SOFTWARE METRICS FROM THE ALGORITHMIC PERSPECTIVE

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

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

Abstract: In this paper, software metrics and processes of program development, verification and maintenance are analyzed in the context of the theory of algorithms. This study is aimed to support the development of high quality production-grade software. The process of program design as a transition from a problem to a program is studied. Analysis of the situation shows that there are different meanings of a problem solution. We formalize these meanings with the aim of better structuring and optimization of the program design process. On the base of algorithmic complexity and of the theory of programs in the sense of Halstead, we analyze qualities of programs represented by software metrics. A classification of software metrics is developed.

Keywords: problem, program, software, problem complexity, program complexity, software metrics

1 INTRODUCTION

The problem of computer and software metrics becomes more and more urgent for the contemporary information technology. As Hennessy [9] writes, “While the past 15 years have seen roughly a factor 1000 in performance growth, other important system characteristics, such as reliability, maintainability, salability, and usability, have clearly shown more modest progress. The potentially explosive growth in information appliances and the universal use of computers is likely to emphasize these other characteristics over performance, and thus, highlight our slow progress in improving these characteristics. It seems that it is time for the computer systems research community to focus its attention on a broader set of metrics ...” Software metrics are specific kind of measures for computer programs.

It is possible to design software metrics in an empirical manner, as it has been done from the beginning of their development. Another way is to create a theoretical approach and construct software metrics in a systematic fashion.

As the experience in the field of software production and maintenance demonstrates, the existing individual complexity metrics are not satisfactory. In [6], the following reasons for this are given:

• Metrics are language and form dependent: a different metrics has to be used for different programming languages, different metric for the machine code, object code, etc. Software can be represented in many forms: requirements, specification, documentation, user interfaces, help files, and all that representations can be manifested in very different appearances: written text, graphical, symbolic, formal languages, etc. Again many different (incompatible and incomparable) metrics have to be used to measure all of them. As a consequence we are not able to measure the software in the holistic manner, compare various products, trace complexity through the design steps, etc.

• The output of a traditional complexity metric is a number, usually without any “physical” meaning (we don't know what are we measuring), without critical values indicating what is large or small. No fundamental conclusions can be deduced or induced.

• The relations between software metrics are rarely known: thereafter such metrics are a poor basis for stating fundamental laws.

So in order to improve the software process, we need to apply theoretical approach and design other types of software metrics.

Our basic assumption is that programs are written not for their own sake, but to solve some problem. This allows us to consider all kinds of software metrics with the emphasis on those that are related to the process of program development.

In the second section, going after Introduction, we consider the process of program design as a transition from a problem to a program. Analysis of the situation shows that there are different meanings of a problem solution. We formalize these meanings with the aim of better structuring and optimization of the program design process and its optimization. The third section contains a brief description of types of software metrics and elements of the theory of programs in the sense of Halstead [7]. We analyze what qualities of programs are represented by different software metrics. A classification of software metrics is developed. In the fourth section, we consider software metrics as algorithmic complexity measures. The research is based on the methods, constructions and techniques developed in [3,4,5]. Conclusion contains some problems related to software metrics.

2 FROM PROBLEMS TO PROGRAMS

Any program is aimed at solution of some problem, that is, there is a problem such that given parameters of this problem as initial data for the program, this program solves this problem. In other words, behind each program there is a problem. Program design begins with some problem that is necessary to solve [11]. So, to estimate complexity of the future program and complexity of its design, it is possible to use complexity of the corresponding problem.
Definition 1 [6]. The complexity of a problem is the amount of resources required for an optimal solution of the problem.

However, the word solution has different meanings. Here we discern three kinds of its meaning, deriving from it three types of solutions: final, intermediate, and start solutions. Each of these solutions has two forms: static as the obtained result and dynamic as a process that brings us to this result. To understand the difference between these kinds of solutions, we need a more exact definition of a problem.

Definition 2. A problem consists of three parts: absence of some object, understanding of this absence, and a feeling of a need for this object. Such absent object may be some information, for example, what weather will be tomorrow, or some physical object such as a house or car.

At first, we consider static solutions.

Definition 3. A final solution to a problem is an object that we or somebody else needs.

Definition 4. An intermediate solution to a problem is a system that will be able to give us a final solution when it will be necessary.

Definition 5. A start solution to a problem is a system that will give us an intermediate solution.

Now let us explain the meaning of these definitions for the programming reality.

For example, biologists want to know the structure of genes in an organism. This is the prime problem of bioinformatics. The final solution of this problem is a description of the necessary structure. However, genetic structures are very complex and biologists cannot discover these structures without help of computers. Thus, an intermediate solution to this problem is a computer program that computes the genetic structure given information of corresponding biological measurements. A start solution to this problem is a program designer or a team of program designers who will write such a program.

Definition 6. A dynamic (start, intermediate, or final) solution to a problem is a process that gives as its result a static (start, intermediate, or final) solution to this problem.

Taking our example, we see that a dynamic start solution is a process of finding or organization of a team of program designers, a dynamic intermediate solution is a process of designing/writing the necessary program, and a dynamic final solution is a computational process that gives the necessary genetic structure.

Remark 1. The triadic approach to solution reflects not only problems of programming, but a multiplicity of aspects of social activity. One of such important aspects is information flows and knowledge exchange in organizations. According to the contemporary approach to the theory of organization, in organizations with low internal and external complexity, people who know how to get the final solution are the most important. When the complexity of organization grows and reaches some level, people who know those who can get the final solution become the most important. Our triadic model of a problem solution implies that if the complexity of organization continues to grow and reaches some new level, then people who know those who know those who can get the final solution become the most important.

The triadic model of a problem solution allows us to give more exact definition of the complexity of a problem in such form that is relevant to problems of computer software.

Definition 7. The start (intermediate or final) complexity of a problem is the amount of resources required for obtaining an optimal start (intermediate or final) solution of the problem.

However, in real life optimal solutions are hard or even impossible to achieve. Consequently, we have to consider complexity of a given solution. Traditionally, the following definition is used [6].

Definition 8. The complexity of a solution is the amount of resources required for that solution of the problem.

With respect to the triadic model of a problem solution, we obtain three types of complexity for a program: the realizational, design and organizational complexity of a program.

The organizational complexity measure is very rarely taken into account in spite of its importance.

3 TYPES OF SOFTWARE METRICS AND THE THEORY OF PROGRAMS

Software metrics are measures used to evaluate software, software development resources, and/or the software development process. In addition, software metrics help to analyze changes in software [16].

One of the first syntactical software metrics is the LOC (lines of code). Estimated and actual source lines are used for productivity estimates, as described by Walston and Felix [15], during the proposal and performance phases of software contracts. The software metrics LOC is mentioned in more than ten thousand papers [17].

The most popular software metrics, which are discussed heavily today and which were created in the middle of the seventies are the measures of McCabe [13] and a cluster of measures of Halstead [7].

Although, more than one thousand software metrics were proposed by researchers and practitioners, and till today more than 5000 papers about software measurement are published [17], it is possible to develop a system, classifying all software metrics.

There are different categories of software parameters, yielding the following classification of software quality metrics.

1. Functional quality of a program reflects what functions are realized by this program. It is measured by functional software metrics.

2. Mission quality of a program reflects the quality of this realization. It is measured by mission software metrics.
3. Resource quality of a program reflects resources necessary for this realization. It is measured by resource software metrics. The latter quality is reciprocal to the complexity of the corresponding program. In such a way, the resource quality of a program reflects complexity of this program. Software metrics can be also classified by the object of measurement:

1. Textual software metrics are determined directly by characteristics of the text of a program.
2. Model software metrics are determined by models of a program.
3. External software metrics are determined by external factors such as cost or the number of a programmers.

4 COMPLEXITY OF PROBLEMS AND SOLUTIONS IN THE ALGORITHMIC CONTEXT

All computer programs are algorithms. So, to find complexity of a computer program, it is natural to use algorithmic measures of complexity. According to Definitions 8, such measures have to determine resources. This is done by direct complexity measures [5]. These measures applied to the realizational complexity of a program estimate such computational resources as time for computation, memory for program storage, memory for program functioning, number and types of computers, number and types of input/output devices, number and types of processors, and so on. Because there are many such measures, it is efficient to study their properties in the context of axiomatic complexity measures [2]. Direct complexity measures allow one to assess functioning of different kind of software, for example expert systems [3].

Direct complexity measures applied to the design complexity of a program estimate such computational resources as time for program development, cost of this process, number and qualification of designers/programmers, number of external interfaces, number of encountered problems and other factors of software production. These resources reflect parameters of the process of program development that are evaluated by means of such repository software metrics as cost, schedule, staffing, size, requirements, risk and others [8].

Importance of the organizational complexity of a program is underestimated and consequently, such measures are not studied. Utilization of corresponding resources are only partially reflected in such software metrics as cost and staffing.

Let us consider some examples.

4.1 The Software Science Effort

In software science [7], 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 [7] are functions of the counts of the unique operators and the unique operands as well as the total occurrences of operators and operands.

A measure of software science effort for a program P reflects the mental work related to the program P. It can be obtained using the estimate of program level, L(P), and program volume, V(P) as defined in [7]. The effort, E, is given by the formula E(P) = V(P) / L(P). By the definition, we have L(P) = V*/V, L(P) = (n1*/n1)(n2*/n2), and V(P) = (N1 + N2)*log (n1 + n2) where n1* is the minimal estimated number of the unique operators for P, n1 is the number of the unique operators in P, n2 is the number of the unique operands in P, N1 is the total the number of operators in P, N2 is the total the number of operands in P and V* is the minimal estimated volume of the program P.

It is possible to demonstrate that the effort E(P) is a compound complexity measure that is built of the dual complexity measure L(P) and the direct complexity measure V(P) [5].

4.2 The Cyclomatic Number

The complexity measure that was proposed based on the control flow graph (CFG) model of a program P [13] is called the cyclomatic number V(P) of a program P and is defined by the formula

\[ V(P) = e - n + 2, \]

where e is the number of edges and n is the number of nodes in the CFG of a program P.

For abstract graphs, we have a more complex expression for the cyclomatic number. Namely, if e is the number of edges, p is the number of connected components, and n is the number of nodes in G, then the cyclomatic number v(G) of a graph G is defined [1] by the formula:

\[ V = e - n + p \]

The cyclomatic number v(G) is equal to the number of independent cycles in G [1].

It is also shown [13] that the cyclomatic number is a function of the number of predicates in the program. In particular, if a CFG represents a program containing P predicates (decision points), then [13] the cyclomatic number is equal to

\[ V = P + 1. \]

Representing a program formally as a structure of operators and operands, we can demonstrate that the cyclomatic number V(P) is a static direct complexity measure [5].

Direct complexity measures usually evaluate completed processes and programs. However, very often it is necessary to make estimates in advance. In this case, we need complexity of a problem. The complexity of a problem often differs from the complexity of its solution. Complex solutions may exist for simple problems. Moreover, as it is proved by [10] many important problems that have hard solutions (those that are P-complete for ESPACE) have low problem complexity, that is, their Kolmogorov complexity or algorithmic information is rather low.

If we estimate optimality of a program by the demanded resources (either for design or for realization or for implementation), then the corresponding mathematical structure is a dual complexity measure [5]. We consider a problem of obtaining some object by

281
means of a given system of algorithms (methods). Such object may be a program or a result of computation controlled by a program. Computers give many useful results: what are parameters of an optimal car, plane or building. A dual complexity measure defines minimal resources that allow one to solve a given problem. Because there are many such measures, it is efficient to study their properties in the context of axiomatic complexity measures [3,4].

It is important to stress on the dependence of problem complexity on means that are used or are achievable for solving this problem. For example, if to develop a program, we allow to use all recursive algorithms (such as Turing machines), then the minimal number of operation, which gives the time estimation in the theory of algorithms, is equal to zero if the input device coincides with the output device. If these devices/tapes are different, then we need only time for rewriting a text from the input tape to the output tape because Turing machines can use as an input any text/word, even the text of the demanded program. However, in real life, we cannot give this program as an input, otherwise it would not be a problem of program design. Thus, restrictions on the utilized means can result in very high time complexity. Many experts in Kolmogorov complexity measure, which is the most popular kind of the dual complexity measures [12], do not understand this and do not study general classes of measures that are dual to computational complexity.

Thus, we can make the following conjecture.

**Software Complexity Thesis.** All software (resource) metrics are algorithmic complexity measures and any complexity measure represents some resource software metrics.

Software metrics are designed and used to evaluate software, software development resources, and/or the software development process. Consequently, in addition to resource software metrics, which measure resources, there are other types of software metrics. They are considered elsewhere.

### 5 CONCLUSION

In spite of problems existing with software metrics, software engineering must continue to automate both the measurement and analysis of software quality and incorporate the reuse of existing software components to satisfy the need for quality software. A healthy metrics program focuses on much more than the measurement of programmer productivity. The following areas of the software development can benefit from a well-planned metrics program: project management, product quality, product performance, product evolution, development process, cost and schedule estimation. Regularly developed software metrics can reduce the burden of quality assurance and increase software reuse.

Existing software metrics are oriented at procedural programming languages. It would be useful to construct such software metrics that measure qualities of programs written in programming languages having other types, descriptive and functional. In addition, it might be even more interesting to design such software metrics that are independent of the type of a programming language

### REFERENCES