Program complexity using hierarchical abstract computers*

by WILLIAM G. BAIL

University of Maryland
College Park, Maryland

and

Automation Industries—Vitro Laboratories Division
Silver Spring, Maryland

and

MARVIN V. ZELKOWITZ

University of Maryland
College Park, Maryland

and

National Bureau of Standards
Washington, DC

INTRODUCTION

There is currently interest in measuring the complexity of a computer program with the evaluation of a program's control structure via its flowgraph representation.1,2 However, the inclusion of the effects of data on these measures is often lacking. This paper proposes a new measure of program complexity that attempts to identify regions of locality by combining a control structure approach with information about data usage to get a valid measure of overall program complexity. The need for objective measures of program structure is important if the area of programming language design and implementation is to take on a more formal basis from the ad hoc techniques currently used.

This work is related to that of Hellerman,3 Savage,4,5 and Chaitin6 in that the complexity of an algorithm is defined as the number of bits in the algorithm's representation, i.e., the number of bits in a program implementing the algorithm. A program implementing a given algorithm is considered to be better than another program implementing the same algorithm if it has fewer bits in its representation. In order to provide a standardized representation for algorithms, the concept of a Hierarchical Abstract Computer is defined (HAC). Based on a primitive-level HAC, hierarchical levels of HAC modules are used to specify the structures and sequences of a specific algorithm. The modules are chosen according to the concept of a Prime Program Parse,7 which defines a particular, unique sequence of subroutine decompositions from an original flowgraph representation of an algorithm. The complexity of the total program is defined to be the sum of complexities of each of the modules used to form the hierarchy.

The difficulty of writing program code to implement an algorithm has been related to the amount of information that must be manipulated in the process of writing the program. There have been many attempts recently to simplify the process, most conspicuously aimed at the control structure complexities. Structured programming has been expounded as a method by which programs can be written with minimum difficulty, and mainly involves restraints on the use of control structures. There are also some methods which are directed at data structure complexities, such as the concepts of data abstraction and information hiding. These methods are concerned with the simplification of the data that is manipulated by the program code. By reducing the data available for referencing to only that which is directly needed by a section of code, the possibility for confusion is reduced. As much as possible, references to data are maintained in small sections of code, and are kept exclusive, in that only those sections are allowed to act on the data. In this manner, local data responsibilities are easy to isolate and to maintain. These local responsibilities are known as clusters, and their exclusiveness and privacy is known as information hiding.

It is assumed that a program is well-structured if clusters of data and control activity are small. The resultant small subroutines are thus accompanied by correspondingly small data spaces. The purpose of this research is to be able to measure different implementations of the same algorithm so as to determine the particular implementation which is best in the sense of being well-structured or least complex. The next section discusses some relationships between an algo-
ALGORITHMS AND PROGRAMS

An algorithm has been defined to be a finite set of rules which specify a sequence of operations used to solve some specific problem. The operations specified by the rules are selected from a set $\Omega$ of operators used by the algorithm. The proper sequencing of these operations depends on the state of the data items used by the algorithm, and is described with the use of a meta-algorithm or program which directly implements the algorithm. The essential difference between an algorithm and its implementing program is the specification of the operation sequencing through the use of control statements. Each algorithm has an infinite number of programs which implement it, some of which are considered to be well-structured, or less complex. It is this class of programs which a complexity measure attempts to identify.

Algorithms are rarely completely specified, in that assumptions are made about the characteristics of the data types and the operators which manipulate the data. Yet each operator is itself described by an algorithm at a lower level, and so on, down to some indivisible, lowest level set of operators and data types. Computer programs are written in high-level languages whose operators are implemented in machine languages, whose operators are in turn implemented in micro-code, whose operators are implemented in hardware gates, etc. The next section proposes a primitive set of operators and data types to be used as a basis upon which complete specifications of algorithms can be hierarchically built.

Rarely, however, does the operator set of the algorithm match the operator set of the programming language. A central problem in programming is identifying and building the inherent operators and data types that make up the algorithm, using the available, lower-level tools. Usually, the algorithm description given to the programmer does not adequately recognize the structures within the algorithm specified, and so, severely complicates the problem facing the programmer. His task is to perform certain transformations to the algorithm specification without altering its basic structure so as to implement the request as simply as possible.

Let $\Delta$ be the set of data items used by an algorithm or program, and let $D$ be a subset of $\Delta$. The Execution Sequence of an algorithm relative to $D$ is defined to be the sequence of states assumed by the members of $D$.

Two programs are execution equivalent relative to $D$ if they have the same execution sequence for all inputs. Different programs can be execution equivalent because the occurrence of control operations is ignored, having no effect on the execution sequence.

Two programs are functionally equivalent if, given the same input values, they produce the same output values.

Two programs are semi-execution equivalent relative to $D$ if, for each $d \in D$, the programs are execution equivalent relative to $d$. That is, each data item of interest takes on the same set of values, but not necessarily all in the same relative order as the other data items.

It is a trivial result that if two programs are semi-execution equivalent, then they are also functionally equivalent. Functional equivalence is the weakest restriction, in that there need be no similarity in the two algorithms used by the programs. Execution equivalence is the strongest restriction, requiring two programs to have identical data transformations. The restriction of semi-execution equivalence is felt to be a reasonable criteria for comparing different programs, in that the implemented algorithms are not only functionally equivalent, but also structurally similar, in that all of their data items are undergoing similar transformations. There is a certain latitude available to the programmer in this situation to strive towards a simple implementation.

HIERARCHICAL ABSTRACT COMPUTERS

The concept of a set of programs interacting at different levels, and working together to implement an algorithm, leads naturally to the following definition of a Hierarchical Abstract Computer (HAC).

Def: A Hierarchical Abstract Computer (HAC) is a quadruple $(\Delta, I, \Omega, T)$ where $\Delta$ is a set of distinct named data items, each of which is associated with a member of $T$, the set of legal data types, $I$ is a set of named instructions, and $\Omega$ is a finite set of operation codes. Each instruction is of the form:

$$\omega d_1 d_2 \ldots d_n x_1 x_2 \ldots x_m$$

where $\omega \in \Omega$, $d_i \in \Delta$ and $x_i$ is the name of an instruction in $I$, or $x_i=0$. The $d_i$'s are the data arguments for the operation $\omega$, and the $x_i$'s represent the possible alternatives for the next instruction to be executed. The data items and the instruction addresses (names) are numbered beginning with 0, and a transfer to address 0 means to halt execution for that HAC.

Operation codes are either composite or primitive. A composite code is itself implemented by a lower-level HAC, carrying out some specific algorithm on the input data. A primitive operation code is, in some sense, indivisible, and is defined to be one of the three operators:

- TEST $d$ $x_1$ $x_2$
- SET $d$ $x_1$
- CLEAR $d$ $x_1$

where $d \in \Delta$, and is associated with the data type BIT, having the value 0 or 1, and $x_1, x_2 \in I$. These instructions have the semantics:

- TEST If the contents of $d=0$, then the next instruction is to be found at address $x_1$; otherwise, the next instruction is at $x_2$.
- SET Set the data item $d$ equal to 1.
- CLEAR Set the data item $d$ equal to 0.

These three codes form the Primitive Basis, $\Omega_p$, and, if
a HAC consists of $\Omega_\text{P}$ together with the data type $T=\{\text{BIT}\}$, then the HAC is said to be a Primitive HAC, or PHAC. An important characteristic of $\Omega_\text{P}$ is that it is complete, that is, it can compute all finite functions $f: \{0,1\}^n \rightarrow \{0,1\}$. This result follows directly from the PHAC implementation of the basic Boolean functions AND, OR, and NOT.

As described, a HAC program can be decomposed into lower and lower level modules until only Primitive HACs are obtained, thus forming a tree of modules. Conversely, given a PHAC implementation of an algorithm, sections of code can be removed and replaced by a single invocation to a new composite operator, with as many parameters as there were global variables across the code segment.

While a HAC is only as powerful as a finite state automata, it is still sufficiently powerful to model real computer programs that do not use auxiliary storage, and, unlike finite state models, can easily be measured with a complexity measure. This model can, however, be easily extended to include tapes. As in real hardware, specific addresses for data, control, and status registers can be defined resulting in a memory mapped I/O. Placing information into these registers causes information to be read or written from a tape and to appear in the data register. This storage may actually be infinite, but accessible only via a finite set of specific addresses.

**PROGRAM COMPLEXITY**

Given a HAC implementation of an algorithm, it is now necessary to define a complexity measure on this set of modules. The complexity of a HAC $H$ is defined to be the sum of the complexities of each of the instructions in the HAC:

$$C(H) = \sum_{j} C(j)$$

Let $L_\Delta = \log_2 \Delta$, $\Delta =$ number of bits needed to address the data space,

$L_1 = \log_2 |I| + 1$, $I =$ number of bits needed to address the instruction space $I$ (with the addition of address zero to be used as a STOP instruction), and

$L_\Omega = \log_2 \Omega$, $\Omega =$ number of bits needed to specify the operation code.

Then, the complexity of an instruction $i$ with $m$ data arguments and $n$ addresses is:

$$C(i) = L_\Delta + mL_\Delta + nL_\Omega$$

For PHAC programs, the complexity of each TEST instruction is $(1.58 + L_\Delta + 2L_1)$ and the complexity of each SET and CLEAR instruction is $(1.58 + L_\Delta + L_1)$.

Thus, the complexity of a HAC implementation of an algorithm is defined to be the sum of the complexities of each of the HAC modules which form the implementation. For each program, however, there are many possible modularizations, any one of which could be chosen by the programmer, but which may not reflect the structure of the program itself. It is desirable to have a canonical modularization for a program to avoid this problem. This representation should be unique for a given program, and should correlate closely with the structural properties of the program. The Prime Program Parse meets these criteria.

A proper program is a flowgraph with a single input arc, a single output arc, and with each node in the flowgraph being on a path from the input arc to the output arc. A prime program is recursively defined as a proper program with no proper prime subprograms of greater than one node. The process of identifying the hierarchy of prime programs which make up a compound program is known as a prime program parse. Such a parse is unique up to associativity of sequences of statements.

**PROPERTIES OF MEASURE**

This measure has been used to measure the complexity of several programs implementing common algorithms. In one example, an 8-bit adder was programmed on a primitive HAC, using 78 instructions, and resulting in a complexity of 918.26. The prime program parse of the program yielded 8 subroutines, all invoked by a single controlling routine. This program was restructured by forming three composite operators, resulting in a complexity of 471.76. This savings in

![Figure 1—Unstructured version of program](From the collection of the Computer History Museum (www.computerhistory.org))
complexity was due primarily to the recognition that 6 of the 8 modules in the prime program parse performed the same function, and thus represented a large degree of redundancy. The two programs are execution equivalent.

As an example of the comparison of two programs which are semi-execution equivalent, consider those shown in Figures 1 and 2. Figure 1 illustrates a nonstructured program which has a complexity of 53.16, as determined from the prime-program parse. Figure 2 illustrates a structured program, having a complexity of 52.02, and consisting entirely of small (size 1 or 2) prime programs.

The second example illustrates a basic result, namely that for HAC programs which consist of two prime programs with all variables being global, the minimum complexity occurs when the number of instructions is equally divided between the two subprograms. This result is obtained directly from a minimization of the expression for the complexity of such a program. Repeated applications of this result yield the conclusion that the minimal complexity for a program occurs when only a structured basis for the control is used, i.e., only the control structures sequence, if-then-else, do-while, repeat-until, do-while-do, if-then, and function.

This proposed measure has a demonstrated sensitivity to both control and data structuring within programs, and, because of its close relationship to the structured programming control graphs and prime programs, is felt to be a valuable tool in the quantification of overall complexity in the effort to formalize the intuitive concept of a good program.

REFERENCES