COMPLEXITY OF ALGORITHMS AND SOFTWARE METRICS

M. Burgin  
Department of Mathematics  
University of California, Los Angeles  
Los Angeles, CA 90095

N. Debnath  
Department of Computer Science  
Winona State University  
Winona, MN 55987

Abstract: The aim of this paper is to introduce a system structure in the diversity of algorithmic complexity measures for a study of computer software. We consider basic concepts and constructions from the theory of algorithmic complexity and develop a system structure in this theory, especially, for the axiomatic theory. An important class of dual complexity measures is studied. We demonstrate how software metrics fit in the lattice of algorithmic complexity measures. Relations between software metrics and algorithmic complexity are analyzed.

Keywords: software, software metrics, complexity measures, algorithmic complexity, axiomatic complexity measures, dual complexity measures

1 INTRODUCTION

Software quality, as quality of any product, is very important. However, quality is a very complex property and to evaluate it, we need a consistent approach based on the theory of evaluation and measurement. As we know, it is impossible to measure or evaluate directly many properties, including quality of programs. However, we can do this and those who develop programs do this in practice. According to the general theory of evaluation [8], the first step in evaluation of software quality is to define criteria. Criteria of good software include such properties as reliability, adequacy, exactness, completeness, convenience, user friendliness etc. However, such properties are also directly immeasurable and to estimate them, it is necessary to use corresponding indicators or indices. With respect to software quality such indicators are called software metrics.

The area of software measurement, is one of the areas in software engineering where researchers are active since more than thirty years. The main tool area in this area is software metrics. A metric is here not considered in the sense of a mathematical metric space, but is treated as a quantitative property of programs obtained in measurement. According to the measurement theory [16], measurement in some domain $D$ is a homomorphism of empirical objects from $D$ to a partially ordered set $L$, which in science is called the scale of the measured property.

Software quality is a property of computer programs that displays in different processes related to these programs. These processes include development maintenance, updating and utilization of program systems. Consequently, software metrics are aimed at determination how well programs fit these processes. The better fitness is achieved, the simpler corresponding process is. This implies that metrics have to estimate or measure complexity of processes based on corresponding programs. As a result, the majority of software metrics measure complexity of program systems. For example, program complexity is defined in software engineering either as the relative difficulty in creating this program compared to other software products or as the difficulty of understanding this program.

At the same time, all programs are algorithms and the theory of algorithms and computation has a developed component that studies complexity of algorithms and computation (cf., for example, [1, 2, 14]). The theory of algorithmic complexity has an advanced apparatus and a diversity of useful results. In spite of this, interaction between the area of software measurement and the theory of algorithmic complexity is minimal (if any at all). Little application of the theory of algorithmic complexity to software metrics is caused by absence of a unified systemic approach to software metrics. Although the theory of complexity is much more developed than the theory of software metrics, its system organization has to be developed further.

The aim of this paper is to introduce a system structure in the diversity of algorithmic complexity measures for a study of computer software. This may remedy many shortcomings of software metrics considered in [9].

In what follows, we consider programs as representations of algorithms and in such a way relate properties and measures of algorithms to computer programs. In this paper, only properties and metrics of existing programs are treated.

2 STRUCTURES OF ALGORITHMIC COMPLEXITY MEASURES

Complexity of a system $R$ is the amount of resources for a process $P$ that involves $R$. There are different kinds of involvement. $P$ may be a process in $R$: $R$ is a computer, $P$ is an electrical process in $R$, and the resource is energy. $P$ may be a process realized by $R$: $R$ is a computer, $P$ is a computational process in $R$, and the resource is memory. $P$ may be a process controlled by $R$: $R$ is a program, $P$ is a computational process controlled by $R$, and the resource is time. $P$ may be a process that builds $R$: $R$ is a software system, $P$ is the process of its design, and the resource is programmers. $P$ may be a process that transforms $R$. $P$ may be a process that utilizes $R$. $P$ may be a process that models $R$. $P$ may be a process that predicts behavior of $R$.

Natural resources consumed by a process $P$: time, space, information, energy/power, minerals, etc. Social resources consumed by a process $P$: people involved,
their time, efforts, expertise, knowledge, etc. Artificial resources consumed by a process \( P \): system time, system space, data, knowledge, memory, system units, system actions, etc.

For systematization, we consider three types (static, dynamic, and processual), three classes, and two kinds (direct and dual) of complexity measures.

**Definition 1.** Static complexity measures depend only on an algorithm/program that is measured.

Examples are the length of an algorithm, which is equal to the number of symbols in its description or number of instructions in a program.

**Definition 2.** Dynamic complexity measures depend both on an algorithm/program that is measured and on the input.

As examples, we can take such measures as the time of processing some given data or the volume of memory that is demanded by this algorithm/program.

**Definition 3.** Processual complexity measures depend on an algorithm/program, its realization, and on the input.

As examples, we can take such measures as the time of processing some given data or the volume of memory that is demanded by this processing. For deterministic algorithm/program the dynamic and processual complexities coincide. However, in non-deterministic cases, one algorithm/program can define different processes, which demand different time or/and memory volume for realization.

**Proposition 1.** Any static complexity measure generates a dynamic complexity measure that is constant for all inputs of a given algorithm and any dynamic complexity measure generates a processual complexity measure that is constant for all realizations of a given algorithm.

Three classes are: axiomatic, semi-axiomatic, and constructive complexity measures. The constructive complexity measures are defined by some construction. Examples are the time of processing, which for an abstract algorithm is taken as the number of steps of this algorithm, or the volume of memory, which for a Turing machine is taken as the number of the cells of its tape that are used in a given computation. The axiomatic complexity measures are defined by axioms, while semi-axiomatic complexity measures involve both axioms and concrete constructions.

### 3 AXIOMATIC COMPLEXITY MEASURES

The first axiomatic complexity measures were introduced by Blum, who defined static complexity measures as the size of machine [3] and dynamic complexity measures as the computational complexity [4]. Further development of this approach is in [5,6].

Let \( A = \{ A_i : i \in I \} \) and \( B = \{ B_i : i \in I \} \) be some classes of algorithms.

**Definition 4** [5]. A function \( \text{Sc} : I \to N \) is called an axiomatic static complexity measure of algorithms from \( A \) (with respect to \( B \)) if the following conditions are satisfied:

(a) \( \text{Sc}(i) \) is a computable in \( A \) (in \( B \)) total function;

(b) for all \( i \in I \), the value of the inverse function \( \text{Sc}^{-1}(i) \) is a finite set with an effective procedure (in \( B \)) that builds all its elements.

An effective procedure here means an algorithm. Such an algorithm may be recursive, subrecursive or super-recursive. It gives three types of static complexity measures. For the class \( B = R \) of all recursive algorithms, this is the size of machines introduced by Blum [3].

Let \( A \) consists of all programs, which are written in some programming language (e.g., Java or FORTRAN). Then the length \( l(p) \) of a program \( p \) as number of letters as number of words in \( p \) is a static complexity measure.

This axiomatization works well for the theory of algorithms. However, for software metrics, we need more general axioms. Thus, we introduce more extended classes of reconstructable and computable static measures, which satisfy axioms (a) and (b’’) or (b’’) instead of (b).

(b’’') \( \text{Sc}(i) \) is a computable in \( A \) (in \( B \)) full function.

All measures defined by Definition 4, will be called finite reconstructable static measures.

**Remark 1.** A static measure \( \text{Sc}(i) \) may be finite, i.e. for all \( i \in N \), \( \text{Sc}(i) \) is a finite set, but it can be non-computable even in such powerful class as the class \( R \) of recursive algorithms, in particular, of all Turing machines. However, \( \text{Sc}(i) \) can be computable by inductive Turing machines.

Let \( A = \{ A_i : i \in I \} \) be a class of algorithms with the domain \( D_A \). Each algorithm \( A_i \) from \( A \) determines a function \( f_i \) such that \( A_i(x) = f_i(x) \) for any \( x \) from the domain \( D_A \). In particular, \( f_i(x) \) is defined if and only if \( A_i(x) \) is defined.

**Definition 5.** A function \( \text{Fc} : N \times I \to N \) is called an axiomatic computational complexity of algorithms from \( A \) (with respect to \( B \)) if the following conditions are satisfied:

(a) \( \text{Fc}(x, i) \) is a computable in \( A \) (in \( B \)) function;

(b) for any \( x \) from the domain \( D_A \) and any \( i \in I \), \( \text{Fc}(x, i) \) is defined if and only if \( f_i(x) \) is defined.

Examples are such popular measures as \( T \) (time) and \( S \) (space).

This axiomatization works well for the theory of algorithms. However, for software metrics, we need more general axioms. Thus, all measures defined by Definition 5, will be called computable.

**Definition 6.** A function \( \text{Fc} : N \times I \to N \) is called an axiomatic sub-computational (up-computational) complexity of algorithms from \( A \) (with respect to \( B \)) if the following conditions are satisfied:

(a) \( \text{Fc}(x, i) \) is a computable in \( A \) (in \( B \)) function;

(b) for any \( x \) from the domain \( D_A \) and any \( i \in I \), \( \text{Fc}(x, i) \) is defined only if \( f_i(x) \) is defined (\( f_i(x) \) is defined only if \( \text{Fc}(x, i) \) is defined).

Proposition 1 cannot be proved for axiomatic complexity measures because the dynamic measure induced by a static one is always defined, while the
axiomatic computational complexity is not defined for those data for which the algorithm give no result. However, we can prove two slightly weaker results.

**Proposition 2.** If all algorithms from U give results for all inputs, then any axiomatic static complexity measure generates an axiomatic dynamic complexity measure that is constant for all inputs of an algorithm.

**Proposition 3.** Any static complexity measure generates a dynamic complexity measure that is constant for all inputs of a given algorithm when this algorithm gives a result.

Some argue (cf., for example, [11]) that axiomatic complexity measures are too general to describe only common features of such popular resource estimation. To eliminate this obstacle, we subdivide axiomatic complexity measures into three groups: abstract, general, and proper.

**Definition 7.** Abstract complexity measures are abstract properties that reflect essential features of resource estimation.

For example, axiomatic computational complexity measures reflect common features of such popular measures as T (time) and S (space). These features allow one to obtain many properties of complexity measures in an axiomatic setting and then to apply these properties to a variety of proper, semi-axiomatic, and constructive measures.

**Definition 8.** General complexity measures are abstract properties that reflect common features of resource estimation.

For example, we can chose some computational resource R (time, number of steps, volume of memory, number of changes of the computation direction, number of interactions etc.). Then the amount Vi(R) of the consumed resource R is determined for each step i of computation. Choosing an appropriate integral operation I [7], we define the dynamic complexity measure cR related to the resource R by the formula

\[ c_R(A, x) = I(V_i(R); i \in I) \]

Here Vi(R) is the amount of the consumed resource R at the step i for all steps of the computation of an algorithm A with the input data x.

In such a way, we obtain a common structure for many popular complexity measures. cR(A, x) is a general dynamic complexity measure.

The concept of a computational resource may be also axiomatized.

**Definition 9.** Proper complexity measures are properties of resource estimation.

The most popular examples of proper complexity measures are the length l(P) of a program, time of the computation T(x) with input data x and memory space S(x) utilized by the computation.

4 DUAL COMPLEXITY MEASURES

All measures considered above are direct, as they are direct properties of algorithms. Dual complexity measures are properties of objects that are constructed and processed by algorithms. Here we consider only static dual complexity measures for algorithms.

Let A be an algorithm and Sc: P → N be a static complexity measure of programs from a set P for the algorithm A.

**Definition 10.** The dual to Sc complexity measure Sc(x) with respect to A is a function from the codomain (the set of all outputs) Y of A that is defined as

\[ Sc(x) = \min \{ Sc(p); A(p) = x \}. \]

The most interesting case is when A is a universal algorithm V for all programs from P.

**Definition 11.** The dual to Sc complexity measure Sc with respect to the class P is defined as

\[ Sc(x) = \min \{ Sc(p); V(p) = x \}. \]

In the theory of algorithms, a lot of dual complexity measures are studied: Kolmogorov complexity, uniform complexity, prefix complexity, monotone complexity, process complexity, conditional Kolmogorov complexity, time-bounded Kolmogorov complexity, resource-bounded Kolmogorov complexity, generalized Kolmogorov complexity etc.

Dual complexity measures with respect to the class P have invariance properties, defining minimal resources that are necessary in P to build/compute objects from Y. The set Y contains such objects that can be computed by programs from P.

Let T and H be two sets of functions.

**Definition 12.** A function f(n) is called (asymptotically) T-optimal in H if there is such h∈T that f(n) ≤ h(g(n)) for any g∈H and (almost) all n∈N.

Let T(h) = { h(n)= h(h(n)+k), k∈N } and P be a set of programs with a universal algorithm U that is properly generated in P.

**Theorem 1** [5], For any axiomatic static complexity measure Sc(p) on P and for some recursively computable function h(n), there is a T(h)-optimal dual measure Sc(h).

**Definition 13.** f(n) ~T(h) g(n) if there is such h∈T that f(n) ≤ h(g(n)) for all n∈N.

**Definition 14.** Functions f(n) and g(n) are called T(h)-equivalent if f(n) ~T(h) g(n) and g(n) ~T(h) f(n).

**Theorem 2** [5], Any two T(h)-optimal functions are T(h)-equivalent.

Theorems 1 and 2 imply minimality and uniqueness for many dual complexity measures from [14].

5 SOFTWARE METRICS AS COMPLEXITY MEASURES

According to Zuse [18], more than one thousand software metrics were proposed by researchers and practitioners. It demands a book to consider all of them. That is why here we consider only three software metrics: the Lines of Code (LOC), the Length of Program (N), and the Volume of Program (V).

5.1 The Lines of Code

In 1974 Wolverton [17] made one of the earliest attempts to formally measure programmer productivity using lines of code (LOC). Studies have shown that the count of the number of lines of code can indicate an estimate of the complexity of understanding of a
program [10,13,15]. The LOC metric is discussed and used till today. It may be computed by counting the number of executable source code lines in a program. A code line may be considered as an elementary unit of a program as formally programs are words in the alphabet consisting of code lines. From this perspective, LOC is such a very popular static complexity measure as it satisfies axioms (a) and (b) from Definition 4.

When the length of a line is bounded (and this is true for all programming languages as compilers demand this restriction), then LOC is a finite reconstructible static complexity measure as it satisfies axioms (a) and (b) from Definition 4.

5.2 The Lengths of Program

In his book *Elements of Software Science* [12], Halstead considers several software metrics. The first of them is the length of the program $N(P)$. In Software Science [12], a computer program is considered to be a sequence of tokens, that are divided into groups of operators and operands. The basic metrics in software science are functions of the counts $n_1$ of the unique operators and $n_2$ of the unique operands, as well as the total occurrences of operators $N_1$ and of operands $N_2$. The simplest of such function is the length of the program $N(P)$ which is defined as

$$N(P) = n_1 + n_2$$

This metric can also indicate an estimate of the complexity of writing and storing a program.

Describing a program formally as a sequence of operators and operands, we see that $N(P)$ is also a static complexity measure, namely, the length $l(P)$ of a program. For a programming language in which the numbers $n_1$ of the unique operators and $n_2$ of the unique operands are finite, $N(P)$ is a finite reconstructible static complexity measure as it satisfies axioms (a) and (b) from Definition 4. However, some languages (at least, potentially) operate with infinite alphabets of operands, for example, with all real numbers. There are also theoretical models in which there are infinitely many unique operands. In such cases, $N(P)$ is not a finite complexity measure. If the sets of operands and operators are computable, then this measure is also computable.

5.3 The Volume of Program

Another software metrics suggested by Halstead [12] is the volume of the program $V(P)$. It is defined by the formula

$$V(P) = N_1 + N_2 \log (n_1 + n_2)$$

When it is defined, it is always a finite reconstructible static complexity measure.

6 Conclusion

Thus, we have systematized complexity measures for algorithms and described their types by means of axioms. This allows us to deduce from these axioms many common properties of complexity measures, to build a correspondence between software metrics and algorithmic complexity measures, and to study software metrics by means of the theory of algorithmic complexity.

7 References