1

**Rapid #: -3994600**

**Ariel**
**IP: 130.39.60.199**

| Status | Rapid Code | Branch Name | Start Date |
|---|---|---|---|
| Pending | IQU | CSEL | 12/16/2010 11:05:58 AM |

**CALL #:** QA76.7 .C647
**LOCATION:** IQU :: CSEL :: sper ,sper1

TYPE: Article CC:CCL
JOURNAL TITLE: Computer languages
USER JOURNAL TITLE: Computer Languages PDF PREFERRED, THANKS!
IQU CATALOG TITLE: Computer languages
ARTICLE TITLE: Computation of Logical Effort in High Level Languages
ARTICLE AUTHOR: S.S.Iyengar
VOLUME: 9
ISSUE: 3
MONTH:
YEAR: 1984
PAGES: 133-148
ISSN: 0096-0551
OCLC #: IQU OCLC #: 2246698
CROSS REFERENCE ID: [TN:4741754][ODYSSEY:216.54.119.188/LUU]
VERIFIED:

REQUEST UPDATED TO FILLED

**BORROWER:** LUU :: Main Library
**PATRON:** karthik nagabandi
PATRON ID: knagab2
PATRON ADDRESS:
PATRON PHONE:
PATRON FAX:
PATRON E-MAIL:
PATRON DEPT:
PATRON STATUS:
PATRON NOTES:

*Comput. Lang.* Vol. 9, No. 3/4, pp. 133–148, 1984
Printed in Great Britain. 

0096-0551/84 $3.00 + 0.00


# COMPUTATION OF LOGICAL EFFORT IN HIGH LEVEL LANGUAGES

STEVEN C. CATER, S. SITHARAMA IYENGAR and JOHN FULLER
Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, U.S.A.

(*Received* 14 *September* 1983; *revision received* 13 *December* 1983)

**Abstract**—A new measure of software complexity is introduced, that of logical effort. This measure is an attempt to quantify program complexity by an analysis of the control structure of the program, using the concepts of language virtuality and segment independence. A program is decomposed by the use of these concepts until measurements can be made at the statement level. These measurements, along with measurements of I, the effort necessary to perform a loop, and Q, the effort necessary to determine which branch to take in a decision, are then used to calculate recursively the effort in larger and larger structures, until the program level is reached. An example of this process is given for a short program, along with a brief comparison of the results obtained with a similar measure, that of cyclomatic complexity.

Algorithms for the computation of I and Q are also given, along with examples of their calculation. This process is essentially a decomposition of a Boolean function into disjunctive normal form, followed by a minimization of the form using a weighting measure involving weights on both the operators and the variables in the function.

Software complexity Computation Logical effort Algorithms Disjunctive normal form Minimization Language virtuality Segment independence

## 1. INTRODUCTION

The concept of program complexity has become an active concern of researchers in software engineering, as practical methods of maximizing program understanding and testability become more necessary[1,6]. Due to a better appreciation of the software life cycle, with its implicit requirement that program maintenance be as effortless as possible, measures of software complexity have become increasingly important in recent years, with measures such as those by Halstead *et al.*, becoming the subject of much research and experimentation[1–8]. A recent report seems to indicate that an explanation of the data structures used in a program or algorithm may be more important in the understanding and writing of a program than is an explanation of the flow of control in the program[19]. Should this prove to be the case, a measure such as that given by Parameswaran and Iyengar[14,15], which attempts to quantify the data structure complexity as a portion of the overall complexity, will prove to be useful.

There remains, however, one aspect of software complexity that has not been discussed fully: that aspect is a consideration of the difficulty in writing a program from a given algorithm, using as an approach the twin concepts of language virtuality and segment independence, formal definitions of which are given later in this report. The emphasis on virtuality and independence is a result of two considerations: first, most recent general purpose programming languages use the concepts in their basic structure, and, second, these twin concepts are beginning to be used in other parts of computer science[20–22]. Thus, these ideas seem to have wide applicability. We therefore introduce a new measure of program complexity, the logical effort in a program, a measure which utilizes independence and virtuality.

Our discussion of logical effort will proceed as follows: we begin with the necessary definitions for the various measures included in logical effort, and an explanation of the underlying concepts inherent in a determination of the overall measure. We then preceed to define the logical effort in a program in a top-down fashion. The next sections continue the definition by considering sequence effort, decision effort, and loop effort. In order to finish the definition of decision effort, the quantity I is quantified, through the use of a "minimal" disjunctive normal form. A similar result is given for loop effort and the quantity Q. There follows a listing of the pseudo-code for calculation of logical effort, and an example using a program written in structured FORTRAN, along with a brief comparison of our results with McCabe's cyclomatic complexity. Finally, we consider some possible directions for future research.

## 2. DEFINITIONS

The two concepts central to the definition of logical effort are those of language virtuality and segment independence. Language virtuality is reflected in the present tendency towards top-down design of programs. At any moment in the design of a large program, one may assume that one is working, not with FORTRAN, for example, but instead with a superset of FORTRAN which includes the necessary features (as subroutines) to implement the desired program. A virtual language is thus the language in which a programmer thinks he is working, and consists of the actual language along with any subroutine libraries which may be present, the two together with any not yet implemented subroutines which might prove useful. Segment independence is the obverse of language virtuality: any segment of a program may be brought out from its program and considered to be an example of a virtual language in its own right (perhaps with minor syntax changes). Thus, segment independence is that property of a programming language that allows any segment to be considered, essentially, as a program in its own right. A program is thus written in its own virtual language, and consists of a collection of individual programs, each written in its own virtual language. The combination of these two principles means that, at any moment, one need only work with a level of detail appropriate to the problem at hand, whether that be the overall flow of control or the details of the printed output. We therefore have a fundamental rule for the computation of logical effort: a subroutine call will count the same as an intrinsic function call. With this basic understanding of language virtuality and segment independence, we can now proceed to define a program and the various measures of logical effort.

For our purposes, a program is a structured, finite collection of code, written in a non-virtual language, designed to implement a given algorithm. By structured we mean that the program has one entry, one exit, no dead code, and is composed only of the allowed structures. As our main concern is complexity in a program, we will be primarily concerned with the subdivisions of a program. Thus, a program is considered to be a collection of one or more program segments, one

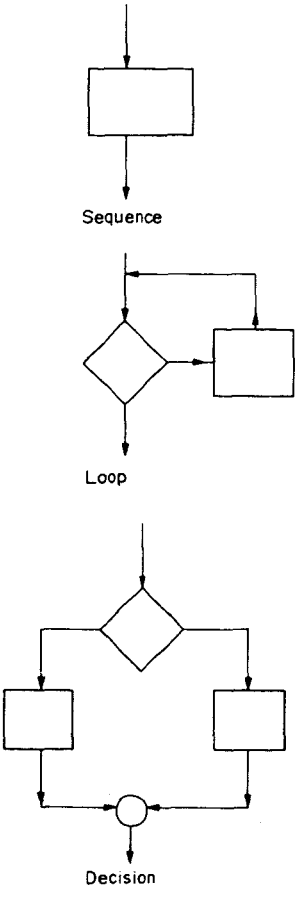


Fig. 1. Standard structures.

called the main program, the others called subroutines or subprograms. A program segment, generally referred to as simply a segment, is a logical collection of code designed to perform a single task. This reflects the modularity and segment independence inherent in the top-down method of program design. We equate a segment with a structure, which is either a sequence, a decision, or a loop. Thus at the upper limit of analysis, a segment is simply an independent structure, which may be part of a larger program.

The structures that we allow are the traditional ones, along with variants of these basic ones, such as elseif, case, and until (Figs 1 and 2). Following the usual practice, structures are composed of underlying structures, until the statement level is reached. We equate a statement with the lowest possible sequence, so the nesting of structures is finite. A statement is the basic unit of code in a programming language, and is defined for each language. Finally, statements are composed of variables, constants, and the operators defined by the language, such as "+", "SQRT", and "=". In view of our position on language virtuality, we shall be interested not only in these primitive operators (those defined by the language), but also with virtual primitive operators (i.e. subroutine calls of any type), and join both under the name of atomic operators. These definitions give rise to the BNF grammar for PROGRAM found in Fig. 3. The INITIALIZATION in the expansion of DECISION is the portion of the decision structure that determines which branch to take, or whether to take a given branch. It is generally seen as the condition portion of an IF statement or one of its variants. The CONDITION in the expansion of LOOP is generally found as the condition portion of a WHILE or UNTIL statement, while the COUNT is usually found in the form of a counting DO-loop. These definitions, along with the given grammar, comprise our definition of program and its constituent parts.

The measurement of logical effort of a program consists of the decomposition of the program into its parts, the measurement of these parts, and then a reconstruction of the effort based on the measurement of the parts, in a top-down, bottom-up fashion. To aid this process, we define various E-measures, which are named with either three or four letters, the first of which is E (Table 1).

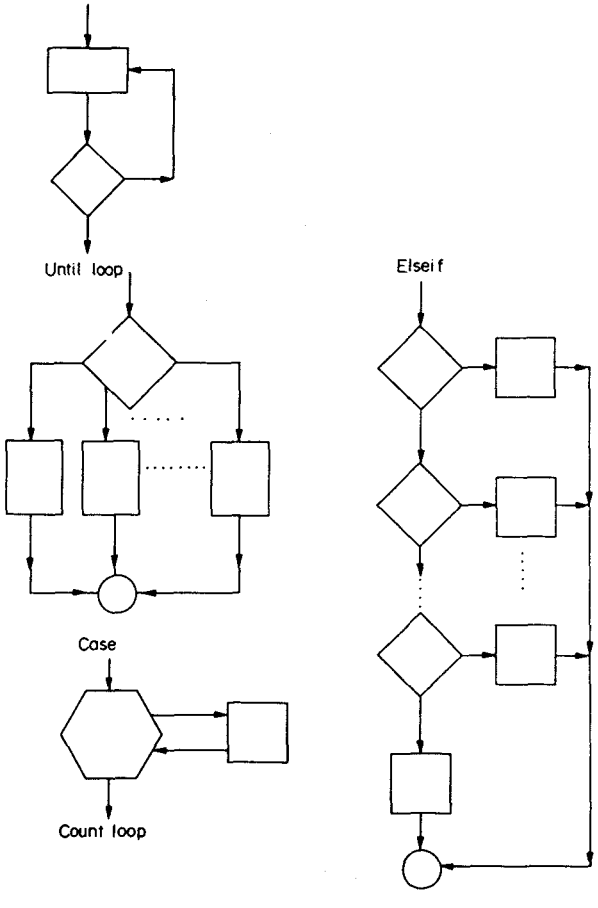


Fig. 2. Some additional structures.

```
⟨PROGRAM⟩        ::= ⟨SEGMENT⟩ | ⟨PROGRAM⟩⟨SEGMENT⟩
⟨SEGMENT⟩        ::= ⟨STRUCTURE⟩
⟨STRUCTURE⟩      ::= ⟨SEQUENCE⟩ | ⟨DECISION⟩ | ⟨LOOP⟩
⟨SEQUENCE⟩       ::= ⟨STATEMENT⟩ | ⟨STRUCTURE⟩⟨STRUCTURE⟩
⟨DECISION⟩       ::= ⟨INITIALIZATION⟩⟨STRUCTURE⟩⟨STRUCTURE⟩ |
                     ⟨INITIALIZATION⟩⟨STRUCTURE⟩⟨DECISION⟩ |
                     ⟨INITIALIZATION⟩⟨STRUCTURE⟩
⟨LOOP⟩           ::= ⟨CONDITION⟩⟨STRUCTURE⟩ |
                     ⟨STRUCTURE⟩⟨CONDITION⟩ |
                     ⟨COUNT⟩⟨STRUCTURE⟩
⟨STATEMENT⟩      ::= language dependent
⟨INITIALIZATION⟩ ::= language dependent
⟨CONDITION⟩      ::= language dependent
⟨COUNT⟩          ::= language dependent
```

Fig. 3. BNF description of PROGRAM.

There are two basic types of measures, total measures and maximal measures, which have T and M as their last letter. There are six values that can occur in the middle, as shown in the second grouping in Table 1. These values are simply the various parts of a program defined earlier. Thus, some of the possible E-measures are EPT, the logical effort for the program as a whole, using the total measure, EPM, the logical effort for the entire program, using the maximal measure, and ESGT, the total logical effort for some segment in a program. These various types of measures are a result of segment independence, and are all measured in essentially the same manner, with the only difference being the level of the program under consideration. The last three groups in the table are ones that are needed in order to define the various E-measures, and will be discussed later. The domain of these measures is indicated by one of three means. First, the name of the

Table 1. Abbreviations used in the calculations of E-measures

| | |
|---|---|
| E | Effort |
| T | Total |
| M | Maximal |
| P | Program |
| SG | Segment |
| SQ | Sequence |
| LP | Loop |
| DC | Decision |
| ST | Statement |
| LN | Length |
| OP | Operator count in a statement |
| I | Effort to decide which branch to take in a decision |
| IB | Effort to decide whether to take a branch in a decision |
| Q | Effort to decide how many times to execute a loop |
| QC | Effort in a count type of loop |
| QS | Effort to decide whether to start a loop |
| QE | Effort to decide whether to end a loop |

Table 2. Definitions of the E-measures

$$EPT = \Sigma ESGT$$
$$ESGT = ESQT$$
$$ESQT = LN + \Sigma ESTT$$
$$ESTT = \begin{cases} OP & \text{if non-structure} \\ EDCT & \text{if decision} \\ ELPT & \text{if loop} \end{cases}$$
$$EDCT = I + \Sigma ESQT$$
$$ELPT = Q + ESQT$$

LN = number of statement-structures
$I = \Sigma IB$
IB = see procedure
$Q = QS + QE + QC$
QS = see procedure
QE = see procedure
QC = see procedure
OP = number of operator-calls

EPM = max (ESGT)
ESGM = ESQM
ESQM = max (LN, max (ESTT))
EDCM = max (I, max (ESQM))
ELPM = max (Q, ESQM)

domain is enclosed in parentheses following the measure name, if the domain has a name. The second and third ways of indicating the domain of a measure involve line numbers. We assume that the program has been assigned line numbers, and refer to a single line by mention in parentheses; for a non-statement segment, we use the form line 1:line 2, to refer to an inclusive set of lines from line 1 to line 2.

The definitions of the measures that will be considered in this paper are given in Table 2. Note that some of the possible measures are not listed (e.g. EDCM), as we consider them to be of secondary importance. In each of these definitions it is understood that the domains are the structure itself on the left side of the equation, and the underlying substructures on the right. In particular, the LN measure is one per statement or substructure, and the value of OP is one per atomic operator. This table is of course not complete in itself. In the next section we begin to justify these definitions, beginning with EPT and EPM.

## 3. EPT, EPM, AND SEQUENCE EFFORT

There are at least two types of programs that may be said to be complex. The first type is a program that is extremely long, even though no individual segment is difficult to understand. The second type of complex program is one that is not necessarily lengthy, but instead is composed of one or more segments that are particularly complex. In order to account for these different types of complexity, we define both total and maximal measures. At any level, a total measure consists of the sum of the efforts of the substructures. Thus, since a program is composed of segments, the measurement of EPT is obtained by taking the sum of the segment efforts, in particular the ESGT of each segment. Similarly, at any level, a maximal measure consists of the maximum of the efforts of the substructures. EPM is thus the maximum of the total efforts of the program segments.

At the segment level, the situation changes slightly. Any segment is just a sequence of one block, since it performs one task, has one entry and one exit with no dead code, and is composed of the allowed structures. We therefore equate segment effort with sequence effort for both the total and maximal measures. Hence, ESGT = ESQT and ESGM = ESQM. However, the definition of sequence effort introduces a new attribute, that of length. This attribute is a reflection of the same possibility mentioned above; it is possible for a sequence to be complex due solely to length. As a result, ESQT is the sum of the underlying structures plus the length. The length, as mentioned above, is simply one per statement if not a structure and one per underlying structure. The definition of ESQM is similarly the maximum of the length with the maximum of the total efforts of the substructures. (This introduction of a new attribute for the sequence structure is actually a special instance of a more general case; each of the three basic structure types will have a new attribute introduced.)

We are now able to calculate the logical effort in any program which contains no decisions or loops. As mentioned earlier, the value of ESTT for a pure statement is OP of the statement, i.e. a count of the atomic operators in the statement. Then, using the previously defined measures, it is possible to calculate ESQT, ESGT, and finally EPT. Similarly, the corresponding maximal measures may be determined for a program of this type. Unfortunately, most interesting programs do contain both loops and decisions. As the calculation of the E-measures for decisions and loops is more involved, we consider these types of calculations in the next sections.

## 4. DECISION EFFORT AND LOOP EFFORT

Decisions and loops may be considered to be compound structures as each is composed of one or more sequences (which may contain substructures), along with additional code to determine either which sequence to take or how many times to execute the sequence. These additional attributes for decisions and loops are separated out in the same manner that length was for sequence and provide the definitions of I and Q. I is defined to be the effort required to determine which branch to take in a decision, while Q is the effort required to determine how many times to execute a loop. Since decisions and loops are simply sequences with additional code, the definitions of the E-measures for decisions and loops are exactly what would be expected, e.g. EDCT is the sum of the ESQT for each subsequence of the decision, along with the value of I; and ELPM is the maximum of Q and the value of ESQM of the underlying sequence. Since the

calculations of sequence effort have already been defined, the only definitions that remain are those for I and Q.

Unlike LN, I and Q are themselves compound measures. The value of I is given by the sum of the efforts to decide whether to take each particular branch in a decision. This value is denoted IB. This definition allows for the non-traditional types of decision structures like CASE and ELSEIF to be used in a program, reflecting standard usage. Similarly, Q can be decomposed into the effort to determine whether to start a loop again, the effort to determine whether to end a loop, and the effort to determine how many times to execute a loop. These values are denoted QS, QE, and QC, and reflect the WHILE, UNTIL, and DO-count structures available in most languages. The method used to calculate values of IB is a generalization of a technique used in logic and switching theory, and will be considered in the next section. Determination of the Q values is similar to determination of IB in two of the three cases and will be considered after the calculation of IB has been discussed.

## 5. CALCULATION OF IB

The procedure we use for the determination of IB is as follows:

(1) The condition used to determine whether to take a given branch is decomposed into a string of atoms and operators, the atoms being the substrings maximal with respect to non-containment of unary and binary Boolean operators.
(2) This decomposition is expressed in disjunctive normal form, and is then minimized with respect to a weighting based on weights of atoms and Boolean operators.
(3) The number of operators, both Boolean and non-Boolean, is then counted; the resulting number is IB.

Distinctive normal form is a standard form for the expression of Boolean formulas in the propositional calculus[23]. There are three advantages of using DNF. First, DNF exists for any proposition which is not a contradiction (of course, a contradiction represents dead code, which is not allowed in structured programming). Second, an expression in DNF requires only the use of NOT, AND, and OR operators in its representation, so, by conversion to this standard form, we obtain an equivalent expression which is simpler with respect to the number of different Boolean operators required. An argument may be made here that one may as well use a minimal set of operators, such as NOR, in the conversion to a standard form. We feel that such a reduction, while certainly advantageous in the reduction of the number of distinct Boolean operators, has as a disadvantage the removal of a certain naturalness. The conversion to DNF retains this naturalness. The final advantage of DNF is that there exist simple algorithms for its computation.

The standard algorithm for conversion to DNF does suffer from a disadvantage: the form that is obtained may sometimes seem much more complicated than the original formulation of the proposition. As an example, consider the proposition $\sim(p \,\&\, (q \mid r))$. Using the truth-table algorithm for conversion to DNF, one obtains:

$$(p \,\&\, \sim q \,\&\, \sim r) \mid$$
$$(\sim p \,\&\, q \,\&\, r) \mid$$
$$(\sim p \,\&\, q \,\&\, \sim r) \mid$$
$$(\sim p \,\&\, \sim q \,\&\, r) \mid$$
$$(\sim p \,\&\, \sim q \,\&\, \sim r).$$

Our solution to this problem is a modification of the standard solution used in switching theory: we modify the standard DNF obtained by this algorithm, using a weighting system for the parts of the proposition[24]. Our conversion to a "minimal DNF" (MDNF) is different from most used in switching theory in that we consider the weights on the Boolean operators to be one, and use non-unit weights on the individual propositional variables. After conversion to our MDNF (using a weight of one for the variables), the above expression becomes simply

$$\sim p \mid (p \,\&\, q \,\&\, \sim r).$$

Thus, MDNF retains the advantages of DNF while reducing the problem inherent in the standard form.

The algorithm we use for conversion to DNF is the truth-table technique[23]. Our version of the algorithm follows:

(1) Call a portion of the condition an atom if it is maximal with respect to non-containment of Boolean unary and binary operators.
(2) Replace each distinct atom by a unique letter.
(3) Construct the truth table of the expression obtained in (2). If the truth table has only values of false, then stop, since the expression is a contradiction.
(4) In each line of the table which has a value of true, construct an expression composed of the conjunction of the atomic variables, each variable prefixed by NOT if and only if the variable has the value false in that line.
(5) Form the disjunction of the expression in (4). This is the atomic DNF of the formula. To obtain the DNF of the formula, replace the atomic variables with their values. This is the equivalent DNF of the original proposition.

As an example of the above procedure, consider the following FORTRAN statement:

IF (A .LT. B – 17 .AND. .NOT. (C – 17 .EQ. 0 .OR. D .GE. 4)) A = B – C.

The atoms are "A .LT. B – 17", "C – 17 .EQ. 0", and "D .GE. 4". Replace these by p, q, and r, respectively, to obtain the proposition p & ~(q | r). The truth table is:

| p | & | ~ | (q | r) |
|---|---|---|---|---|
| T | F | F | T | T | T |
| T | F | F | T | T | F |
| T | F | F | F | T | T |
| T | T | T | F | F | F |
| F | F | F | T | T | T |
| F | F | F | T | T | F |
| F | F | F | F | T | T |
| F | F | T | F | F | F |

The only true result appears in line four of the table where p has value true and q and r have value false. Hence, the atomic DNF is p & ~q & | r, while expansion of the atoms gives A .LT. B – 17 .AND. .NOT. C – 17 .EQ. 0 .AND. .NOT. D .GT. 4.

A slightly more complicated example can be given from PL/I with the statements:

```
IF (A & B ~ = 17 | BOOL (A, C, '1001'B))
THEN B = ABS (B);
ELSE B = 0;
```

There are two values of IB that must be calculated here: the value in the first line, and the value in the third line. The third line contains a null conditional, so that its value is zero. In the first statement, the atoms are "A", "C", and "B ~ = 17". Replacement by atomic variables a, c, and b, gives (a & b) | (a < = > c) which has the following truth table:

| (a | & | b) | | (a | < = > | c) |
|---|---|---|---|---|---|---|
| T | T | T | T | T | T | T |
| T | T | T | T | T | F | F |
| T | F | F | T | T | T | T |
| T | F | F | F | T | F | F |
| F | F | T | F | F | F | T |
| F | F | T | T | F | T | F |
| F | F | F | F | F | F | T |
| F | F | F | T | F | T | F |

All lines except four, five, and seven have value true, so the atomic DNF is:

$$(a \,\&\, b \,\&\, c) \mid (a \,\&\, b \,\&\, \sim c) \mid$$
$$(a \,\&\, \sim b \,\&\, c) \mid (\sim a \,\&\, b \,\&\, \sim c) \mid$$
$$(\sim a \,\&\, \sim b \,\&\, \sim c).$$

As before, one replaces a, b, and c by their values to obtain the DNF of the expression.

The problem with conversion to DNF in this second case is that the resulting expression is much more complicated than the original, and the expansion seems somehow artificial. We overcome this problem by conversion to MDNF, as follows. We define the weight of the variables and operators in an expression in DNF by assigning a weight of one to each of the Boolean operators, and a weight of OP(atom) to each of the variables. Thus in the PL/I example the weights are $w(a) = w(c) = 0$, and $w(b) = 1$, for the operator "$\sim =$". After the weights are assigned, find an equivalent DNF of minimal total weight, by eliminating redundancies in the original DNF. Continue the process until all redundancies have been eliminated. Therefore, the conclusion of our algorithm for conversion to MDNF and the calculation of IB becomes:

(6) Using the atomic DNF from step (5), assign a weight of one to each Boolean operator, and a weight of OP(atom) to each of the variables in the DNF.

(7) Convert from DNF to MDNF by eliminating the redundant variables, until an equivalent DNF of minimal total weight is obtained, i.e. find all places where two disjuncts are exactly the same except for one atom, which will appear in both negated and non-negated forms. Replace these two disjuncts with a single disjunct which eliminates reference to the redundant atom. Continue until there are no more redundancies to eliminate.

(8) Once the atomic MDNF is obtained, convert to MDNF by replacing the variables with their values. The value of IB is then OP(MDNF), the operator count of the resulting MDNF.

Step seven is actually more complicated than it perhaps could be. The choice of redundancies to eliminate must be ones that will result in a minimal total weight over all possibilities of elimination, i.e. the choice must guarantee that the resulting weight is the least that could be obtained and still leave the expression in DNF. One way to do this is by considering all possibilities of elimination; this makes the algorithm at best exponential in the number of variables. In practice this will not be much of a problem, since there will be few times that the number of atoms will be over three or four.

Continuing the above examples, it is clear that in the FORTRAN example, the MDNF is just the DNF, since there is only one disjunct. The PL/I example is not minimal, e.g. the first two disjuncts are redundant. Applying step seven to this example gives

$$(a \,\&\, c) \mid (\sim a \,\&\, \sim c) \mid (a \,\&\, b \,\&\, \sim c).$$

As a result, the value of IB for the FORTRAN example is nine, while the values of IB in the PL/I case are ten for the IF conditional and zero for the (null) ELSE conditional.

Using this eight step procedure, the value of IB can always be determined, and in most practical cases, the amount of time spent in determining this value will not be too large. The advantages of using this MDNF form include obtaining a standard form for conversion which is both natural (using only NOT, AND, and OR) and minimal with respect our measure. Since equivalent atomic propositions will result in the same truth table, and therefore in the same DNF according to our algorithm for conversion to DNF, equivalent non-atomic propositions with the same weights on the atoms will result in the same values for IB. Thus we have a standard, uniquely determined, natural measure for IB. This algorithm for conversion to MDNF will therefore be used in the calculations of QS and QE, whose measures we now consider.

## 6. CALCULATION OF Q

As mentioned in Section 2, the computation of Q can be decomposed into three parts. Of these three parts, the calculation of QS and QE are similar to the calculation of IB. In the case of a WHILE type statement, a check is made before the execution of the body of the loop; in the case of an UNTIL type, the check is made at the end of the loop. These checks are expressed in the form of Boolean functions and thus may be converted into MDNF. A count of OP(MDNF) of

the resulting form is made, and the resulting value is the value of QS or QE. The calculation of QC is problematic in that there is no explicit condition that can be converted to MDNF.

There are two basic approaches to a definition of QC. The first is to create a Boolean function that will execute in the same manner as the counting loop. An example of this type of conversion would be the change of the FORTRAN statement DO 10 I = 1, 10 to a structure of the form LOOP... UNTIL (I .LT. 1 .OR. I .GT. 10). There are two problems with this approach. The first is that this change is somewhat artificial in the creation of a new conditional where none was before. Secondly, we must know the exact manner of execution of the counting loop; in PL/I, the equivalent structure would be DO WHILE (I > = 1 & I < = 10);. If this knowledge is not available, this method breaks down. This third, and most problematic, is that in the conversion to a conditional we have changed an extremely simple DO statement into a less simple conditional. Conversion to MDNF would aid in reducing the complexity of the resulting conditional, but the combination of the other problems with this one would seem to indicate that there should be a better way to handle counting loops. The second approach is to accept the count loop as a separate paradigm, and to construct a distinct way of measuring QC. This approach will overcome the problems inherent in the first approach, and has as an additional advantage in that it will allow with minor modifications most of the other count type loops, such as REPEAT in PL/I and the various types of enumerative counts that appear in most modern languages. This is the approach that we use.

To construct this measure of QC, we begin by considering the standard type of count, which has three parameters: starting value, ending value, and increment. We define the simplest case of the standard count to be a count of positive integers, beginning with one, and with an increment of one. To this simplest case, we assign a QC value of one. A similar case is that of the appearance of simple variables for the three components, e.g. DO 10 I = J, K, L. A value of one is assigned to this case, and to the case of a simple enumeration such as "DO I = 1, 3, 5, 6;". The remaining cases are considered to be modifications of these two cases, and additional values are added for the modifications. For constants, the possible modifications are: starting value not equal to one, ending value not equal to k (increment) + start for some value of k, negative increments, and non-integer increments. Each of these modifications adds one to the value of QC. Thus a statement like "DO I = 10 TO 3.27 BY − 0.5;" would have a value of five, and would represent the worst case of a count loop that involved only constants.

The other possibility is the appearance of expressions in the three parameters. This possiblilty is handled by evaluating OP (expression) and adding that value to the resulting count, e.g. "DO I = 1 TO K + L − 3;" would have a value of three, one for the basic count loop and two for the operator count of the expression. This would include the REPEAT option in PL/I, as an expression like "DO I = 1, 3, REPEAT I ** 2;" would have a value of one for the enumeration plus one for the operator count in the expression.

One final possibility that should be considered is that of the endless loop with inside escape, such as the LEAVE statement in PL/I. Although not a count type of loop but a separate paradigm, the procedure of assigning a base value with additions will work here also. Accordingly, we assign the standard base value of one to an endless loop. The LEAVE option is assigned a value of the number of structures left, so that an endless loop which exits only from itself (and not from any other encompassing procedure) would have a value of two. Since the LEAVE option is generally encountered along with an IF statement, we include the value of IB to be a part of the loop, but still name it as IB. (This is due primarily to the inherent vagueness of present languages with regard to this structure. If ever a language is written that has a structure such as LOOP... LEAVE name WHEN (condition)... END LOOP, this artificiality will no longer be necessary.)

Now that it is possible to measure QS, QE, and QC, the possibility of combinations of these measures occurs. The solution to this problem is simply to remember that the value of Q is the sum of the three loop types, and thus to consider and calculate each separately, then sum to get the value of Q for the loop. With this discussion of the Q measure finished, the measure of logical effort is complete. We thus turn to an example to illustrate the total algorithm.

## 7. AN ALGORITHM

In order to demonstrate the general algorithm for calculation of EPT, we consider the program NEWRAP, written in FORTRAN WATFIV-S (Listing 1). As the name NEWRAP suggests, the

Listing 1. Program NEWRAP

```
00010 $ JOB        NOEXT
00020C
00030C
00040C
00050C      THIS PROGRAM WILL FIND ROOTS OF FUNCTIONS USING METHODS OF
00060C      BISECTION AND NEWTON'S METHOD. TWO ASSUMPTIONS ARE MADE:
00070C      FIRST, THAT THE ROOTS ARE ALL WITHIN THE INPUT INTERVAL,
00080C      AND SECOND, THAT ALL ROOTS ARE AT LEAST A UNIT APART.
00090C      A UNIT, BEGINNING FROM THE LEFT TO THE INTERVAL, IS FOUND
00100C      WHICH CONTAINS A ROOT, THEN BISECTION IS USED UNTIL AN
00110C      APPROXIMATE ROOT WITHIN .005 IS FOUND. USING THE MIDPOINT
00120C      OF THE BISECTION INTERVAL RESULT, NEWTON'S METHOD IS
00130C      THEN RUN UNTIL GUESSES ARE NO MORE THAN 1.E-6 APART.
00140C      APPROPRIATE TESTS ARE MADE TO DISTINGUISH BETWEEN EASILY
00150C      FOUND ROOTS AND OTHER ROOTS.
00160C
00170C
00180C
00190       REAL HBOUND, TESTL, TESTH, PANS, ANS
00200       READ(5,*) TESTL, HBOUND
00210C
00220C        CHECK FOR ROOTS ON LEFT BOUNDARY
00230       WHILE(F(TESTL) .EQ. 0. .AND. TESTL .LE. HBOUND)
00240          WRITE(6,*) 'EXACT ANSWER = ', TESTL
00250          TESTL = TESTL + 1.
00260       ENDWHILE
00270C
00280C        BEGIN SEARCHING UNITS. IF EXACT ROOT, IT MUST BE RIGHT SIDE.
00290       WHILE(TESTL .LT. HBOUND)
00300          TESTH = TESTL + 1.
00310          IF(F(TESTL) * F(TESTH) .LT. 0.) THEN
00320            CALL BISECT(TESTL, TESTH, PANS)
00330            CALL NEWTON(PANS,ANS)
00340            WRITE(6,*) 'APPROXIMATE ROOT = ', ANS
00350            TESTL = TESTH
00360          ELSEIF(F(TESTL) * F(TESTH) .EQ. 0.) THEN
00370            WRITE(6,*) 'EXACT ROOT = ', TESTH
00380            TESTL = TESTH + 1.
00390          ELSE
00400            TESTL = TESTH
00410          ENDIF
00420       ENDWHILE
00430       STOP
00440       END
00450       SUBROUTINE BISECT (LOW,HIGH,ANS)
00460C        THIS IS A STANDARD BISECTION ROUTINE.
00470C        LOW AND HIGH ARE THE INPUT LEFT AND RIGHT HAND END POINTS
00480C        OF THE INTERVAL, ANS IS THE RETURN VARIABLE, AND MIDPT IS
00490C        THE MIDPOINT OF THE SEARCH INTERVAL.
00500         REAL LOW, HIGH, ANS, MIDPT
00510         WHILE(ABS(LOW-HIGH) .GE. .005)
00520           MIDPT = (LOW + HIGH)/2.
00530           IF(F(LOW) * F(MIDPT) .LT. 0.) THEN
00540             HIGH = MIDPT
00550           ELSIF(F(HIGH) * F(MIDPT) .LT. 0.) THEN
00560             LOW = MIDPT
00570           ELSE
00580             WRITE(6,*) 'EXACT ANSWER = ', MIDPT
00590             LOW = MIDPT
00600             HIGH = MIDPT
00610           ENDIF
00620         ENDWHILE
00630         ANS = (LOW + HIGH)/2.
00640         RETURN
00650         END
00660       SUBROUTINE NEWTON(INPUT,OUTPUT)
00670C        STANDARD NEWTON-RAPHSON METHOD.
00680           REAL INPUT, OUTPUT
00690           INTEGER I, FLAG
00700           I = 0
00710           FLAG = 0
00720           WHILE(I .LE. 15 .AND. FLAG .EQ. 0)
00730             OUTPUT = INPUT - F(INPUT)/FPRIME(INPUT)
00740             IF(ABS(INPUT - OUTPUT) .LT. 1.E-6) FLAG = 1
00750             INPUT = OUTPUT
00760             I = I + 1
00770           ENDWHILE
00780           IF(I .GT. 15) THEN
00790           WRITE(6,*) 'TOO MANY ITERATIONS'
00800           ENDIF
```

```
00810          RETURN
00820          END
00830        FUNCTION F(VALUE)
00840C         THE FUNCTION FOR WHICH THE ROOTS ARE FOUND
00850          REAL VALUE
00860          F = (((VALUE - 9.) * VALUE - 2.) * VALUE + 120.)
00870     *     * VALUE - 130
00880        RETURN
00890        END
00900        FUNCTION FPRIME(VALUE)
00910        DERIVATIVE OF SAID FUNCTION
00920          REAL VALUE
00930          FPRIME = ((4. * VALUE - 27.) * VALUE - 4.) * VALUE + 120.
00940          RETURN
00950          END
00960        $ENTRY
```

Listing 2. Pseudo-code for calculation of EPT and EPM

```
begin calc_EPT
        EPT: = 0
        EPM: = 0
          repeat until (end_of_program)
                call calc_ESGT (segment, ESGT)
                EPM: = max (EPM, ESGT)
                EPT: = EPT + ESGT
           end repeat
end calc_EPT

begin calc_ESGT (segment, ESGT)
        LN: = 0
        ESGT: = 0
        repeat until (end_of_segment)
                get statement
                LN: = LN + 1
                if (statement .NE. decision .AND. statement .NE. loop) then
                        call count OP (statement, OP)
                        ESGT: = ESGT + OP
                elseif (statement = decision) then
                        I: = 0
                        EDCT: = 0
                        repeat until (end_of_decision)
                                call calc_IB (statement, IB)
                                call calc_ESDT (decision_subsegment, ESQTD)
                                EDCT: = ESGTD + EDCT
                        end repeat
                        EDCT: = EDCT + I
                        ESGT: = ESGT + EDCT
                elseif (statement = decision) then
                        call calc_QS (statement, QS)
                        call calc_QE (statement, QE)
                        call calc_QC (statement, QC)
                        Q: = QS + QE + QC
                        call calc_ESGT (loop_segment, ELPT)
                        ELPT: = ELPT + Q
                        ESGT: = ESGT + ELPT
                endif
        end repeat
        ESGT: = ESGT + LN
end calc_ESGT

begin count_OP (statement, OP)
        /*this is language dependent*/
end count_OP

begin calc_IB (statement, IB)
        /*see paper*/
end calc_IB

begin calc_QS (statement, QS)
        /*see paper*/
end calc_QS

begin calc_QE (statement, QE)
        /*see paper*/
end calc_QE

begin calc_QC (statement, QC)
        /*see paper*/
end calc_QC
```

program is designed to calculate the roots of the polynomial in function F, using the method of bisection to find initial approximations to the roots, then using the Newton-Raphson method to obtain good approximations. NEWRAP consists of five segments: MAIN, the driver and initial root isolator (unnamed in this version of FORTRAN), BISECT, the bisection routine, NEWTON, the Newton-Raphson routine, and F and FPRIME, routine to calculate the needed functional values.

The pseudo-code for our algorithm is given in Listing 2. It consists of seven routines, the last five of which are not given in full. Of these last five, four are discussed in the preceding sections and involve calculation of MDNF; the other procedure, count_OP, is straightforward, being essentially a table of the various operators in the language, along with a recognizer to determine when a subprocedure is called. The main portion of the algorithm is the listing of the evaluation procedure for EPT and EPM; it is this portion that will be used in the calculation of logical effort for NEWRAP.

To begin our calculation, we note that the occurrence of procedure count_OP in procedure calc_ESGT occurs unmodified, so that we can count the operators in each line which is not used in the calculation of either I or G. Also, the procedures given earlier for calculation of I and Q allow us to count the values of IB, QS, QE, and QC in each line. The result of these counts occur in Table 3. Note that declaration statements have a count of zero, while input–output statements

Table 3. Statement efforts for NEWRAP

| Line | Value | Type | Line | Value | Type |
|---|---|---|---|---|---|
| 10 | 0 | JCL | 490 | 0 | COM |
| 20 | 0 | COM | 500 | 0 | ST |
| 30 | 0 | COM | 510 | 3 | QS |
| 40 | 0 | COM | 520 | 3 | ST |
| 50 | 0 | COM | 530 | 4 | IB |
| 60 | 0 | COM | 540 | 1 | ST |
| 70 | 0 | COM | 550 | 4 | IB |
| 80 | 0 | COM | 560 | 1 | ST |
| 90 | 0 | COM | 570 | 0 | IB |
| 100 | 0 | COM | 580 | 1 | ST |
| 110 | 0 | COM | 590 | 1 | ST |
| 120 | 0 | COM | 600 | 1 | ST |
| 130 | 0 | COM | 610 | 0 | CMP |
| 140 | 0 | COM | 620 | 0 | CMP |
| 150 | 0 | COM | 630 | 3 | ST |
| 160 | 0 | COM | 640 | 1 | ST |
| 170 | 0 | COM | 650 | 0 | CMP |
| 180 | 0 | COM | 660 | 0 | CMP |
| 190 | 0 | ST | 670 | 0 | COM |
| 200 | 1 | ST | 680 | 0 | ST |
| 210 | 0 | COM | 690 | 0 | ST |
| 220 | 0 | COM | 700 | 1 | ST |
| 230 | 4 | QS | 710 | 1 | ST |
| 240 | 1 | ST | 720 | 3 | QS |
| 250 | 2 | ST | 730 | 5 | ST |
| 260 | 0 | CMP | 740 | 3+1 | I,ST |
| 270 | 0 | COM | 750 | 1 | ST |
| 280 | 0 | COM | 760 | 2 | ST |
| 290 | 1 | QS | 770 | 0 | CMP |
| 300 | 2 | ST | 780 | 1 | IB |
| 310 | 4 | IB | 790 | 1 | ST |
| 320 | 1 | ST | 800 | 0 | CMP |
| 330 | 1 | ST | 810 | 1 | ST |
| 340 | 1 | ST | 820 | 0 | CMP |
| 350 | 1 | ST | 830 | 0 | CMP |
| 360 | 4 | IB | 840 | 0 | COM |
| 370 | 1 | ST | 850 | 0 | ST |
| 380 | 2 | ST | 860 | 8 | ST |
| 390 | 0 | IB | 870 | — | CNTN |
| 400 | 1 | ST | 880 | 1 | ST |
| 410 | 0 | CMP | 890 | 0 | CMP |
| 420 | 0 | CMP | 900 | 0 | CMP |
| 430 | 1 | ST | 910 | 0 | COM |
| 440 | 0 | CMP | 920 | 0 | ST |
| 450 | 0 | CMP | 930 | 7 | ST |
| 460 | 0 | COM | 940 | 1 | ST |
| 470 | 0 | COM | 950 | 0 | CMP |
| 480 | 0 | COM | 960 | 0 | JCL |

have a count of one. In general, any compiler directive such as END or FORMAT will have a count of zero, as they have no effect on the structural complexity of the program.

Of the values for Q and IB in the tables, the calculations for most are trivial. The exceptions are lines 230, 310, 510, and 720; we examine these calculations more closely.

Line 230: The Boolean operator is "F(TESTL) .EQ. 0 .AND. TESTL .LE. HBOUND". Let a = "F(TESTL) .EQ. 0" and b = "TESTL .LE. HBOUND".

The general form is thus a & b, which has truth table

| a | & | b |
|---|---|---|
| T | T | T |
| T | F | F |
| F | F | T |
| F | F | F. |

The MDNF is just the proposition itself. Proposition a has an operator count of two, one for the function F and one for the operator .EQ.. Proposition b has a count of one, for the operator .LE.. Hence QS(230) = 4.

Line 310: The count here is four, one for each occurrence of function F, one for the operator "*", and one for the operator .LT..

Line 510: The count here is three, one for each of the operators ABS, "−", and .GE..

Line 720: This is similar to line 230, with a count of 3.

We now turn to a calculation of ESGT(MAIN). MAIN has the form of a sequence of five blocks: line 190, line 200, lines 230–260, lines 290–420, and line 430. Thus LN(MAIN) = 5, as statement 440 is a compiler directive and has no length attribute. We have given the declaration statement a length, as we feel that our measure should reflect programming difficulties, and it is certainly possible to encounter problems with declarations, while an experienced programmer will generally encounter no problems with the other compiler directives. Our calculation is thus

$$\begin{aligned}\text{ESGT(MAIN)} = {} & \text{LN(MAIN)} + \text{ESTT(190)} + \text{ESTT(200)} \\ & + \text{ELPT(230:260)} + \text{ELPT(290:420)} \\ & + \text{ESTT(430)}.\end{aligned}$$

We have LN(MAIN) = 5, and the values for the three statement efforts are given in the statement effort table. The only remaining values are those for the loops, which require more calculation.

For the first loop, we have

$$\begin{aligned}\text{ELPT(230:260)} &= \text{Q(230:260)} + \text{ESQT(240:250)} \\ &= \text{QS(230)} + (\text{LN(240:250)} + \text{ESTT(240)} \\ &\quad + \text{ESTT(250)}) \\ &= 4 + (2 + 1 + 2) \\ &= 9\end{aligned}$$

where the values are taken from the table. The length is two, since both statements are executable.

For the second loop,

$$\begin{aligned}\text{ELPT(290:420)} &= \text{Q(290:420)} + \text{ESGT(300:410)} \\ &= \text{Q(290)} + (\text{LN(300:410)} + \text{ESTT(300)} \\ &\quad + \text{EDCT(310:410)}.\end{aligned}$$

From the table, QS(290) = 1 and ESTT(300) = 2.We have LN(300:410), since the structure is a

Table 4. Summary of values for EPT(NEWRAP)

| |
|---|
| ESGT(MAIN) = 44 |
| LN(MAIN) = 5 |
| ELPT(230:260) = 9 |
| ELPT(290:420) = 28 |
| EDCT(310:410) = 23 |
| ESGT(BISECT) = 34 |
| LN(BISECT) = 4 |
| ELPT(510:620) = 26 |
| EDCT(530:610) = 18 |
| ESGT(NEWTON) = 33 |
| LN(NEWTON) = 7 |
| ELPT(720:770) = 20 |
| EDCT(740) = 5 |
| EDCT(780:800) = 3 |
| ESGT(F) = 12 |
| LN(F) = 3 |
| ESGT(FPRIME) = 11 |
| LN(FPRIME) = 3 |

sequence of two blocks. For the calculation of EDCT(310:410), we obtain

$$\begin{aligned}\text{EDCT}(310{:}410) &= \text{I}(310{:}410) + \text{ESGT}(320{:}350) \\ &\quad + \text{ESGT}(370{:}380) + \text{ESTT}(400) \\ &= (\text{IB}(310) + \text{IB}(360) + \text{IB}(390)) \\ &\quad + (\text{LN}(320{:}350) + \text{ESTT}(320) + \text{ESTT}(330) \\ &\quad + \text{ESTT}(340) + \text{ESTT}(350)) \\ &\quad + (\text{LN}(370{:}380) + \text{ESTT}(370) + \text{ESTT}(380)) \\ &\quad + (\text{LN}(400) + \text{ESTT}(400)).\end{aligned}$$

Each of these values may be found in the table (all of the LN values are just the statement counts, as each statement is executable), with the result that

$$\begin{aligned}\text{EDCT}(310{:}410) &= (4+4+0) + (4+1+1+1+1) \\ &\quad + (2+1+1) + (1+1) \\ &= 23.\end{aligned}$$

As a result, ELPT(290:420) = 1 + 2 + 2 + 23 = 28, and the final tabulation for ESGT(MAIN) is thus 5 + 0 + 1 + 9 + 28 + 1 = 44.

The segment efforts for the other four segments are computed similarly, with a summary of results given in Table 4. Hence EPT(NEWRAP) = 44 + 34 + 33 + 12 + 11 = 134 and EPM(NEWRAP) = ESDT(MAIN) = 44.

As a comparison, we calculate McCabe's cyclomatic complexity for the program NEWRAP. The graph of the problem is given in Fig. 4, and a count gives the resulting complexity value of 17. Logical effort and McCabe's cyclomatic complexity [2] are at least partially independent. It is clear by inspecting Fig. 4, the graph of the program, that statements in a sequence structure or substructure could be added or deleted, which would leave the cyclomatic complexity fixed, but could possibly change the logical effort of the structure. Thus a major difference between our measure and cyclomatic complexity is the additional consideration of the sequence structure.

## 8. CONCLUDING REMARKS

Now that logical effort has been defined, the question of its usefulness arises. Given the other, more established, measures of complexity, what is the purpose of a new measure? We feel that the advantages of logical effort are three in number: First, logical effort reflects the current concerns in programming languages, in the use of virtuality and segment independence. These concepts appear to be unifying a study of programming languages, and a measure of complexity that reflects these concepts should be a useful aid in the continuation of this study. Second, logical effort is a

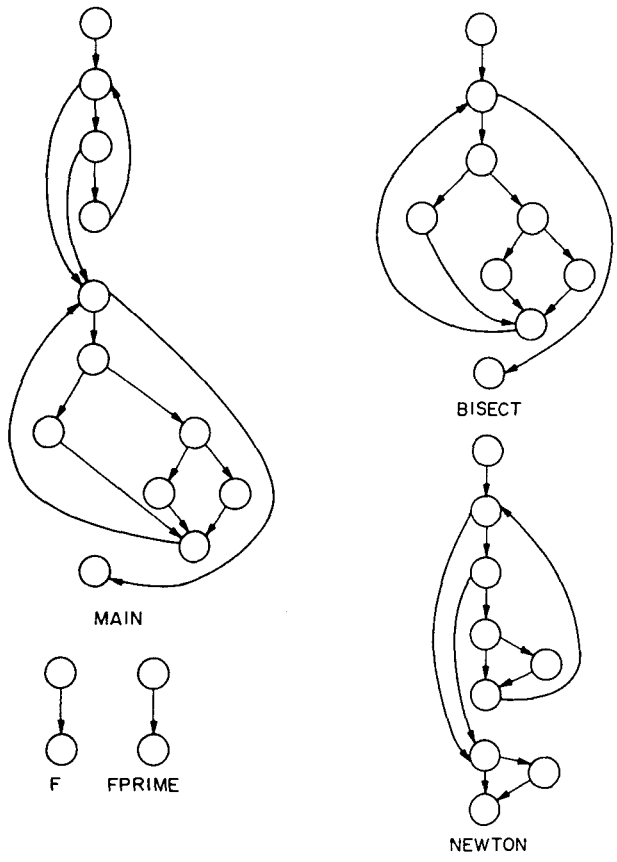


Fig. 4. Complexity graphs for NEWRAP.

quantification of the control structures in a program, and could be very easily modified to calculate complexity from an algorithm instead of a program, if one had knowledge of the intended implementation language. (Such knowledge is essential, since computation of logical effort depends on a knowledge of the available primitive operators.) As a result, logical effort has the ability to predict the complexity of a program before the program has been written. Predictive measures of this type are important in the allocation of resources for program implementation, and it appears that logical effort is able to aid in determinations of this type. Third, logical effort is a detailed measure of the control structures in a program, and, when combined with an analysis of the data structures used in either a program or algorithm, should allow an essentially complete quantitative description of programs and algorithms.

The current research is therefore proceeding in two basic directions. The first is a conversion of logical effort so that it will work for algorithms, in order to measure the predictive ability of such a measure. The other, and perhaps more ambitious research effort, is to integrate a data structure complexity measure with logical effort in order to give a more complete view of the general complexity problem. The final result would then be an attempt to utilize this new measure to predict programming effort, based on a given algorithm and a description of the data structures used in the algorithm. Such a measure would be of great aid in the allocation of resources for software production.

*Acknowledgements*—We wish to thank the referees and editor of the journal for several helpful criticisms of an earlier draft of this paper.

## REFERENCES

1. Halstead M. H., *Elements of Software Science.* Elsevier/North-Holland, Amsterdam (1977).
2. McCabe T., A complexity measure. *IEEE Trans. Software Engng* **SE-2,** 105–113 (1977).
3. Woodward M., Hennell M. and Headley D., A measure of control flow complexity in program test. *IEEE Trans. Software Engng* **SE-5,** 45–50 (1979).
4. Fitzimmons A. and Love T., A review and evaluation of software science. *ACM Comput. Surv.* **10,** 3–18 (1978).

5. Baker A. L., Software science and program complexity. Ph.D. Dissertation, Department of Computer and Information Science, Ohio State University (July 1979).
6. Schneider G. M., Sedlmeyer R. L. and Kearney J., On the complexity of measuring software complexity. *Proceedings of the National Conference (NCC)*, pp. 317–322 (1981).
7. Atwood M. A. and Ramsey H. R., Cognitive structures in the comprehension and memory of computer programs: an investigation of computer program debugging. ARI Technical Report TR-78-A21 (August 1978).
8. Hennell M. A., Woodward M. R. and Headley D., On program analysis. *Information Processing Lett.* **5**, 136–140 (1976).
9. Dunsmore H. and Gannon J., Programming factors—language features that help explain programming complexity. In *Proceedings of the ACM 1978 Annual Conference*, Washington (1978).
10. Curtis W., Sheppard S., Milliman P. M., Borst M. A. and Love T., Measuring the psychological complexity of software maintenance tasks with Halstead and McCabe metrics. *IEEE Trans. Software Engng* **SE-5**, 95–104 (1979).
11. McClure C. L., A model for program complexity analysis. *Proceedings of the 3rd Conference on Software Engineering*, pp. 149–157 (1978).
12. Love T., An experimental investigation of the effect of program structure on program understanding. *SIGPLAN* **12**, 105–113 (1977).
13. Chapin N., A measure of software complexity. *Proceedings of the 1979 National Computer Conference*, New York, pp. 995–1002 (1979).
14. Parameswaran N. and Iyengar S. S., A measure of logical complexity as related to program complexity. Submitted for publication.
15. Iyengar S. S., Parameswaran N. and Fuller J., A measure of logical complexity of programs. *Comput. Lang.* **7**, 147–160 (1982).
16. Cater S. C., Iyengar S. S. and Fuller J., Algorithms for the computation of logical effort in a program. Technical Report 82-023, Department of Computer Science, Louisiana State University (1982).
17. Ledgard H. and Marcotty M., A genealogy of control structures. *Commun. ACM* **8**, 626–639 (1975).
18. Gordon R., Measuring improvements in program clarity. *IEEE Trans. Software Engng* **SE-5**, 79–90 (1979).
19. Shneiderman B., Control flow and data structure documentation: two experiments. *Commun. ACM* **25**, 55–63 (1982).
20. Knuth D. E., *The Art of Computer Programming*, Vol. 1. Addison-Wesley, Reading, Massachusetts (1973).
21. Tannenbaum A. S., *Structured Computer Organization*. Prentice-Hall, Englewood Cliffs, New Jersey (1976).
22. Madnick S. E. and Donovan J. J., *Operating Systems*. McGraw-Hill, New York (1974).
23. Hamilton A. G., *Logic for Mathematicians*. Cambridge University Press (1978).
24. Prather R. E., *Introduction to Switching Theory: A Mathematical Approach*. Allyn & Bacon, New York (1967).
25. Parameswaran N., Iyengar S. S. and Ramesh S., A complexity measure that takes into account the complete program semantics. Technical Report 83-022, Department of Computer Science, Louisiana State University (1983).
26. Parameswaran N. and Iyengar S. S., Logical complexity of programs. *Proceedings of the IEEE–SMC Conference*, Bombay (December 1983).
27. Iyengar S. S. and Bruno G., Formalization of logical complexity of programs. Technical Report 83-013, Department of Computer Science, Louisiana State University (1983).
28. Iyengar S. S., Fuller J., Parameswaran N. and Ambhardhar S., A comparison of logical complexity with Halstead and McCabe measures. *Kybernates*. To appear.

**About the Author**—STEVEN C. CATER received the M.S. degree in mathematics from Louisiana State University at Baton Rouge in 1981. Since then he has served as an instructor in the Computer Science Department at LSU, where he is now also a student in the Ph.D. program. His current research interests are in the areas of automata, formal language theory and mathematical structure simulation.

**About the Author**—Professor S. S. IYENGAR is a faculty member in the Department of Computer Science at Louisiana State University. His research interests are in analysis of algorithms, data structure complexity in programs and intelligent information systems. Professor Iyengar has published more than 40 papers in *IEEE-SMC, Computer Networks, IEEE-PAMI, CACM, JCIS, Simulation, Applied Mathematics and Computation, Journal of Information Sciences*, etc. He is also a reviewer to *JACM, CACM, Operation Research Journal*, etc.

**About the Author**—JOHN FULLER is a graduate student in the Department of Computer Science and is working on his thesis entitled "Complexity of Programs" with Professor Iyengar.