

Information-theoretic Complexity of Program Specifications

N. S. COULTER, R. B. COOPER AND M. K. SOLOMON

Department of Computer and Information Systems, Florida Atlantic University, Boca Raton, FL 33431, USA

Various models and metrics that are based on cognitive theories and assumptions have been proposed in an effort to formalise the concept of programming difficulty. The model and metric proposed here are, instead, based on a problem-space formalisation and an information-theoretic measure (entropy) that provides the complexity of a program specification; this metric can be applied before the specification's implementation is complete. The problem space for a program specification is the set of possible state descriptions that could arise from the execution of the program on that input, as ascertainable from the specification. We show how to define the entropy metric based on the problem space definition, and demonstrate the applicability of this metric to modularity problems.

Received January 1986

1. INTRODUCTION

Models of aspects of the computer programming process have been proposed by many researchers. Some studies deal primarily with the cognitive processes involved in programming,^{7, 16, 18–20, 27–30} while others discuss metrics that model the programming process.^{2, 5, 11, 14, 15, 18, 21, 34} The former studies probe the underlying processes of program comprehension, such as the acquisition of programming skills, and they often incorporate theories from cognitive psychology that involve different levels of human memory.^{10, 22, 32} The latter studies deal mostly with similar topics but result in conclusions expressed as mathematical functions instead of qualitative and statistical appraisals. Still other researchers have explored programming style in an effort to discover ways to produce reliable and comprehensible programs.^{12, 13, 17, 35} That work requires the assumption that the mental processing or 'cognitive load'⁸ involved in understanding how a program should function is a predictor of how difficult the program would be to implement.

Inconsistencies are apparent in many studies that involve software metrics. The metrics proposed often do not address the programming process in a unified fashion from the initial problem space to its final representation. In most cases the metrics apply only to completed programs, so that little predictive value exists. Some of these theories rely heavily on other theories from cognitive psychology, which are constantly evolving and may not be relevant in computer science because the experimental conditions do not match the underlying psychological assumptions.⁹ Finally, the attempts to estimate the cognitive load that can be handled in a single module have been relatively crude. Most estimates of that limit also refer to human memory models and are difficult to evaluate in the light of ongoing psychological research.

2. A NEW APPROACH

The approach we take in this paper is to analyse the (presumably formalised²⁴) program specification, not the programmer, using the elementary information theory developed by Shannon,²⁵ reprinted in Ref. 26. Our approach is also motivated by Bar-Hillel.⁶ We will base a metric on the amount of 'information' present in a program specification, as formally represented in a problem space. Shannon has given a measure of in-

formation that satisfies certain 'reasonable' postulates. We will examine this idea and propose some guidelines for applying it to the determination of the complexity of various specifications.

Consider an experiment with a finite number n of distinct possible outcomes, and assume the probabilities associated with those outcomes are p_1, p_2, \dots, p_n . Before the experiment is performed there is an 'uncertainty' as to which one of the n possible outcomes will occur; this can be interpreted as the amount of *information* that will be obtained by performing the experiment and observing the outcome. As part of an investigation of the coding and transmission of information in the presence of noise, Shannon demonstrated that there is a unique mathematical function that satisfies certain postulates that describe the abstract concept of 'information'. He showed that there is one and only one way to assign a quantitative value to measure information, provided only that the information measure satisfy the following two (loosely stated) postulates: (i) if the outcomes are equally likely (i.e. $p_i = 1/n$, for $i = 1, 2, \dots, n$) then the information measure is a strictly increasing function of the number of possible outcomes (i.e. all things being equal, the more possibilities an experiment admits, the more information it provides); and (ii) the amount of information provided by an 'answer' is independent of the way in which the experiment found the answer. (Shannon's statement of these postulates is given on page 19 of Ref. 26. (i) is Shannon's postulate (2); (ii) is postulate (3). Shannon's postulate (1) is merely technical, a continuity requirement.)

Let $H_n(p_1, \dots, p_n)$ denote the information measure. Shannon proved that the only function that satisfies these postulates is

$$H_n(p_1, \dots, p_n) = -K \sum_{i=1}^n p_i \log p_i, \quad (1)$$

where K is an arbitrary positive constant. (The mathematical properties of Shannon's information measure are given in many works, such as Ref 2.) It is most convenient to take the base of the logarithm as 2 and to take K as unity. Then the unit of H is called the *bit*. As noted by Shannon, the function H is the same as the well-known *entropy* function of statistical mechanics. Shannon's work is widely known and has been applied in various disciplines. It has been mentioned that Shannon's information measure can be used to calculate

the information content of problem statements and resulting computer programs (Refs 23, 31 p. 182), but this assertion has not been supported by detailed argument or evidence. In this paper we will sharpen the argument for applying Shannon's measure to quantify the abstract concept of information content of a program specification.

A program can be viewed as a transformation that, based on a specification, assigns or changes the values of entities occupying memory locations. (Effectively, a program is the end result of a sequence of refined specifications.) We can take the view that a program execution is an experiment that selects, from all possible ways that a program specification allows, the entities to occupy the memory locations (i.e. a *problem space*, PS), the single correct configuration (point) for that execution. In the case where all n points in the PS have the same *a priori* probability (the least presumptive assumption), the expression (1) for H reduces to

$$H = \log n. \quad (2)$$

(In the description to follow, n will be the cardinality $|S|$ of a finite set S .) Thus, for example, if there are x distinct memory locations and a program specification contains as its only requirement that each location hold exactly one of y distinct strings, there are $n = y^x$ distinct possible configurations in memory that satisfy the program specification. The purpose of executing the program is to find the single configuration that is the 'answer'. If we assume that all of the y^x possible outcomes are *a priori* equally likely, then the information provided by executing a program that chooses the correct configuration is, according to (2), $\log y^x = x \log y$ bits of information.

More generally, we associate with a program specification a finite set of memory locations X , a finite set of location values Y , a problem space PS , and an information measure $H(PS)$. PS is the set of state descriptions (SD s) permitted by the program specification, where an SD is a binary predicate over X , Y . An SD specifies, for each location x and value y , whether or not x takes the value y . Consistent with equation (2), we define $H(PS) = \log |PS|$. That is, we take as our measure of a specification's complexity the amount of information it provides when, through the execution of a resulting program, it chooses the unique correct answer from among the $|PS|$ *a priori* equally likely answers allowed by the program specification. Because $|PS| \leq 2^{|X||Y|}$, it follows that $H(PS) \leq |X||Y|$. We interpret $H(PS)$ as measuring the conceptual complexity (i.e. inherent programming difficulty) of the underlying program specification. In the next two sections, we elaborate on this model and its application through a series of examples.

3. SOME SAMPLE APPLICATIONS

When considering the following examples, note that the SD s involve only the termination states of the output variables. Considering only output variables in the SD s seems to provide the most reasonable models for specifications, but it is not an inherent restriction of our model, as will be demonstrated by Program Specification 6. The cases that follow are relatively simple, but provide guidance for analysing more complex problems.

Program Specification 1. The only information provided by the program specification (and input) is the number of locations $|X|$ and number of values $|Y|$. We give two different models of this *a priori* specification.

Model a. $PS_a = \{\text{all } SD\text{s over } X \text{ and } Y\}$. Hence $H(PS_a) = |X||Y|$.

Model b. $PS_b = \{SD\text{s} \mid \text{each location in } X \text{ is assigned exactly one of the values in } Y\}$. Thus PS_b can be interpreted as the set of all functions from X into Y , and $H(PS_b) = \log |Y|^{|X|} = |X| \log |Y|$. PS_b is obtained from PS_a by excluding 'ambiguous' SD s that do not assign every location a value, or that assign some locations more than one value. However, the first type of ambiguity may be considered useful, because it can be interpreted as specifying a solution in which some locations take on undefined values. In any case, PS_a represents a PS with a greater number of possible solutions (hence more uncertainty), so that $H(PS_a) > H(PS_b)$ is consistent with the intended interpretation of H .

Program specification 2 (an abstract sort). y input values are to be arranged, according to some given criterion, in y memory locations.

Model. PS_2 represents the set of permutations of the y values. Hence $H(PS_2) = \log y!$.

At first glance it may seem counter-intuitive that the problem of sorting ten numbers should have a greater conceptual complexity than the problem of sorting, say, five numbers. This objection is equivalent to suggesting that the complexity of Program Specification 2 be a fixed value, instead of being a function of the dataset size. However, the conceptual complexity of divide-and-conquer algorithms (Ref. 1, pp. 306–307) provides strong evidence that a problem's dataset size can contribute to the conceptual complexity of that problem. The divide-and-conquer technique yields a solution by reduction to instances of the original problem on smaller datasets. Certainly, when such solutions are conceptually simple the reduction must have been to simpler problems, implying that the problem on smaller datasets must be considered simpler than the same problem on larger datasets. Program Specifications 6 and 7, below, will illustrate this divide-and-conquer insight for the problem of finding a maximum.

Program Specification 3. One out of y input values (say, the maximum of the y values) is to be selected.

Model. PS_3 represents the y output values, and hence $H(PS_3) = \log y$. Note that $H(PS_2) > H(PS_3)$, again consistent with our intuition that, for equal numbers of input values, Specification 2 is more complex than Specification 3.

Program Specification 4 (computation of an identity function). Input a value and then output that same value.

Program Specification 5 (computation of a constant function). Output a constant value regardless of the input value.

Models. $|PS_4| = |PS_5| = 1$. Thus $H(PS_4) = H(PS_5) = \log 1 = 0$. In both of these specifications, knowledge of the output requires, at most, knowledge of only the input (without further computation). Hence neither of these problems contains any uncertainty, so that each of them is measured as having complexity 0.

We conclude this section with two observations concerning the application of our theory. In this paper we mainly restrict our attention to the important class of problems involving the rearrangement and selection of

data values. The model can also be easily applied to other types of specifications. Secondly, our model is not a very high-level one, because it deals directly with variables and values. However, the level of abstraction is under the control of the person constructing the model. For example, because of data typing, variables and values for a program description intended for APL or SNOBOL are likely to be of higher level than for a program description intended for an assembly language. A related point is that the purpose of subrange declarations in specifications based on Pascal or Ada,* as interpreted within our model, is to reduce the associated *a priori* problem space, and therefore reduce the complexity of the specification.

4. IMPLICATIONS FOR MODULARITY

Program specifications are often given in terms of conceptually simpler *subspecifications* or *modules*. A benefit of this modularisation is that each module can be considered separately, independently of the others, and perhaps developed in parallel with the others. The overhead in conceptual complexity due to increased detail, in particular the overhead associated with interfacing the modules, may be considered a cost of the modularisation. In this section we show, through a series of examples, how to use our metric to quantify these notions of the cost and benefit of modularisation.

Consider the following specification for selecting the maximum of y elements, which exploits the associativity of the maximum operation, and is a modular refinement of Program Specification 3.

Program Specification 6. (a) Find the maximum m of the initial $y-1$ elements of a list of y ($y \geq 2$) elements; and then, (b) find the maximum of m and the y th element.

Model. Let X consist of two memory locations: the location to hold m (which has $y-1$ possible values), and the location to hold the final answer (which, given m , has two possible values). Then $H(PS_6) = \log((y-1) \cdot 2) = \log(y-1) + 1 \geq \log y = H(PS_3)$, with equality if and only if $y = 2$.

Note that unlike the examples in the preceding section, the detail of this problem specification forces us to examine the 'intermediate' location that holds m , in addition to the location that holds the final answer. It is this additional detail that causes $H(PS_6)$ to be greater than $H(PS_3)$ (i.e. it causes PS_6 to represent more uncertainty, from the variables/values perspective, than PS_3). We can, however, consider Program Specification 6 to consist of a Program Specification Module 6a and a Program Specification Module 6b. Then each module can be modelled using only its output locations. Because PS_6 can be viewed as the Cartesian product of PS_{6a} and PS_{6b} , we have $H(PS_{6a}) + H(PS_{6b}) = \log(y-1) + 1$.

It seems reasonable to define the *cost* C , of the modular decomposition of Program Specification 6 into Program Specification 3 as the difference between the complexities of the original specification PS_3 and PS_6 :

$$C(PS_6; PS_3) = H(PS_6) - H(PS_3) = \log(y-1) + 1 - \log y.$$

Motivated by the potential for the independent implementation and inspection of modules, we define the

benefit B of a modularisation as the difference between the complexity of the original specification and the complexity of the most complex module:

$$\begin{aligned} B(PS_6; PS_3) &= H(PS_3) - \max(H(PS_{6a}), H(PS_{6b})) \\ &= \log y - \log(y-1) = \log(y/(y-1)). \end{aligned}$$

We emphasise that one expects Module 6a to be conceptually less complex than Program Specification 3, although they both involve finding the maximum member of a list of elements; this is because decreasing the size of the data set for a problem makes it easier to find a solution for that problem. Also, this reduction in conceptual complexity can be considered the basis for converging on solutions having faster execution times, as in divide-and-conquer algorithms.

We observe that the *relative value* RV of the module can be defined as the ratio of the complexity of the module to the complexity of the entire specification. For example,

$$RV(PS_{6a}; PS_6) = H(PS_{6a})/H(PS_6).$$

Hence, $RV(PS_{6a}; PS_6) > RV(PS_{6b}; PS_6)$. In our present example, because PS_{6a} contains the major percentage of 'uncertainty' in the problem, we would be willing to pay more to obtain Module 6a than to obtain Module 6b.

Before examining some alternative specifications for the maximum problem, we note that Program Specification 6 illustrates how a specification involving an intermediate variable can be functionally decomposed into two modules that involve only output locations, so as to obtain a positive conceptual benefit from divide-and-conquer. Thus, Program Specification 6 very simply illustrates one of the key concepts of the functional programming philosophy.⁴ One can argue that program specifications, as well as programs, are best expressed as a composition of functional modules, thereby eliminating the need to consider intermediate variables.

Program Specification 7 (maximum by balanced divide-and-conquer). (a) Find the maximum m_1 of the first $y/2$ elements in a list; and then (b) find the maximum m_2 of the last $y/2$ elements; and then (c) find the maximum of m_1 and m_2 .

$$\begin{aligned} \text{Model. } H(PS_7) &= H(PS_{7a}) + H(PS_{7b}) + H(PS_{7c}) \\ &= 2 \log(y/2) + 1 = -1 + 2 \log y. \end{aligned}$$

We next verify the soundness of our metric by showing that it identifies an obviously poor decomposition.

Program Specification 8. Find the maximum of a list of y elements by (a) sorting the y elements; and then (b) extracting the element placed last in the list by Module a.

$$\begin{aligned} \text{Model. } H(PS_8) &= H(PS_{8a}) + H(PS_{8b}) = \log y! + \log 1 \\ &= \log y!. \end{aligned}$$

Note that Program Specification 8 is inferior to Program Specifications 6 and 7 with respect to both cost and benefit. Because $B(PS_7; PS_3) = 1$ and $B(PS_6; PS_3) = \log(y/(y-1))$, we see that Program Specification 7 is superior relative to benefit. However, the cost of Program Specification 6 is less than the cost of Program Specification 7. (Of course, there is no unique way to measure cost versus benefit; these values must be appropriately weighted in each situation.)

Next, we give a program specification where the cost is 0, but the benefit is positive. This implies that there is no 'overhead' for the modularisation, but a benefit, nevertheless.

* Ada is a trademark of the U.S. Department of Defense.

Program Specification 9 (insertion sort). Given y variables, (a) sort the first $y-1$ variables; and then (b) insert the y th value appropriately in the list (a variation of Program Specification 2).

$$\begin{aligned} \text{Model. } H(PS_9) &= H(PS_{9a}) + H(PS_{9b}) \\ &= \log(y-1)! + \log y \\ &= \log y!. \end{aligned}$$

$$C(PS_9; PS_2) = \log y! - \log y! = 0.$$

$$B(PS_9; PS_2) = \log y! - \log(y-1)! = \log y.$$

Finally, we observe that our model provides insight into a different kind of modular decomposition, namely the decomposition of a join query in the relational data manipulation language SQL. The conceptual simplicity gained by such a decomposition is intuitively obvious and has been verified empirically (Ref. 33, p. 642). Our model accounts for the gained simplicity in the following way. Let A and B be files with $|A| = m$, $m \geq 2$, and $|B| = n$, $n \geq 2$. The complexity of a join having the form.

SELECT attributes FROM A, B WHERE condition

has complexity $\log 2^{mn} = mn$, because a join is defined (and therefore appropriately conceptualised) as being a subset of the Cartesian product of A and B . However, the 'nested select' form of this query

SELECT A-attributes FROM A WHERE A-attribute IN
(SELECT A-attribute FROM B WHERE condition)

has complexity $\log(2^m \cdot 2^n) = m + n \leq mn$, because this SELECT is defined as first getting a subset of B and then retrieving the records in A that appropriately match that subset.

Incidentally, observe that we can consider the above two selects as program specifications, say, for corre-

sponding COBOL programs, instead of specific expressions of queries. Then our computations provide a quantitative comparison of the conceptual complexities of these COBOL specifications.

5. SUMMARY AND DISCUSSION

The model presented here formalises the concept of specification problem space, and it provides a consistent metric for evaluating the complexity of each problem space. Because the complexity of a specification can be computed before its implementation is completed, the metric can be used to partition large problem spaces into smaller ones objectively, so as to simplify the programming process. The model also quantifies the cost and benefit of such decompositions. Research into the capabilities of programmers to implement modules of different complexities correctly would provide important guidelines for subdividing computer programs.

A related topic concerns the relative complexity of each different implementation of the same specification. Because the number of locations and values associated with a program module depends on the level of abstraction supported by the implementation language (for example, compare sort routines in a machine language and APL), insights into programming languages are possible. In particular, it seems promising to compare program declaration sections for the same specification, as implemented in different languages, in an attempt to quantify the levels of those languages. The declaration sections, which can be generated before a program's detail is supplied, are then sufficient for the calculation of a program's complexity.

REFERENCES

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, Reading, Mass. (1983).
2. A. J. Albrecht and J. G. Gaffney, Jr. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE Transactions on Software Engineering SE-9* (6), 639-648 (1983).
3. R. Ash, *Information Theory*. Wiley, New York (1965).
4. J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21* (8), 631-641 (1978).
5. A. L. Baker and S. H. Zweben, The use of software science in evaluating modularity concepts. *IEEE Transactions on Software Engineering SE-5* (2), 110-210 (1979).
6. Y. Bar-Hillel, *Language and Information*. Addison-Wesley, Reading, Mass. (1964).
7. R. E. Brooks, Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies 9*, 737-751 (1977).
8. J. S. Bruner, J. J. Goodnow and G. A. Austin. *A Study of Thinking*. Wiley, New York (1956).
9. N. S. Coulter, Software science and cognitive psychology. *IEEE Transactions on Software Engineering SE-9*, (2), 166-171 (1983).
10. F. Craik and R. S. Lockhart, Levels of processing: a framework for memory research. *Journal of Verbal Learning and Verbal Behavior 11*, 671-684 (1972).
11. B. Curtis, S. B. Shepard, P. Milliman, M. A. Borst and T. Love, Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering SE-5* (2), 96-104 (1979).
12. E. W. Dijkstra, The humble programmer. *Communications of the ACM 15* (10), 859-866 (1972).
13. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J. (1976).
14. T. Gilb, *Software Metrics*. Winthrop, Cambridge, Mass. (1977).
15. M. H. Halstead, *Elements of Software Science*. Elsevier, New York (1977).
16. J. D. Gould, Some psychological evidence on how people debug programs. *International Journal of Man-Machine Studies 7*, 151-182 (1975).
17. R. C. Linger, H. D. Mills and D. I. Witt, *Structured Programming: Theory and Practice*. Addison-Wesley, Reading, Mass. (1979).
18. T. Love and A. Fitzsimmons, A review and evaluation of software science. *ACM Computing Surveys 10* (1), 3-18 (1978).
19. H. C. Lucas and R. B. Kaplan, A structured programming experiment. *The Computer Journal 19*, 136-138. (1976).
20. R. E. Mayer, The psychology of learning computer programming by novices. *ACM Computing Surveys 13* (1), 121-141 (1981).
21. T. McCabe, A complexity measure. *IEEE Transactions on Software Engineering SE-2* (4), 308-320 (1976).
22. G. A. Miller, The magical number 7, plus or minus two. Some limits on our capacity for processing information. *Psychological Review 63* (2), 81-97 (1956).

23. H. D. Mills, Measurements of program complexity. In *Software Productivity*, edited H. D. Mills, pp. 57–63. Little, Brown, Boston (1983).
24. D. L. Parnas, A technique for software module specifications with examples. *Communications of the ACM* **15** (5), 330–336 (1972).
25. C. E. Shannon, A mathematical theory of communication. *The Bell System Technical Journal* **27**, 379–423, 623–656 (1948). Reprinted in Ref. 26.
26. C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*. University of Illinois Press, Chicago (1949).
27. B. A. Sheil, The psychological study of programming. *ACM Computing Surveys* **13** (1), 101–120 (1981).
28. S. Shepard, B. Curtis, P. Millman and T. Love, Modern coding practices and programmer performance. *Computer* **12**, 41–49 (1979).
29. B. Shneiderman, *Software Psychology*. Winthrop, Cambridge, Mass. (1980).
30. B. Shneiderman, F. Mayer, D. McKay and P. Heller, Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM* **20** (6), 373–381 (1977).
31. M. L. Shooman, *Software Engineering*. McGraw-Hill, New York (1983).
32. H. A. Simon, How big is a chunk? *Science* **183**, 482–488 (1974).
33. C. Welty and D. W. Stemple, Human factors comparisons of a procedural and a nonprocedural query language. *ACM Transactions on Database Systems* **6**, (4), 626–649 (1981).
34. M. Woodfield, M. Hennell and D. Hedley, A measure of control complexity in program text, *IEEE Transactions on Software Engineering* **SE-5** (1), 45–50 (1979).
35. E. Yourdon, *Techniques of Program Structure and Design*. Prentice-Hall, Englewood Cliffs, N.J. (1975).