bits). The reason for this difference is that by encapsulating the rational data type as a single concept, the program can deal with a single data object x, instead of multiple data objects, x_1 and x_2.

4.1.2 Data coupling. An important factor in overall program complexity is the degree of data coupling experienced between modules within the program. Consider the diagram shown in Fig. 8(a). There are many possible ways to modularize this structure, two of which are shown in Figs 8(b) and 8(c). The first decomposition, however, is accompanied by 9 dependencies between the modules implying an unnatural modularization, while the second realizes a small coupling (2 dependencies) implying a more natural modularization and a smaller complexity. Figure 9(a) illustrates a program which possesses this coupling pattern, Fig. 9(b) illustrates a PSF modularization of this program according to the first decomposition, and Fig. 9(c) illustrates a PSF modularization according to the second decomposition. Application of the HAC complexity on these programs yields a complexity of 71.18 + 58.44 + 73.32 = 202.94 bits for the unnatural decomposition and 18.24 + 49.74 + 61.18 = 129.16 bits for the more natural decomposition, demonstrating that the measure is sensitive to this software complexity influence factor.

4.1.3 Cyclomatic complexity. The cyclomatic complexity measure of McCabe is based on the number of linearly independent circuits through a program when represented as a graph [9]. Cyclomatic complexity, v, is defined as:

$$v = e - n + 2$$

where e is the number of edges in G, and n is the number of nodes in G. For graphs whose predicate nodes are all binary, i.e. those nodes which emit exactly two edges, the complexity is defined as:

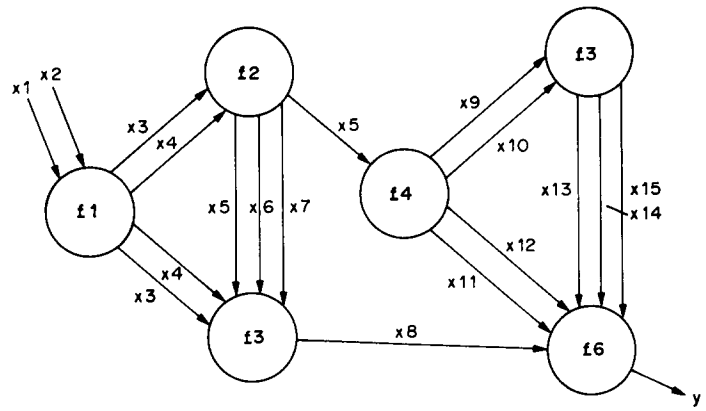
$$v = s + 1$$

where s is the number of predicate nodes in the graph. McCabe also defines an essential complexity, ev, as the cyclomatic complexity v of a reduced graph G, where the reduced graph is the outer module in the prime decomposition of the original graph.

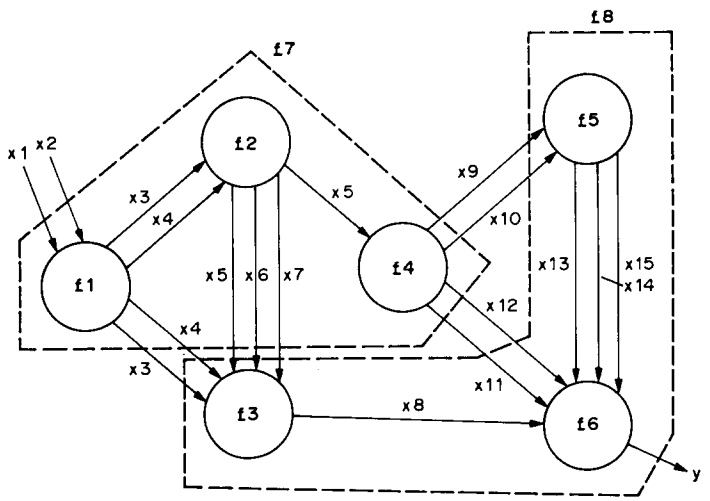
Cyclomatic complexity and CMPLX both share a sensitivity to the program control flow structure; however, CMPLX is measured from a linear model of the program rather than from a directed graph and is sensitive to deviations from this linear control flow. Such deviations are strong contributors to overall program complexity.

Consider the program to compute the Fibonacci sequence given in Fig. 10(a) with flowchart in Fig. 10(b). This program can be decomposed into three prime programs: P1, P2 and P3. The PSF and the CMPLX measures are computed as follows:

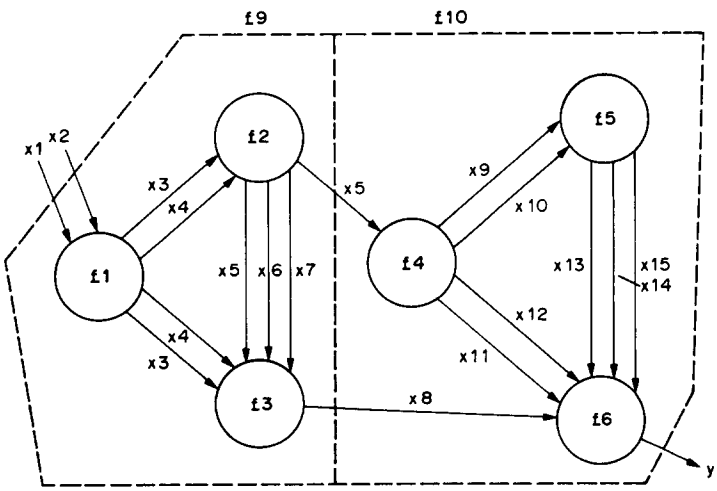
```
P1:  sub     n,1,t
     p3      t,p
     sub     n,2,t
     p3      t,q
     add     p,q,fib

P2:  test    n,1
     assign  1,fib
     else
     p1      n,fib

P3:  test    n,0
     assign  0,fib
     else
     p2      n,fib
```



(a)



(b)



(c)

Fig. 8. Data coupling.

```
PROGRAM:  L1: F1 X1,X2,X3,X4
          L2: F2 X3,X4,X5,X6,X7
          L3: F3 X3,X4,X5,X6,X7,X8
          L4: F4 X5,X9,X10,X11,X12
          L5: F5 X9,X10,X13,X14,X15
          L6: F6 X8,X11,X12,X13,X14,X15,Y
```

a. Original Program before Decomposition

```
PROGRAM:  F7 X1,X2,X3,X4,X5,X6,X7,X9,X10,X11,X12
          F8 X3,X4,X5,X6,X7,X9,X10,X11,X12,Y

F7:       F1 X1,X2,X3,X4
          F2 X3,X4,X5,X6,X7
          F4 X5,X9,X10,X11,X12

F8:       F3 X3,X4,X5,X6,X7,X8
          F5 X9,X10,X13,X14,X15
          F6 X8,X11,X12,X13,X14,X15,Y
```

b. Prime Decomposition of Figure 8(b)

```
PROGRAM:  F7 X1,X2,X5,X8
          F8 X5,X8,Y

F7:       F1 X1,X2,X3,X4
          F2 X3,X4,X5,X6,X7
          F3 X3,X4,X5,X6,X7,X8

F8:       F4 X5,X9,X10,X11,X12
          F5 X9,X10,X13,X14,X15
          F6 X8,X11,X12,X13,X14,X15,Y
```

c. Prime Decomposition of Figure 8(c)

Fig. 9. Prime descriptions of Fig. 8.

For P1, $|I| = 3$, $\log_s |I| = 1.58$
$|D|7$, $\log_2 |D|2.81$
CMPLX(P1) = 44.43

For P2, $|I|4$, $\log_2 |I| = 2$
$|D| = 3$, $\log_2 |D| = 1.58$
CMPLX(P2) = 11.48

For P3, $|I| = 4$, $\log_2 |I| = 2$
$|D| = 3$, $\log_2 |D| = 1.58$
CMPLX(P3) = 11.48

$$\text{CMPLX} = 44.43 + 11.48 + 11.48 = 67.39.$$

However, this program can also be represented by the flowchart of Fig. 10(c) which can be decomposed into two prime programs—the P1 prime of Fig. 10(b) and a more complex P4 prime.

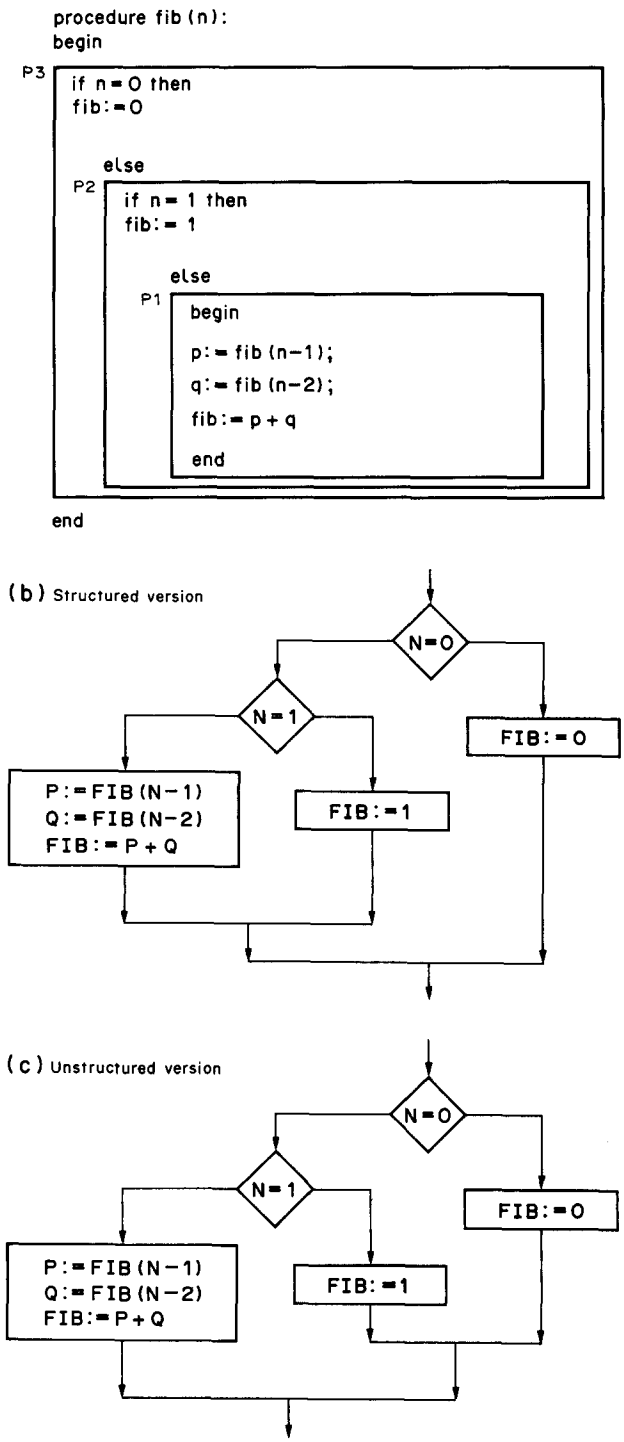


Fig. 10. Comparison with cyclomatic complexity.

Computing the PSF and the CMPLX measures gives:

P4:	test	n,0
	assign	0,fib
	test2	n,1
	assign	1,fib
	branch	
	P1	n,fib

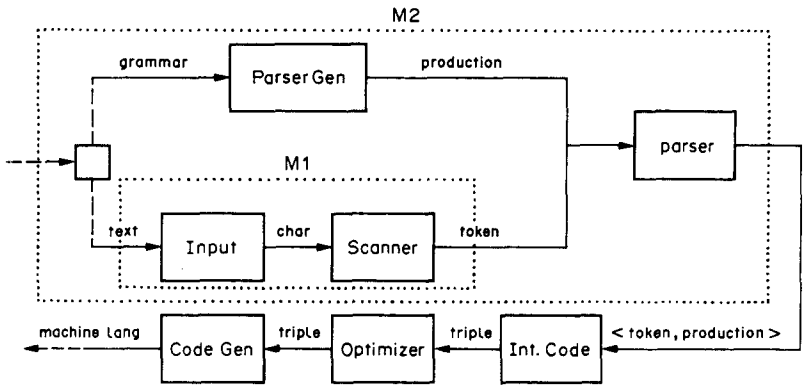


Fig. 11. Compiler structure.

For P4, $|I| = 5$, $\log_2 |I| = 2.32$
$|D| = 4$, $\log_2 |D| = 2$
CMPLX(P4) = 33.92

$$\text{CMPLX} = \text{CMPLX(P1)} + \text{CMPLX(P4)} = 44.43 + 33.92 = 78.35.$$

Note that this figure of 78.35 is greater than the figure of 67.39 of the more structured program; however, both versions of the program have the same cyclomatic complexity measure of 3.

4.2 CASE complexity

Of great interest today is the concept of integrated environments and CASE (Computer Aided Software Engineering) tools. It is important to store knowledge of the programming process within the computer to give the programmer or system analyst benefits from this knowledge. A common feature for many of these tools is a graphically-based specifications methodology using data flow diagrams, "bubble charts" or other pictorial structure [10]. Measuring the structure of these designs would be an important analysis tool, but such measurements are lacking. Most existing measures are based upon a textual source code representation of the program. We propose that a measure like the HAC CMPLX measure could play a role in these environments.

Consider the structure chart of a typical compiler in Fig. 11. In this example, the compiler has two inputs (grammatical description of the programming language and a given source program to compile), and has one output (the machine language translation of the input source program). We can represent this as a HAC program, convert it to its PSF and compute the CMPLX measure on the resulting program:

(1) Convert the structure chart to a proper program by adding one input node with no input data items that branches to the given input nodes (dashed lines in Fig. 11). Do a similar transformation to the output data, but in this case there is only one output item already specified.
(2) Compute the prime program decomposition of this flowgraph (dotted lines surrounding boxes M1 and M2 in Fig. 11).
(3) Derive the PSF for each prime and compute its CMPLX measure.

For Fig. 11, we get the following decomposition:

M1 scanner module—

Input	text, char
Scanner	char, token

which has $|I| = 2$, $|D| = 3$, $|I_1| = \log_2 2 + 2*\log_2 3 = 4.16$, $|I_2| = \log_2 2 + 2*\log_2 3 = 4.16$, and CMPLX(M1) = 8.32.

M2 parser module—

Start	text, grammar
M1	text, token
Exit	
ParserGen	grammar, production
Parser	token, production, ⟨token, production_number⟩

which has $|I| = 5$, $|D| = 5$, $|I_1| = \log_2 5 + 2*\log_2 5 = 6.96$, $|I_2| = \log_2 5 + 2*\log_2 5 = 6.96$, $|I_3| = \log_2 5 = 2.32$, $|I_4| = \log_2 5 + 2*\log_2 5 = 6.96$, $|I_5| = \log_2 5 + 3*\log_2 5 = 9.28$, and CMPLX(M2) = 32.48.

M3 Entire compiler—

M2	⟨token,production_number⟩
IntCode	⟨token,production_number⟩, triple
Optimizer	triple, triple
CodeGen	triple, machine_language

which has $|I| = 4$, $|D| = 3$, $|I_1| = \log_2 4 + \log_2 3 = 3.58$, $|I_2| = \log_2 4 + 2*\log_2 3 = 5.16$, $|I_3| = \log_2 4 + 2*\log_2 3 = 5.16$, $|I_4| = \log_2 4 + 2*\log_2 3 = 5.16$, and CMPLX(M3) = 19.06.

The total complexity of this top level design is then $8.32 + 32.48 + 19.06 = 59.86$.

Using this measure, alternative designs can be evaluated early in the development cycle and give indications of alternative strategies that can be applied. All too often the only criteria used in such an evaluation is the experience of the designer. In this case, we have an objective measure that is easily programmable into such a CASE environment that can be used as a design aid before source (or even design) code has been written. With many CASE tools based upon such graphical representations of the process, a prime decomposition measure captures its internal structure naturally.

We do not claim that our CMPLX measure is the ultimate decider of good design; however, we do claim that a measure, similar to the CMPLX measure on the prime decomposition of a flowgraph, does agree with our intuitive notions of good designs, and that measures based upon this prime decomposition will ultimately provide effective measurements in this area.

5. CONCLUSIONS

One common view of complexity is the idea that simplicity and complexity are dependent on the number of structural features contained within an organization rather than simply on the number of its basic elements. That is, pure size is not as strong an influence as the interrelationships that exist among those elements. The study of algorithm and program structures reflects this view in that the discipline of software engineering directs its energies at managing the coupling patterns within programs by suitable partitioning of the algorithmic components. Examples of such efforts include the structured programming methodologies, formalized specification and design techniques, and the utilization of integrated development environments containing tools designed to aid the management of complexity.

This paper has defined a view of algorithm structure based on a finite-state graph model. This view allows a hierarchical decomposition of the program from its level of expression down to an atomic, primitive level and thus reveals the entire internal organization of the algorithm as it is actually interpreted on a physical processor. By taking this view, we have observed that there exists a natural level to each program or operation in its distance from the atomic level. We have also demonstrated that within this internal organization, there is the potential for recognizing patterns or clusters of activity. Some programs do not have easily recognizable patterns, and are considered to be inherently more complex than those that do. The programs that contain recurring patterns can be described, using the HAC notation, in a much shorter representation than at their primitive level. The degree of reduction in size is inversely proportional to the inherent complexity of the program, and is directly proportional to its simplicity or capability of being remembered.

Measures were defined for several HAC program representations. These measures were demonstrated to detect those structural features which are considered to be influential in affecting program complexity. In particular, the model is useful for determining complexity of graphically-

based specification and design processes when we do not yet have concrete textual representation of the desired solution.

The prime program decomposition has a strong connection to the concept of good module design, therefore, we believe that a complexity measure based upon this decomposition will be a better predictor of adherence to good software engineering principles than more *ad hoc* program measures. We also believe that the overall approach of using the HAC computational model to analyze program structures is valid and useful in that it presents a view that has not been taken by previous models.

Acknowledgement—Research on this grant was partially supported by NASA grant NSG-5123 to the University of Maryland.

REFERENCES

1. Chaitin G. J., A theory of program size formally identical to information theory. *J. ACM* (July 1975).
2. Bail W. and Zelkowitz M., Program complexity using hierarchical abstract computers. *National Computer Conference* (1978).
3. Halstead M., *Elements of Software Science*. Elsevier, New York (1977).
4. Shannon C. and Weaver W., *The Mathematical Theory of Communication*. The University of Illinois Press (1974).
5. Bail W., Algorithmic structure analysis using hierarchical abstract computers. Ph.D. dissertation, Computer Science Department, University of Maryland (1985).
6. Maddux R., A study of computer program structure. Ph.D. dissertation, University of Waterloo (July 1975).
7. Gannon J. D., Hecht M. S. and Herbold R. S., Prime program decomposition. *16th Hawaii International Conference on Systems Science*, pp. 25–29 (January 1983).
8. McCabe T. J., Structured testing: a software testing methodology using the cyclomatic complexity measure. *NBS Special Publication 500–99* (December 1982).
9. McCabe T. J., A complexity measure. *IEEE Trans. Software Engng* **2**(6), 308–320 (December 1976).
10. *ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Palo Alto, Calif. (December 1986). [*ACM SIGPLAN Notices* (January 1987).]

About the Author—WILLIAM BAIL is the Deputy Division Director for the Washington Division of Intermetrics. His current interests include program complexity measures, language design and translation, logic and expert systems development, and software development methodologies. Dr Bail has been with Intermetrics since 1979 as a Senior Computer Scientist, and was a Visiting Lecturer at the University of Maryland from 1984 to 1987. From 1966 to 1979 Dr Bail was with Vitro Laboratories. He received his B.S. in Mathematics from Carnegie Institute of Technology in 1966, and his M.S. and Ph.D. in Computer Science from the University of Maryland in 1973 and 1985 respectively.

About the Author—MARVIN V. ZELKOWITZ is Associate Professor of Computer Science at the University of Maryland. He obtained the B.S. degree in Mathematics from Rensselaer Polytechnic Institute and the M.S. and Ph.D. degrees in Computer Science from Cornell University. In the past he has had various positions including Chairman of ACM SIGSOFT, Chairman of the Computer Society's Technical Committee on Software Engineering, and both Associate Chairman and Acting Chairman of the Department of Computer Science. Dr Zelkowitz's research interests include language and compiler design, programming environments and the measurement of the programming process, a topic which led to the work reported here.