Function Point Measurement from Java Programs

Shinji Kusumoto, Masahiro Imagawa, Katsuro Inoue
Graduate School of Engineering Science
Osaka University
Toyonaka, Osaka, Japan
{kusumoto, imagawa, inoue}@ics.es.osaka-u.ac.jp

Shuuma Morimoto, Kouji Matsusita, Michio Tsuda
Hitachi Systems & Services, Ltd.
Otaku, Tokyo, Japan
{sh-morimoto, k-matsushita, m-tsuda}@hitachi-system.co.jp

ABSTRACT

Function point analysis (FPA) was proposed to help measure the functionality of software systems. It is used to estimate the effort required for the software development. However, it has been reported that since function point measurement involves judgment on the part of the measurer, differences for the same product may occur even in the same organization. Also, if an organization tries to introduce FPA, FP will have to be measured from the past software developed there, and this measurement is cost-consuming. In this paper, we intend to examine the possibility to measure FP from source code automatically. At first, we propose measurement rules to count data and transactional functions for object-oriented program based on IFPUG method and develop the function point measurement tool. Then, we have applied the tool to practical Java programs in a computer company and examined the difference between the FP values obtained by the tool and those of an FP measurement specialist. As the results, the number of data and transactional functions extracted by the tool is similar to ones by the specialist though for the classification of each function there is some difference between them.

1. INTRODUCTION

As the size and the complexity of software increase, it becomes increasingly important to develop high-quality software cost-effectively within a specified period. In order to achieve this goal, the entire software development processes need to be managed based on an effective project plan.

In order to construct a distinct project plan, it is essential to estimate various undesirable phenomena which happened during the project and take measures to prevent them in advance. The subjects of estimation in the area of software development are size, effort invested, development time, technology used and quality. Particularly, development effort is the most important issue. So far, several effort models [3][4][14] have been proposed and most of them include software "size" as an important parameter. In the models, LOC (lines of code) is often adopted as a measurement criterion. However, using LOC for software size assessment has difficulties because the definition of LOC is very vague and LOC depends on the programming language.

Function point is a measure of software size that uses logical functional terms business owners and users more readily understand [2]. Since it measures the functional requirements, the measured size stays constant despite the programming language, design technology, or development skills involved. Also, it is available early in the development process, making its use opportune for planning design and development projects. Up to the present, various FPA versions based on the Albrecht's version have been proposed. The IFPUG (International Function Point Users Group) version [6], MarkII version [12] and COSMIC method [5] have been frequently used in software organizations. Many studies on function points have been reported in conferences and journals[1][11].

However, several unsolved problems still remain. One of them is that differences for the same product may occur even in the same organization since function point measurement involves judgment on the part of the measurer [8]. For example, Low and Jeffery [10] reported that a 30-percent variance was caused within one organization and more than 30-percent variance was caused across organizations. In order to get a consistent value, it is important to automate the function point measurement[2]. Also, if an organization tries to introduce FPA, FP will have to be measured from the past software developed there, and this measurement is cost-consuming.

We have been dealing with automatic function point measurement for object-oriented software[9][13]. Through the studies, we have found that many organizations want to measure function point from source code. Because sometimes there are some functional differences between requirements/design specification and source code and the actual functions are existed only in source code. Also, organizations sometimes remain only source code.

In this paper, we intend to examine the possibility to measure function point from source code automatically and report the early results of this study. At first, we propose measurement rules to count data and transactional functions for object-oriented program based on IFPUG method. Then, we develop the function point measurement tool based on the rules. Finally, we apply the tool to practical Java pro-

grams in a computer company and examined the difference between the FP values obtained by the tool and those of an FP measurement specialist. As the results, the number of data and transactional functions extracted by the tool is similar to ones by the specialist though for the classification of each function there is some difference between them.

Section 2 briefly explains function point analysis. Next, Section 3 proposes the rules to count data and transactional functions from object-oriented program. Section 4 shows the function point measurement tool based on the proposed rules and Section 5 describes the case study. Section 6 concludes the paper.

2. FUNCTION POINT

2.1 Overview

Function point measures the functionality provided by software. It can be determined from the requirements specification, design specification and program code. Unlike LOC, since function point measures functionality, it is said to be independent of the technology and language used for the software implementation.

Allan Albrecht first proposed original function point analysis [2]. Albrecht's function point is computed by counting the following software characteristics: (1) External inputs and outputs, (2) User interactions, (3) External interfaces and (4) Files used by the system. Each of them is then individually assessed for complexity and given a weighting value which varies from 3 (simple) to 15 (complex).

Albrecht's function point has been widely used but it has some weakness. Thus, many kinds of function point, such as IFPUG version[6], COSMIC method[5]. 3D Function Points version[7], Feature Points version[7] and MarkII version[12], have been proposed. In this paper, we address the IFPUG version since it provides the detail procedures and rules for function point counting compared to other versions.

2.2 IFPUG version

IFPUG version is a modified-version of the Albrecht's function point. In the modification, the evaluation of the complexity of the software was objectively established and the rules of the counting procedures were also described minutely and precisely.

In the IFPUG version, the counting procedure of function point consists of seven steps[6], Step1: Determine the Type of Function Point Count, Step2: Identify the Counting Boundary, Step3: Count Data Function Types, Step4: Count Transactional Function Types, Step5: Determine the Unadjusted Function Point Count, Step6: Determine the Value Adjustment Factor and Step7: Calculate the Final Adjusted Function Point Count.

Here, we explain Step 2, 3, 4 and 5 to understand the proposed rules in Section 3.

Step2 (Identify the Counting Boundary): A boundary indicates the border between the application or project being measured and the external applications or the user domain. A boundary establishes which functions are included in the function point count.

Step3 (Count Data Function Types): Data function types represent the functionality provided to the user to meet internal and external data requirements. Data function types are classified into the following two types: Internal logical file(ILF) and External interface file(EIF).

The definition of data functions are described as follows:

Internal Logical File(ILF): (1)The group of data is user identifiable group of data. (2)The group of data is maintained within the application boundary. (3)The group of data identified has not been counted as an EIF for the application.

External Interface File(EIF): (1)The group of data is user identifiable group of data. (2)The group of data is not maintained by the application being counted. (3)The group of data identified has not been counted as an ILF for the application.

Here, the term "file" refers to a logically related group of data and not to the physical implementation of those group of data.

Then, assign each identified ILF or EIF a functional complexity based on the number of data element types (DETs) and record element types (RETs) associated with the ILF or EIF using the RET/DET complexity matrix(See Table 1). A data element type (DET) is a unique user recognizable, nonrecursive field on the ILF or EIF. A record element type(RET) is a user recognizable subgroup of data elements within an ILF or EIF.

Table 1: RET/DET complexity matrix

RET\DET	1-19	20-50	51-
1	Low	Low	Average
2-5	Low	Average	High
6-	Average	High	High

Step4 (Count Transactional Function Types):

Transactional function types represent the functionality provided to the user for the processing of data by an application. They are defined as the following three types: External input(EI), External output(EO) and External inquiry(EQ). The definition of transactional functions are described as follows:

External input(EI): An external input processes data or control information that comes from outside the application's boundary. The external input itself is an elementary process.

External output(EO): An external output is an elementary process that generates data or control information sent outside the application's boundary.

External inquiry(EQ): An external inquiry is an elementary process made up of an input-output combination that results in data retrieval. The output side contains no derived data. Here, derived data is data that requires processing other than direct retrieval and editing of information from internal logical files and/or external interface files. No internal logical file is maintained during processing.

Then, assign each identified EI or EO a functional complexity based on the number of file types referenced (FTRs) and data element types (DETs).A file type referenced is ,(1) An internal logical file read or maintained by a function type, or (2) An external interface file read by a function type. Also, assign each EQ a functional complexity based on the number of file types referenced (FTRs) and data element types (DETs) for each input and output component. Use the higher of the two functional complexities for either the input or output side of the inquiry to translate the external inquiry to unadjusted function points. For each of EI, EO and EQ, there is a FTR/DET complexity matrix. Table 2 shows the FTR/DET complexity matrix for EI.

Table 2: FTR/DET complexity matrix of EI

FTR\DET	1-4	5-15	16-
0-1	Low	Low	Average
2	Low	Average	High
3-	Average	High	High

Step5 (Determine the Unadjusted Function Point Count):

As the result of Step3 and Step4, the counts for each function type are classified according to complexity and then weighted using the matrix. The total of all the function types is the unadjusted function point count.

3. PROPOSED FUNCTION POINT MEASUREMENT RULES FOR OO PROGRAMS

3.1 Key Idea

Here, we explain the key idea of the proposed function point measurement rules for object-oriented programs. The proposed approach deal with the function point measurement based on IFPUG, the main process is extracting data functions and transactional functions from the target program.

It seems to be difficult to judge the types of functions only from the static information about source codes. So, we use the dynamic information collected from the program execution based on a set of testcases which should correspond to all functions of the target program. It is desirable that the testcases are used for acceptance test by the user because functions not used by the user (for example, functions used by only the developers for debug or maintenance activities) are not tested and such functions should not be counted as function point. Also, since the testcases used in the acceptance test would be kept in the organization, this assumption is appropriate.

An example of the dynamic information collected from program execution is shown in Figure 1. In Figure 1, there are four classes[1] (A, B, C and D). It shows an interaction, which is a set of messages exchanged among the classes like a sequence diagram. We call this kind of sequence as *method calling sequence*. By analyzing the contents of the messages and the type of the classes in the method calling sequence constructed by each testcase, we measure the function point.

[1]Actually, they are objects based on the classes. "Class" in this paper is almost the same as "object".

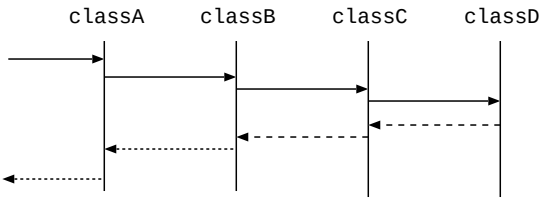

Figure 1: Example of dynamic information

In the following sections, we explain the details of counting data and transactional functions.

3.2 Counting data function types

We consider that some of the classes included in the target program are data functions because, in IFPUG, data functions are defined as the functionality provided to the user to meet internal and external data requirements.

From our previous experience[9][13], most classes (except actor classes) on the object-oriented requirements/design specification directly corresponded to data functions. However, in the actual program even if it was developed based on the design specification, there are a lot of classes which are not appeared on the specification. So, it is quite difficult to identify which classes should be corresponded to data functions. Here, we assume that the function point analyst select the classes that would be the data functions from the program.

Then, the data functions are classified into two types: Internal Logical File (ILF) and External Input File (EIF) as follows:

ILF: Among the classes selected as data function, during the program execution, the classes some of which methods are called with some arguments are regarded as ILF. That is, we consider that the arguments represents the data and the methods update the data which the class obtains.

EIF: Among the classes selected as data function, other classes that are not regarded as ILF, are regarded as EIF.

Finally, we need to decide the complexity of the ILF/EIF based on the number of data element type(DET) and the record element type(RET). DET is a unique user recognizable, nonrecursive field on the ILF or ELF and RET is a user recognizable subgroup of data elements within an ILF or EIF. As described above, since classes in the program are corresponded to data functions, we define that DET is the number of simple variables (int, chat, boolean) in the class and RET is the number of the variables defined as the class type. That is, class variables defined as class type is considered to be the meaningful group of data.

3.3 Counting transactional function types

In IFPUG, transactional functions are defined as the input/output processing, which updates or refers to the data function, from outside the application's boundary. Regarding the classes as data functions, we consider that the method

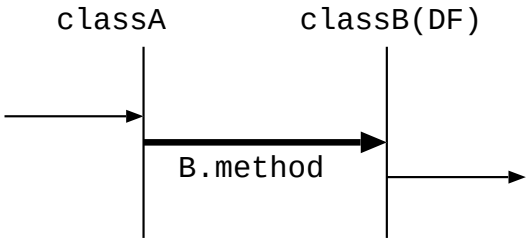

Figure 2: Basic element

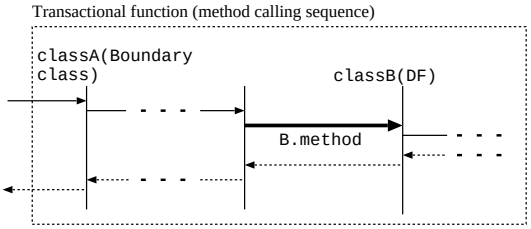

Figure 3: Identification of transactional function

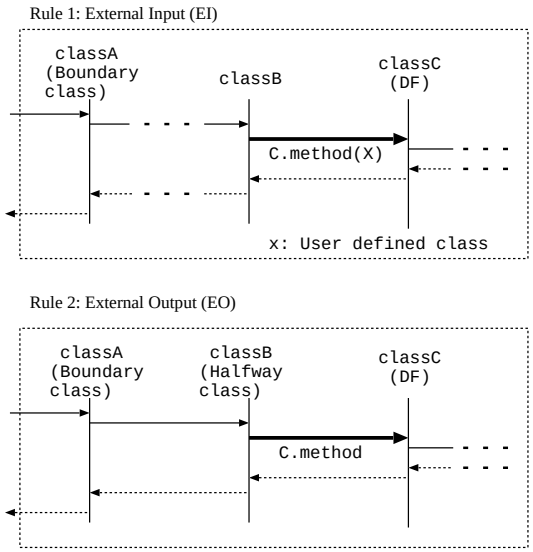

Figure 4: Rules for transactional functions

that updates or refers to the data in the class can be used to extract the transactional function.

Base on the idea, we define *basic element* in the method calling sequence. Basic element is the sub-method calling sequence where a method, that is defined in a data function class, is called by other class as shown in Figure 2.

At first, we collect the method calling sequence for all the testcases and the sequences are candidates of transactional function. Next, to identify transactional function, we have to take account of the application boundary. Here, we identify the class, whose methods are inevitably called when the user input some data into the program, as *boundary class*. For example, GUI classes or Java Servlet classes would be the boundary classes. Then, in a method calling sequence which starts when the method defined in the boundary class is called, and ends when the method call is finished, if the basic element is appeared, then the sequence is regarded as a transactional function(See Figure 3).

If there exist transactional functions in which name of the called method, type and the number of the argument, the order of called are the same, they are regarded as the same transactional function.

Then, for each of the identified transactional functions, we classify them into three types: External Input(EI), External Output(EO) and External inquiry(EQ) based on the proposed rules in Figure 4.

In Rule 1, since the user-defined class is delivered to data function class as a argument, such sequence is regarded as EI.

In Rule 2, before delivering the return value to the boundary class from data function class, some class (called *halfway class*), that are not identified as data function class, modifies the return value and delivers it to the boundary class. In IFPUG, a processing in which data maintained by data function is processed and outputted to application boundary is regarded as EO. Thus, such sequence is regarded as EO.

For the transactional functions that do not satisfy the Rules 1 and 2, we consider that they don't include renewal of data function and output the processed data outside the application boundary. So, they are regarded as EQ.

Finally, we need to decide the complexity of the EI/EO/EQ based on the number of data element type(DET) and the types referenced record element type(FTR). DET is the number of data items which are comings and goings through application boundary and FTR is the number of data function updated or referred to in the transactional function.

In our proposed rule, since boundary classes are application boundary, we count DET as follows:

- EI: DET is the total number of the argument of the methods called by the boundary class in the processing of EI.
- EO, EQ: DET is the total number of the return values of the methods called by the boundary class in the processing of EO. If the return value is a class, then the number of variables defined in the class is also counted.

FTR is the number of data function appeared in the transactional function. In the case that the same data function class is appeared repeatedly, it is counted just once.

4. FUNCTION POINT MEASUREMENT TOOL

Based on the proposed method, we have developed a function point measurement tool from Java programs. The tool has developed on Windows2000 PC with Java.

Figure 5 shows the overview of the system. It includes the following components:

- Syntax analyzer: It analyses the target program and accumulates the syntax information used in the FP calculation of it into syntax information file.
- Executor: It executes the target program using a set of testcase, collects the information about program execution and accumulates it into execution log file.
- FP calculator: It calculates the value of function point based on the data of syntax database, execution log

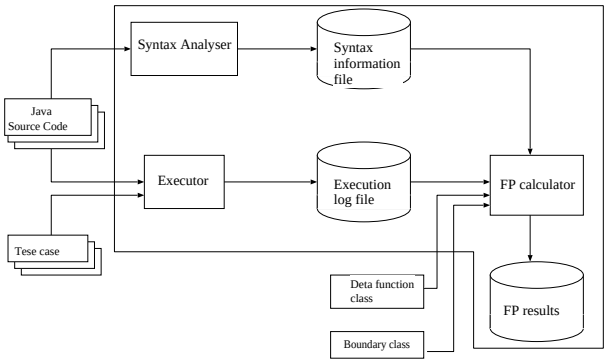

Figure 5: Overview of FP tool

Table 3: Format of syntax information file

C	Class name		
CV	Class variable name	Type	DETflag
⋮	⋮	⋮	⋮
M	Method name		
MV	Method variable name	Type	DETflag
⋮	⋮	⋮	⋮
M			
MV			

database using the specified data function classes and boundary classes.

The format of the syntax information file is shown in Table 3. In Table 3, C means a label of the name of a class. Then, CV means a label of the name, type, and a flag (DETflag) of each class variable defined in the class. The flag (DETflag) is used to judge whether the variable is regarded as DET or RET. Then, M means a label of the name of the method defined in the class. MV means a label of the name, type, and a flag of each class variable defined in the method. DETflag is also used to judge whether the variable is regarded as DET or RET. Table 3 shows the information for just one class. Actually, the information is described for all classes in the target program.

An example of the execution log file is shown in Table 4. In Table 4, the line labeled "Begin" indicates that the following method is called in the program execution and the line labeled "End" indicates that the following method execution is finished. For example, the first line "Begin Sakaya.Sakaya.1.String" means that a method "Sakaya" in a class "Sakaya" is called with one argument and the type of the argument is String. Next, a method "Souko" in a class "Souko" is called with two arguments and the each type of the arguments is String and int. Then, a method "Haisou" in a class "Haisou" is called with no arguments. Finally, these methods are recursively finished. Figure 6 shows Table 4 schematically.

Table 4: Example of execution log file

Begin	Sakaya.Sakaya.1.String
Begin	Souko.Souko.2.String.int
Begin	Haisou.Haisou.0
End	Haisou.Haisou.0
End	Souko.Souko.2.String.int
End	Sakaya.Sakaya.1.String

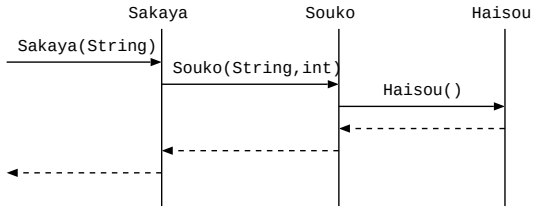

Figure 6: Corresponding sequence to Table 4

5. CASE STUDY

5.1 Overview

In order to evaluate the appropriateness of the proposed rules, we applied our tool to an application program developed in Hitachi Systems & Services. Then, we compared the FP values obtained by our system to those obtained by the FP counting specialist. The FP counting specialist counted the FP from requirements specifications of the application.

The target program is a typical Web application. Since the application has been developed based on the object oriented approach and such kind of application would be developed repeatedly, we consider that it is appropriate for the case study. The structure of the application is shown in Figure 7. Java program in the application server is the target program. The size of the application is about 10K steps.

In order to measure function point by the tool, it is necessary to identify the data function classes in the program and boundary classes. In this case study, we selected them as follows:

Data function classes: The classes whose methods access and update the table of the database are selected as data function classes.

Boundary classes: Java Servlet classes except the classes for showing confirmation message and assistance for data input are selected as boundary classes. Because in the target program, the screen for data input includes several sub-screens for data input assistance. So, we consider that it is appropriate to select only the classes that implement the main screens for data input.

Table 5 shows the FP values calculated by the system and the function point specialist. The size of the execution log files was 15MB. As shown in Table 5, the values of unadjusted function point are quite similar (174 and 170). Based on Table 5, we analyze and examine the discuss the results.

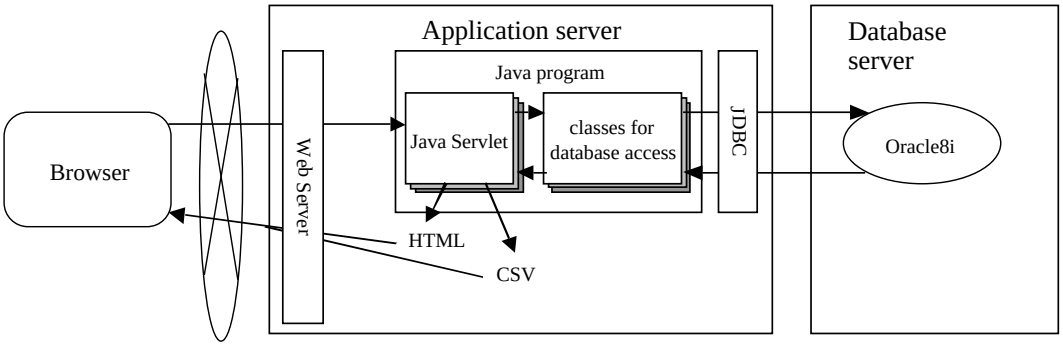

Figure 7: Target application program

Table 5: Result of FP

		Tool	Specialist
Data Function	ILF	11	4
	EIF	1	7
Transactional Function	EI	9	6
	EO	13	14
	EQ	0	0
FP		174	170

5.2 Analysis of data function

It is not always possible to correspond the data functions on the requirements specification to ones on the program. However, in this case study, by selecting the classes, that conceal the access to database, as data functions, both the number of data functions by the system (12=11+1) and the specialist (11=4+7) are quite similar. It indicates that it would be possible to get the values of data functions from program that is similar to ones from requirements specification.

On the other hand, the classification result of data function types are different between the tool and the specialist. The tool regarded most data functions as ILF. The result is due to the fact that most methods in the data function class have arguments. In the classification rule described in Section 3.2, such classes are regarded as ILF.

In order to cope with this problem, it is necessary to revise the classification rule. The original rule is, among the classes selected as data function, during the program execution, the classes some of whose methods are called with some arguments are regarded as ILF. It is conceivable to make conditions to the arguments. For example, if the argument of the method in a class indicates the meaningful data, the class is regarded as ILF. Also, it may be necessary to collect the detail information from program executions whether the data in the class is updated.

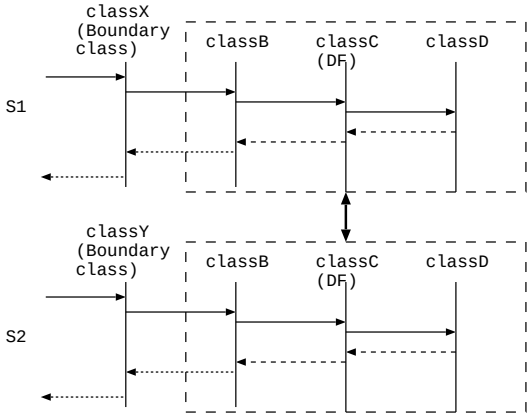

Figure 8: Cause of classification error

5.3 Analysis of transactional function

Both the number of transactional functions by the tool (22=9+13) and the specialist (20=6+14) are quite similar. Each of the transactional functions counted by the specialist was also counted by the tool. So, the tool counted two transactional functions excessively. The reason is that we extracted the transactional function as the method calling sequence and if the same method calling sequence was extracted, then it was not counted as twice. In this case, if the starting class of the calling sequence is not the same, they are decided as different functions. Figure 8 shows an example of this case. The method calling sequence S1 and S2 include the same sub-sequence (surrounded by a dotted line). However, the specialist judged such ones are essentially the same and he regarded them as one transactional function. To cope with this problem, we need to improve the rules for transactional function. For example, some calling sequences each of which includes the same sub-calling sequence, though the length of the sequence should be determined in some way, regard as the same one.

For some transactional functions that handle the PDF or CSV file, in counting DET, we did not count the inside of the files and so the value of DET becomes lower. To cope with this problem, the syntax information file should include the detail information about the variables in the class.

6. CONCLUSIONS

We have intended to examine the possibility to measure

function point from source code automatically. Here, we have proposed detailed function point measurement method for object-oriented program and developed the function point measurement tool. Then, we have applied the tool to practical Java programs in a computer company and examined the difference between the function point values obtained by the tool and those of an function point measurement specialist. As the results, we got the similar number of data and transactional functions. However, in the classification of functions, there exists some differences between the tool and the specialist.

In the case study, the target Java program did not include EQ. So, it is necessary to apply the proposed tool to other programs which include EQ. Simultaneously, we are going to revise the rule of the classification of each function as described in Section 5 and pursuit the possibility of automatic function point measurement from source code.

7. ACKNOWLEDGMENTS

This research was supported in part by Grant-in-Aid for Encouragement of Young Scientists (No: 12780220), Japan Society for the Promotion of Science. The tool used in the case study was developed by RISE(Research Institute of Software Engineering) under support from IPA's (Information-technology Promotion Agency) "support program for young software researchers".

8. REFERENCES

[1] A. Abran, P. N. Robillard: "Function point analysis: An empirical study of its measurement processes", *IEEE Transactions on Software Engineering, 22(12),* pp.895-909(1996).

[2] A. J. Albrecht: Function point analysis, *Encyclopedia of Software Engineering, 1.* John Wiley & Sons (1994).

[3] V. R. Basili and K. Freburger: "Programming measurement and estimation in the Software Engineering Laboratory", *Journal of Systems & Software, 2*, pp. 47-57 (1981).

[4] B. W. Boehm: *Software Engineering Economics*, Prentice-Hall(1981).

[5] Common Software Measurement International Consortium, *COSMIC-FFP Version 2.0* (2000). http://www.cosmicon.com/.

[6] IFPUG. 2000. *Function Point Counting Practices Manual, Release 4.1.* International Function Points Users Group.

[7] C. Jones: *Applied Software Measurement*, McGraw-Hill(1996).

[8] B. A. Kitchenham: "The problem with function points", *IEEE Software, 14(2):*, pp. 29-31 (1997).

[9] S. Kusumoto, K. Inoue, T. Kasimoto, A. Suzuki, K. Yuura and M. Tsuda: "Function Point Measurement for Object-Oriented Requirements Specification", *Proc. of International Computer Software and Applications Conference*, pp. 543-548(2000).

[10] G. C. Low and D. R. Jeffery: "Function points in the estimation and evaluation of the software process", *IEEE Transactions on Software Engineering, 16(1),* pp. 64-71(1990).

[11] C. J. Lokan: "An empirical study of the correlations between function point elements", *Proc. of the 6-th International Symposium on Software Metrics,* pp. 200-206 (1999).

[12] C. Symons: *Software Sizing and Estimating*. John Wiley & Sons (1991).

[13] T. Uemura, S. Kusumoto and K. Inoue: "Function point analysis for design specifications based on the Unified Modeling Language", *Journal of Software Maintenance and Evolution*, Vol. 13, No. 4, pp.223-243 (2001).

[14] C. E. Walston and C. P. Felix: "A method of program measurement and estimation", *IBM Systems Journal, 16(1),* 54-73(1977).