State of The Art in Software Size & Effort Estimation

Charan Singh Chandna, Shourabh Dave

Abstract— In this paper, we present an overview of existing size and effort estimates for software. All these estimates are described more or less on their own. Size & effort estimation is a very popular task. We also explain the fundamentals of size & effort estimation. We describe today's approaches for size & effort estimation. From the broad variety of size & effort estimation models that have been developed we will compare the most important ones. Their strengths and weaknesses are also investigated. It turns out that the behavior of the size & effort estimates is much more similar as to be expected.

Index Terms— COCOMO-II, Project planning and Software size & Estimation.

I. INTRODUCTION

Software engineering cost (and schedule) models and estimation techniques are used for a number of purposes. These include:

- Budgeting
- Tradeoff and risk analysis
- Project planning and control
- Software improvement investment analysis

Need of Software Size & Effort Estimation

Small Projects are very easy to estimate and accuracy is not very important. But as the size of project increases, required accuracy is not very important. But as the size of project increases, required accuracy is very important which is very hard to estimate. A good estimate should have amount of granularity so it can be explained. Since the effort invested in a project is one of the most important and most analyzed variables. So the prediction of this value while we start the Software projects, it helps to plan any forthcoming activities adequately. Estimating the effort with a large value of reliability is a problem which has not been solved yet.

II. LITERATURE SURVEY

Size is one of the most important attributes of a software product. It is a key indicator of software cost and time; it is also a base unit to derive other metrics for software project measurements, such as productivity and defect density. This section describes the most popular sizing metrics and techniques that have been proposed and applied in practice. These techniques can be categorized into code-based sizing metrics and functional size measurements.

CODE-BASED SIZING METRICS

Manuscript received on April, 2013.

Charan Singh Chandna, Computer Science Department, MITM, Indore, India.

Sourabh Dave, Computer Science Department, MITM, Indore, India.

Code-based sizing metrics measure the size or complexity of software using the programmed source code. Because a significant amount of effort is devoted to programming, it is believed that an appropriate measure correctly quantifying the code can be a perceivable indicator of software cost. Halstead's software length equation based on program's operands and operators, McCabe's Cyclometic Complexity, number of modules, source lines of code (SLOC) among others have been proposed and used as code-based sizing metrics. Of these, SLOC is the most popular. It is used as a primary input by most major cost estimation models, such as SLIM, SEER-SEM, PRICE-S, COCOMO, and Knowledge Plan.

Many different definitions of SLOC exist. SLOC can be the number of physical lines, the number of physical lines excluding comments and blank lines, or the number of statements commonly called logical SLOC, etc. To help provide a consistent SLOC measurement, the SEI published a counting framework that consists of SLOC definitions, counting rules, and checklists [Park 1992]. Boehm *et al.* adapted this framework for use in the COCOMO models [Boehm 2000b]. USC's Center for Systems and Software Engineering (USC CSSE) has published further detailed counting rules for most of the major languages along with the CodeCount tool.

In COCOMO, the number of statements or logical SLOC is the standard SLOC input. Logical SLOC is less sensitive to formats and programming styles, but it is dependent on the programming languages used in the source code [Nguyen 2007].

For software maintenance, the count of added/new, modified, unmodified, adapted, reused, and deleted SLOC can be used. Software estimation models usually aggregate these measures in a certain way to derive a single metric commonly called effective *SLOC* or *equivalent SLOC* [Boehm 2000b, SEER-SEM, Jorgensen 1995, Basili 1996, De Lucia 2005]. Surprisingly, there is a lack of consensus on how to measure SLOC for maintenance work. For example, COCOMO, SLIM, and SEER-SEM exclude the *deleted SLOC* metric while KnowledgePlan and PRICE-S include this metric in their size measures. Several maintenance effort estimation models proposed in the literature use the size metric as the sum of SLOC added, modified, and deleted [Jorgensen 1995, Basili 1996].

SLOC has been widely accepted for several reasons. It has been shown to be highly correlated with the software cost; thus, they are relevant inputs for software estimation models [Boehm 1981, 2000b]. In addition, code-based metrics can be easily and precisely counted using software tools, eliminating inconsistencies in SLOC counts, given that the same counting rules are applied. However, the source code is not available in early project stages, which means that it is difficult to accurately measure SLOC until the source code is available. Another limitation is that SLOC is dependent on the programmer's skills and programming

styles. For example, an experienced programmer may write fewer lines of code than an inexperienced one for the same purpose, resulting in a problem called *productivity paradox* [Jones 2008]. A third limitation is a lack of a consistent standard for measurements of SLOC. As aforementioned, SLOC could mean different things, physical lines of code, and Physical lines of code excluding comments and blanks, or logical SLOC. There is also no consistent definition of logical SLOC [Nguyen 2007]. The lack of consistency in measuring SLOC can cause low estimation accuracy as described in [Jeffery 2000]. A forth limitation is that SLOC is technology and language dependent. Thus, it is difficult to compare productivity gains of projects of varying technologies.

FUNCTIONAL SIZE MEASUREMENT (FSM)

Having faced these limitations, Albrecht developed and published the Function Point Analysis (FPA) method as an alternative to code-based sizing methods [Albrecht 1979]. Albrecht and Gaffney later extended and published the method in [Albrecht and Gaffney 1983]. The International Function Point Users Group (IFPUG), a non-profit organization, was later established to maintain and promote the practice. IFPUG has extended and published several versions of the FPA Counting Practices Manual to standardize the application of FPA [IFPUG 2004, 1999]. Other significant extensions to the FPA method have been introduced and widely applied in practice, such as Mark II FPA [Symons 1988] and COSMIC-FFP [Abran 1998].

F UNCTION POINTS ANALYSIS (FPA)

FPA takes into account both static and dynamic aspects of the system. The static aspect is represented by data stored or accessed by the system, and the dynamic aspect reflects transactions performed to access and manipulate the data. FPA defines two data function types (*Logical Internal File*, *External Interface File*) and three transaction function types (*External Input, External Output, Query*). Function point counts are the sum of the scores that are assigned to each of the data and the transactional functions.

The score of each of the data and transaction functions is determined by its type and its complexity. Function counts are then adjusted by a system complexity factor called *Value Adjustment Factor (VAF)* to obtain adjusted function point counts.

The IFPUG's Function Point Counting Practices Manual (CPM) provides guidelines, rules, and examples for counting function points. The manual specifies three different types of function point counts for the Development project, the Enhancement project, and the Application count. The Development project type refers to the count of functions delivered to the user in the first installation of the new software. The Enhancement project type refers to Function point counts for modifications made to the preexisting software. And the Application function point count measures the functions provided to the user by a software product and is referred to as the baseline function point count. It can be considered the *actual* function point count of the development project developing and delivering the system. Thus, the application function point count can be determined by applying the same process given for the development project if the user requirements can be obtained from the working software.

In FPA, the enhancement project involves the changes that result in functions added, modified or deleted from the *existing* system. The procedure and complexity scales are the same as those of the development, except that it takes into account changes in complexity of the modified functions and the overall system characteristics. The process involves identifying added, modified, deleted functions and determining value adjustment factors of the system before and after changes.

III. CONCLUSION

In this paper, we surveyed the list of existing size estimation techniques for software development. We restricted ourselves to the modern size estimation techniques. We observed that the overall adjustment factor can be improved by making small changes in adjustment factor by introducing some new factors. It may result in increasing the effort estimates. In a forthcoming paper, we will use the results obtained from this survey to develop a more efficient estimation model.

REFERENCES

1. Abran A., St-Pierre D., Maya M., Desharnais J.M. (1998), "Full function points for embedded and real-time software", Proceedings of the UKSMA Fall Conference, London, UK, 14.
2. Albrecht A.J. (1979), "Measuring Application Development Productivity," Proc. IBM Applications Development Symp., SHARE-Guide, pp. 83-92.
3. Albrecht A.J. and Gaffney J. E. (1983) "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," IEEE Transactions on Software Engineering, vol. SE-9, no. 6, November
4. Basili V.R., Briand L., Condon S., Kim Y.M., Melo W.L., Valett J.D. (1996), "Understanding and predicting the process of software maintenance releases," Proceedings of International Conference on Software Engineering, Berlin, Germany, pp. 464–474.
5. Boehm B.W. (1981), "Software Engineering Economics", Prentice-Hall, Englewood Cliffs, NJ, 1981.
6. Boehm B.W., Abts C., Chulani S. (2000), "Software development cost estimation approaches: A survey," Annals of Software Engineering 10, pp. 177-205.
7. De Lucia A., Pompella E., and Stefanucci S. (2005), "Assessing effort estimation models for corrective maintenance through empirical studies", Information and Software Technology 47, pp. 3–15
8. IFPUG (1999), "IFPUG Counting Practices Manual - Release. 4.1," International Function Point Users Group, Westerville, OH
9. IFPUG (2004), "IFPUG Counting Practices Manual - Release. 4.2," International Function Point Users Group, Princeton Junction, NJ.
10. Jeffery D.R., Ruhe M., Wieczorek I. (2000), "A comparative study of cost modeling
11. techniques using public domain multi-organizational and company-specific data", Information and Software Technology 42 (14) 1009–1016.
12. Jones T.C. (2008), "Applied Software Measurement: Global Analysis of Productivity and
13. Quality", 3rd Ed., McGraw-Hill.
14. Nguyen V., Deeds-Rubin S., Tan T., Boehm B.W. (2007), "A SLOC Counting Standard," The 22nd International Annual Forum on COCOMO and Systems/Software Cost Modeling.
15. Park R.E. (1992), "Software Size Measurement: A Framework for Counting Source Statements," CMU/SEI-92-TR-11, Sept.
16. Symons C.R. (1988) "Function Point Analysis: Difficulties and Improvements," IEEE Transactions on Software Engineering, vol. 14, no. 1, pp. 2-11
17. UKSMA (1998) MkII Function Point Analysis Counting Practices Manual. United Kingdom Software Metrics Association. Version 1.3.1