2013 Fifth International Conference on Multimedia Information Networking and Security

An Empirical Study of Source Level Complexity

Liu Xiao and Peraphon Sophatsathit
Advanced Virtual and Intelligent Computing (AVIC) Center
Department of Mathematics and Computer Science
Faculty of Science, Chulalongkorn University, Bangkok 10330, Thailand
liuxiaolxy@163.com, peraphon.s@chula.ac.th

***Abstract*—This paper presents a source level complexity evaluation method using a set of well-established measurements. The objective is to gauge the variations of program complexity being written in different programming languages, thereby performance assessment can be reached. Experiments show that source code written in compiled languages have greater complexity than those in interpreted languages. The results could aid in language selection decision for software development so as to attain higher quality, lower effort, and shorter development time.**

Keywords-software metrics; source code complexity; compiled language; interpreted language.

I. INTRODUCTION

Effective management of development process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of software development process model. Thus productivity and quality can be improved.

As software becomes more complex, the cost inevitably increases. Software organizations are trying to find ways to reduce it. Research efforts are spent finding the relation of software feature and the extent of the problem that would lessen the cost burden. One of the reasons for proceeding to investigate software complexity and its measurement is controlling the expenditure of software development, operation, and maintenance over its life time. Unfortunately, software complexity is an inherent property that cannot be straightforwardly identified, described, and measured. Worse yet, it is often disregarded in the development planning process and incorporated as an after-thought artifact. This is particularly apparent during the maintenance phase where considerable amount of efforts are expended to modify the source code. The overwhelming magnitude of complexity poses a challenging problem for researchers to reckon with.

In early 70's, software complexity had attracted the public attentions. The modularization program style and the object-oriented paradigms were both introduced to lower such complexity. Meanwhile, a number of useful metrics were employed to measure various level of software complexity, yet were inadequate to settle this problem.

This paper aims to providing a straightforward measurement for source level complexity of program using a set of well-established measurements, namely, operators, operands, parameters, inputs and outputs, files operations, externals functions or libraries, variable declarations, and flow graph. We have selected popularly used programming languages such as C, C#, Java, Python, PHP, and Perl to effectively measure and assess source code complexity.

The rest of the paper is organized as follows. Section 2 introduces some related work on source code complexity measurements and metrics. The proposed method is described in Section 3. Section 4 demonstrates the experimental results. Some discussion and final thoughts are given in the last section.

II. RELATED WORK

Many research works on software complexity have been carried out in recent years. Several metrics have been defined and tested in specific environments. Although remarkable successes have been reported in the initial application and validation of these metrics, subsequent attempts to test or apply the metrics in different situations have yielded different results. One problem could stem from failure to identify a commonly accepted set of software properties. Moreover, there were virtually no theoretical models and metrics to support the measurement. Principally, there are three types of well-practiced software complexity metrics, namely, process metrics, project metrics, and product metrics [1]. Some classical and efficient software complexity metrics introduced in [2] were popularly applied to measure the complexity of software. These metrics were compared in [3] and identified which metric was the most suitable one to be adopted in the state-of-the-practice development. They are McCabe's cyclomatic complexity (CCM) [4], Halstead's software science [5] complexity metrics (HCM), and Shao and Wangs' cognitive functional size [6].

While the extent of research in this field is still relatively limited, particularly when compared with research on static metrics, the field is growing given the inherent advantages of dynamic metrics. Tahir, et al. [7] systematically investigated the body of research on dynamic software metrics to identify issues associated with their selection, design, and implementation. Current measures can be used to compute complexity, but these methods are not sufficient to express complexity variations among programming languages. New methods are being searched for predicting complexity since high degree of complexity in a module is considered inefficient as oppose to low degree of complexity [9]. In addition, the measurement helps estimate other quality attributes such as testability and maintainability [8].

978-1-4799-2107-2/13 $31.00 © 2013 IEEE
DOI 10.1109/MINES.2013.108

III. PROPOSED METHOD

This study aims to measuring the complexity of source code written in different programming languages. We propose a method using metrics that focus on operators, function parameters, file operations, and flowchart. The proposed method can be divided into four stages, namely, initial metrics statistics, flow graph transformation, data quantified analysis, and comparative evaluation.

The source programs employed were collected from different languages and grouped based on their operational characteristics. They are 1) Compiled programming languages, such as C, C++, C#, Java, and 2) Interpreted programming languages, such as Python, JavaScript, PHP, and Perl. The reason was that they represented the most popular programming languages used in modern software application development.

A. Initial metrics statistics

For source level complexity evaluation of software, we choose essential programming tokens to be evaluated, namely, operators, operands, parameters, inputs and outputs, file operation, external function references, and variable declarations.

The first step is to count the following tokens: 1) operators such as +, -, *, /, %, ->, (), +=, -=, ++, --; 2) operands in executable statements; 3) parameters in and out of each module or function; 4) inputs and outputs of the program; 5) files operations including open, close, read, and write; 6) external references including library functions, external source files, and external user-defined functions; and 7) variable declarations in the program.

B. Flow graph transformation

This step transforms each program into a flow graph and counts the number of nodes and edges of that flow graph. Figure 1 shows parts of the flow graph converted from Java source code by using Visustin v7 Flow Chart Generator. The number of nodes and edges are 31 and 33, respectively.

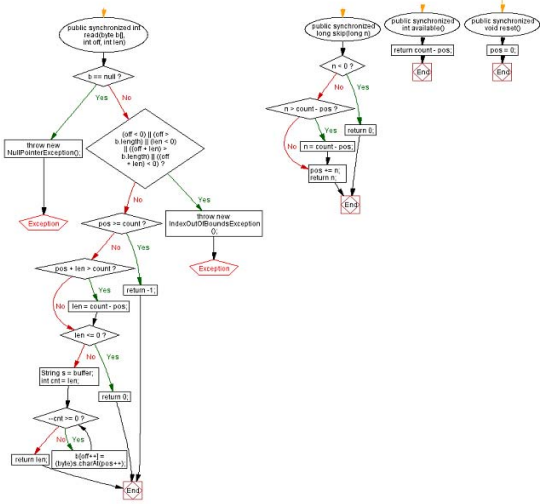

Figure 1. Part of a flow graph converted from Java source code

C. Data quantified analysis

Based on the results of the first two steps, we compute the following statistics: 1) range of the numerical spread, 2) normalized data to standardize the results, 3) standard deviation of the dispersion from the average, and 4) group average and standard deviation.

1) Range

The range can be obtained from the difference between the maximum and minimum values. Table I and II show the minimum and maximum values, while Table III shows the range of each group.

2) Normalized data

All numbers except CCM and HCM are normalized by taking each value and divided by the range of that value as shown in Table IV. For example,

LOC of C = 46/59 = 0.78
OR of Java = 16/16 = 1.00
EL of C# = 1/5 =0.20

3) Average and standard deviation

The average and standard deviation of all the normalized values are shown in Table V, VI, VII, VIII, denoting by group and language, respectively.

D. Comparative evaluation

The above results are plotted by group and programming language to visually compare the resulting measurements. Any discernible proportion of the groups and programming languages will reflect the level of variations in program complexity. Thus, proper development can be planned and administered to attain lesser project effort and cost.

IV. EXPERIMENT

The experiment on source code complexity evaluation by the proposed method is elaborated. All source code was collected from the Internet. There were 20 programs of different sizes written in C, C#, Java, Python, PHP, and Perl. The first three were grouped as compiled programming language, whereas the remaining three as interpreted programming language. The following abbreviations are used to denote all metrics being collected: LOC denotes lines of code, OR and OD denote the number of operators and operands, PR denotes the number of formal arguments, IO denotes the number of inputs and outputs invoked by each function, FE denotes the number of file operations, EL denotes the number of external functions, libraries, and files linked, VE denotes the number of variable declarations, FG denotes the sum of nodes and edges derived from program flowchart, and CCM and HCM are software complexity evaluated by Cyclomatic Complexity and Halstead Complexity Metrics. McCabe's CCM is based on program flow graph and is defined as V(G) = e-n+2, where *e* and *n* represent the number of edges and nodes, respectively. On the other hand, HCM is defined as D = (n1*N2)/(2*n2), where *n1* denotes the number of unique or distinct operators, *n2* denotes the number of unique or distinct operands, and *N2* denotes total operands.

The values of LOC varied from 46 to 607, where the minimum and maximum sizes of both groups were relatively

indifferent. We used Visustin v7 Flow Chart Generator to generate flow graph for each program. Table I and Table II show the maximum and minimum collected from the source programs.

TABLE I. MINIMUM METRICS COUNT OF EACH LANGUAGE

PL	LOC	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
C	46	7	9	0	0	0	1	5	27	2	3
C#	54	2	1	0	0	0	1	1	70	1	0.2
Java	59	16	30	1	0	0	3	2	72	1	24
Python	47	13	9	3	0	0	3	3	30	1	2
PHP	52	1	1	0	0	0	1	3	29	1	0.1
Perl	51	4	4	0	0	0	5	5	28	1	1

TABLE II. MAXIMUM METRICS COUNT OF EACH LANGUAGE

PL	LOC	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
C	585	357	262	12	60	15	140	59	497	10	83
C#	436	99	103	26	13	11	39	52	699	10	39
Java	395	390	571	23	27	0	127	46	494	10	263
Python	309	290	328	119	7	12	38	20	279	10	15
PHP	607	411	401	86	179	6	61	61	351	10	160
Perl	471	210	342	54	27	27	51	72	679	10	54

TABLE III. RANGE OF EACH GROUP

Group	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
Compiled	388	570	26	60	15	139	58	672	9	263
Interpreted	410	400	119	179	27	60	72	651	10	160

TABLE IV. NORMALIZED MINIMUM OF EACH LANGUAGE

PL	LOC	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
C	.78	0.44	0.30	0.00	0.00	0.00	0.20	1.00	0.38	10	19
C#	.92	0.13	0.03	0.00	0.00	0.00	0.20	0.20	0.97	3	11
Java	1.0	1.00	1.00	0.33	0.00	0.00	0.60	0.40	1.00	8	112
Python	.80	0.30	0.30	1.00	0.00	0.00	0.60	0.60	0.42	2	9
PHP	.88	0.03	0.03	0.00	0.00	0.00	0.20	0.60	0.40	8	18
Perl	.86	0.13	0.13	0.00	0.00	0.00	1.00	1.00	0.39	4	22

TABLE V. AVERAGE OF EACH GROUP

Group	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
Compiled	0.368	0.217	0.186	0.231	0.126	0.064	0.243	0.319	10	47
Interpreted	0.301	0.173	0.179	0.197	0.039	0.066	0.250	0.219	6	16

TABLE VI. AVERAGE OF EACH LANGUAGE

PL	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
C	0.421	0.171	0.090	0.131	0.254	0.123	0.309	0.289	10	19
C#	0.349	0.100	0.068	0.315	0.039	0.070	0.089	0.369	3	11
Java	0.334	0.380	0.398	0.248	0.084	0.000	0.331	0.299	8	112
Python	0.247	0.230	0.189	0.256	0.002	0.022	0.232	0.213	2	9
PHP	0.323	0.127	0.120	0.213	0.099	0.031	0.265	0.174	8	18
Perl	0.333	0.162	0.229	0.121	0.016	0.144	0.253	0.271	4	22

TABLE VII. STANDARD DEVIATION OF EACH GROUP

Group	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
Compiled	0.277	0.271	0.260	0.233	0.194	0.203	0.263	0.236	10	60
Interpreted	0.232	0.186	0.204	0.198	0.131	0.179	0.203	0.182	8	23

TABLE VIII. STANDARD DEVIATION OF EACH LANGUAGE

PL	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
C	0.374	0.233	0.110	0.158	0.263	0.268	0.277	0.260	9	18
C#	0.214	0.086	0.059	0.273	0.064	0.210	0.080	0.262	10	10
Java	0.201	0.343	0.346	0.213	0.118	0.000	0.299	0.166	10	63
Python	0.139	0.180	0.158	0.252	0.009	0.097	0.165	0.124	8	3
PHP	0.274	0.217	0.222	0.185	0.212	0.060	0.252	0.138	8	34
Perl	0.250	0.138	0.210	0.101	0.033	0.271	0.179	0.246	7	16

We omitted the function point (FP) metric from the statistics because the interpreted source was relatively small. They lacked the principal attributes that made up the FP computations such as I/O and external references. This is apparent in Table I, II and III, where the spread of these parameters is quite noticeable. This finding might not hold in larger production code. As a consequence, the resulting statistics so obtained from FP metric will not be an accurate measure to participate in this study.

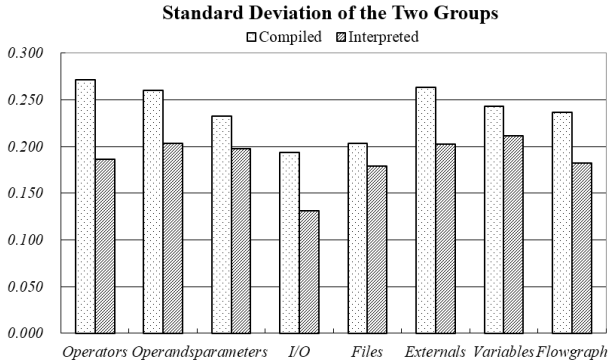

Figure 2. Standard deviation of compiled and interpreted groups

Figure 2 shows the proportional distribution of both groups. Obviously, the compiled language group exhibits discernible complexity variations than its interpreted counterpart owing to program size.

Figure 3 shows the proportional dispersion of different compiled languages. Notice that Java program exhibits the least variation in secondary storage operations, e.g., I/O and Files, yet internally is complicated to maneuver as the values of HCM exceed other languages by many folds. Meanwhile, C and C# are somewhat less sporadic. Such indicators can be conducive toward decision on language selection to suit specific requirements for application development.

The interpreted languages, on the other hand, exhibit wider dispersion than the compiled ones. Particularly, Perl and Python show exceptionally high dispersion of complexity in all but two categories, namely, I/O, Files for Python, and Parameters, I/O for Perl. These are depicted in Figure 4. A noteworthy result is that all languages in the same group exhibit alternate high complexity across the measuring metrics, thereby no single language is suitable for all operational characteristics of the underlying application domain.

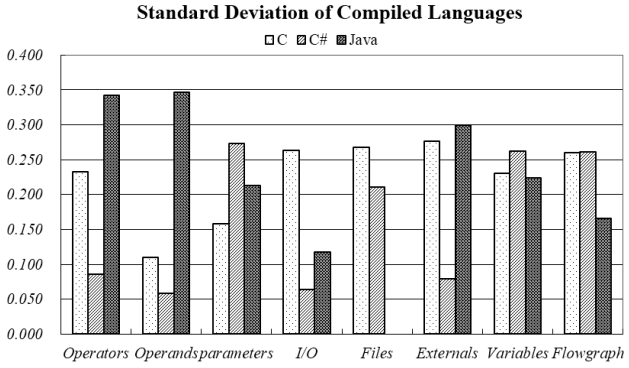

Figure 3. Standard deviation of compiled language: C, C# and Java

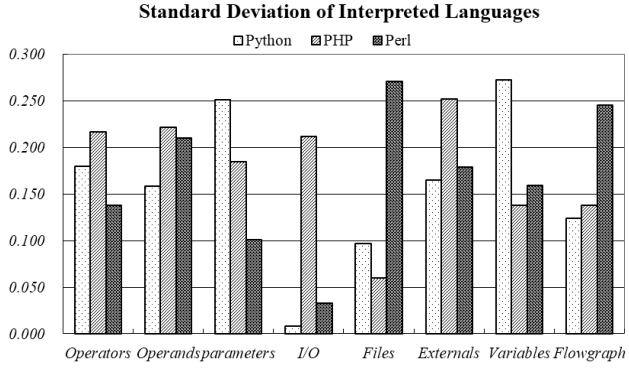

Figure 4. Standard deviation of interpreted language, Python, PHP and Perl

V. DISCUSSION AND CONCLUSION

With the rapid advancement in software industries, software metrics become the basis for software management and are crucial to the accomplishment of software development. We proposed a straightforward method to measure the complexity of source code written in C, C#, Java, Python, PHP, and Perl by means of eight metrics, namely, operators, operands, parameters, inputs and outputs, file operations, external functions or libraries, variable declarations, and flow graph. Our findings revealed that source code written in compiled languages were inherently more complex than those of interpreted languages.

The reason may depend primarily on the nature of application to be performed by the target software, as interpreted software is likely to be smaller and less involved than its compiled counterpart. Despite the significant role played by software metrics, studies and researches in this field are still immature. As new paradigms and programming languages are being invented, in particular, design patterns, automated code generation, 5GL, and user computing. Unfortunately, these techniques bring about accidental and inherent complexities [10] that grow exponentially out of control. The effect renders software project management to inevitably fall behind technology in terms of productivity measurement, cost estimation, project planning, and the likes. In addition, there are no adequate international standards to warrant the software products being distributed. We envision that more work needs to be done to supplement the absence of firm theoretical foundation and assurance of methods and metrics. Such endeavors will foster the development of software applications that could serve the insatiable needs in this evolving digital society.

REFERENCES

[1] T. Honglei, S. Wei, and Z Yanan. “The Research on Software Metrics and Software Complexity Metrics,” Proc. IEEE International Forum on Computer Science-Technology and Application (IFCSTA'09), IEEE Press, Jan. 2009, pp. 131-136, doi:10.1109/IFCSTA.2009.39.

[2] S. Yu and S. Zhou. “A Survey on Metric of Software Complexity,” Proc. IEEE International Conference on Information Management and Engineering (ICIME 2010), IEEE Press, Mar. 2010, pp. 352-356, doi:10.1109/ICIME.2010.5477581.

[3] D. I. De Silva, N. Kodagoda, and H. Perera. “Applicability of Three Complexity Metrics,” Proc. IEEE International Conference on Advances in ICT for Emerging Regions (ICTer 2012), IEEE Press, Dec. 2012, pp. 82-88, doi:10.1109/ICTer.2012.6421409.

[4] T. J. McCabe. “A Complexity Measure,” IEEE Transactions on Software Engineering, vol. 2, Dec. 1976, pp. 308-320, doi:10.1109/TSE.1976.233837.

[5] M. H. Halstead. “Elements of Software Science,” Elsevier Science Inc., New York, NY, 1977.

[6] Y. Wang and J. Shao. “Measurement of the Cognitive Functional Complexity of Software,” Proc. IEEE International Conference on Cognitive Informatics (ICCI 2003), IEEE Press, Aug. 2003. Pp. 67-74, doi:10.1109/COGINF.2003.1225955.

[7] A. Tahir, S. G. MacDonell. “A Systematic Mapping Study on Dynamic Metrics and Software Quality,” Proc. IEEE International Conference on Software Maintenance (ICSM 2012), IEEE Press, Sept. 2012, pp. 326-335, doi:10.1109/ICSM.2012.6405289.

[8] S. Sabharwal, R. Sibal, and P. Kaur. “Software Complexity: A Fuzzy Logic Approach,” Proc. IEEE International Conference on Communication, Information & Computing Technology (ICCICT 2012), IEEE Press, Oct. 2012, pp. 1-6, doi:10.1109/ICCICT.2012.6398233.

[9] M. K. Debbarma, S. Debbarma, N. Debbarma, K. Chakma, and A. Jamatia. "A Review and Analysis of Software Complexity Metrics in Structural Testing," International Journal of Computer and Communication Engineering, vol. 2, Mar. 2013, pp. 129-133.

[10] Frederick P. Brooks, “No Silver Bullet”, Computer, April 1987, pp. 10-19.