An Empirical Study on Software Defect Prediction using Function Point Analysis

Xinghan Zhao[1,2] and Cong Tian[1,*]

[1]Xidian University, Xi'an, Shanxi, China, 710071

[2]the 27th Research Institute of CETC, Zhengzhou, Henan, China, 450047

zhaoxinghan@126.com, ctian@xidian.edu.cn

*corresponding author

***Abstract*—The software defect prediction method based on requirement specification is proposed to address the defect prediction needs in the requirements phase when the organization adopts the W-model of software development. The theoretical synthesis presents that the function point and the number of defects should be positively correlated. The theory's correctness is verified by analyzing the correlation between function point and defect distribution of eight software applications. Then, the mathematical equations for software configuration testing defects are derived, and the specific meaning of the equation is explained. Finally, the shortcomings of this study and the subsequent research directions are pointed out.**

***Keywords*—software defect prediction; function point analysis; software configuration testing; IFPUG;**

I. Introduction

Software defect prediction is one of the key research directions in software testing and software engineering. This is because failures caused by software defects can have very severe consequences including property damage, monetary loss, or even human casualty [34], [36]. Software defect prediction can predict the number of potential software bugs and their distribution based on the code, documents, configuration process information, etc. The results of software defect prediction are very instructive for the organization's software testing resource allocation, software product quality judgment, software process quality management, and software testing result assessment. Defect prediction methods are generally based on the size of software code [2], complexity [8], and various design parameters [11], [35]. After the preprocessing, fitting, and regression of the inherent properties of software and process information, a targeted prediction model is formed. The model predicts the likelihood of the distribution of defects in the target software.

Software configuration testing is the process of testing software configuration items' functionality, performance, and other characteristics. This type of testing is typically performed by an organization-level software testing department and always occurs when the development team has completed the code work and unit testing. Configuration testing is crucial throughout the software life cycle as the last step of the organization's software quality control. In the traditional waterfall model [3], the overall activity of configuration testing begins when the source code of SUT (software under test) is fully developed. The software testing member can use the source code and the process information from the configuration management system to predict the number and the distribution of defects. However, in the present day, to increase the software tester's involvement, the V-model, or the more explicit W-model [32] always be chosen. In W-model, testers' activity tends to shift left [21]. In this model, the configuration testing plan starts in the requirements phase of the project. Testers must decide on test strategies and allocate test resources and complete the main test case design work before the requirements phase is completed. Unlike the Waterfall and V-model, the design and coding work is not carried out then. Software defect prediction based on source code or process information is impossible.

This study proposes a method for predicting software defects using software requirement specifications as input. Thus, the managers can obtain software fault prediction results in the requirement phase, which may be helpful for software activities such as software quality assurance and test resource allocation. In terms of input parameter selection, function point for every requirement item are used to calculate which requirement item is more likely to have bugs. Because we didn't find an open-source database that we could use, we chose eight software projects from our organization's historical software repository. These software projects were randomly selected. All function point for each requirement item have been analyzed with the IFPUG method [16]. The configuration tests' bugs have been selected from the software configuration testing reports. This study involves 374 functional items, 5011.24 functional points, and 233 software defects. The results show that the probability of occurrence and number of functional errors of software configuration items positively correlate with the number of function point of software requirements. Then we use the least-squares method with curve fitting to calculate a mathematical model for defect risk calculation. The mathematical model can use the function point of each requirement as input and calculate the relative risk level of defects. At last, the differences between the actual data and the theory are given, and our insights are provided.

The subsequent chapters are organized as follows: Chapter 2 gives a brief description of the development of defect prediction and related research works; Chapter 3 introduces the theoretical basis of this study; Chapter 4 presents the information of the selected validation software and the calculation of function point, as well as the difficulties encountered in data collection and our solutions; Chapter 5 discusses the relationship between the final function point and defects, gives the fitting equation and analyze the engineering significance of each parameter of the equation; Then, we give some results of data distribution in Chapter 6. Chapter 7 explores the reasons

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00027

for the data distribution in the context of the research. Chapter 8 describes the follow-up research plan. Chapter 9 shows the summary and outlook of this study.

II. Related Works

The original defect prediction methods are mainly based on lines of code and their complexity-derived metrics. In 1971, Akiyama proposed a measure of complexity based on lines of code and used this for defect prediction [2]. Akiyama also gave a correspondence between software defects and lines of code based on his research sample. In 1973, Ferdinand used the relevant information theory to analyze the complexity of a system, the number, and the density of defects. It shows that the larger the system size, the more possible defects [11].

There is one obvious problem with using the code line to predict defects. This method does not include the complexity of the software system. Based on Akiyama's research, Halstead proposed complexity metrics derived from operations and operands in the following years. He calculated the correlation between these metrics and defects, and he got the result that the correlation coefficient is greater than 0.9 [13]. Lipow et al. generalize based on Halstead's study by proposing a polynomial correspondence between the LOC and defects which coefficients depend on language-related operands [18]. But Lipow's conclusions still had some arguments. Gaffney proposed that the correspondence between defects and LOC is language-independent, and he uses Lipow's sample to derive a more simplified relational expression independent of the programming language [12].

Unlike the route taken by Halstead et al., who based their calculations on the metrics derived from code and operands, McCabe used a language-independent, structure-based approach to the complexity measure. He utilized a graphical method to calculate the complexity of programs by computing the circle complexity of the program's flowchart and used this as the primary metric for defect distribution prediction [8]. These studies, like Halstead and McCabe's, were prevalent around the 1980s. The effectiveness of these metrics in practice has been evaluated and compared [4].

Most of the studies in the 1970s and early 1980s focused on the fit of prediction models to existing program data. In contrast, the performance of the models and their different parameters were not validated for the new SUT. Shen et al. selected three commercial programs as test subjects to address this issue and validated the correlation between models and metrics using linear regression [30]. Munson et al. stated that the current regression model yielded inaccurate predictions of the number of defects and proposed the classification of modules into high and low-risk categories to replace the number of predictions. The classification model obtained 92% prediction accuracy on their target software [23].

The above studies are based on code, and many more studies try to establish available metrics in another way. Henry and Kafura et al. established a metric system for module complexity based on statements and data flow in design documents [15]. Ohlsson et al. implemented the automatic collection of information from design documents and made a successful defects prediction on Ericsson's telephone switch software [27]. In addition, another research direction is complexity analysis and defect prediction based on object-oriented (OO). Basili and Chidamber et al. have published their study results in this area [4], [6].

After 2000, with the popularity of version control systems such as SVN and Git, using process information to predict defects has become a new direction [24]. Moser et al. build metrics from records of code revisions, refactoring, and bug fixes, and he illustrates that the model of using change records is much more accurate than the code-based model [22].

Since 2000, machine learning has become more and more popular. Using machine learning in software defect prediction has become a new research hotspot. In the direction of cross-project prediction, machine learning has helped researchers a lot in processing and analyzing large amounts of data, and many scholars have given their own models for cross-project prediction [5], [14], [17], [19], [25], [33]. However, how to evaluate these cross-project models is also a problem that needs to be faced. Zimmermann [38] and He [14] et al. have done research in this area.

There are a lot of studies on software defect predictions, but there is little literature about how to predict defects in the requirement phase. Air force's Rome laboratory developed a model for early software reliability prediction, which is based on the software requirement specification data collection [20]. Smidts suggest a reliability prediction model based on the requirement changes during the life cycle [31]. Yadav [37] et al. propose a software defect prediction model using fuzzy logic to solve the problem of early-stage defect prediction, but he uses not only the requirement information but also the metrics about the size and historical quality information. Sangeeta et al. show a failure rate model centered on iterative software development life cycle [29]. However, all the studies above are centered on the defect distribution of the different phases of the software development life cycle across programs. They haven't answered the question of how to analyze the defects distribution across the different functions and how a software manager can predict defects when he gets a new requirement specification.

III. Theoretical Analysis

There is no suitable theory or method for predicting defects from software requirement specifications. However, it is possible to achieve it indirectly using some relevant research findings. The ideas involved are mainly in the following two areas.

Theory 1: The more function point there are, the larger LOC will be.

Function point analysis (FPA) is one of the most mature and popular methods of software size prediction today. The main FPA methods are IFPUG [16] and COSMIC [9]. Although the calculation methods of these two are different, both ways calculate function point that are proportional to LOC.

Theory 2: The larger LOC of the software module, the larger number of BUGs.

In the discussion in Chapter 2, whether it is Halstead or Akiyama, or later Lipow, Gaffney, and other subsequent studies, although the final prediction models of their studies differ, their results show that the larger the software size (LOC), the higher the number and likelihood of bugs embedded in the software modules.

By combining theory 1 and theory 2, we can draw the inference below.

Inference: The more function point of the software module, the more probability and larger the number of bugs in the software.

Using this inference, we can predict the distribution of defects in software by calculating the function point of the software requirement specification in the early stage of the software life-cycle, when only the software requirement specification is available.

IV. Data and Methods

A. Target Programs

We did not find a publicly available database about software requirement function point and defects. In order to make the experimental data more realistic, we randomly selected eight commercial programs from the organization's project database for validation. The descriptions of these projects are in Table 1.

These projects were developed by different teams and have been completely done unit tests and integration tests, and then handed over to the organization-level software testing department to complete the configuration testing. The requirements of all projects were formulated according to the relevant standards, and all functions were described in natural language. All projects have been in operation for more than one year, and no escape defects were found during the operation that was not detected by the configuration tests.

B. FPA method

Nowadays, the most widely used FPA methods are IFPUG and COSMIC [10], [26]. Although COSMIC is simpler than IFPUG, we still use IFPUG because it is the recommended method in our organization. According to the related research [1], [7], [28], the difference between the results using IFPUG and COSMIC methods is not much for the final prediction results. Since the difference brought by their methods is within our tolerance range, it does not affect our research results.

The IFPUG method is with the following steps.

1. Analyze user functional requirements The functional requirements are identified by analyzing the documentation related to the software requirements. In this study, the functional requirements do not include performance, quality, and environmental requirements.

2. Decompose functional requirements

Decompose the requirement entries according to the functional unit in Table 2, down to the smallest functional unit possible.

Internal Logic File (ILF): A set of logic-related data or control information that can be identified by the user and maintained within this software. The primary use of an internal logic file is to control data through one or more of the software's basic processes.

External Interface File (EIF): A set of logically related data or control information that can be identified by the user and referenced by software but maintained by other software. The primary purpose of the external interface file is to control data references through one or more fundamental processes of this software, i.e., the external interface file of one software should be the internal logic file of another software.

External Input (EI): A basic processing of data or control information that comes unexpectedly from the boundaries of this software. The main purpose of external input is to maintain one or more internal logic files and (or) to change the behavior of the system.

External Output (EO): A basic process of sending data or controlling information outside the boundaries of this software. The main purpose of external output is to provide information to the user through processing logic or through the retrieval of data or control information. The process should contain at least one mathematical formula or calculation, generate data everywhere, maintain one or more internal logic files, or change system behavior.

External Query (EQ): A basic processing of sending data or controlling information outside the boundaries of this software. The main purpose of the external query is to provide information to the user by retrieving data or controlling information from internal logic files in external interface files. This processing logic does not contain mathematical formulas or calculations, does not produce exported data, and the process neither maintains the internal logic files nor changes the system behavior.

3. Determine the weighting factor

The functions are divided into different levels according to high, average, or low. The level is determined by the number of data element types and the number of record element types or the number of reference file types involved in a particular function together. Different functional units are involved in different levels with different corresponding weights.

4. Calculate the number of unadjusted function point

The number of external inputs (EI), external outputs (EO), external queries (EQ), internal logic files (ILF), and external interface files (EIF) are multiplied by their corresponding weighting factors, and then the products are added together, and the result is the number of unadjusted function point (UFP).

$$UFP = N_{EI} * \theta_{EI} + N_{EO} * \theta_{EO} + N_{EQ} * \theta_{EQ} + \dots \quad (1)$$

5. Determine the adjustment factor

Each function was analyzed for system impact according to fourteen system characteristics: data communication, distributed data processing, performance, system configuration

Table 1. Experimental Validation Software

Index	Software Name	Software Type	Platform	Language	Size
P1	Display and control software	Non-Embedded	QT	C++	Medium
P2	Photoelectric tracking software	Embedded	IAR	C	Small
P3	Resource allocation management software	Non-Embedded	Eclipse	JAVA	Medium
P4	Data census analysis software	Non-Embedded	QT	C++	Medium
P5	Data customization platform software	Non-Embedded	Eclipse	JAVA	Medium
P6	Data processing and forwarding software	Embedded	Keil	C	Small
P7	Information processing and control software	Non-Embedded	VS 2015	C#	Medium
P8	Control software	Non-Embedded	VS 2015	C++	Small

Table 2. Functional Unit

Data Functions	Operation Function
Internal logic files	External Input
External interface files	External Output
	External queries

requirements, processing rate, online data entry, end-user efficiency, online updates, complex processing, reusability, ease of installation, ease of operation, multiple workplaces, and ease of change, and each system characteristic was scored on a scale of 0 to 5 for system impact, where 0 indicates no impact and 5 indicates strong impact. After that, the total impact degree was obtained by adding up all the system characteristics, and the value of the adjustment factor (VAF) was calculated by equation 2.

$$VAF = 0.65 + (\sum_{i=1}^{n} N_i/100) \tag{2}$$

Where n is the number of system performance characteristics (not limited to 14) determined based on the actual impact, and N_i is the degree of influence of the i^{th} influence factor.

6. Calculate the number of delivered function point

Multiplying the unadjusted function point (UFP) and the adjustment factor (VAF) yields the IFPUG function point (FP).

$$FP = UFP \times VAF \tag{3}$$

C. Difficulties and Solutions in Implementation

In the actual implementation process, we also found some difficulties in the implementation of the IFPUG method, the details, and solutions of which are described as follows.

1. Function point calculation for data-related functional units

The IFPUG calculation method involves five main parameters, which are external input (EI), external output (EO), external query (EQ), internal logic file (ILF), and external interface file (EIF). EI, EO, and EQ are relatively easy to analyze, but ILF and EIF may be highly subjective if the conclusion is drawn only from the textual descriptions in software requirement specifications. For example, as a certain complex logic control information, we can divide it into several small ILF, or we can calculate it to one larger ILF. However, the final result about FP would be somewhat different.

We made some adaptations for this situation when we calculate the function point. For ILF, when the requirement specification clearly shows that there is a more complex process or logical relationship, or when the process is complex according to our judgment, we will consider making the ILF level to medium or high, so this ILF's function point would be larger. Vice versa. If the requirements clearly state that the internal logic is divided into several parts or the processing is divided into several steps, we will divide the ILF into several parts according to the description of the requirements, otherwise, they will all be processed into one ILF.

2. Treatment of adjustment factors

The adjustment factor in the IFPUG method requires the estimation of 14 factors. However, in the actual implementation, we found that a large part of the impact factors recommended by IFPUG was not suitable for our project. So we did not follow the method recommended by IFPUG in the selection of impact factors but based on the understanding of the project requirement. We made an overall estimate of the adjustment factor VAF directly. We estimated a value between 0.65 to 1 for the VAF by judging the requirement in terms of performance, reliability, fault tolerance, and criticality. We should notice that this method may cause a decrease in estimation accuracy compared to the recommended method of IFPUG, but it would greatly reduce the estimation effort of VAF.

3. Handing of interfaces, performance, and other non-functional requirements

Although the interface specification description is often a very important section in the requirement document, there is no specific treatment for interface requirements in IFPUG. In the target projects we selected, the interface requirements are all external interfaces, so we merged all contents in the interface requirement specification into the relevant functional requirement entries and treated them as external inputs (EI) to the functional requirement units.

The IFPUG approach does not give an explicit treatment on performance and other non-functional requirements. We choose to reflect such elements in the adjustment factors. If a functional unit has performance or other non-functional requirements, such as special safety and reliability requirements, we adjust its adjustment factor (VAF) by adding a value of 0.05 to 0.1 to reflect the requirement.

4. Statistical methods for software defects

We use projects that have passed the organization-level testing and have operated for a long time to ensure that there are no obvious remaining defects in the software. Our defect statistics are derived from the organization's software configuration test report. Our program defects are derived from the organization's software configuration test reports. The re-

Table 3. Function Point and Defects Information

Software	Function items	Function point	error	omissions
P1	32	578.1	19	4
P2	12	196.8	2	0
P3	81	812.95	32	2
P4	62	764.53	34	4
P5	94	1264.91	92	0
P6	20	308.35	3	1
P7	58	915.4	33	1
P8	15	170.2	6	0

ports only contain the software defects related to functionality, performance, reliability, etc. which were found by the software testing department at the organization level during the software configuration testing, and do not include the defects found by the static analysis, code review, unit testing, and integration testing.

In the statistics, we also eliminated some low-level errors that existed in the software, because these defects are useless to analyze the statistical results. For example, in P3 software, all the input controllers in this program did not have a length limit, and we intentionally eliminated these defects from statistics.

In terms of classifying functional defects, this study classifies defects into two types: functional errors and functional omissions. If a function is not as expected due to incorrect design or coding, the defect is classified as a functional error, while if human negligence causes a function to be ignored in whole or in part, the defect is classified as a functional omission.

In terms of the number of defects counted, we use different treatments for different defects in the software.

1) Defects that appear separately in functional tests and have no obvious correlation with other defects are counted according to the number of defects, and each regular defect is counted as one defect. However, if there are several defects of the same kind in one requirement item, we count them as one defect. But if they are in two different requirement items, we count them as two defects.
2) These items also involve fault-tolerant, environment-adaptive, and performance-related requirements. For fault-tolerant requirements, we count the defects int the corresponding functional requirement items. In the defects statistics of this study, we ignore environmental adaptability defects and performance defects.
3) This study ignored defects classified as documentation bugs found in all inspections in the software defect statistics.

V. Function point statistics and defects distribution

A. Function point and defects data

After the calculation of function point, the number of function point and defects of the software selected for this study shows in Table 3.

The distribution of function point and defects of the target software is shown in Figure 1[1]. The horizontal coordinate is the index of the function items, the vertical axis is the function point, the red vertical axis on the left is the function error defects, and the right vertical axis is the function omission defects. The blue dots in the figure indicate the function point of the corresponding requirement items, the red dots indicate the number of functional error defects of the corresponding requirement items, and the green dots indicate the number of functional omission defects of the corresponding requirement items. In order to represent the relationship between function point, function errors and functional omissions more clearly, the requirement point in the figure are reordered from smallest to largest function point.

The relationship between defects and function point in Figure 1 can support our conclusion in Chapter 3. It is obvious from Figure 1 that the number and probability of defects in software increases as the number of function point increases. However, we also need to note that it is not always the case that more function point will result in defects; there are also some functions with more function point that do not have defects, and similarly, requirement items with relatively low function point also have a certain probability of having defects. In addition, functional omissions often occur in requirements with fewer function point.

B. Modeling

We use the results of FPA and the defect distributions as inputs to generate the predictive model by curve fitting, the specific steps are described as follows.

1. Normalize the function point

The function point of different software vary greatly, so we normalize the function point in order to allow horizontal comparison between software. For the m^{th} requirement, the corresponding normalization method is

$$f_m = \frac{F_m}{\sum_{i=0}^{n} F_i}$$

Where F_m is the number of function point corresponding to the original requirement, and f_m is the number of function point after normalization.

2. Normalize the software defects

As the same reason, we use the same method to normalize the software defects. For the m^th requirement, the normalization of defects is

$$e_m = \frac{E_m}{\sum_{i=0}^{n} E_i}$$

Where E_m denotes the number of defects corresponding to the original requirement, and e_m denotes the normalized value of defects.

In the statistics here, we did not include functional omission defects in the calculation of the model generation because

[1]The raw data has been published at git@github.com:zhaoxinghan/FPA.git

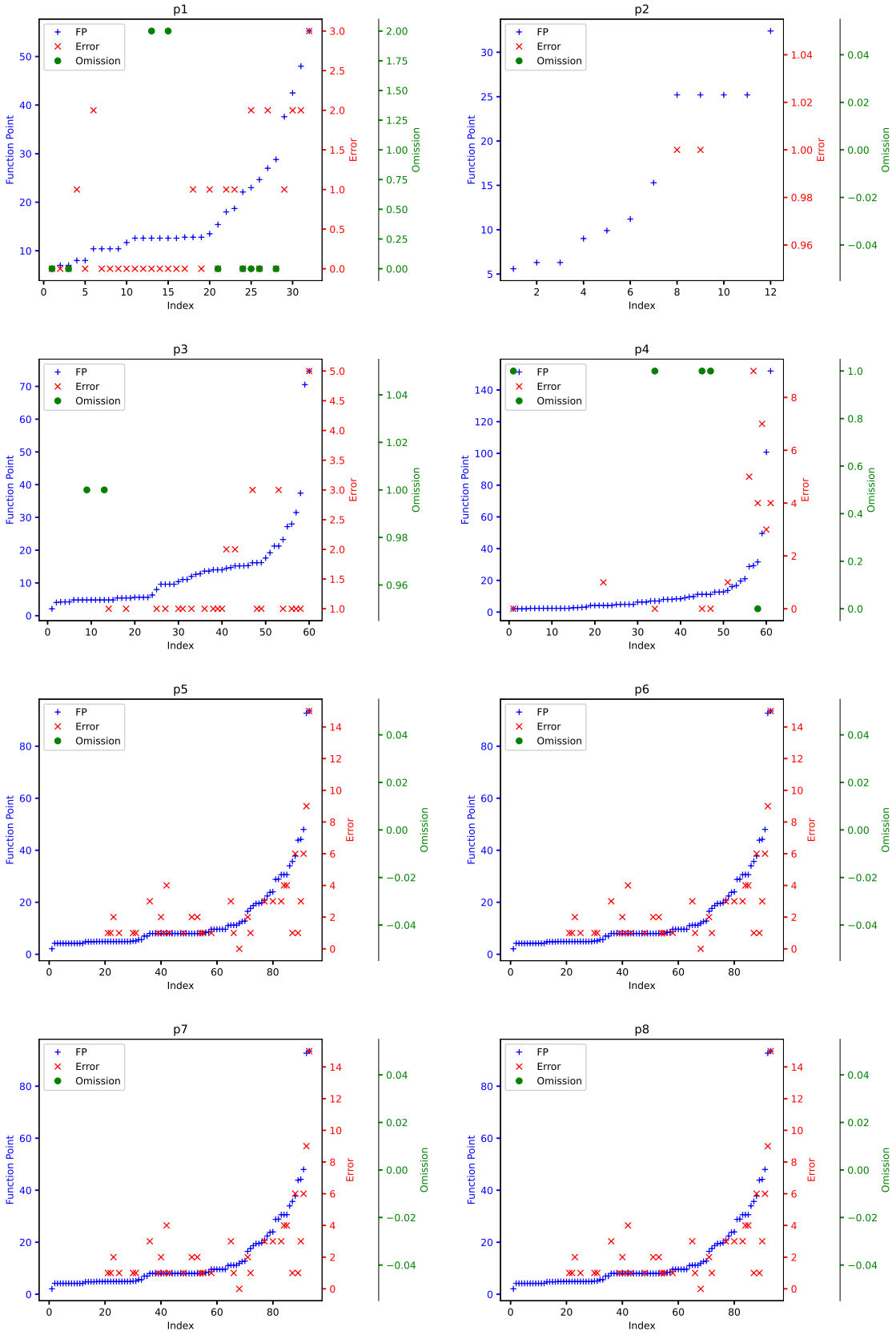

Figure 1. Defects Distribution

the number of functional omission defects is small and not sufficient to support the mathematical model generation.

3. Form the set of function point and defects

After normalization, each software will have a normalized set of function point and defects, and then these sets will be grouped into a universal set A.

$$A = \bigcup_{j=1}^{8} \bigcup_{i=0}^{n_j} (f_m^j, e_m^j)$$

Where n_j denotes the number of the requirement items in the j^{th} software, and (f_m^j, e_m^j) denotes the data pair corresponding to the m^{th} function point and defects in the j^{th} program.

4. Construct the required mathematical expression using curve fitting

A curve fitting approach is used to construct the most suitable mathematical expression for the set of A, so that the error between the mathematical model and the actual value is minimized.

In the case of independent requirements, there should be a linear relationship between the requirements of the software and the corresponding defects. In another word, if we combine any two requirement items, the number of defects after their combination should be equal to the sum of the number of defects corresponding to the original two requirements. We suppose the fitting function is $f(x)$, the function point corresponding to any two requirement items (x_1, x_2) and the number of defects corresponding to them (y_1, y_2), the relationship between them should be

$$y_1 + y_2 = f(x_1 + x_2) \tag{4}$$

we assume the form of fitting function is

$$y = f(x) = ax^2 + bx + c$$

Because y should satisfy equation 4, we can get

$$a = 0$$

If a requirement item is empty, the function point should be 0.

$$f(0) = 0$$

we can deduce that

$$c = 0$$

So the final form of the fitting function should be

$$y = bx$$

we use the least squares pair for fitting, and the value of b calculated to be 1.21, which corresponds to a residual sum of squares of 0.61. The prediction model for the defects is

$$y = 1.21x \tag{5}$$

The distribution of the normalized relationship in A and the fitting curve are shown as Figure 2. The horizontal coordinates are the normalized function point and the vertical coordinates are the number of defects after normalization.

C. Interpretation of the prediction equation

In equation 5, x represents the proportion of a certain function point to the total function point of the software, while y represents the proportion of defects of a certain functional item to the total number of defects.

In the software engineering context, y in equation 5 denotes the probability that one certain functional item has bugs. The higher the value, the greater the probability of failure and the more defects that may exist.

The final value of the coefficient b is calculated as 1.21, which is named as 'configuration item maturity factor'. In the software engineering context, the smaller the value, the slower the growth of defects in the software's requirement items in the case of growing function point (size), which can be interpreted that the software is more stable. Of course, stability here does not mean excellent quality.

The sum squared residual is 0.61 after fitting in Figure 2, it can be seen that there are some data that still deviate from the fit to a greater extent. This is because the appearance of software defects is not a logical event and can be influenced by the designer's condition and many external circumstances. Our prediction model can only represent a trend and probability, but the defect data in the specific software will not exactly match the prediction results.

VI. Statistical Results

By combining two acknowledged theories, we obtain a theoretical inference that the more function point that a function requirement item has, the more probability that the item has defects. We randomly choose 8 empirical programs to verify this theory and get some results as follow.

1) The distribution of function point and defects is generally in line with the trend of the theoretical conclusion. The probability and number of defects in most requirement items in the configuration test increase as the number of function point increases.
2) However, there are still some function items data which is not followed this trend. Some requirement items with large function point have not found defects in the configuration test, and vice versa.

VII. Issues and Analysis

A. Function point is not in keeping with the LOC

From our statistics, the LOC of the software is basically consistent with the function point, but not completely consistent. Specifically, in the case of the same function point, the scale of embedded software is always larger than the scale of non-embedded software. From our investigation and analysis, the reasons are mainly reflected in two aspects.

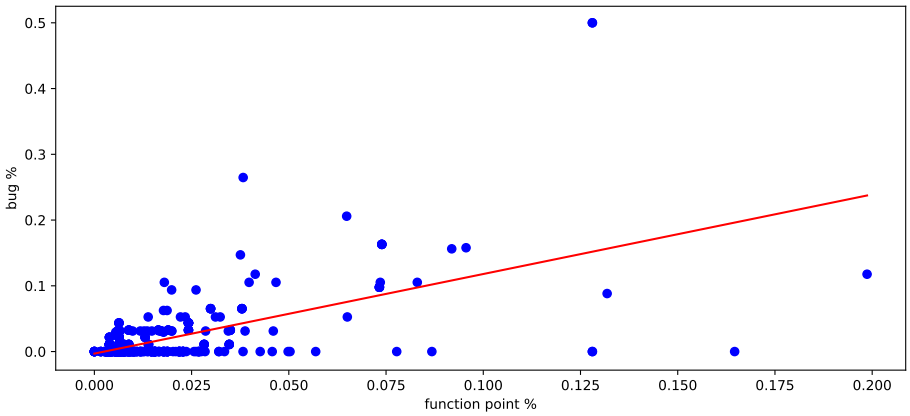

Figure 2. Distribution relationship between FPs and defects

1. In terms of support for the development environment, the embedded development environment is not as supportive as the non-embedded development environment.

Non-embedded development environments, such as Java, C sharp, etc. various standard libraries and algorithm libraries are more mature, and most of them are integrated with the IDE, while most embedded development environments do not have the corresponding standard libraries, many basic functions need to be implemented by the designers themselves.

2. Embedded software project often contains some other code that is not written by the developers.

The codes of non-embedded software are often closely related to the program's functions, and most of them were written by the program developers. However, the embedded software often contains some manufacturers to provide the basic library, such as bootloader and some basic functions of the package, as well as some general standard library, such as TCP/IP library, etc. These codes are often integrated into the source code and counted into the LOC statistics.

B. The difference between the distribution of defects and function point

When conducting the data statistics, it was found that although the distribution of function point and defects of the software satisfied the equation 5 in terms of the total statistical tendency, there would be many cases with a large amount of deviation. For example, in program P4 there were 9 defects in the function item with not large function point, but in several other projects, there were a large number of function point corresponding to a function item with no defects found. After analysis and verification, the main reasons for this phenomenon are as follows.

1) The degree of unit test coverage within the project team
2) Difficulty of the processing involved in the function point corresponding to the requirement item
3) The project team has no similar engineering experience
4) It is also possible that the designer was in a god or bad state at that time

C. The reason why we choose function point as prediction input

In recent years, since deep learning become widely used, defect prediction technology also has a trend that the input parameters become more and more complex. Of course, if more input parameters were used, the result would be more accurate. However, in another way, the costs would be more expensive.

The FPA is the most accurate program size prediction technology so far. Many organizations use FPA as the main method to support the development plan. If we use function point as the input parameter to predict defects distribution, this would be no additional costs when the development team uses FPA to predict the program size. Thus, it would be easy to spread across the industrial organization.

VIII. Future Research Plan

A. Enlarge study samples

Due to limited resources, there are only eight target programs. Although it shows some basic patterns of defect distribution in statistics and analysis, the fitting error is large and the persuasive power is still lacking. We will add more programs to the study samples so that the requirement items and defects would be richer.

B. Increase the input parameters

In this project, only the function point of the requirement items were used as the input to the study. The input form was homogeneous. We also found that the shake of the distribution of defects among the fitting curve is still significant. Exploring the causes of these shakes and the weights of these factors is helpful for software quality improvement of organizations. Therefore, the next step should be to add various types of inputs, such as the composition of the development team, historical data, process information from configuration management systems, etc. Then, machine learning would be used to calculate the impact weights.

C. Use natural language processing methods

Using manual parsing of software requirements and calculating the function point is not only a heavy workload but also prone to errors. The use of natural language processing (NLP) methods can automatically parse out various types of entities in the requirement description, which not only greatly improves efficiency but also ensures the correctness of the parsing process. This is the only method to enlarge the number of samples quickly.

IX. Summary and Outlook

By exploring the relationship between the function point of software requirement items and the defects of configuration items, we explore the basic law that the defects of configuration items are consistent with the growth trend of the number of requirement point, and through the method of least squares, we derive the mathematical model equation of the relationship between function point and defects, elaborate the significance of the parameters of the equation in practical engineering, and analyze the causes of the problems found during the calculation and statistics of requirement function point.

However, we only extracted eight projects as the study target due to resource constraints and selected only one input parameter of function point. From the final results, although the conclusions from the general trend are consistent with our theoretical derivation, the shakes between the fitting curve are a bit larger when applied to specific projects. To achieve more accurate prediction results, it is necessary to upgrade the number of data samples for the study and obtain more input parameters as well as various historical and process data to form a more accurate prediction model.

References

[1] A. Z. Abualkishik and L. Lavazza, "Ifpug function points to cosmic function points convertibility: A fine-grained statistical approach," *Information and Software Technology*, vol. 97, pp. 179–191, 2018.

[2] F. Akiyama, "An example of software system debugging," pp. 353–359, 1971.

[3] B. Barry *et al.*, "Software engineering economics," *New York*, vol. 197, 1981.

[4] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[5] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 519–531.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[7] J. J. Cuadrado-Gallego, L. Buglione, M. J. Domínguez-Alda, M. F. De Sevilla, J. A. G. De Mesa, and O. Demirors, "An experimental study on the conversion between ifpug and cosmic functional size measurement units," *Information and Software Technology*, vol. 52, no. 3, pp. 347–357, 2010.

[8] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Transactions on software engineering*, no. 2, pp. 96–104, 1979.

[9] R. Dumke and A. Abran, *COSMIC Function Points: Theory and Advanced Practices*. CRC Press, 2016.

[10] C. Eduardo Carbonera, K. Farias, and V. Bischoff, "Software development effort estimation: A systematic mapping study," *IET Software*, vol. 14, no. 4, pp. 328–344, 2020.

[11] A. E. Ferdinand, "A theory of system complexity," *International Journal of General System*, vol. 1, no. 1, pp. 19–33, 1974.

[12] J. E. Gaffney, "Estimating the number of faults in code," *IEEE Transactions on Software Engineering*, no. 4, pp. 459–464, 1984.

[13] M. H. Halstead, "Natural laws controlling algorithm structure?" *ACM Sigplan Notices*, vol. 7, no. 2, pp. 19–26, 1972, aCM New York, NY, USA.

[14] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.

[15] S. Henry and D. Kafura, "The evaluation of software systems' structure using quantitative software metrics," *Software: Practice and Experience*, vol. 14, no. 6, pp. 561–573, 1984.

[16] IFPUG., *The IFPUG guide to IT and software measurement*. CRC Press, 2012.

[17] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, 2012.

[18] M. Lipow, "Number of faults per line of code," *IEEE Transactions on software Engineering*, no. 4, pp. 437–439, 1982.

[19] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.

[20] J. McCall, W. Randall, C. Bowen, N. McKelvey, and R. Senn, "Methodology for software reliability prediction," *Rome Air Development Center (RADC) Technical Reports, RADC-TR-87-171 (Volumes 1 and 2)*, 1987.

[21] S. Miller and D. Firesmith, "Four types of shift left testing," CARNEGIE-MELLON UNIV PITTSBURGH PA, Report, 2021.

[22] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*, Conference Proceedings, pp. 181–190.

[23] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on software Engineering*, vol. 18, no. 5, p. 423, 1992.

[24] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, Conference Proceedings, pp. 284–292.

[25] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th international conference on software engineering (ICSE)*. IEEE, Conference Proceedings, pp. 382–391.

[26] M. Nasir, "A survey of software estimation techniques and project planning practices," in *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)*. IEEE, 2006, pp. 305–310.

[27] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[28] C. Quesada-López, D. Madrigal-Sánchez, and M. Jenkins, "An empirical analysis of ifpug fpa and cosmic ffp measurement methods," in *International Conference on Information Technology & Systems*. Springer, Conference Proceedings, pp. 265–274.

[29] K. Sharma, M. Bala *et al.*, "New failure rate model for iterative software development life cycle process," *Automated Software Engineering*, vol. 28, no. 2, pp. 1–22, 2021.

[30] V. Y. Shen, T.-j. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—an empirical study," *IEEE Transactions on Software Engineering*, no. 4, pp. 317–324, 1985.

[31] C. Smidts, M. Stutzke, and R. W. Stoddard, "Software reliability modeling: an approach to early reliability prediction," *IEEE Transactions on Reliability*, vol. 47, no. 3, pp. 268–278, 1998.

[32] A. Spillner and H. Bremenn, "The w-model. strengthening the bond between development and test," in *Int. Conf. on Software Testing, Analysis and Review*, Conference Proceedings, pp. 15–17.

[33] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2018.

[34] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok, "Recent catastrophic accidents: Investigating how software was responsible," in

2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. IEEE, 2010, pp. 14–22.

[35] W. E. Wong, J. R. Horgan, M. Syring, W. Zage, and D. Zage, "Applying design metrics to predict fault-proneness: a case study on a large-scale software system," *Software: Practice and Experience*, vol. 30, no. 14, pp. 1587–1608, 2000.

[36] W. E. Wong, X. Li, and P. A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *Journal of Systems and Software*, vol. 133, pp. 68–94, 2017.

[37] D. K. Yadav, S. Chaturvedi, and R. B. Misra, "Early software defects prediction using fuzzy logic." *International Journal of Performability Engineering*, vol. 8, no. 4, 2012.

[38] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, Conference Proceedings, pp. 91–100.