Research

Function-point analysis using design specifications based on the Unified Modelling Language

Takuya Uemura, Shinji Kusumoto*† and Katsuro Inoue

Graduate School of Engineering Science, Osaka University, Osaka, Japan

SUMMARY

Function-point analysis was introduced to help measure the functionality of software systems. For more than a decade, function points have been widely used to measure the size of information systems, often as a part of estimating the effort required for software development and maintenance processes. Limiting the use of function point measurement have been concerns about variable judgements on the part of the personnel doing the measurement, yielding differences in function-point measures for the same software product even in the same organization. Also, if an organization tries to introduce function-point analysis, the process normally starts with measurements from the organization’s own past software products—a time consuming task with start-up costs. In this paper, we propose detailed function-point analysis measurement rules using design specifications based on the Unified Modelling Language and describe a function-point measurement tool, whose inputs are design specifications developed on Rational Rose®. Then in this paper, we report tool validation work on software involved in software evolution at an organization where we have applied the tool to actual design specifications and examined the differences between the function point values obtained by the tool and those of an experienced function point measurement specialist at the organization. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: project sizing; object-oriented software; program comprehension; empirical study; software metrics; automated FP analysis

1. INTRODUCTION

As the size and the complexity of software increases, it becomes increasingly important to maintain and develop high-quality software cost-effectively within a specified period. In order to achieve this

*Correspondence to: Dr. Shinji Kusumoto, Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 560-8531, Japan.
†E-mail: kusumoto@ics.es.osaka-u.ac.jp

Contract/grant sponsor: Telecommunications Advancement Foundation and Grant-in-Aid for the Encouragement of Young Scientists; contract/grant number: 12780220

Received 9 November 2000
Revised 4 June 2001

Copyright © 2001 John Wiley & Sons, Ltd.
goal, the entire software evolution process from initial development to final retirement needs to be managed based on effective project plans.

In order to construct a specific project plan, it is essential to estimate various undesirable phenomena which happened during the project and take measures to prevent them in advance. The common subjects of estimation are size, effort invested, person-time, calendar time, technology and resources used, and quality of product. In particular, development effort is usually the most important issue. So far, several effort models such as [1–3] 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 unit of measure for size. However, using LOC for software size assessment has difficulties because the definition of LOC is very vague and LOC depends on the programming language [4].

LOC is less useful, furthermore, as a size measure in the maintenance phase of the software life cycle. Many research studies have reported that large companies spend a lot more on maintenance of their existing systems [5–6] than on their development. Historically, three different types of software maintenance have been recognized: corrective, adaptive and perfective [6–8]. It is generally said that most maintenance is perfective maintenance, and that it includes all changes, insertions, deletions, modifications, extensions and enhancements made to a system to meet the evolving and/or expanding needs of the user [7]. Maintenance processes need to be managed based on the amount of change made to the system.

Function points (FPs) are a measure of software size that uses logical functional terms that business managers and system users readily understand [9]. Since FPs measure the functionality, the measured size stays constant despite the programming language, design technology or information-system skills involved. Also, FPs are available early in the maintenance and development processes, making its use opportune during the planning and design of software projects. Up to the present, various FP analysis (FPA) versions based on the Albrecht’s original version have been proposed. The IFPUG (International Function Points Users Group) version [10], the MarkII version [11] and the COSMIC version [12] have been frequently used. Many studies on FPs have been reported in conferences and journals, as for example [13–15].

However, several unsolved problems still remain. One of them is that differences for the same specification or software product may occur, even in the same organization, since FP measurement involves judgments on the part of the measurer [16]. For example, Low and Jeffery [17] reported a 30% variance within one organization and more than a 30% variance across organizations. Also, when an organization introduces FP analysis, the process normally starts with measurements from the organization’s own past software products—a time consuming task with start-up costs. In order to arrive at a consistent value, it is important to automate the FP measurement process [9]. In the past, FP measurement could not be completely automated because specifications were not machine readable. Since CASE (computer-aided software engineering) tools are now providing machine-readable specifications, it becomes possible to develop a FP measurement tool for the specifications produced by a specific CASE tool. As a general approach, Paton and Abran have proposed a formal notation for the rules of FP analysis [18]. If the specifications on several case tools can be transformed to the formal notation, FPs can be measured unequivocally.

Recently, many organizations have started to introduce object-oriented technology into their software maintenance and development environments. However, this has created new challenges for organizations and researchers using software metrics as a tool for managing the software and the process [19]. That is, it is necessary to establish the method to calculate the software metrics from
the object-oriented software. For example, with respect to FP analysis, Caldiera et al. proposed FP-like metrics for object-oriented software [20].

We deal with automatic FP measurement for object-oriented software. First of all in this paper, we propose detailed FPA measurement rules for the design specifications described by the Unified Modelling Language (UML) and present a FP measurement tool based on these rules. In order to support the UML-based design, several CASE tools have been developed. Among them, Rational Rose® by Rational Software is the most prevalent one and it is widely used by software professionals. Thus, as the input for the CASE tool, we address the design specifications developed on Rational Rose®. Next, we apply our tool to the actual design specifications of software involved in a software evolution situation. There we compare the value obtained by our tool and that obtained by an FPA specialist in order to examine the difference between them. Finally, we examine the application of our tool to maintenance processes.

Hence, Section 2 overviews the IFPUG version of FP analysis, and presents several UML diagrams. Section 3 proposes detailed rules to measure FPs from UML diagrams. Section 4 describes a FP measurement system based on the proposed rules and briefly describes a case study that evaluates the maintenance applicability of our system. Finally, Section 5 concludes this paper.

2. PRELIMINARIES

2.1. FPA

FPA measures the functionality provided by a particular software [10]. It can be determined from the requirements specification, or from the design specification, or from the program code. Since FPs measure functionality, they (unlike LOC) should be independent of the technology and language used for designing and implementing the software and any changes to the software. COCOMO II [21], a recent version of a COCOMO cost model, uses FPs as the cost factor.

Albrecht first proposed FPA [9]. Albrecht’s FPs are computed by counting the following software characteristics:

- external inputs and outputs,
- user interactions,
- external interfaces, and
- files used by the system.

Each of these counts is then individually assessed for complexity and given a weighting value which varies from three (simple) to 15 (complex).

Albrecht’s FPs have been widely used but have some weaknesses. To address these, many kinds of FPs, such as the IFPUG [10], 3D Function Points [15], Feature Points [22], MarkII [11] and COSMIC method [12], have been proposed. In this paper, we use the IFPUG version since compared to other versions, it provides more detailed coverage of the procedures and rules for FP counting.

2.2. IFPUG version

The IFPUG version is based upon but modifies Albrecht’s rules and procedures. In the modification, the evaluation of the functional size of the software is objectively established and the rules of the counting
procedures are also described minutely and precisely. The IFPUG version has been widely used and become one of the de facto standards for FPA.

In the IFPUG version, the counting procedure of FPA consists of the following seven steps [10]. Appendix A provides some supporting explanation of each step.

**Step 1.** Determine the type of FP count.

**Step 2.** Identify the counting boundary.

**Step 3.** Count data function types.

**Step 4.** Count transactional function types.

**Step 5.** Determine the unadjusted FP count.

**Step 6.** Determine the value adjustment factor.

**Step 7.** Calculate the final adjusted FP count.

2.3. The UML

UML [23] was developed to provide a common language for object-oriented modelling. It was designed to be extensible in order to satisfy a wide variety of needs, and was also intended to be independent of the particular programming language and development methods.

The concrete syntax of UML is dominated by a graphical notation. UML defines several different diagrams that are divided into three categories: static structure, behaviour and implementation diagrams.

Static structure diagrams describe the structure of the system and include class and object diagrams. Behaviour diagrams describe the behaviour/dynamic perspective of the system and include use-case, interaction, sequence, collaboration, state and activity diagrams. Implementation diagrams provide information about the actual source code and include component and deployment diagrams.

In order to calculate FPs from UML diagrams, we elected to use the sequence and class diagrams, because they show the functions and the data manipulated in the system. In Sections 2.4 and 2.5 we briefly explain the class and sequence diagrams [24].

2.4. Class diagrams

Class diagrams describe the static structure of the model—that is, the objects and classes, and the relations between these entities including generalization and aggregation. They also represent the attributes and operations of the classes.

For example, Figure 1 shows the class diagrams for Person, Student and Teacher that are used in the office system in a university. In Figure 1, the class Person has two attributes (Name and Address) and four operations (SetName, SetAddress, GetName and GetAddress). Other classes, Student and Teacher, inherit the data in the parent class Person.
2.5. Sequence diagrams

A sequence diagram shows an interaction arranged in a time sequence. In particular, it shows the objects participating in the interaction by their 'lifelines', and the messages that they exchange arranged in a time sequence. The sequence diagram represents an interaction taking the form of a set of messages exchanged among objects within a collaboration to effect a desired operation or result. It does not show time associations among the objects. Sequence diagrams show the explicit sequence of messages and are better for real-time specifications and for more complex scenarios.

The sequence diagram has two dimensions: vertical, which represents time, and horizontal which represents different objects. Normally, time proceeds down the page. There is no significance to the horizontal ordering of the objects. Objects can be grouped into 'swimlanes' on a diagram.

Figure 2 shows an example of a sequence diagram of the office system in a university. Here, there are nine messages and the set of these messages corresponds to the following processing: (1) a student hands the registration sheet for the desired classes to the receptionist in the office of the university; (2) the receptionist inputs the data into the database through the Registration system; and (3) the student receives the result of the registration.
3. PROPOSED COUNTING RULES

3.1. Overview

Our purpose is to calculate unadjusted IFPUG FPs. We propose the following five steps to apply the first five steps of the IFPUG version to the requirements/design specifications based on the UML class and sequence diagrams.

**Step 1—select the type of FP count.** We begin with handling only the development project type, as discussed later.

**Step 2—identify the counting boundary.** The counting boundary is determined by the type of objects that appear in the sequence diagrams. That is, actor objects (defined in Section 3.2) are outside the boundary and other objects are inside the boundary.

**Step 3—count the data-function types.** Data function types are automatically decided based on the data in the class and on the sequence diagrams, according to the rules explained in Section 3.2.

**Step 4—count the transactional-function types.** Transactional function types are automatically decided based on the data in the class and on the sequence diagrams according to the rules explained in Section 3.3.
Step 5—calculate the unadjusted FP count. Using the results of Step 3 and Step 4, the counts for each-function type are automatically classified according to complexity and then weighted. The total of all of the function types is the unadjusted FP count.

As a practical matter, Steps 3 and 4 (counting data function types and counting transactional functions) are done together. In the following sections, we describe them separately.

3.2. Rules for counting data function types

Based on the definitions of data function (DF, in Appendix Step 3), we propose the following rules to extract the data functions from the class and sequence diagrams. Here, we classify the objects into ‘actor’ and ‘non-actor’ objects. Since actor objects exist outside of the application, they should not be regarded as data functions.

Step 3.1—select the candidates for data functions. From the sequence diagrams, we select the objects that have some attributes, and exchange data, with non-actor objects, as the candidates for data functions. All sequence diagrams are searched to identify a list of unique candidates for data functions.

Step 3.2—determine function type. For each of the candidates selected in Step 3.1, we determine the function type. Objects that have attributes changed by the operations of other objects are regarded as internal logical files (ILFs) and others are regarded as external interface files (EIFs).

Based on the sequence diagrams, it is decided whether such an operation (message) changes the attributes of another object. That is, in determining the DF type of the object, if, among all transactional functions that are related to the object, there exists a transactional function that receives data from an external input (EI), then the object is regarded as an ILF. If not, then the object is regarded as an EIF.

Step 3.3—judge the complexity of the data function. The complexities of the ILF and EIF are determined by the data element type (DET) and the record element type (RET). Since the DET is a unique user-recognizable non-recursive field in the ILF or EIF, we count the number of attributes of the corresponding class. If the class is derived from another class, the number of attributes from the inherited class is also added. Unlike the DET, the RET cannot be counted from the class and sequence diagrams. However, from our previous experience, the RET is almost always 1 in the requirements/design specifications, so we treat RET as a constant with a value of 1. Finally, the functional complexity is rated based on the RET/DET complexity matrix (see Table AIV in Appendix A).

3.3. Rules for counting transactional function types

In accordance with the IFPUG rules, we regard each of the messages exchanged by the object specified as a data function in sequence diagrams, as a candidate for a transactional function. If a message has no arguments, we consider that it does not exchange data and therefore that it is not a transactional function. In order to count the transactional functions from the class and sequence diagrams, we assume here the following conditions.
Condition 1. Assume that the sender object sends a message to another object (the receiver object) in the sequence diagram. If the receiver object returns a meaningful message to the sender object, we assume that the reply message must be precisely described in the sequence diagram. According to the definition of the sequence diagram in UML, this description is not necessary.

Condition 2. Data exchange must be written as the arguments of the messages in the sequence diagram.

Condition 3. When an argument of the message is the same as the sender object’s attribute, the data stored in the argument is simply copied from the sender object.

Condition 4. If the same messages appear repeatedly in the sequence diagrams, each of the arguments and message names must be the same.

For each actor object in the sequence diagrams, we apply the following two steps to count the transactional function types. These steps are based on the fact that transactional function types can be determined by renewal, or by reference to data functions, or by comparison of the data elements output.

Step 4.1—select candidates for transactional function. List the sequence of messages in which the first message is sent by the actor object and the last message is received by the actor or non-actor object in the sequence diagram. For example, in Figure 2, a sequence of messages (1, 2, 3, 4, 5, 6, 7, 8 and 9) is listed.

Step 4.2—determine the type of transactional function. For each sequence listed in Step 4.1, use the following five patterns to determine the type of the transactional function and its complexity—i.e. the DET and file types referenced (FTR—see Appendix A).

Pattern 1. An actor object sends a message to a DF, such as shown in Figure 3. We usually regard this pattern as an external input (EI). However, if no arguments are given for the message, we do not regard it as EI. DET is the number of arguments for the message. The FTR is 1 since there is one DF.
**Pattern 2.** A DF sends a message to an actor object, such as shown in Figure 4. If all the arguments of the message are the same as the attributes of the DF, we regard it as external inquiry. Otherwise, it means that the message contains derived data and we regard it as external output. DET is the number of arguments of the message. FTR is 1 since there is one DF.

**Pattern 3.** An actor object sends a message (‘message1’) to a DF, and returns a message from the DF to the actor object, such as shown in Figure 5. In this case, we pay attention to a ‘message2’. If all the arguments of ‘message2’ are the same as the attributes of the DF, then we regard the two messages (‘message1’ and ‘message2’) as one external inquiry. Otherwise, it means that ‘message2’ contains an elementary process and we regard the two messages as one external output. For example, we consider that ‘message1’ from the actor is the input of the key for data retrieval. DET is the number of arguments of the output message (‘message2’). FTR is 1 since there is one DF.
Pattern 4. An actor object sends a message to a DF and the DF sends a message to another actor object, such as shown in Figure 6. We divide this into two transactional functions. That is, Pattern 4 is the combination of Pattern 1 and Pattern 2.

Pattern 5. An actor object sends a message to a DF, and finally the actor object receives the reply message through several DFs, such as shown in Figure 7. Pattern 5 handles a sequence of messages. This is the case in which, when an actor object sends a message,
the reply message comes through several DFs. For example, in Figure 7, a sequence of five messages (messages 1, 2, 3, 5 and 6) are candidates for a transactional function. In this case, we apply Pattern 3 to ‘message1’ and ‘message6’. In Figure 7, since ‘message4’ is not included in the sequence, we apply another pattern (in this case, Pattern 2) separately to that message. DET is the number of arguments of the output message (‘message6’). FTR is 2 since there are two DFs in the sequence.

3.4. Related works

Several researchers have already proposed measuring FPs from object-oriented design specifications, as for example, the work done at the University of Quebec [25,26]. In [25], they have defined the mappings from UML design specifications to FP counting notations. For example, functional processes, sequence of subprocesses, and data groups correspond to use case diagrams, scenarios, and class diagrams. Usually, use case diagrams are firstly developed. So, the approach can be applied to the early design phases of maintenance and development. With respect to the mapping rules, our proposed approach is similar to that in [25].

On the other hand, we only use the sequence and class diagrams and impose several assumptions on them. So, the applicability of our method is less than that of the method in [25]. However, since our main objective is to measure FPs automatically, we consider that the assumptions are unavoidable to count the FPs correctly. So, our method is one of the candidate ones to apply with the method in [25] on some specific project in order to measure FPs automatically. The two methods should complement each other.

4. OUR TOOL AND ITS APPLICATION

4.1. Outline of the tool

We have designed and implemented a software tool that can count FPs from the design specifications developed by Rational Rose® version 4.0 and Rational Rose® 98 using the Visual C++ language on Windows 98. The inputs for the tool are sequence diagrams and class diagrams produced from Rational Rose® and the output includes the unadjusted FPs, transactional functions, data functions and objects that may be related to the FPs calculation.

Figure 8 shows the system structure of the tool. The system consists mainly of three units and two databases: analysis unit, analysis database, counting unit, counting database, and interface unit.

The analysis unit does a syntax analysis of the input sequence diagrams and class diagrams from Rational Rose®, extracts data functions and stores them in the analysis database. The counting unit calculates the value of the FPs using the data in the analysis database and stores them into the counting database. The interface unit shows the measurement results to the user. These results include the values of the FPs, and the candidates for the data functions and transactional functions.

The user can supervise the action taken by the tool. Usually, some details of the design specifications (number of attributes, arguments of message, etc.) are not described in the early design phase. In such a case, the user can calculate the FPs for the ‘rough’ design specification for the time being, and the results can be refined through the interface unit. For example, non-actor objects not regarded as data
functions by our tool, but which in fact, should be treated as data functions in the later design phase, can be changed into data functions and the user can have the tool recalculate the FPs. This capability is very useful in practice.

4.2. Case study 1

In order to evaluate the appropriateness of the FP values calculated by the tool, we applied the tool to the following design specifications provided in UML by Rational Rose®.

- Purchase processing system: creates and outputs several documents involved in the purchase of office equipment.
- Order processing system: successfully estimates the price for the office equipment and decides upon the source from which it is to be ordered.
- Stock control system: controls the stock of office equipment and makes production plans.

We compared the FP values obtained by our tool to those obtained by the FP counting specialist. Table I shows the number of diagrams, messages and objects in these design specifications. The purchase processing, order processing and stock control systems are relatively small in size.

Table II shows the FP values of the three systems. As can be observed, all FP values obtained by the tool are smaller than those of the specialist. These differences are caused by the fact that in the several sequence diagrams there is only one external inquiry, but the specialist considered that this transactional function would consist of two transactional functions and counted two external inquiries. For example, in Figure 9, our tool identified the messages ‘2: Confirm a purchase order’ and ‘3: Display a purchase order (Purchase order)’ as the external inquiry based on Pattern 3. However, the specialist considered that the staff in charge of purchasing gets the list of purchase orders by sending message 2, and then selects the target purchase order. So, the specialist extracted two external inquiries from messages 2 and 3. If the detailed descriptions had existed in the sequence diagrams according to our
Table I. Data describing four case study systems.

<table>
<thead>
<tr>
<th>System</th>
<th>No. of diagrams</th>
<th>No. of messages</th>
<th>No. of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase processing system</td>
<td>4</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>Order processing system</td>
<td>6</td>
<td>73</td>
<td>12</td>
</tr>
<tr>
<td>Stock control system</td>
<td>1</td>
<td>17</td>
<td>10</td>
</tr>
<tr>
<td>Monitoring</td>
<td>8</td>
<td>127</td>
<td>25</td>
</tr>
</tbody>
</table>

Table II. FP values for the three small systems.

|                | FP by specialist | FP by tool |              |              |              |
|----------------|------------------|------------|--------------|--------------|
|                | Purchase Order Stock | Purchase Order Stock |              |              |
| DF             | 14               | 14         | 26           | 26           |
| TF             | 18               | 15         | 63           | 47           | 33           | 30           | 30           |
| FP             | 32               | 29         | 92           | 76           | 59           | 56           |

Figure 9. Sequence diagram from the purchase processing system.
Table III. FP values for the monitoring system.

<table>
<thead>
<tr>
<th>FP type</th>
<th>Specialist</th>
<th>Initial</th>
<th>Refined</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILF</td>
<td>28</td>
<td>21</td>
<td>28</td>
</tr>
<tr>
<td>EIF</td>
<td>45</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>EI</td>
<td>24</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td>EO</td>
<td>22</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>EQ</td>
<td>49</td>
<td>3</td>
<td>49</td>
</tr>
<tr>
<td>Unadjusted FP</td>
<td>168</td>
<td>128</td>
<td>168</td>
</tr>
</tbody>
</table>

assumption in Condition 1, our tool would have counted the transactional functions in the same way as the specialist did.

4.3. Case study 2

We also applied our tool to another design specification done on Rational Rose®. It is a relatively large monitoring system for the collection and delivery of business data for retail stores. It includes the following functions.

- extract the delivery data from the operating log of the system;
- display the status of collecting and delivering for all of the retail stores; and
- display the status of collecting and delivering for any specific store, etc.

We compared the FP value obtained by our tool with the value provided by the FP counting specialist. Table I shows the number of diagrams, messages and objects in the specification. Table III shows the FP calculation results for the monitoring system. As observed in Case study 1, all FP values obtained by the tool are smaller than those of the specialist. Also, with respect to the type of function, there are some differences in the data and transactional functions. We analysed the differences and found the following causes to which they could be attributed.

**Cause 1.** In describing a process in which some messages are output to the display screen, the messages from the screen to the actor object are sometimes not described in the UML sequence diagram. Figure 10 shows an example; there are no messages between an actor ‘Warden’ and an object ‘Display of detailed data type’. Thus, Condition 1 from Section 3.3 is not satisfied, but the specialist has judged that there is one EQ or EO by complementing the omitted messages. On the other hand, our tool cannot automatically complement them and has considered the messages 1 and 2 as internal processing.

**Cause 2.** The specialist regarded the automatic processing by the scheduler in the system as a transactional function. Since the scheduler is a non-actor object, our tool regarded the processing between the scheduler and the DF as an internal processing. However, the specialist regarded
For Cause 1, if the designer writes the messages to the actor object and writes the arguments, the tool can correctly count the function. In the case of omission, if the display screen is regarded as an actor, our tool can count the function. Also, for Cause 2, if we describe the scheduler as an actor, our tool can also regard the messages that are exchanged between the scheduler and the DF as transactional functions.

After we modified the specification as described above and had our tool recalculate the FPs, it obtained the same values as those provided by the FP specialist, not only at the total level but also the subtotal level.

4.4. Application to maintenance processes

In the two cases reported above and used for validation of our tool, we used the development project type of FP count in order not to bias the comparisons with the specialist’s measurements. The two cases are systems undergoing software evolution, and have documentation available in UML via Rational Rose®. Our measurements and those of the specialist focus on the system being maintained, not on the change being made. Also, our focus is on the UML available for object-oriented systems from the design process, a process present in both the software development life cycle (SDLC) and the software maintenance life cycle (SMLC) [27].
We agree that a key difference between the SDLC and the SMLC is the much greater effort in the SMLC on program understanding [27]. For object-oriented software with UML support, using our automated tool offers the potential of a reduction in the program understanding effort. This reduction may be quantitatively measurable and useful in the major types of software maintenance and evolution [7].

In the planning of a new release during software evolution, the IFPUG enhancement project FP type noted in Appendix A is applicable [12]. The enhancement project FP calculation recognizes three components of functionality: the application functionality included in the user requirements for the project, the conversion functionality, and the application value adjustment factor. Here, even as it now stands, our tool can be used to measure the application functionality. Measuring that functionality consists of counting the FPs for three aspects of the project.

- The functionality added by the project enhancements,
- The existing functionality changed during the project; and
- The functionality deleted during the project.

If these functionalities are described in UML sequence and class diagrams, our tool can count the corresponding FPs. The counted FPs could be used in the maintenance project planning and in focussing the program comprehension effort.

5. CONCLUSIONS

In this paper, we have proposed detailed FP analysis rules for a design specification documented in UML class and sequence diagrams. Based on the proposed rules, we have produced a FP measurement tool to automate the FP analysis process. We have validated the tool on several design specifications on typical business applications undergoing software evolution. Initially, the values provided by our tool were consistently smaller than those provided by a specialist in FP analysis. The differences mostly came from assumed conditions not being satisfied. After refinement, the values calculated by our tool were nearly the same as those obtained by the specialist in IFPUG FP analysis.

As described in Section 3.3, we have assumed four conditions in the UML diagrams in order to measure FPs automatically from the UML documented specifications. Assuming conditions on the counting process has been traditional in FP work [10], and seems to us as to some degree inevitable in order to enforce consistency in the measurement process. As far as we know, such an approach in some form is used in most organizations to measure FPs from the specifications, even in manual FP counting.

During the design process, whether in development or maintenance, UML class and sequence diagrams may not be available in some organizations, although other UML documents are available. Since class and sequence diagrams can be produced from use case diagrams, these may offer an alternative source.

Our future research work is expected to include exploring the use of other UML diagrams, such as use case diagrams, as input to our tool, as well as modifying our tool to cope with newer versions of UML, IFPUG and Rational Rose®. We expect to validate our tool additionally on specific enhancive-type maintenance projects in different organizations.
### APPENDIX A. IFPUG COUNTING STEPS

**Step 1: determine the type of FP count.** Select the type of FP from the following three: (1) development project FP count, (2) enhancement project FP count and (3) application FP count.

**Step 2: 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 FP count.

**Step 3: count data function types.** Data function types represent the functionality provided to the user to meet internal and external data requirements. There are two such types, classified as ILF and EIF. The respective three criteria for each are as follows.

- **ILF.** (1) The group of data is a 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 this application.

- **EIF.** (1) The group of data is a 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 this application.

Here, the term ‘file’ refers to a logically related group of data and not to the physical implementation of those of data or of the group. First, assign to each identified ILF or EIF a functional complexity value based on the number of DETs and RETs associated with the ILF or EIF using the RET/DET complexity matrix (see Table A1). A DET is a unique user-recognizable, nonrecursive field in the ILF or EIF. A RET is a user-recognizable subgroup of data elements within an ILF or EIF.

**Step 4: 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 in three types:

- **EI.** An external input is an elementary process receiving control data, or data for processing, that comes from outside the application’s boundary.

- **EO.** An external output is an elementary process that generates control data, or data for processing that are sent outside the application’s boundary.

---

Table A1. RET/DET complexity matrix.

<table>
<thead>
<tr>
<th>RET/DET</th>
<th>1–19</th>
<th>20–50</th>
<th>51+</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Low</td>
<td>Low</td>
<td>Average</td>
</tr>
<tr>
<td>2–5</td>
<td>Low</td>
<td>Average</td>
<td>High</td>
</tr>
<tr>
<td>6+</td>
<td>Average</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

---

Table AI. FRT/DET complexity matrix for EI.

<table>
<thead>
<tr>
<th>FRT/DET</th>
<th>1–4</th>
<th>4–15</th>
<th>16+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–1</td>
<td>Low</td>
<td>Low</td>
<td>Average</td>
</tr>
<tr>
<td>2</td>
<td>Low</td>
<td>Average</td>
<td>High</td>
</tr>
<tr>
<td>3+</td>
<td>Average</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Table AII. Unadjusted FP calculation table.

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
</tr>
<tr>
<td>ILF</td>
<td>$\times 7 = \Box$</td>
</tr>
<tr>
<td>EIF</td>
<td>$\times 5 = \Box$</td>
</tr>
<tr>
<td>EI</td>
<td>$\times 3 = \Box$</td>
</tr>
<tr>
<td>EO</td>
<td>$\times 4 = \Box$</td>
</tr>
<tr>
<td>EQ</td>
<td>$\times 3 = \Box$</td>
</tr>
</tbody>
</table>

Unadjusted function point

An EQ (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 are data that require processing other than direct retrieval and editing of data from internal logical files and/or external interface files. No internal logical file is maintained during the processing.

At this stage, assign each identified EI or EO a functional complexity value based on the number of FTR and DET. A FTR is (1) an ILF 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 value based on the number of FTR and DET 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 FPs. For each of EI, EO and EQ, there is a FTR/DET complexity matrix. Table AI shows the FTR/DET complexity matrix for EI.

**Step 5: Determine the unadjusted FP count.** The counts for each function type from Step 3 and Step 4, are classified according to complexity and then weighted using Table AII. The total of all the function types is the unadjusted FP count.

**Step 6: Determine the value adjustment factor.** The value adjustment factor (VAF) indicates the general complexity of the application functionality provided to the user, and may range from 0.65 through 1.35. The VAF takes into account fourteen general system characteristics (GSC) as summarized in Table AIII that address different complexity aspects of the specific functionalities of the application. Each characteristic has associated descriptions that help assess the degree of influence...
Table AIV. Influencing general system characteristics.

<table>
<thead>
<tr>
<th>Article</th>
<th>GSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data communications</td>
</tr>
<tr>
<td>2</td>
<td>Distributed data processing</td>
</tr>
<tr>
<td>3</td>
<td>Performance</td>
</tr>
<tr>
<td>4</td>
<td>Heavily used configuration</td>
</tr>
<tr>
<td>5</td>
<td>Transaction rate</td>
</tr>
<tr>
<td>6</td>
<td>Online data entry</td>
</tr>
<tr>
<td>7</td>
<td>End-user efficiency</td>
</tr>
<tr>
<td>8</td>
<td>Online update</td>
</tr>
<tr>
<td>9</td>
<td>Complex processing</td>
</tr>
<tr>
<td>10</td>
<td>Reusability</td>
</tr>
<tr>
<td>11</td>
<td>Installation ease</td>
</tr>
<tr>
<td>12</td>
<td>Operational ease</td>
</tr>
<tr>
<td>13</td>
<td>Multiple sites</td>
</tr>
<tr>
<td>14</td>
<td>Facilitate change</td>
</tr>
</tbody>
</table>

of that characteristic. Assign for each of the 14 GSC items a degree of influence on a scale of 0 to 5, for from no influence to very strong influence. Then sum the 14 degrees of influence to get the total degrees of influence, that may range from 0 to 70. Calculate the VAF as 0.65 plus one % the total degrees of influence. For enhancement projects, the VAF is calculated for the system both before and after the maintenance changes.

**Step 7: Calculate the final adjusted FP count.** The final adjusted FP count is calculated using a specific formula based upon the type identified in Step 1. For a development project, the adjusted count is just the product of the VAF times the unadjusted FP count for the entire system. For an enhancement project, the adjusted FP count is the ‘after’ VAF times the sum of the unadjusted counts for the added and the modified parts of the system, plus the ‘before’ VAF times the unadjusted count for the deleted parts of the system.

**ACKNOWLEDGEMENTS**

We thank Mr. Katuhiko Yuura and Ms. Ayane Suzuki of Hitachi, Ltd., and Mr. Michio Tsuda, Mr. Hiroshi Fukuhara and Mr. Kazuhiro Hama of Hitachi System and Service, Ltd., for their discussions and advice on this paper. We gratefully acknowledge the valuable comments and suggestions of the referees. This research was supported in part by a grant from the Telecommunications Advancement Foundation and Grant-in-Aid for the Encouragement of Young Scientists, number 12780220.

**REFERENCES**

AUTHORS' BIOGRAPHIES

Takuya Uemura currently works for Sony Corporation. Takuya received the ME degree in Information and Computer Sciences from Osaka University in 2000. E-mail: uemura@ics.es.osaka-u.ac.jp

Shinji Kusumoto is currently an Associate Professor at Osaka University. His research interests are software metrics and software quality assurance. He is a member of the IEEE, IPSJ, IEICE and JFUG. Shinji received the BE, ME and DE degrees in Information and Computer Sciences from Osaka University in 1988, 1990 and 1993, respectively. E-mail: kusumoto@ics.es.osaka-u.ac.jp

Katsuro Inoue is a Professor at Osaka University, where he has been a faculty member since 1989. Previously, he was an Associate Professor at the University of Hawaii at Manoa from 1984 to 1986. Katsuro is engaged in the study of software engineering, and is a member of the IEEE, ACM, IPSJ, IEICE and JSSST. He received the BE, ME and DE degrees in Information and Computer Sciences from Osaka University in 1979, 1981 and 1984, respectively. E-mail: inoue@ics.es.osaka-u.ac.jp