A FRAMEWORK FOR AUTOMATIC FUNCTION POINT COUNTING FROM SOURCE CODE

Vinh T. Ho and Alain Abran
Software Engineering Management Research Laboratory
Université du Québec à Montréal (Canada)
vho@lrgl.uqam.ca abran.alain@uqam.ca

ABSTRACT

The paper proposes a general framework to build a model for automatic Function Point Analysis (FPA) from the source code of COBOL system using program slicing technique. The COBOL system source code is scanned by the model to produce Function Point counts. The application's source files are used to define the application's boundary for the count. The model takes into account the structure of the COBOL language to identify physical files and transactions. Reserved words as FDs, file input/output statements (READ and WRITE) and user interface and data manipulation statements (ACCEPT, DISPLAY and MOVE) are used as basic information for program slicing technique to identify candidate physical files and transactions. Some heuristic rules will be proposed in order to map candidate physical files and transactions into candidate logical files and transactions. These candidate files and transactions are then assessed with regards to the IFPUG' identifying rules in order to identify data function types and transactional function types to be counted. The proposed framework helps to build models for automating Function Point Analysis from source code in compliance with the IFPUG Counting Practices Manual.

I INTRODUCTION

I.1 Automation of Function Points counting: scope and objectives

Function Point Analysis (FPA) is the measurement of the functional size of software. Function points are now being used for software quality studies, software contract management, business process re-engineering, and software portfolio control (Jones, 1995). The latest release of the method is the version 4.1 of the Counting Practices Manual (IFPUG, 1999). The objectives of Function Point Analysis (IFPUG, 1994) are: to measure what the user requested and received, to measure independently of the technology used for implementation, to provide a sizing measure to support quality and productivity analysis, to provide a vehicle for software estimation, and to provide a normalization factor for software comparison.

The process of Function Points counting is a multi-step manual process that requires a labor intensive and time consuming work and expertise of the counter. Moreover, it depends on availability and quality of documentation related to the application to be counted. It is very useful that some type of software support should be possible and available to count Function Points. Our research is interested in developing a tool that can automatically count Function Points from the source code of COBOL systems.

IFPUG defines automatic Function Point counting as: "Where the system counts the function points automatically based on stored descriptions of the application functions, records the count and performs appropriate calculations". The stored descriptions are referred to as elements in the IFPUG case study no. 1 (IFPUG, 1994). The definition emphasizes on the fact that these elements must be stored on some computer-readable medium. Examples of descriptions in the case study no1 are: user requirements, database physical structure, interfaces and reports layout. Application functions represent a sub-set of all user requirements by representing the user practices and procedures that the software must perform to fulfill the users' needs. Appropriate calculations are referred to those based on the rules of the Counting Practices Manual. That is to obtain an accurate count with regards to the IFPUG Counting Practices Manual (IFPUG CPM).

I.2 Literature review on automatic Function Point counting from source code

The ability to count function point directly from source code is quite interesting for large organizations that spend the majority of their time maintaining software. One of the most pervasive problems while maintaining software systems is that most of the programs being maintained are poorly documented and unstructured. As a result, to manually apply Function Point Analysis to these programs would not be realistic in terms of time, cost and accuracy. However, there are few attempts concerning automation of Function Point counting from source code (Couturier, 1993; Mendes, 1996; Edge, 1997; April, 1997).

In their theoretical research on automation of IFPUG function point analysis method, Paton and Abran (1995) mentioned that the method lacks a formal basis since the object to be counted is nowhere defined in a precise manner. It is clear from the IFPUG CPM that the object to be counted must include such things as file, records, fields, processes and that it must be known which process read and write which fields in which records in which files. They proposed a formal notation for the rules of function point analysis. By breaking function point counting process, they figured out 17 tasks in counting function points. It is indicated that twelve tasks could be programmed while five other tasks need user's judgment. In (Paton and Abran, 1997), a structured analysis of the counting rules was presented using the specification techniques of decision tables for the tasks that could be automated. It is important to mention that all proposed decision tables are justified by referring to the IFPUG CPM. The analysis suggests the following four applications: reviewing existing tools based on IFPUG CPM 4.0, building a validation protocol to facilitate the verification of tools that claim to have automated the function point counting process, extending the rules to include uncodified practice used by expert counters, and building a tool with which a standard body can investigate the effect of changing rules. Although their research has not yet reach the construction of a prototype, their works are the first detail theoretical research published in the literature concerning automation of function point counting process. It is worth noting that their works fully support the IFPUG counting rules.

Edge et al (1997) have proposed a model as a solution to the problem of automating a function point approximation count for COBOL legacy system source code. As they indicated, an important difference between this automated process and manual counting is that the automatic model cannot readily identify individual logical transactions. It is mainly for this reason that the model output is termed a function point "approximation" rather than a function point count. This model addresses both IFPUG and Mk II function point counting methods. Note that this work is still in progress.

April et al (1997) have proposed that there would be a benefit in developing models that can be easily automated, in particularly that would take into account the CPM counting concepts and rules. The CPM rules are expressed in natural language and must be subject to formal definition in order to automate them. Based on the semi-formal representation proposed by Paton and Abran (1995), they have put forward an extension of the formalism to include a logical to physical translation. Using this logical representation, IFPUG counting rules could be formalized while assuming that the primitives of the FPA (i.e. file and process) were identified. An example of formalization of four CPM identifying rules associated to an external inquiry (EQ) was presented. Reverse engineering techniques will be used to identify the primitives (i.e., file and process) of the FPA from the source code. This work is original in the sense that it proposed an approach for automation of function point counting process that aims to fully support the IFPUG concepts and rules. While this approach appears as a promising research direction still a large amount of theoretical and practical works, as to refine the formal representation and to select candidate reverse engineering techniques, need to be done before a tool can be developed.

Software reverse engineering technologies are a promising basis for building tools in order to automate Function Point counting from source code. In the next section, we propose using program slicing technique to identify automatically both data function types and transactional function types from source code of COBOL system.

II AUTOMATIC FUNCTION POINT ANALYSIS USING PROGRAM SLICING TECHNIQUES

II.1 IFPUG Function Point Counting Process

The function point measurement includes five major visible externally components of software applications: inputs, outputs, inquiries, logical files, and interfaces. The Counting Practices Manual published by the IFPUG (IFPUG, 1994) provides a method with a set of rules to measure functional size from the user point of view. The method can be devised in five major steps: determine the type of the count, identify the boundary of the count, determine the unadjusted function point count, determine the value adjustment factor, compute the final adjusted count. Most counting time and effort are spent in the third step: determine the unadjusted function point count (Fig. 1), especially to identify data and transactional function types. Once identified the counting procedure is quite straightforward.

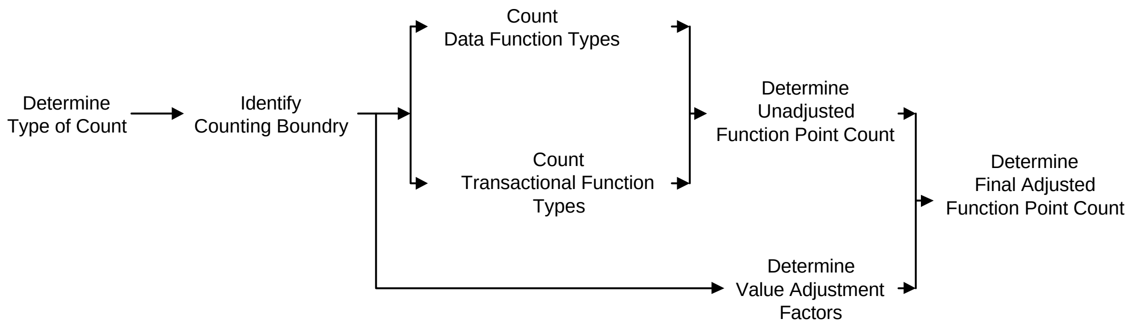
Determine Type of Count
Identify Counting Boundry
Count Data Function Types
Count Transactional Function Types
Determine Unadjusted Function Point Count
Determine Value Adjustment Factors
Determine Final Adjusted Function Point Count

Figure 1. Function Point counting process (IFPUG, 1994)

The CPM provides a set of rules to identify each of five major components that comprise an application: internal logical file (ILF), external logical file (ELF), external input (EI), external output (EO) and external inquiry (EQ). Our main interest is to propose a framework based on program slicing techniques to identify automatically data and transactional function types from source code.

II.2 Background of program slicing technology

Program slicing is a program-analysis technique that extracts from program statements relevant to a particular computation. Such a particular computation is referred as a slicing criterion and is typically specified by a pair (program point, set of variables). The parts of a program that have a direct or indirect effect on the computation at a slicing criterion constitute the program slice with respect to that criterion. The task of computing program slices is called program slicing (Tip, 1995). For example, a slice can answer the question "What program statements potentially affect the computation of variable *v* at statement *s*?" (Binkley and Gallagher, 1996). There are various slightly different notions of program slices as well as a number of methods to compute slices (Tip, 1995). Two kinds of program slices: static and dynamic. The former is computed without making assumptions regarding a program's input whereas the latter relies on some specific input cases. A *backward* slice of a program with respect to a set of program elements S is the set of all programs elements that might affect the values of the variables used at members of S. A *forward* slice with respect to S is the set of all program elements that might be affected by the computations performed at members of S. A *chop* is a kind of "filtered" slice that answers questions of the form "Which program elements serve to transmit effects from a given source element s to a given target element t?" Applications of slicing include program understanding, debugging, testing, parallelizasion, re-engineering, maintenance, etc (Gramma Tech, 1998; Huang et al, 1998; Ning et al, 1994).

II.3 A model for automatic Function Point counting

Most works on automation of Function Point counting have taken a *black box* approach by which the counting process is not transparent to users. The counting tool reads some description of the system being counted and provides the

Function Point counts as final output. For this research we adopt a *white box* approach and the counting process is thus transparent to users. This approach can help the users to assess whether the counting process is in compliance with the IFPUG rules.

A model based on program slicing technique is proposed to derive Function Point counts from source code of COBOL system (Fig. 1). In principle, the COBOL system source code is scanned by the model to produce Function Point counts. The application's source files is used to define the application's boundary for the count. The model takes into account the structure of the COBOL language to identify physical files and transactions. Reserved words as FDs, file input/output statements (READ and WRITE) and user interface and data manipulation statements (MOVE, ACCEPT and DISPLAY) are used as basic information for program slicing technique to identify candidate physical files and transactions. Some heuristic rules will be proposed in order to map candidate physical files and transactions into candidate logical files and transactions. These candidate files and transactions are then assessed with regards to the IFPUG' identifying rules in order to identify data function types and transactional function types to be counted. Determining contribution of these objects is thus straightforward to produce a final adjusted Function Point counts.

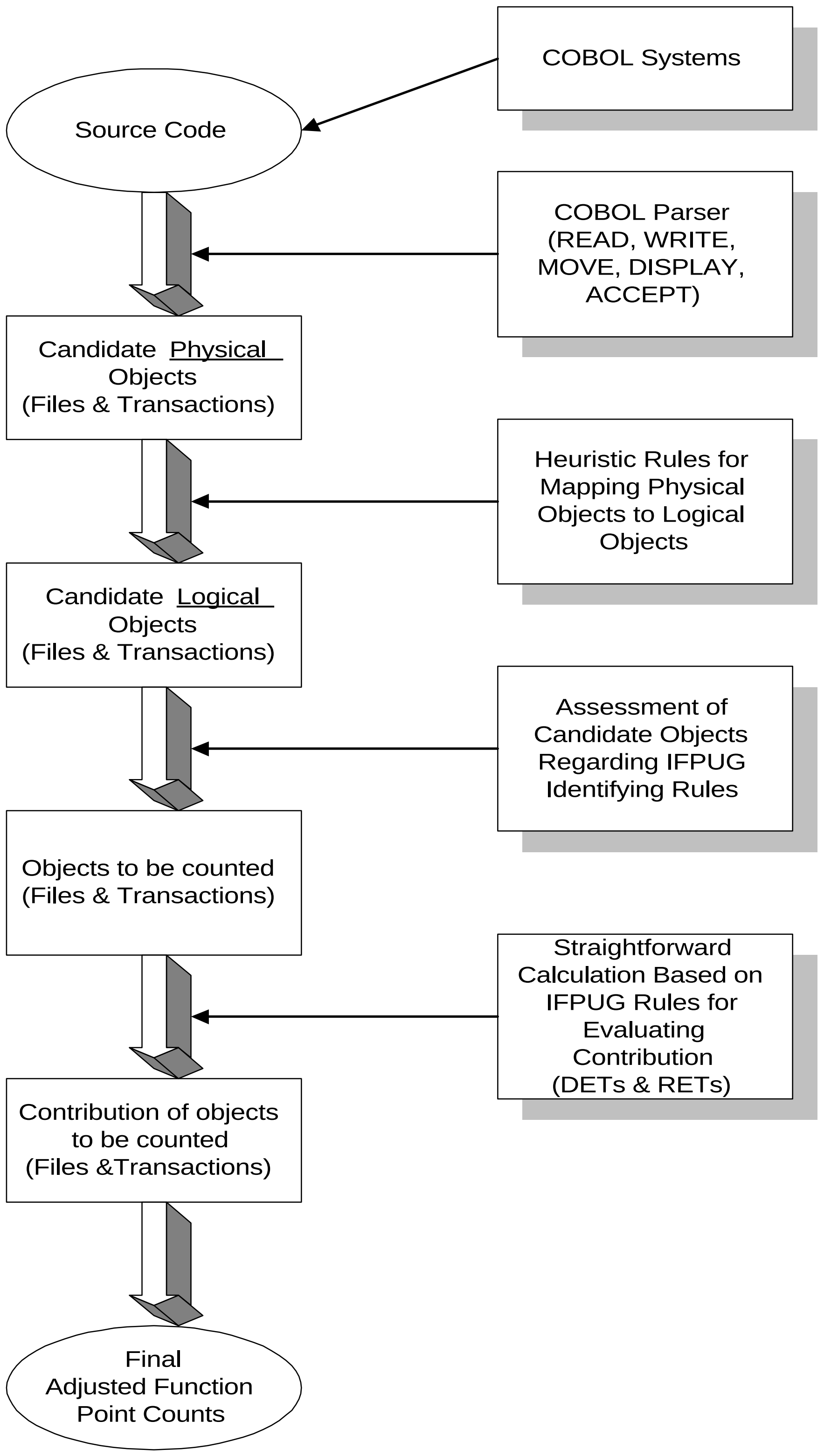
COBOL Systems
Source Code
COBOL Parser (READ, WRITE, MOVE, DISPLAY, ACCEPT)
Candidate Physical Objects (Files & Transactions)
Heuristic Rules for Mapping Physical Objects to Logical Objects
Candidate Logical Objects (Files & Transactions)
Assessment of Candidate Objects Regarding IFPUG Identifying Rules
Objects to be counted (Files & Transactions)
Straightforward Calculation Based on IFPUG Rules for Evaluating Contribution (DETs & RETs)
Contribution of objects to be counted (Files &Transactions)
Final Adjusted Function Point Counts

Figure 2. Proposed model for automatic FPA

III IDENTIFYING DATA AND TRANSACTIONAL FUNCTION TYPES FOR FUNCTION POINT ANALYSIS

III.1 Identifying Data Function Types

From Data Division of the COBOL code and based on the level indicator FD and the level number 01, all physical files, their records and data groups within records are identified. The record's name is then used to track READ, WRITE and REWRITE statements operated on these files (and their "Parent" Procedures).

Candidate physical files are selected as Data Function Type being counted based on the following criteria:

- A file is an Internal Logical File (ILF) if it is updated (by WRITE and/or REWRITE statements) with information entered from outside of the application (i.e., with ACCEPT statement).
- A file is an External Interface File (EIF) if is operated in read-only mode (i.e. by READ statement).

III.2 Identifying Transaction Function Types

Candidate transactions are identified using a file-oriented method. Based on the record and their data groups of the identified logical files, the model can supposedly track (with slicing technique) all procedures and/or statements operated on these files. Note that procedures and statements are identified for each logical file respectively.

The candidate transaction are then analyzed based on a formal notation proposed in (Paton & Abran, 1995). An application is represented by process, user and date flows as in Figure 3. This representation was used to develop a classifying signature for transactional function types of Function Point based on the CPM rules (Abran & Paton, 1997). The signature in Table 1 can be interpreted in the following manner:

- A transaction that reads (i.e., with ACCEPT and/or READ statements) outside of the boundary but does not write (i.e., with WRITE and/or REWRITE statements) outside of the boundary is an External Input (EI).
- A transaction that writes (i.e., with DISPLAY statement) outside of the boundary but does not read outside of the boundary is an External Output (EO).
- A transaction that reads and writes outside of the boundary is an External Enquiry (EQ) if it reads inside of the boundary too.
- A transaction that carries on all four activities (read and write outside and inside of the boundary) is a Double transaction and must be decomposed as an External Input plus an External Output.

Note that a flow is assigned 1 in Table 1 if this flow does exist. Otherwise it is assigned 0.

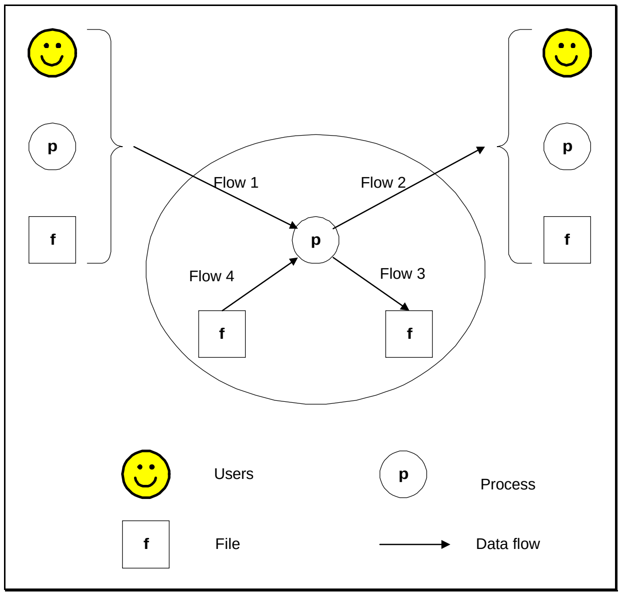

Figure 3. A formal representation of an application (Paton & Abran, 1995)

Table 1. Classifying signatures for transactional function types

	Flow 1	**Flow 2**	**Flow 3**	**Flow 4**
EI	1	0	1	0
EO	0	1	0	1
EQ	1	1	0	1
EI + EO	1	1	1	1

IV CONCLUSION

We propose a framework that can be used to build a model for automatic Function Point counting in compliance to the IFPUG Counting Practices Manual. Slicing technique in reverse engineering is identified as a principle program analysis technique which can help to develop a tool for automating Function Point Analysis from source code of COBOL systems. In addition, the proposed model produces information that can be used to assess the uniqueness of files and transactions while identifying files and transactions for the Function Point count.

It is important to note that the realization of the proposed model is highly dependent on the ability and the efficiency of the slicing tool being implemented. Continuing research will address design a prototype of a tool to automatically compute Function Point from COBOL source codes. Another research direction is the empirical validation of the proposed model by comparing the count produced by the tool to that of a manual count.

ACKNOWLEDGEMENT

This work has been financially supported by Bell Canada and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. Abran, A. & K. Paton (1995). *A Formal Notation for the Rules of Function Points Analysis*. Research Report No 247, LRGL-UQAM.
2. Abran, A. & K. Paton (1997). *Tool Builders Require Structured Rules Specifications for Building Accurate Automated FP Counting Tools*. IFPUG Spring Conference, Cincinnati.
3. Albrecht, A. J. (1988). *Function Points Fundamentals*. IBM: 22.
4. April, A., E. Merlo, et al. (1997). *A Reverse Engineering Approach to Evaluate Function Point Rules*. WCRE97 - Fourth Working Conference on Reverse Engineering, CWI, Amsterdam, The Netherlands, IEEE Computer Society Press.
5. Bellay B. & Gall H. (1998). *An Evaluation of Reverse Engineering Tool*. Software Maintenance: Research and Practice, 10, 305-331.
6. Binkley, D. & Gallagher K. (1996). Program Slicing. In Advances in Computers, Volume 43, 1996. Marvin Zelkowitz, Editor, Academic Press San Diego, CA.
7. Couturier, G. W. (1993). *Automatic function point calculations in maintenance environment*. Proceedings of the 31st Annual Southeast Conference, Birmingham, AL, USA, ACM New York, NY, USA.
8. Edge, N., Finnie, G. Wittig, G*. Automating Function Point Approximation Counts For COBOL Legacy*. ACOSM 97, Australian Software Metrics Association, pp. 80-87.
9. GrammaTech (1998*). CodeSurfer Technology Overview: Dependence Graphs and Program Slicing*.
10. Huang Hai et al. (1998). *Business Rule Extraction Techniques for COBOL Programs*. Software Maintenance: Research and Practice, 10, 3-35.
11. IFPUG (1994*). Function Point Counting Practices Manual*, Release 4.0. Westerville. Ohio, International Function Point Users Group.
12. Jones, C. (1998). *Sizing Up Software*. Scientific American: 104-109.

13. Jones, E. L. (1995). *Automated Calculation of Function Points*. Proceedings for the 4th software engineering research forum, Boca Raton, Florida, SERF.

14. MacDonell, S. G. (1994). *Comparative Review of Functional Complexity Assessment Methods for Effort Estimation*. Software Engineering Journal: 107-116.

15. Mendes, O. (1996). *Développement d'un protocole d'évaluation pour les outils informatisés de comptage automatique de points de fonction*. Département Informatique. Montréal, Université du Québec a Montreal: 430.

16. Ning J. Q et al. (1994). *Automated Support for Legacy Code Understanding*. Communication of the ACM, Vol. 37, 50-57.

17. Tip Frank (1995). *A Survey of program slicing techniques*. Journal of Programming Languages, 3, 121-189.

18. Wittig, G. E. (1995). *Artificial Neural Networks with Function Point Analysis for Software Development Effort Estimation*. School of Information Technology, Bond University: 249.

19. Wittig, G. E. and G. R. Finnie (1994). *Software Design for The Automation of Unadjusted Function Point Counting*. Business Process Re-Engineering Information Systems Opportunities and Challenges. IFIP TC8 Open Conference, Gold Coast, Qld., Australia.