Software Quality Assurance Methodologies and Techniques

Guest Editors: Chin-Yu Huang, Hareton Leung,
Wu-Hon Francis Leung, and Osamu Mizuno

Software Quality Assurance Methodologies and Techniques

Advances in Software Engineering

Software Quality Assurance Methodologies and Techniques

Guest Editors: Chin-Yu Huang, Hareton Leung,
Wu-Hon Francis Leung, and Osamu Mizuno

Editorial Board

Contents

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 872619, 2 pages
doi:10.1155/2012/872619

Editorial
Software Quality Assurance Methodologies and Techniques

Chin-Yu Huang,[1] Hareton Leung,[2] Wu-Hon Francis Leung,[3] and Osamu Mizuno[4]

[1] *Department of Computer Science and Institute of Information Systems and Applications, National Tsing Hua University, Hsinchu 30013, Taiwan*
[2] *Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong*
[3] *Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA*
[4] *Graduate School of Science and Technology, Kyoto Institute of Technology, Kyoto 606-8585, Japan*

Correspondence should be addressed to Chin-Yu Huang, cyhuang@cs.nthu.edu.tw

Received 18 July 2012; Accepted 18 July 2012

Software quality assurance (SQA) is a planned and systematic pattern of actions necessary to provide adequate confidence that a software product conforms to requirements during software development. SQA consists of methodologies and techniques of assessing the software development processes and methods, tools, and technologies used to ensure the quality of the developed software. SQA is typically achieved through the use of well-defined standard practices, including tools and processes, for quality control to ensure the integrity and reliability of software. This special issue presents new research works along these directions, and we received 21 submissions and accepted five of them after a thorough peer-review process. The acceptance rate of this special issue is around 24%. The resultant collection provides a number of useful results. These accepted papers cover a broad range of topics in the research field of SQA, including software validation, verification, and testing, SQA modeling, certification, evaluation, and improvement, SQA standards and models, SQA case studies, data analysis and risk management.

For example, in "*Specifying process views for a measurement, evaluation, and improvement strategy,*" P. Becker, P. Lew, and L. Olsina developed a specific strategy called SIQinU (strategy for understanding and improving quality in use), which recognizes problems of quality in use through evaluation of a real system-in-use situation and proposes product improvements by understanding and making changes to the product's attributes. They used UML 2.0 activity diagrams and the SPEM profile to stress the functional, informational, organizational, and behavioral views for the SIQinU process.

In the paper "*Program spectra analysis with theory of evidence,*" R. Hewett proposed a spectrum-based approach to fault localization using the Dempster-Shaffer theory of evidence. Using mathematical theories of evidence for uncertainty reasoning, the proposed approach estimates the likelihood of faulty locations based on evidence from program spectra. Evaluation results show that their approach is at least as effective as others with an average effectiveness of 85.6% over 119 versions of the programs.

In the paper entitled "*An empirical study on the impact of duplicate code,*" K. Hotta et al. presented an empirical study on the impact of the presence of duplicate code on software evolution. They assumed that if duplicate code is modified more frequently than nonduplicate code, the presence of duplicate code affects software evolution, and compared the stability of duplicate code and non-duplicate code. They conducted an experiment on 15 open-source software systems, and the result showed that duplicate code was less frequently modified than nonduplicate code and, in some cases, duplicate code was intensively modified in a short period though duplicate code was more stable than nonduplicate code in the whole development period.

The next paper by X. Xiao and T. Dohi, "*A comparative study of data transformations for wavelet shrinkage estimation with application to software reliability assessment,*" applied the wavelet-based techniques to estimate the software intensity function. Some data transformations were employed to preprocess the software-fault count data. Throughout the numerical evaluation, the authors concluded that the

wavelet-based estimation methods have much more potential applicability than the other data transformations to the software reliability assessment.

In the last paper "*Can faulty modules be predicted by warning messages of static code analyzer?*," O. Mizuno and M. Nakai proposed a detection method of fault-prone modules based on the spam filtering technique—fault-prone filtering. For the analysis, the authors tried to state two questions: "can fault-prone modules be predicted by applying a text filter to the warning messages of static code analyzer?" and "is the performance of the fault-prone filtering becomes better with the warning messages of a static code analyzer?". The results of experiments show that the answer to the first question is "yes." But for the second question, the authors found that the recall becomes better than the original approach.

In summary, this special issue serves as a platform for researchers and practitioners to present theory, results, experience, and other advances in SQA. Hopefully, you will enjoy this publication, and we look forward to your feedback and comments.

Chin-Yu Huang
Hareton Leung
Wu-Hon Francis Leung
Osamu Mizuno

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 938296, 22 pages
doi:10.1155/2012/938296

Research Article

An Empirical Study on the Impact of Duplicate Code

Keisuke Hotta, Yui Sasaki, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto

Graduate School of Information Science and Technology, Osaka University, Osaka 565-0871, Japan

Correspondence should be addressed to Keisuke Hotta, k-hotta@ist.osaka-u.ac.jp

Received 4 January 2012; Accepted 5 March 2012

Academic Editor: Osamu Mizuno

It is said that the presence of duplicate code is one of the factors that make software maintenance more difficult. Many research efforts have been performed on detecting, removing, or managing duplicate code on this basis. However, some researchers doubt this basis in recent years and have conducted empirical studies to investigate the influence of the presence of duplicate code. In this study, we conduct an empirical study to investigate this matter from a different standpoint from previous studies. In this study, we define a new indicator "modification frequency" to measure the impact of duplicate code and compare the values between duplicate code and nonduplicate code. The features of this study are as follows the indicator used in this study is based on modification places instead of the ratio of modified lines; we use multiple duplicate code detection tools to reduce biases of detection tools; and we compare the result of the proposed method with other two investigation methods. The result shows that duplicate code tends to be less frequently modified than nonduplicate code, and we found some instances that the proposed method can evaluate the influence of duplicate code more accurately than the existing investigation methods.

1. Introduction

Recently, duplicate code has received much attention. Duplicate code is also called as "code clone." Duplicate code is defined as identical or similar code fragments to each other in the source code, and they are generated by various reasons such as copy-and-paste programming. It is said that the presence of duplicate code has negative impacts on software development and maintenance. For example, they increase bug occurrences: if an instance of duplicate code is changed for fixing bugs or adding new features, its correspondents have to be changed simultaneously; if the correspondents are not changed inadvertently, bugs are newly introduced to them.

Various kinds of research efforts have been performed for resolving or improving the problems caused by the presence of duplicate code. For example, there are currently a variety of techniques available to detect duplicate code [1]. In addition, there are many research efforts for merging duplicate code as a single module like function or method, or for preventing duplications from being overlooked in modification [2, 3]. However, there are precisely the opposite opinions that code cloning is a good choice for design of the source code [4].

In order to answer the question whether duplicate code is harmful or not, several efforts have proposed comparison methods between duplicate code and nonduplicate code. Each of them compares a characteristic of duplicate code and nonduplicate code instead of directly investigating their maintenance cost. This is because measuring the actual maintenance cost is quite difficult. However, there is no consensus on this matter.

In this paper, we conduct an empirical study that compares duplicate code to nonduplicate code from a different standpoint of previous research and reports the experimental result on open source software. The features of the investigation method in this paper are as follows:

(i) every line of code is investigated whether it is duplicate code or not; such a fine-grained investigation can accurately judge whether every modification conducted to duplicate code or to nonduplicate code;

(ii) maintenance cost consists of not only source code modification but also several phases prior to it; in order to more appropriately estimate maintenance cost, we define an indicator that is not based on modified lines of code but the number of modified places;

(iii) we evaluate and compare modifications of duplicate code and nonduplicate code on multiple open source software systems with multiple duplicate code detection tools, that is, because every detection tool detects different duplicate code from the same source code.

We also conducted a comparison experiment with two previous investigation methods. The purpose of this experiment is to reveal whether comparisons between duplicate code and nonduplicate code with different methods yield the same result or not. In addition, we carefully analyzed the results in the cases that the comparison results were different from each method to reveal the causes behind the differences.

The rest of this paper is organized as follows: Section 2 describes related works and our motivation of this study. Section 3 introduces the preliminaries. We situate our research questions and propose a new investigation method in Section 4. Section 5 describes the design of our experiments, then we report the results in Sections 6 and 7. Section 8 discusses threats to validity, and Section 9 presents the conclusion and future work of this study.

2. Motivation

2.1. Related Work. At present, there is a huge body of work on empirical evidence on duplicate code shown in Table 1. The pioneering report in this area is Kim et al.'s study on clone genealogies [5]. They have conducted an empirical study on two open source software systems and found 38% or 36% of groups of duplicate code were consistently changed at least one time. On the other hand, they observed that there were groups of duplicate code that existed only for a short period (5 or 10 revisions) because each instance of the groups was modified inconsistently. Their work is the first empirical evidence that a part of duplicate code increases the cost of source code modification.

However, Kapser and Godfrey have different opinions regarding duplicate code. They reported that duplicate code can be a reasonable design decision based on the empirical study on two large-scale open source systems [4]. They built several patterns of duplicate code in the target systems, and they discussed the pros and cons of duplicate code using the patterns. Bettenburg et al. also reported that duplicate code does not have much a negative impact on software quality [6]. They investigated inconsistent changes to duplicate code at release level on two open software systems, and they found that only 1.26% to 3.23% of inconsistent changes introduced software errors into the target systems.

Monden et al. investigated the relation between software quality and duplicate code on the file unit [7]. They use the number of revisions of every file as a barometer of quality: if the number of revisions of a file is great, its quality is low. Their experiment selected a large-scale legacy system, which was being operated in a public institution, as the target. The result showed that modules that included duplicate code were 40% lower quality than modules that did not include duplicate code. Moreover, they reported that the larger duplicate code a source file included, the lower quality it was.

Lozano et al. investigated whether the presence of duplicate code was harmful or not [8]. They developed a tool, CloneTracker, which traces which methods include duplicate code (in short, duplicate method) and which methods are modified in each revision. They conducted a pilot study, and found that: duplicate methods tend to be more frequently modified than nonduplicate methods; however, duplicate methods tend to be modified less simultaneously than nonduplicate methods. The fact implies that the presence of duplicate code increased cost for modification, and programmers were not aware of the duplication, so that they sometimes overlooked code fragments that had to be modified simultaneously.

Also, Lozano and Wermelinger investigated the impact of duplicate code on software maintenance [9]. Three barometers were used in the investigation. The first one is *likelihood*, which indicates the possibility that the method is modified in a revision. The second one is *impact*, which indicates the number of methods that are simultaneously modified with the method. The third one is *work*, which can be represented as a product of *likelihood* and *impact* ($work = likelihood \times impact$). They conducted a case study on 4 open source systems for comparing the three barometers of methods including and not including duplicate code. The result was that *likelihood* of methods including duplicate code was not so different from one of methods not including duplicate code; there were some instances that *impact* of methods including duplicate code were greater than one of methods not including duplicate code; if duplicate code existed in methods for a long time, their *work* tended to increase greatly.

Moreover, Lozano et al. investigated the relation between duplicate code, features of methods, and their changeability [10]. Changeability means the ease of modification. If changeability decreased, it will be a bottleneck of software maintenance. The result showed that the presence of duplicate code can decrease changeability. However, they found that changeability was more greatly affected by other properties such as length, fan-out, and complexity of methods. Consequently, they concluded that it was not necessary to consider duplicate code as a primary option.

Krinke hypothesized that if duplicate code is less stable than nonduplicate code, maintenance cost for duplicate code is greater than for nonduplicate code. He conducted a case study in order to investigate whether the hypothesis is true or not [11]. The targets are 200 revisions (a version per week) of source code of 5 large-scale open-source systems. He measured *added*, *deleted*, and *changed* LOCs on duplicate code and nonduplicate code and compared them. He reported that nonduplicate code was more *added*, *deleted*, and *changed* than duplicate code. Consequently, he concluded that the presence of duplicate code did not necessarily make it more difficult to maintain source code.

Göde and Harder replicated Krinke's experiment [12]. Krinke's original experiment detected line-based duplicate code meanwhile their experiment detected token-based duplicate code. The experimental result was the same as Krinke's one. Duplicate code is more stable than nonduplicate code in the viewpoint of added and changed. On

Table 1: Summarization of related work.

	How to investigate	Impact of duplicate code
Kim et al. [5]	Using clone linages and clone genealogies	A part of duplicate code is negative
Kapser and Godfrey [4]	Build several patterns of duplicate code and discuss about them	Nonnegative
Bettenburg et al. [6]	Investigate inconsistent changes to duplicate code at the release revel	Nonnegative
Monden et al. [7]	Calculate the number of revisions on every file	Negative
Lozano et al. [8]	Count the number of modifications on methods including duplicate code	Negative
Lozano and Wermelinger [9]	Using *work*	A part of duplicate code is negative
Lozano et al. [10]	Using changeability (the ease of modification)	Negative but not so high
Krinke [11]	Using stability (line level)	Nonnegative
Göde and Harder [12]	Using stability (token level)	Nonnegative
Krinke [13]	Using ages	Nonnegative
Rahman et al. [14]	Investigate the relationship between duplicate code and bugs	Nonnegative
Göde and Koschke [15]	Count the number of changes on clone genealogies	A part of duplicate code is negative

the other hand, from the deleted viewpoint, nonduplicate code is more stable than duplicate code.

Also, Krinke conducted an empirical study to investigate ages of duplicate code [13]. In this study, he calculated and compared average ages of duplicate lines and nonduplicate lines on 4 large-scale Java software systems. He found that the average age of duplicate code is older than nonduplicate code, which implies duplicate code is more stable than nonduplicate code.

Eick et al. investigated whether source code decays when it is operated and maintained for a long time [16]. They selected several metrics such as the amount of *added* and *deleted* code, the time required for modification, and the number of developers as indicators of code decay. The experimental result on a 15-year-operated large system showed that cost required for completing a single requirement tendS to increase.

Rahman et al. investigated the relationship between duplicate code and bugs [14]. They analyzed 4 software systems written in C language with bug information stored in Bugzilla. They use Deckard, which is an AST-based detection tool, to detect duplicate code. They reported that only a small part of the bugs located on duplicate code, and the presence of duplicate code did not dominate bug appearances.

Göde modeled how type-1 code clones are generated and how they evolved [17]. Type-1 code clone is a code clone that is exactly identical to its correspondents except white spaces and tabs. He applied the model to 9 open-source software systems and investigated how code clones in them evolved. The result showed that the ratio of code duplication was decreasing as time passed; the average life time of code clones was over 1 year; in the case that code clones were modified inconsistently, there were a few instances that additional modifications were performed to restore their consistency.

Also, Göde and Koschke conducted an empirical study on clone evolution and performed a detailed tracking to detect when and how clones had been changed [15]. In their study, they traced clone evolution and counted the number of changes on each clone genealogy. They manually inspected the result in one of the target systems and categorized all the modifications on clones into consistent or inconsistent. In addition, they carefully categorized inconsistent changes into intentional or unintentional. They reported that almost all clones were never changed or only once during their lifetime, and only 3% of the modifications had high severity. Therefore, they concluded that many of clones do not cause additional change effort, and it is important to identify the clones with high threat potential to manage duplicate code effectively.

As described above, some empirical studies reported that duplicate code should have a negative impact on software evolution meanwhile the others reported the opposite result. At present, there is no consensus on the impact of the presence of duplicate code on software evolution. Consequently, this research is performed as a replication of the previous studies with solid settings.

2.2. Motivating Example. As described in Section 2.1, many research efforts have been performed on evaluating the influence of duplicate code. However, these investigation methods still have some points that they did not evaluate. We explain these points with the example shown in Figure 1. In this example, there are two similar methods and some places are modified. We classified these modifications into 4 parts, modification A, B, C, and D.

Investigated Units. In some studies, large units (e.g., files or methods) are used as their investigation units. In those investigation methods, it is assumed that duplicate code has a negative impact if files or methods having a certain amount of duplicate code are modified, which can cause a problem. The problem is the incorrectness of modifications count. For example, if modifications are performed on a method which has a certain amount of duplicate code, all the modifications are assumed as performed on the duplicate code even if they are actually performed on nonduplicate code of the method. Modification C in Figure 1 is an instance of this problem. This modification is performed on nonduplicate

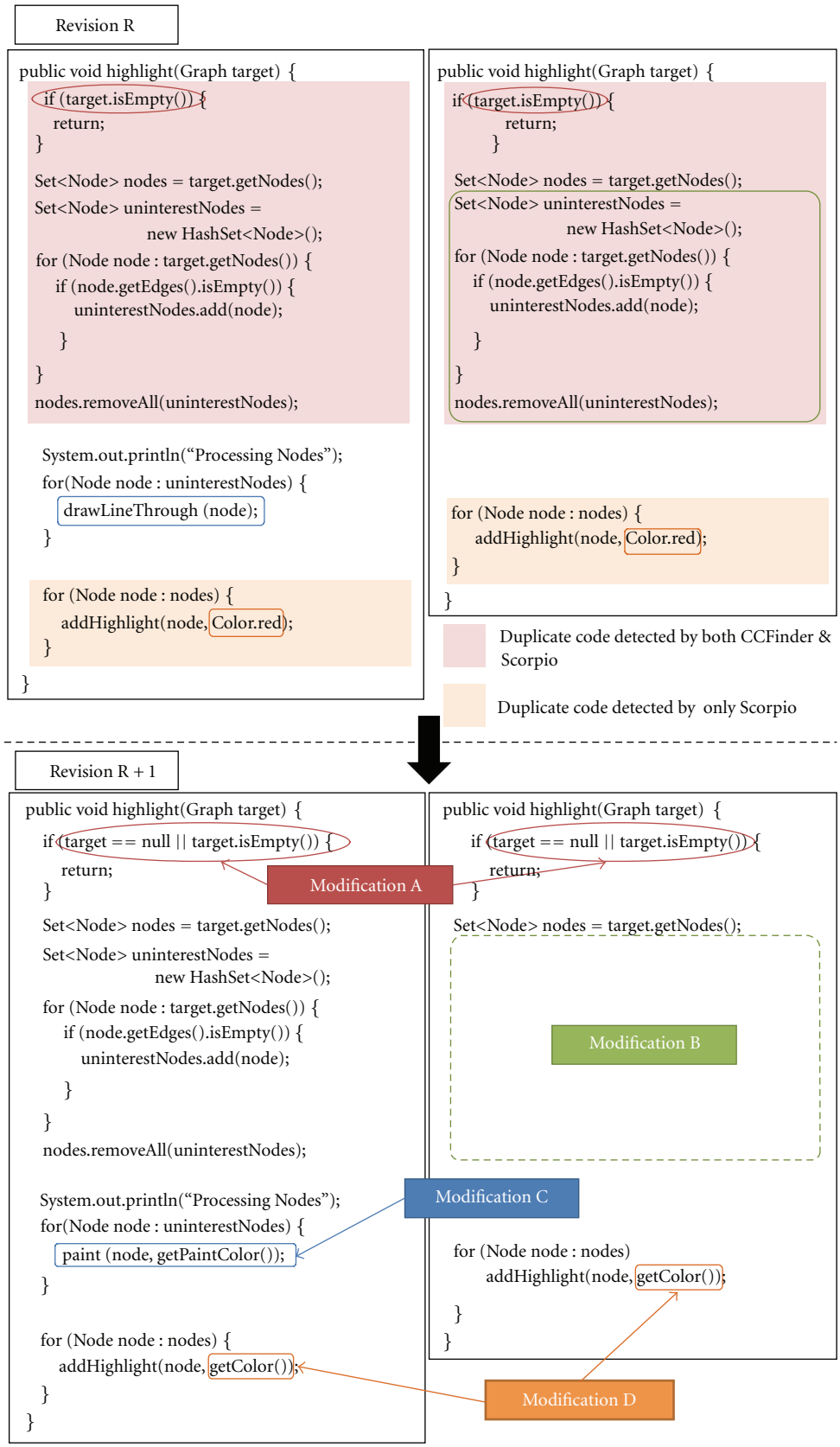


Figure 1: Motivating example.

code; nevertheless, it is regarded that this modification is performed on duplicate code if we use method as the investigation units.

Line-Based Barometers. In some studies, line-based barometers are used to measure the influence of duplicate code on software evolution. Herein, the line-based barometer indicates a barometer calculated with the amount of added/changed/deleted lines of code. However, line-based barometer cannot distinguish the following two cases: the first case is that *consecutive 10 lines of code were modified for fixing a single bug*; the second case is that *1 line modification was performed on different 10 places of code for fixing 10 different bugs*. In real software maintenance, the latter requires much more cost than the former because we have to conduct several steps before the actual source code modification such as identifying buggy module, informing the maintainer about the bugs, and identifying buggy instruction.

In Figure 1, Modification A is 1 line modification, and performed on 2 places, meanwhile Modification B is 7 lines modification on a single place. With line-based barometers, it is regarded that Modification B has the impact 3.5 times larger than Modification A. However, this is not true because we have to identify 2 places of code for modifying A meanwhile 1 place identification is required for B.

A Single Detection Tool. In the previous studies, a single detection tool was used to detect duplicate code. However, there is neither a generic nor strict definition of duplicate code. Each detection tool has its own unique definition of duplicate code, and it detects duplicate code based on the own definition. Consequently, different duplicate code is detected by different detection tools from the same source code. Therefore, the investigation result with one detector is different from the result from another detector. In Figure 1, a detector `CCFinder` detects lines highlighted with red as duplicate code, and another detector `Scorpio` detects not only lines highlighted with red but also lines highlighted with orange before modification. Therefore, if we use `Scorpio`, Modification D is regarded as being affected with duplicate code, nevertheless, it is regarded as not being affected with duplicate code if we use `CCFinder`. Consequently, the investigation with a single detector is not sufficient to get the generic result about the impact of duplicate code.

2.2.1. Objective of This Study. In this paper, we conducted an empirical study from a different standpoint of previous research. The features of this study are as follows.

Fine-Grained Investigation Units. In this study, every line of code is investigated whether it is duplicate code or not, which enables us to judge whether every modification is conducted on duplicate code or nonduplicate code.

Place-Based Indicator. We define a new indicator based on the number of modified places, not the number of modified lines. The purpose of place-based indicator is to evaluate the impact of the presence of duplicate code with different standpoints from the previous research.

Multiply Detector. In this study, we use 4 duplicate code detection tools to reduce biases of each detection method.

3. Preliminaries

In this section, we describe preliminaries used in this paper.

3.1. Duplicate Code Detection Tools. There are currently various kinds of duplicate code detection tools. The detection tools take the source code as their input data, and they provide the position of the detected duplicate code in it. The detection tools can be categorized based on their detection techniques. Major categories should be line based, token based, metrics based, AST (Abstract Syntax Tree) based, and PDG (Program Dependence Graph) based. Each technique has merits and demerits, and there is no technique that is superior to any other techniques in every way [1, 18]. The following subsections describe 4 detection tools that are used in this research. We use two token-based detection tools, which is for investigating whether both the token-based detection tools always introduce the same result or not.

3.1.1. `CCFinder`. `CCFinder` is a token-based detection tool [19]. The major features of `CCFinder` are as follows.

(i) `CCFinder` replaces user-defined identifiers such as variable names or function names with special tokens before the matching process. Consequently, `CCFinder` can identify code fragments that use different variables as duplicate code.

(ii) Detection speed is very fast. `CCFinder` can detects duplicate code from millions lines of code within an hour.

(iii) `CCFinder` can handle multiple popular programming languages such as C/C++, Java, and COBOL.

3.1.2. `CCFinderX`. `CCFinderX` is a major version up from `CCFinder` [20]. `CCFinderX` is a token-based detection tool as well as `CCFinder`, but the detection algorithm was changed to *bucket sort* to *suffix tree*. `CCFinderX` can handle more programming languages than `CCFinder`. Moreover, it can effectively use resources of multi core CPUs for faster detection.

3.1.3. `Simian`. `Simian` is a line-based detection tool [21]. As well as `CCFinder` family, `Simian` can handle multiple programming languages. Its line-based technique realizes duplicate code detection on small memory usage and short running time. Also, `Simian` allows fine-grained settings. For example, we can configure that duplicate code is not detected from *import* statements in the case of Java language.

3.1.4. `Scorpio`. `Scorpio` is a PDG-based detection tool [22, 23]. `Scorpio` builds a special PDG for duplicate code detection, not traditional one. In traditional PDGs, there are two types of edge representing data dependence and control dependence. The special PDG used in `Scorpio` has one

(a) *before modification*
(1) A
(2) B
(3) line will be changed 1
(4) line will be changed 2
(5) C
(6) D
(7) line will be deleted 1
(8) line will be deleted 2
(9) E
(10) F
(11) G
(12) H

(b) *after modification*
(1) A
(2) B
(3) line changed 1
(4) line changed 2
(5) C
(6) D
(7) E
(8) F
(9) G
(10) line added 1
(11) line added 2
(12) H

(c) *diff output*
```
3,4c3,4
<    line will be changed 1
<    line will be changed 2
---
>  line changed 1
>  line changed 2
7,8d6
<  line will be deleted 1
<  line will be deleted 2
11a10,11
>  line added 1
>  line added 2
```

Algorithm 1: A simple example of comparing two source files with `diff` (changed region is represented with identifier "c" like 3,4c3,4; deleted region is represented with identifier "d" like 7,8d6, added region is represented with identifier "a" like 11a10,11). The number before and after the identifier shows the correspond lines.

more edge, execution-next link, which allows detecting more duplicate code than traditional PDG. Also, `Scorpio` adopts some heuristics for filtering out false positives. Currently, `Scorpio` can handle only Java language.

3.2. Revision. In this paper, we analyze historical data managed by version control systems for investigation. Version control systems store information about changes to documents or programs. We can specify changes by using a number, "*revision*". We can get source code in arbitrary revision, and we can also get modified files, change logs, and the name of developers who made changes in arbitrary two consecutive revisions with version control systems.

Due to the limit of implementation, we restrict the target version control system to Subversion. However, it is possible to use other version control systems such as CVS.

3.3. Target Revision. In this study, we are only interested in changes in source files. Therefore, we find out revisions that have some modifications in source files. We call such revisions as *target revisions*. We regard a revision R as the target revision, if at least one source file is modified from R to $R + 1$.

3.4. Modification Place. In this research, we use the number of places of modified code, instead lines of modified code. That is, even if multiple consecutive lines are modified, we regard it as a single modification. In order to identify the number of modifications, we use UNIX diff command. Algorithm 1 shows an example of diff output. In this example, we can find 3 modification places. One is a change in line 3 and 4, another is a deletion in line 7 and 8, and the other is an addition at line 11. As shown in the algorithm, it is very easy to identify multiple consecutive modified lines as a single modification; all we have to do is just parsing the output of diff so that the start line and end line of all the modifications are identified.

4. Proposed Method

This section describes our research questions and the investigation method.

4.1. Research Questions and Hypotheses. The purpose of this research is to reveal whether the presence of duplicate code really affects software evolution or not. We assume that *if duplicate code is more frequently modified than nonduplicate code, the presence of duplicate code has a negative impact on software evolution*. This is because if much duplicate code is included in source code though, it is never modified during its lifetime, the presence of duplicate code never causes inconsistent changes or additional modification efforts. Our research questions are as follows.

RQ1: Is duplicate code more frequently modified than nonduplicate code?

RQ2: Are the comparison results of stability between duplicate code and nonduplicate code different from multiple detection tools?

RQ3: Is duplicate code modified uniformly throughout its lifetime?

RQ4: Are there any differences in the comparison results on modification types?

To answer these research questions, we define an indicator, *modification frequency* (in short, MF). We measure and compare MF of duplicate code (in short, MF_d) and MF of nonduplicate code (in short, MF_n) for investigation.

4.2. Modification Frequency

4.2.1. Definition. As described above, we use MF to estimate the influence of duplicate code. MF is an indicator based on the number of modified code, not lines of modified code. This is because this research aims to investigate from a different standpoint from previous research.

We define MF_d in the formula:

$$\mathrm{MF}_d = \frac{\sum_{r \in R} \mathrm{MC}_d(r)}{|R|}, \tag{1}$$

where R is a set of target revisions, $\mathrm{MC}_d(r)$ is the number of modifications on duplicate code between revision r and $r+1$. We also define MF_n in the formula:

$$\mathrm{MF}_n = \frac{\sum_{r \in R} \mathrm{MC}_n(r)}{|R|}, \tag{2}$$

where $\mathrm{MC}_n(r)$ is the number of modifications on nonduplicate code between revision r and $r+1$.

These values mean the average number of modifications on duplicate code or nonduplicate code per revision. However, in these definitions, MF_d and MF_n are very affected by the amount of duplicate code included the source code. For example, if the amount of duplicate code is very small, it is quite natural that the number of modifications on duplicate code is much smaller than nonduplicate code. However, if a small amount of duplicate code is included but it is quite frequently modified, we need additional maintenance efforts to judge whether its correspondents need the same modifications or not. We cannot evaluate the influence of duplicate code in these situations in these definitions.

In order to eliminate the bias of the amount of duplicate code, we normalize the formulae (1) and (2) with the ratio of duplicate code. Here, we assume that

(i) $\mathrm{LOC}_d(r)$ is the total lines of duplicate code in revision r,

(ii) $\mathrm{LOC}_n(r)$ is the total lines of nonduplicate code on r,

(iii) $\mathrm{LOC}(r)$ is the total lines of code on r, so that the following formula is satisfied:

$$\mathrm{LOC}(r) = \mathrm{LOC}_d(r) + \mathrm{LOC}_n(r). \tag{3}$$

Under these assumptions, the normalized MF_d and MF_n are defined in the following formula:

$$\begin{aligned} \text{normalized } \mathrm{MF}_d &= \frac{\sum_{r \in R} \mathrm{MC}_d(r)}{|R|} \times \frac{\sum_{r \in R} \mathrm{LOC}(r)}{\sum_{r \in R} \mathrm{LOC}_d(r)}, \\ \text{normalized } \mathrm{MF}_n &= \frac{\sum_{r \in R} \mathrm{MC}_n(r)}{|R|} \times \frac{\sum_{r \in R} \mathrm{LOC}(r)}{\sum_{r \in R} \mathrm{LOC}_n(r)}. \end{aligned} \tag{4}$$

In the reminder of this paper, the normalized MF_d and MF_n are called as just MF_d and MF_n, respectively.

4.2.2. Measurement Steps. The MF_d and MF_n are measured with the following steps,

Step 1. It identifies target revisions from the repositories of target software systems. Then, all the target revisions are checked out into the local storage.

Step 2. It normalized all the source files in every target revision.

Step 3. It detects duplicate code within every target revision. Then, the detection result is analyzed in order to identify the file path, the lines of all the detected duplicate code.

Step 4. It identifies differences between two consecutive revisions. The start lines and the end lines of all the differences are stored.

Step 5. It counts the number of modifications on duplicate code and nonduplicate code.

Step 6. It calculates MF_d and MF_n.

In the reminder of this subsection, we explain each step of the measurement in detail.

Step 1. It obtains Target Revisions. In order to measure MF_d and MF_n, it is necessary to obtain the historical data of the source code. As described above, we used a version control system, Subversion, to obtain the historical data.

Firstly, we identify which files are modified, added, or deleted in each revision and find out target revisions. After identifying all the target revision from the historical data, they are checked out into the local storage.

Step 2. It normalizes Source Files. In the Step 2, every source file in all the target revisions is normalized with the following rules:

(i) deletes blank lines, code comments, and indents,

(ii) deletes lines that consist of only a single open/close brace, and the open/close brace is added to the end of the previous line.

The presence of code comments influences the measurement of MF_d and MF_n. If a code comment is located within a duplicate code, it is regarded as a part of duplicate code even if it is not a program instruction. Thus, the LOC of duplicate code is counted greater than it really is. Also, there is no common rule how code comments should be treated if they are located in the border of duplicate code and nonduplicate code, which can cause a problem that a certain detection tool regards such a code comment as duplicate code meanwhile another tool regards it as nonduplicate code.

As mentioned above, the presence of code comments makes it more difficult to identify the position of duplicate code accurately. Consequently, all the code comments are removed completely. As well as code comments, different detection tools handle blank lines, indents, lines including only a single open or close brace in different ways, which also influence the result of duplicate code detection. For this reason, blank lines and indents are removed, and lines that consist of only a single open or close brace are removed, and

Table 2: Target software systems—Experiment 1.

(a) Experiment 1.1

Name	Domain	Programming language	Number of Revisions	LOC (latest revision)
EclEmma	Testing	Java	788	15,328
FileZilla	FTP	C++	3,450	87,282
FreeCol	Game	Java	5,963	89,661
SQuirrel SQL Client	Database	Java	5,351	207,376
WinMerge	Text Processing	C++	7,082	130,283

(b) Experiment 1.2

Name	Domain	Programming language	Number of Revisions	LOC (latest revision)
ThreeCAM	3D Modeling	Java	14	3,854
DatabaseToUML	Database	Java	59	19,695
AdServerBeans	Web	Java	98	7,406
NatMonitor	Network (NAT)	Java	128	1,139
OpenYMSG	Messenger	Java	141	130,072
QMailAdmin	Mail	C	312	173,688
Tritonn	Database	C/C++	100	45,368
Newsstar	Network (NNTP)	C	165	192,716
Hamachi-GUI	GUI, Network (VPN)	C	190	65,790
GameScanner	Game	C/C++	420	1,214,570

Table 3: Overview of Investigation Methods.

Method	Krinke [11]	Lozano and Wermelinger [9]	Proposed method
Target Revisions	A revision per week	All	All
Investigation Unit	Line	Method	Place (consecutive lines)
Measure	ratio of Modified lines	*Work*	Modification frequency

the removed open or close brace is added to the end of the previous line.

Step 3. It detects Duplicate Code. In this step, duplicate code is detected from every target revision, and the detection results are stored into a database. Each detected duplicate code is identified by 3-tuple (v, f, l), where v is the revision number that a given duplicate code was detected; f is the absolute path to the source file where a given duplicate code exists; l is a set of line numbers where duplicate code exists. Note that storing only the start line and the end line of duplicate code is not feasible because a part of duplicate code is non-contiguous.

This step is very time consuming. If the history of the target software includes 1,000 revisions, duplicate code detection is performed 1,000 times. However, this step is fully automated, and no manual work is required.

Step 4. It identifies Differences between Two Consecutive Revisions. In Step 4, we find out modification places between two consecutive revisions with UNIX diff command. As described above, we can get this information by just parsing the output of diff.

Step 5. It Counts the Number of Modifications. In this step, we count the number of modifications of duplicate code and nonduplicate code with the results of the previous two steps. Here, we assume the variable for the number of modifications of duplicate code is MC_d, and the variable for nonduplicate code is MC_n. Firstly, MC_d and MC_n are initialized with 0, then they are increased as follows; if the range of specified modification is completely included in duplicate code, MC_d is incremented; if it is completely included in nonduplicate code, MC_n is incremented; if it is included in both of duplicate code and nonduplicate code, both MC_d and MC_n are incremented. All the modifications are processed with the above algorithm.

Step 6. It calculates MF_d and MF_n. Finally, MF_d and MF_n defined in the formula (4) are calculated with the result of the previous step.

5. Design of Experiment

In this paper, we conduct the following two experiments.

Experiment 1. Compare MF_d and MF_n on 15 open-source software systems.

Experiment 2. Compare the result of the proposed method with 2 previous investigation methods on 5 open-source software systems.

Table 4: Target software systems—Experiment 2.

Name	Domain	Programming language	Number of Revisions	LOC (latest revision)
OpenYMSG	Messenger	Java	194	14,111
EclEmma	Testing	Java	1,220	31,409
MASU	Source Code Analysis	Java	1,620	79,360
TVBrowser	Multimedia	Java	6,829	264,796
Ant	Build	Java	5,412	198,864

We describe these experiments in detail in the reminder of this section.

5.1. Experiment 1

5.1.1. Outline. The purpose of this experiment is to answer our research questions. This experiment consists of the following two subexperiments.

Experiment 1.1. We compare MF_d and MF_n on various size software systems with a scalable detection tool, `CCFinder`.

Experiment 1.2. We compare MF_d and MF_n on small size software systems with 4 detection tools, described in Section 3.1.

The reason why we choose only a single clone detector, `CCFinder`, on Experiment 1.1 is that the experiment took much time. For instance, we took a week to conduct the experiment on SQuirrel SQL Client.

The following items are investigated in each subexperiment.

Item A. Investigate whether duplicate code is modified more frequently than nonduplicate code. In this investigation, we calculate MF_d and MF_n on the entire period.

Item B. Investigate whether MF tendencies differ according to the time.

To answer RQ1, we use the result of Item A of Experiments 1.1 and 1.2. For RQ2, we use the result of Item A of Experiment 1.1. For RQ3, we use Item B of Experiment 1.1 and Experiment 1.2. Finally, for RQ4, we use Item A of Experiment 1.1 and Experiment 1.2.

5.1.2. Target Software Systems. In Experiment 1, we select 15 open source software systems shown in Table 2 as investigation targets. 5 software systems are investigated in Experiment 1.1, and the other software systems are investigated in Experiment 1.2. The criteria for these target software systems are as follows:

(i) the source code is managed with Subversion;

(ii) the source code is written in C/C++ or Java;

(iii) we took care not to bias the domains of the targets.

Table 5: Ratio of duplicate code—Experiment 1.

(a) Experiment 1.1

Software Name	ccf	ccfx	sim	sco
EclEmma	13.1%	—	—	—
FileZilla	22.6%	—	—	—
FreeCol	23.1%	—	—	—
SQuirrel	29.0%	—	—	—
WinMerge	23.6%	—	—	—

(b) Experiment 1.2

Software Name	ccf	ccfx	sim	sco
ThreeCAM	29.8%	10.5%	4.1%	26.2%
DatabaseToUML	21.4%	25.1%	7.6%	11.8%
AdServerBeans	22.7%	18.2%	20.3%	15.9%
NatMonitor	9.0%	7.7%	0.7%	6.6%
OpenYMSG	17.4%	9.9%	5.8%	9.9%
QMailAdmin	34.3%	19.6%	8.8%	—
Tritonn	13.8%	7.5%	5.5%	—
Newsstar	7.9%	4.8%	1.5%	—
Hamachi-GUI	36.5%	23.1%	18.5%	—
GameScanner	23.1%	13.1%	6.6%	—

5.2. Experiment 2

5.2.1. Outline. In Experiment 2, we compare the results of the proposed method and two previously described investigation methods on the same targets. The purpose of this experiment is to reveal whether comparisons of duplicate code and nonduplicate code with different methods always introduce the same result. Also, we evaluate the efficacy of the proposed method comparing to the other methods.

5.2.2. Investigation Methods to Be Compared. Here, we describe 2 investigation methods used in Experiment 2. We choose investigation methods proposed by Krinke [11] (in short, Krinke's method) and proposed by Lozano and Wermelinger [9] (in short, Lozano's method). Table 3 shows the overview of these methods and the proposed method. The selection was performed based on the following criteria.

(i) The investigation is based on the comparison some characteristics between duplicate code and nonduplicate code.

(ii) The method has been published at the time when our research started (at 2010/9).

In the experiments of Krinke's and Lozano's papers, only a single detection tool `Simian` or `CCFinder` was selected. However, in this experiment, we selected 4 detection tools for bringing more valid results.

We developed software tools for Krinke's and Lozano's methods based on their papers. We describe Krinke's method and Lozano's method briefly.

Krinke's Method. Krinke's method compares stability of duplicate code and nonduplicate code [11]. Stability is calculated based on ratios of modified duplicate code and modified nonduplicate code. This method uses not all the revisions but a revision per week.

First of all, a revision is extracted from every week history. Then, duplicate code is detected from every of the extracted revisions. Next, every consecutive two revisions are compared for obtaining where added lines, deleted lines, and changed lines are. With this information, the ratios of added lines, deleted lines, and changed lines on duplicate and nonduplicate code are calculated and compared.

Lozano's Method. Lozano's method categorized Java methods, then compare distributions of maintenance cost based on the categories [9].

Firstly, Java methods are traced based on their owner class's full qualified name, start/end lines, and signatures. Methods are categorized as follows:

AC-Method. Methods that always had duplicate code during their lifetime;

NC-Method. Methods that never had duplicate code during their lifetime;

SC-Method. Methods that sometimes had duplicate code and sometimes did not.

Lozano's method defines the followings where m is a method, P is a period (a set of revisions), and r is a revision.

(i) $ChangedRevisions(m, P)$: a set of revisions that method m is modified in period P,

(ii) $Methods(r)$: a set of methods that exist in revision r,

(iii) $ChangedMethods(r)$: a set of methods that were modified in revision r,

(iv) $CoChangedMethods(m, r)$: a set of methods that were modified simultaneously with method m in revision r. If method m is not modified in revision r, it becomes 0. If modified, the following formula is satisfied:

$$ChangedMethod(r) = m \cup CoChangedMethod(m, r). \tag{5}$$

Table 6: Overall results—Experiment 1.

(a) Experiment 1.1

Software Name	ccf	ccfx	sim	sco
EclEmma	N	—	—	—
FileZilla	N	—	—	—
FreeCol	N	—	—	—
SQuirrel	N	—	—	—
WinMerge	N	—	—	—

(b) Experiment 1.2

Software Name	ccf	ccfx	sim	sco
ThreeCAM	N	C	N	N
DatabaseToUML	N	N	N	N
AdServerBeans	N	N	N	N
NatMonitor	C	C	N	C
OpenYMSG	C	C	C	N
QMailAdmin	C	C	C	—
Tritonn	N	C	N	—
Newsstar	N	N	N	—
Hamachi-GUI	N	N	N	—
GameScanner	C	C	N	—

Then, this method calculates the following formulae with the above definitions. Especially, *work* is an indicator of the maintenance cost:

$$\begin{aligned} likelihood(m, P) &= \frac{ChangedRevisions(m, P)}{\sum_{r \in P} |\, ChangedMethods(r) \,|}, \\ impact(m, P) &= \frac{\sum_{r \in P} |\, CoChangedMethods(m, r) \,| / |Methods(r)|}{|\, ChangedRevisions(m, P) \,|}, \\ work(m, P) &= likelihood(m, P) \times impact(m, P). \end{aligned} \tag{6}$$

In this research, we compare *work* between AC-Method and NC-Method. In addition, we also compare SC-Methods' *work* on duplicate period and nonduplicate period.

5.2.3. Target Software Systems. We chose 5 open-source software systems in Experiment 2. Table 4 shows them. Two targets, OpenYMSG and EclEmma, are selected as well as Experiment 1. Note that the number of revisions and LOC of the latest revision of these two targets are different from Table 2. This is because they had been being in development between the time-lag in Experiments 1 and 2. Every source file is normalized with the rules described in Section 4.2.2 as well as Experiment 1. In addition, automatically generated code and testing code are removed from all the revisions before the investigation methods are applied.

6. Experiment 1—Result and Discussion

6.1. Overview. Table 5 shows the average ratio for each target of Experiment 1. Note that "ccf," "ccfx," "sim," and "sco" in

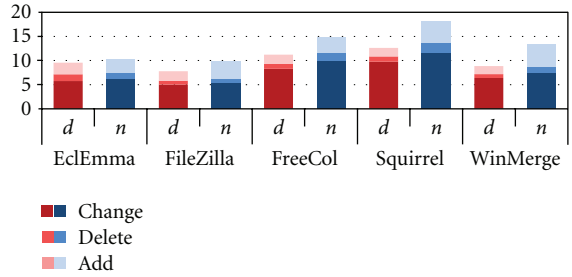


Figure 2: Result of Item A on Experiment 1.1.

Table 7: The average values of MF in Experiment 1.1.

Modification Type	MF	
	Duplicate code	Nonduplicate code
Change	7.0337	8.1039
Delete	1.0216	1.4847
Add	1.9539	3.7378
ALL	10.0092	13.3264

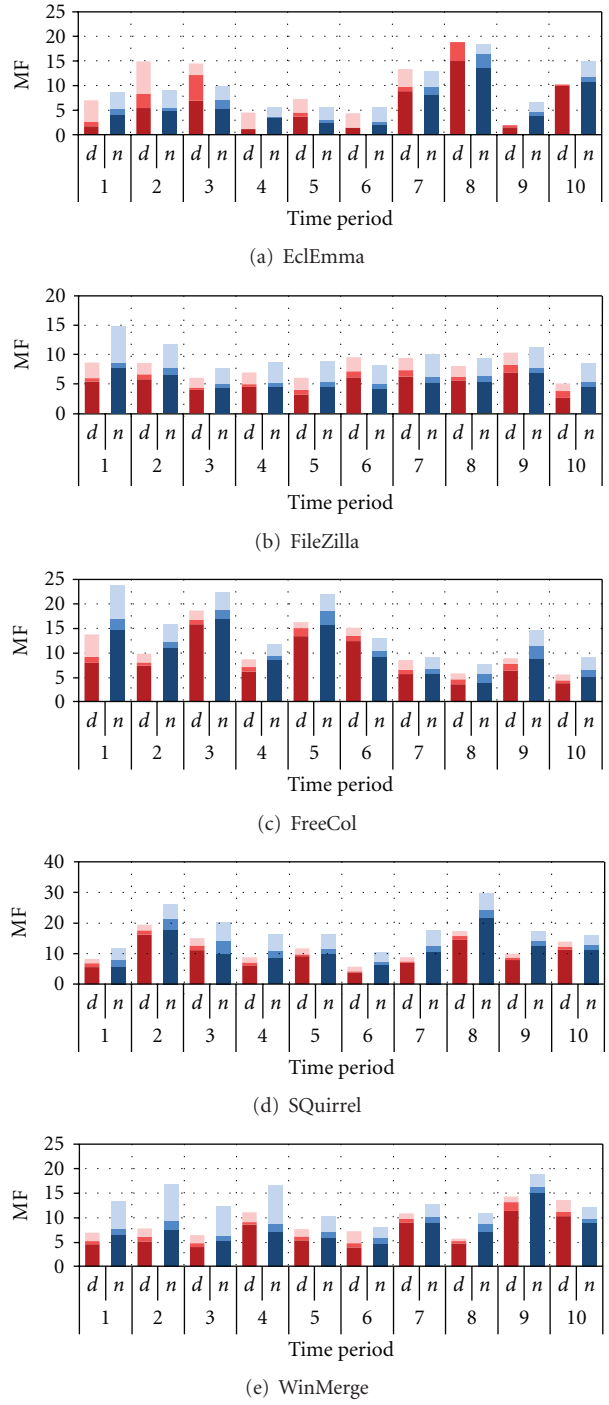
the table are the abbreviated form of `CCFinder`, `CCFinderX`, `Simian`, and `Scorpio`, respectively.

Table 6 shows the overall result of Experiment 1. In this table, "C" means $MF_d > MF_n$ in that case, and "N" means the opposing result. For example, the comparison result in ThreeCAM with `CCFinder` is $MF_d < MF_n$, which means duplicate code is not modified more frequently than nonduplicate code. Note that "—" means the cases that we do not consider because of the following reasons: (1) in Experiment 1.1, we use only `CCFinder`, so that the cases with other detectors are not considered; (2) `Scorpio` can handle only Java, so that the cases in software systems written in C/C++ with `Scorpio` are not considered.

We describe the results in detail in the following subsections.

6.2. Result of Experiment 1.1. Figure 2 shows all the results of Item A on Experiment 1.1. The labels "*d*" and "*n*" in *X*-axis means MF in duplicate code and nonduplicate code, respectively, and every bar consists of three parts, which means *change*, *delete*, and *add*. As shown in Figure 2, MF_d is lower than MF_n on all the target systems. Table 7 shows the average values of MF based on the modification types. The comparison results of MF_d and MF_n show that MF_d is less than MF_n in the cases of all the modification types. However, the degrees of differences between MF_d and MF_n are different for each modification type.

For Item B on Experiment 1.1, first, we divide the entire period into 10 sub-periods and calculate MF on every of the sub periods. Figure 3 shows the result. *X*-axis is the divided periods. Label "1" is the earliest period of the development, and label "10" is the most recent period. In the case of EclEmma, the number of periods that MF_d is greater than MF_n is the same as the number of periods that MF_n is greater than MF_d. In the case of FileZilla, FreeCol, and WinMerge, there is only a period that MF_d is greater than MF_n. In



(a) EclEmma

(b) FileZilla

(c) FreeCol

(d) SQuirrel

(e) WinMerge

Figure 3: Result of Item B on Experiment 1.1 (divided into 10 periods).

the case of Squirrel SQL Client, MF_n is greater than MF_d in all the periods. This result implies that if the number of revisions becomes large, duplicate code tends to become more stable than nonduplicate code. However, the shapes of MF transitions are different from every software system.

For WinMerge, we investigated period "2," where MF_n is much greater than MF_d, and period "10," where is only the

Table 8: Comparing MFs based on programming language and detection tool.

(a) Comparison on programming language

Programming language	MF	
	Duplicate code	Nonduplicate code
Java	20.4370	24.1739
C/C++	49.4868	57.2246
ALL	32.8869	38.3384

(b) Comparison on detection tool

Detection tool	MF	
	Duplicate code	Nonduplicate code
`CCFinder`	38.2790	40.7211
`CCFinderX`	40.3541	40.0774
`Simian`	26.0084	42.1643
`Scorpio`	20.9254	24.1628
ALL	32.8869	38.3384

Table 9: The average values of MF in Experiment 1.2.

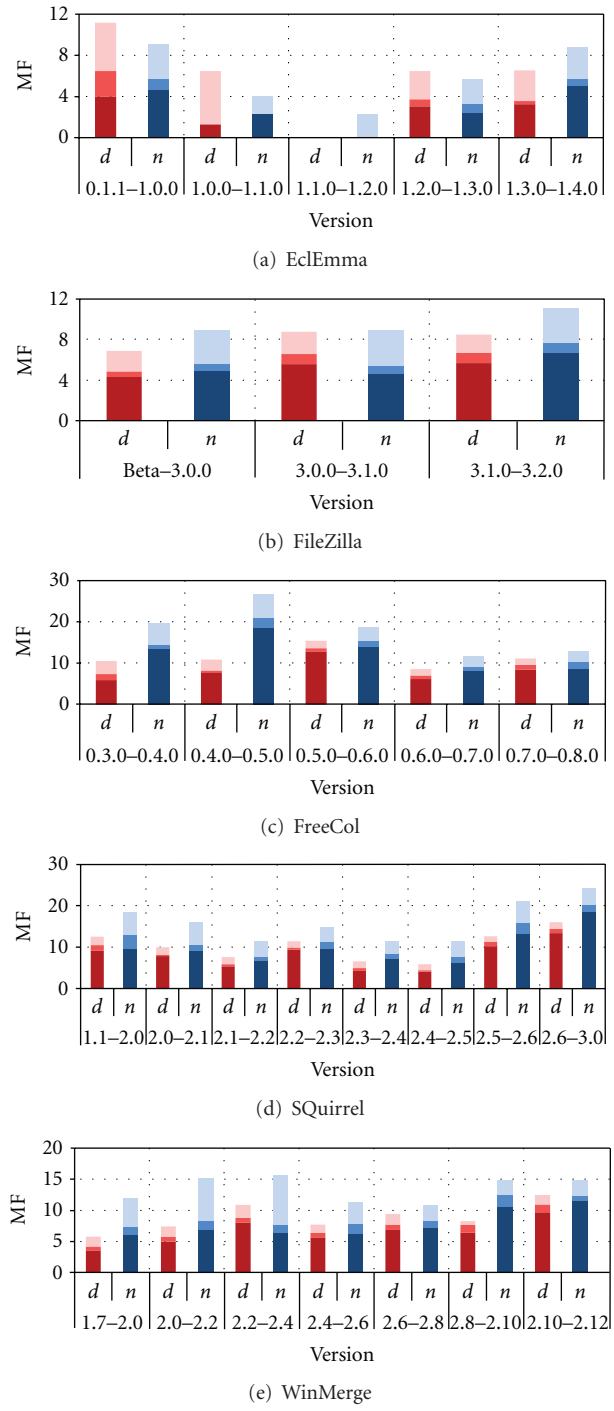
Modification type	MF	
	Duplicate code	Nonduplicate code
Change	26.8065	29.2549
Delete	3.8706	3.5228
Add	2.2098	5.5608
ALL	32.8869	38.3384

period that MF_d is greater than MF_n. In period "10," there are many modifications on test cases. The number of revisions that test cases are modified is 49, and the ratio of duplicate code in test cases is 88.3%. Almost all modifications for test cases are performed on duplicate code, so that MF_d is greater than MF_n. Omitting the modifications for test cases, MF_d and MF_n became inverted. However, there is no modification on test cases in period "2," so that MF_d is less than MF_n in this case.

Moreover, we divide the entire period by release dates and calculate MF on every period. Figure 4 shows the result. As the figure shows, MF_d is less than MF_n in all the cases for FileZilla, FreeCol, SQuirrel, and WinMerge. For EclEmma, there are some cases that $MF_d > MF_n$ at the release level. Especially, duplicate code is frequently modified in the early releases.

Although MF_d is greater than MF_n in the period "6" in Freecol and the period "10" in WinMerge, MF_d is less than MF_n in all cases at the release level. This indicates that duplicate code is sometimes modified intensively in a short period, nevertheless it is stable than nonduplicate code in a long term.

The summary of Experiment 1 is that duplicate code detected by `CCFinder` was modified less frequently than nonduplicate code. Consequently, we conclude that duplicate code detected by `CCFinder` does not have a negative impact on software evolution even if the target software is large and its period is long.



(a) EclEmma

(b) FileZilla

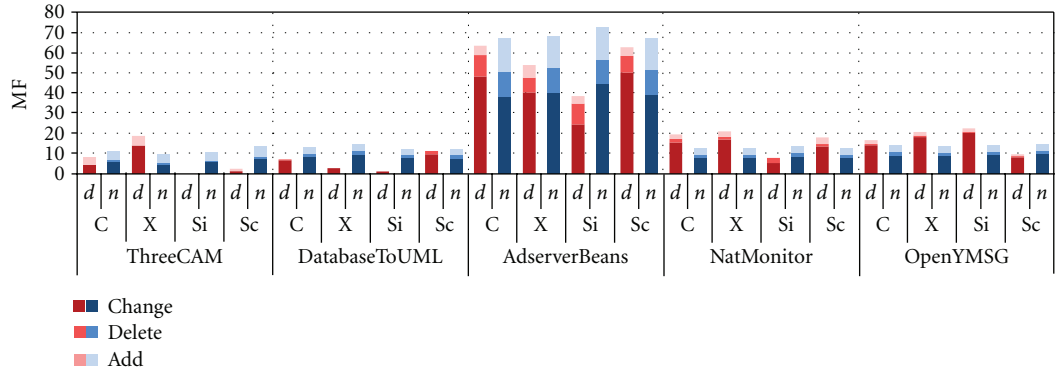
(c) FreeCol

(d) SQuirrel

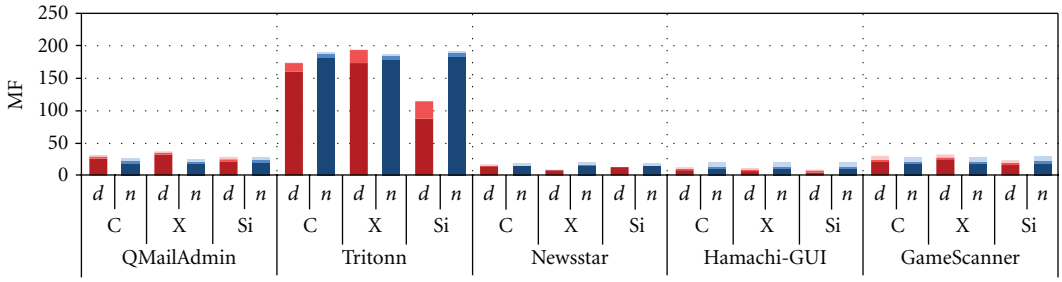
(e) WinMerge

Figure 4: Result of Item B on Experiment 1.1 (divided by releases).

6.3. Result of Experiment 1.2. Figure 5 shows all the results of Item A on Experiment 1.2. In Figure 5, the detection tools are abbreviated as follows: `CCFinder` → `C`; `CCFinderX` → `X`; `Simian` → `Si`; `Scorpio` → `Sc`. There are the results of 3 detection tools except `Scorpio` on C/C++ systems, because `Scorpio` does not handle C/C++. MF_d is less than MF_n in the 22 comparison results out of 35. In the 4 target systems out of 10, duplicate code is modified less frequently than



(a) Java software



(b) C/C++ Software

Figure 5: Result of Item A on Experiment 1.2.

nonduplicate code in the cases of all the detection tools. In the case of the other 1 target system, MF_d is greater than MF_n in the cases of all the detection tools. In the remaining systems, the comparison result is different for the detection tools. Also, we compared MF_d and MF_n based on programming language and detection tools. The comparison result is shown in Table 8. The result shows that MF_d is less than MF_n on all the programming language, and MF_d is less than MF_n on the 3 detectors, `CCFinder`, `Simian`, and `Scorpio`, meanwhile the opposing result is shown in the case of `CCFinderX`. We also compared MF_d and MF_n based on modification types. The result is shown in Table 9. As shown in Table 9, MF_d is less than MF_n in the cases of change and addition, meanwhile the opposing result is shown in the case of deletion.

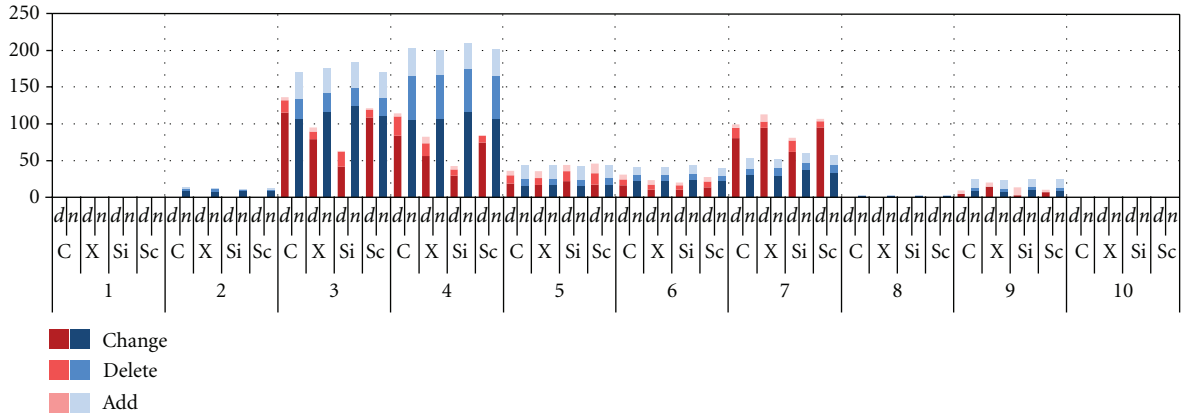
We investigated whether there is a statistically significant difference between MF_d and MF_n by t-test. The result is that, there is no difference between them where the level of significance is 5%. Also, there is no significant difference in the comparison based on programming language and detection tool.

For Item B on Experiment 1.2, we divide the whole period into 10 subperiods likewise Experiment 1.1. Figure 6 shows the result. In this experiment, we observed that the tendencies of MF transitions loosely fall into three categories: (1) MF_d is lower than MF_n almost of all the divisions; (2) MF_d is greater than MF_n in the early divisions, meanwhile the opposite tendency is observed in the late divisions; (3) MF_d is less than MF_n in the early divisions, meanwhile the opposite tendency is observed in the late divisions. Figure 6 shows the result of the 3 systems on which we observed remarkable tendencies of every category.

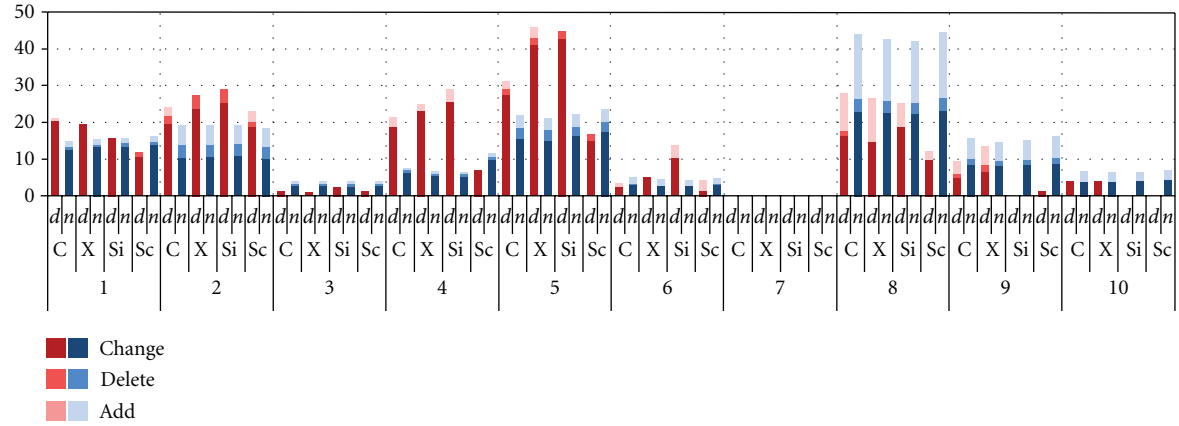
In Figure 6(a), period "4" shows that MF_n is greater than MF_d on all the detection tool meanwhile period "7" shows exactly the opposite result. Also, in period "5," there are hardly differences between duplicate code and nonduplicate code. We investigated the source code of period "4." In this period, many source files were created by copy-and-paste operations, and a large amount of duplicate code was detected by each detection tool. The code generated by copy-and-paste operations was very stable meanwhile the other source files were modified as usual. This is the reason why MF_n is much greater than MF_d in period "4."

Figure 6(b) shows that duplicate code tends to be modified more frequently than nonduplicate code in the anterior half of the period meanwhile the opposite occurred in the posterior half. We found that there was a large number of duplicate code that was repeatedly modified in the anterior half. On the other hand, there was rarely such duplicate code in the posterior half.

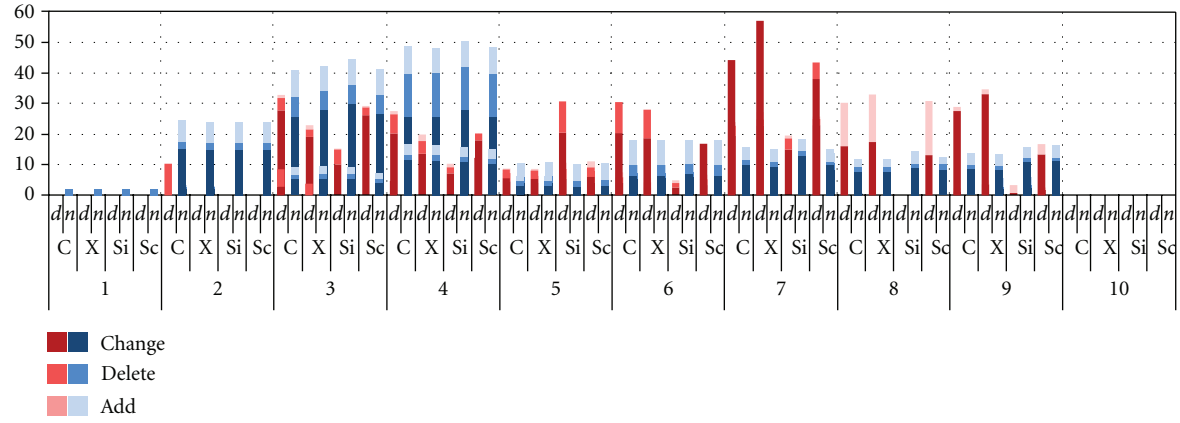
Figure 6(c) shows the opposite result of Figure 6(b). That is, duplicate code was modified more frequently in the posterior half of the period. In the anterior half, the amount of duplication was very small, and modifications were rarely performed on it. In the posterior half, amount of duplicate code became large, and modifications were performed on it repeatedly. In the case of `Simian` detection, no duplicate code was detected except period "5." This is because `Simian`



(a) AdServerBeans



(b) OpenYMSG



(c) NatMonitor

Figure 6: Result of Item B on Experiment 1.2 (divided into 10 periods).

detects only the exact-match duplicate code meanwhile the other tools detect exact match and renamed duplicate code in the default setting.

In Experiment 1.1, we investigate MF tendencies at the release level. However, we cannot apply the same investigation way to Experiment 1.2. This is because the target software systems in Experiment 1.2 is not enough mature to have multiple releases. Instead, we investigate MF tendencies at the most fine-grained level, at the revision level. Figure 7 shows the result of the investigation at the revision level for AdServerBeans, OpenYMSG, and NatMonitor. The X-axis of each graph indicates the value of $\mathrm{MF}_d - \mathrm{MF}_n$. Therefore, if the value is greater than 0, MF_d is greater than MF_n at the revision and vice versa. For AdServerBeans, MF tendencies are similar for every detection tool except revision 21 to 26. For other 2 software systems, MF comparison results differ

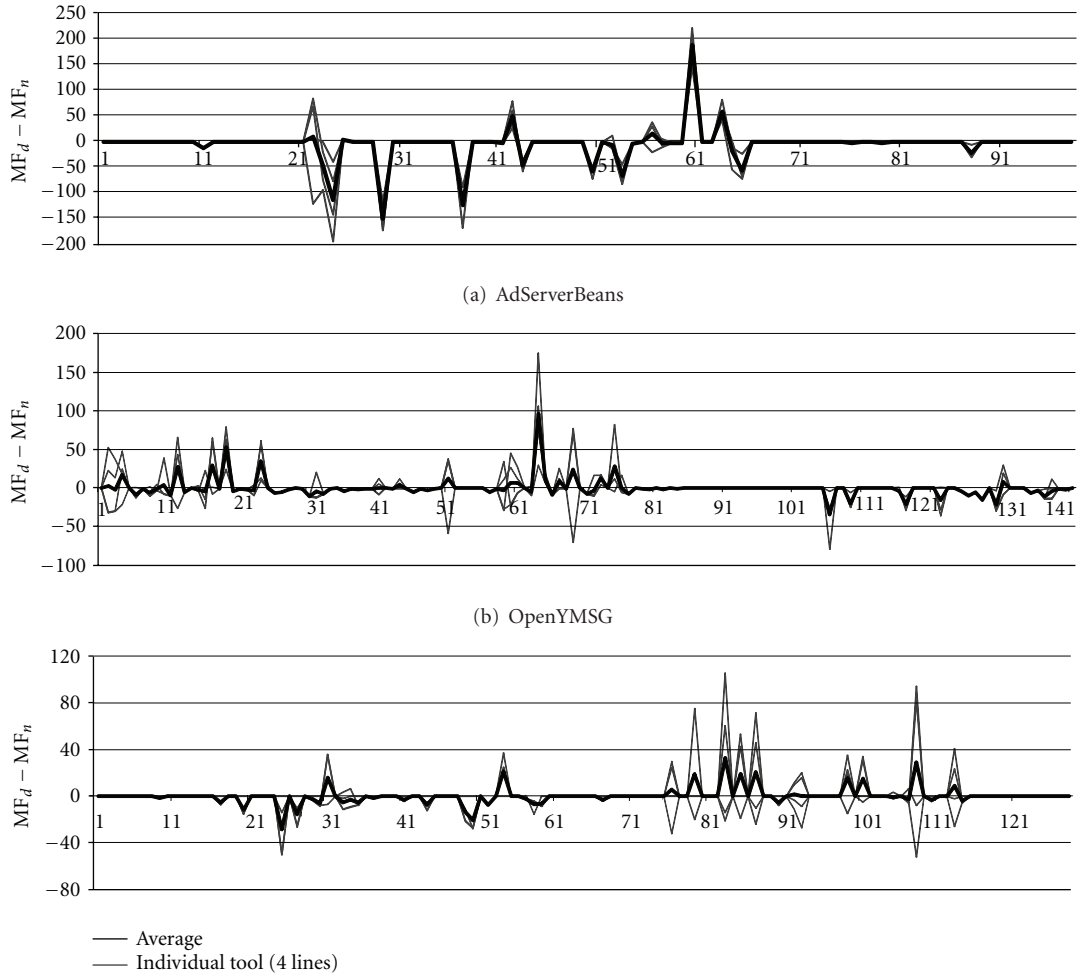


Figure 7: Result of Item B on Experiment 1-2 (for each revision).

from each detection tool in most of the revisions. As the figures show, tendencies of MF transition differ from clone detectors nevertheless there seems to be small differences between clone detectors in 10 sub-periods division. However, these graphs do not consider modification types. Therefore, we cannot judge what type of modification frequently occurred from the graphs.

The summary of Experiment 1.2 is as follows: we found some instances that duplicate code was modified more frequently than nonduplicate code in a short period on each detection tool; however, in the entire period, duplicate code was modified less frequently than nonduplicate code on every target software with all the detection tools. Consequently, we conclude that the presence of duplicate code does not have a seriously-negative impact on software evolution.

6.4. Answers for RQs

RQ1: Is duplicate code more frequently modified than nonduplicate code? The answer is *No*. In Experiment 1.1, we found that MF_d is lower than MF_n in all the target systems. Also, we found a similar result in Experiment 1.2: 22 comparison results out of 35 show that MF_d is lower than MF_n, also MF_d is lower than MF_n in average. This result indicates that the presence of duplicate code does not seriously affect software evolution, which is different from the common belief.

RQ2: Are the comparison results of stability between duplicate code and nonduplicate code different from multiple detection tools? The answer is *Yes*. In Experiment 1.2, the comparison results with `CCFinderX` are different from the results with other 3 detectors. Moreover, MF_n is much greater than MF_d in the case of `Simian`. At present, we cannot find the causes of the difference of the comparison results. One of the causes may be the ratio of duplicate code. The ratio of duplicate code is quite different for each detection tool on the same software. However, we cannot see any relation between the ratio of duplicate code and MF.

RQ3: Is duplicate code modified uniformly throughout its lifetime? The answer is *No*. In Item B of Experiments 1.1

and 1.2, there are some instances that duplicate code was modified more frequently than nonduplicate code in a short period though MF_d is less than MF_n in the whole period. However, these MFs tendencies depend on target software systems, so that we cannot find characteristics of such variability.

RQ4: Are there any differences in the comparison results on modification types? The answer is *Yes*. In Experiment 1.1, MF_d is less than MF_n on all the modification types. However, there is a small difference between MF_d and MF_n in the case of deletion, meanwhile there is a large difference in the case of addition. In Experiment 1.2, MF_d is less than MF_n in the cases of change and addition. Especially, MF_n is more than twice as large as MF_d in the case of addition. However, MF_d is greater than MF_n in the case of deletion. These results show that deletion tends to be affected by duplicate code, meanwhile addition tends not to be affected by duplicate code.

6.5. Discussion. In Experiment 1, we found that duplicate code tends to be more stable than nonduplicate code, which indicates that the presence of duplicate code does not have a negative impact on software evolution. We investigated how the software evolved in the period, and we found that the following activities should be a part of factors that duplicate code is modified less frequently than nonduplicate code.

Reusing Stable Code. When implementing new functionalities, reusing stable code is a good way to reduce the number of introduced bugs. If most of duplicate code is reused stable code, MF_d becomes less than MF_n.

Using Generated Code. Automatically generated code is rarely modified manually. Also, the generated code tends to be duplicate code. Consequently, if the amount of generated code is high, MF_d will become less than MF_n.

On the other hand, there are some cases that duplicate code was more frequently modified than nonduplicate code in a short period. The period "7" on AdServerBeans (Experiment 1.2, Item B) is one of these instances. We analyzed the source code of this period to detect why MF_d was greater than MF_n in this period though the opposite results were shown in the other periods. Through the analysis, we found that there are some instances that the same modifications were applied to multiple places of code.

Algorithm 2 shows an example of unstable duplicate code. There are 5 code fragments that are similar to this fragment. Firstly, lines labeled with "%" (shown in Algorithm 2(b)) were modified to replace the getter methods into directly accesses to fields. In the next, a line labeled with "#" is removed (shown in Algorithm 2(c)). These two modifications were concentrically conducted in period "7." Reusing unstable code like this example can cause additional costs for software maintenance. Moreover, a code fragment was not simultaneously changed with its correspondents at the second modification. If this inconsistent change was introduced unintentionally, it might cause a bug. If so, this is a typical situation that duplicate code affects software evolution.

Table 10: Ratio of duplicate code—Experiment 2.

Software Name	ccf	ccfx	sim	sco
OpenYMSG	12.4%	6.2%	2.7%	5.5%
EclEmma	6.9%	4.8%	2.0%	3.7%
MASU	25.6%	26.5%	11.3%	15.4%
TVBrowser	13.6%	10.9%	5.4%	19.0%
Ant	13.9%	12.1%	6.2%	15.6%

Table 11: Overall results—Experiment 2.

Software Name	Method	Tools			
		ccf	ccfx	sim	sco
OpenYMSG	Proposed	N	C	C	N
	Krinke	N	C	C	N
	Lozano	—	—	N	—
EclEmma	Proposed	N	N	N	N
	Krinke	N	N	N	C
	Lozano	N	N	—	—
MASU	Proposed	C	N	C	C
	Krinke	C	C	C	C
	Lozano	C	C	C	C
TVBrowser	Proposed	N	N	N	N
	Krinke	C	C	C	C
	Lozano	C	C	C	C
Ant	Proposed	N	N	N	N
	Krinke	C	C	C	C
	Lozano	C	C	C	C

7. Experiment 2—Result and Discussion

7.1. Overview. Table 10 shows the average ratios of duplicate code in each target, and Table 11 shows the comparison results of all the targets. In Table 11, "C" means that duplicate code requires more cost than nonduplicate code, and "N" means its opposite. The discriminant criteria of "C" and "N" are different in each investigation method.

In the proposed method, if MF_d is lower than MF_n, the column is labeled with "C," and the column is labeled with "N" in its opposite case.

In Krinke's method, if the ratio of *changed* and *deleted* lines of code on duplicate code is greater than *changed* and *deleted* lines on nonduplicate code, the column is labeled with "C," and in its opposite case the column is labeled with "N." Note that herein we do not consider *added* lines because the amount of *add* is the lines of code added in the next revision, not in the current target revision.

In Lozano's method, if *work* in AC-Method is statistically greater than one in NC-Method, the column is labeled with "C." On the other hand, if *work* in NC-Method is statistically greater than one in AC-Method, the column is labeled with "N." Here, we use Mann-Whitney's U test under setting

(a) *Before Modification*

```
int offsetTmp = dataGridDisplayCriteria
        .getItemsPerPage() *
           (dataGridDisplayCriteria.getPage() -1);
    if (offsetTmp > 0) --offsetTmp;
    if (offsetTmp < 0) offsetTmp = 0;
final int offset = offsetTmp;
 String sortColumn =
         dataGridDisplayCriteria.getSortColumn();
Order orderTmp =
        dataGridDisplayCriteria.getOrder()
             .equals(AdServerBeansConstants.ASC) ?
                    Order.asc(sortColumn) :
                       Order.desc(sortColumn);
```

(b) *After 1st Modification*

```
int offsetTmp = dataGridDisplayCriteria
        .getItemsPerPage() *
             (dataGridDisplayCriteria.getPage() -1);
    if (offsetTmp > 0) --offsetTmp;
    if (offsetTmp < 0) offsetTmp = 0;
final int offset = offsetTmp;
String sortColumn =
%         dataGridDisplayCriteria.sortColumn;
Order orderTmp =
%            dataGridDisplayCriteria.order
                .equals(AdServerBeansConstants.ASC) ?
                       Order.asc(sortColumn) :
                          Order.desc(sortColumn);
```

(c) *After 2nd Modification*

```
int offsetTmp = dataGridDisplayCriteria
        .getItemsPerPage() *
             (dataGridDisplayCriteria.getPage() -1);
#
    if (offsetTmp < 0) offsetTmp = 0;
final int offset = offsetTmp;
String sortColumn =
        dataGridDisplayCriteria.sortColumn;
Order orderTmp =
        dataGridDisplayCriteria.order
             .equals(AdServerBeansConstants.ASC) ?
                    Order.asc(sortColumn) :
                       Order.desc(sortColumn);
```

Algorithm 2: An example of unstable duplicate code.

5% as the level of significance. If there is no statistically significant difference in AC- and NC-Method, we compare *work* in duplicate period and nonduplicate period in SC-Method with Wlcoxon's singed-rank test. We also set 5% as the level of significance. If there is no statistically significant difference, the column is labeled with "—."

As this table shows, different methods and different tools brought almost the same result in the case of EclEmma and MASU. On the other hand, in the case of other targets, we get different results with different methods or different tools. Especially, in the case of TVBrowser and Ant, the proposed method brought the opposite result to Lozano's and Krinke's method.

7.2. Result of MASU. Herein, we show comparison figures of MASU. Figure 8 shows the results of the proposed method. In this case, all the detection tools except `CCFinderX` brought the same result that duplicate code is more frequently modified than nonduplicate code. Figure 9 shows the results of Krinke's method on MASU. As this figure shows, the comparison of all the detection detectors brought the same result that duplicate code is less stable than nonduplicate code. Figure 10 shows the results of Lozano's method on MASU with `Simian`. Figure 10(a) compares AC-Method and NC-Method. X-axis indicates maintenance cost (*work*) and Y-axis indicates cumulated frequency of methods. For readability, we adopt logarithmic axis on X-axis. In this

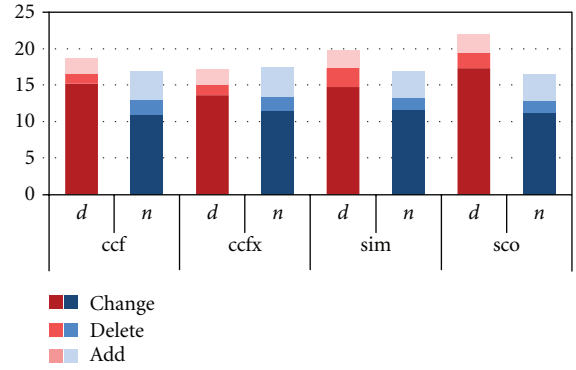


Figure 8: Result of the proposed method on MASU.

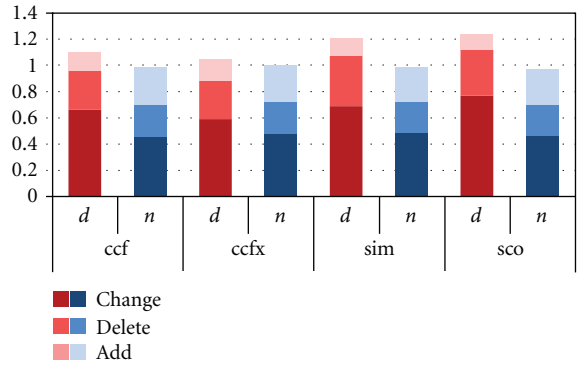
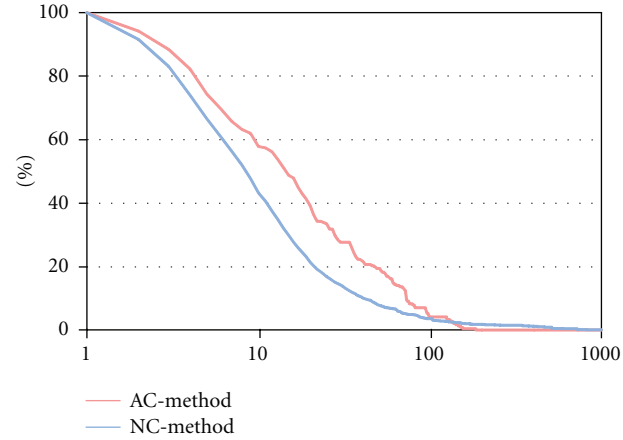
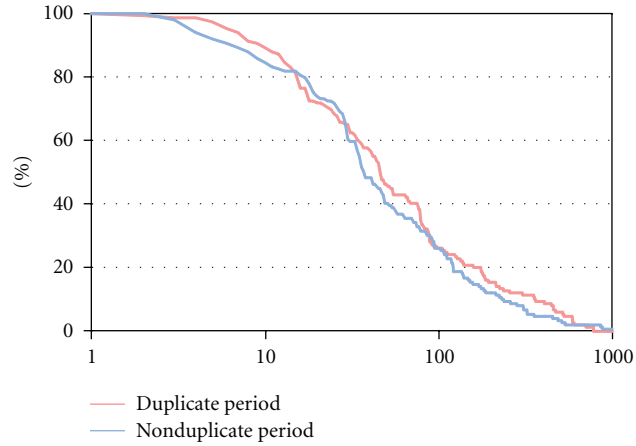


Figure 9: Result of Krinke's method on MASU.



(a) AC-Method versus NC-Method



(b) SC-Method

Figure 10: Result of Lozano's Method on MASU with `Simian`.

case, AC-Method requires more maintenance cost than NC-Method. Also, Figure 10(b) compares duplicate period and nonduplicate period of SC-Method. In this case, the maintenance cost in duplicate period is greater than in nonduplicate period.

In the case of MASU, Krinke's method and Lozano's method regard duplicate code as requiring more cost than nonduplicate code in the cases of all the detection tools. The proposed method indicates that duplicate code is more frequently modified than nonduplicate code with `CCFinder`, `Simian`, and `Scorpio`. In addition, there is little differences between MF_d and MF_n in the result of the proposed method with `CCFinderX`, which is the only case that duplicate code is more stable than nonduplicate code. Considering all the results, we can say that duplicate code has a negative impact on software evolution on MASU. This result is reliable because all the investigation methods show such tendencies.

7.3. Result of OpenYMSG. Figures 11, 12, and 13 show the result of the proposed method, Krinke's method, and Lozano's method on OpenYMSG. In the cases of the proposed method and Krinke's method, duplicate code is regarded as having a negative impact with `CCFinderX` and `Simian`, meanwhile the opposing results are shown with `CCFinder` and `Scorpio`. In Lozano's method with `Simian`, duplicate code is regarded as not having a negative impact. Note that we omit the comparison figure on SC-Method because there are only 3 methods that are categorized into SC-Method.

As these figures show, the comparison results are different for detection tools or investigation methods. Therefore, we cannot judge whether the presence of duplicate code has a negative impact or not on OpenYMSG.

7.4. Discussion. In the case of OpenYMSG, TVBrowser, and Ant, different investigation methods and different tools brought opposing results. Figure 14 shows an actual modification in Ant. Two methods were modified in this modification. The hatching parts are detected duplicate code and frames in them mean pairs of duplicate code between two methods. Vertical arrows show modified lines between this modification and the next (77 lines of code were modified).

This modification is a refactoring, which extracts the duplicate instructions from the two methods and merges them as a new method. In the proposed method, there are 2 modification places in duplicate code and 4 places in nonduplicate code, so that MF_d and MF_n become 51.13 and 18.13, respectively. In Krinke's method, DC + CC and

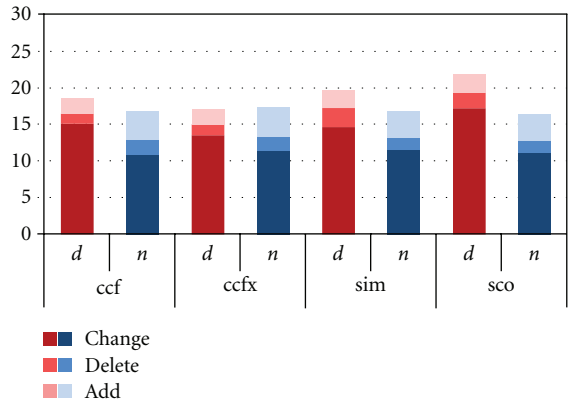


Figure 11: Result of the proposed method on OpenYMSG.

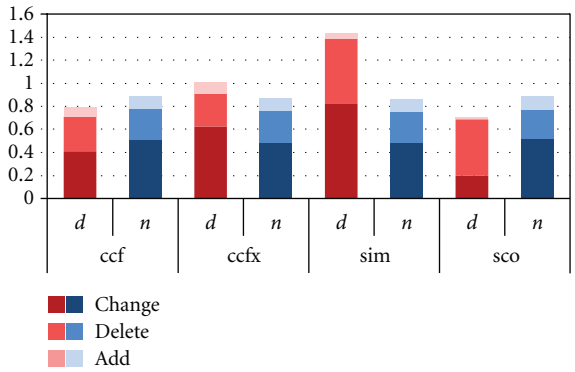


Figure 12: Result of Krinke's method on OpenYMSG.

DN + CN become 0.089 and 0.005, where DC, CC, DN, and CN indicate the ratio of deleted lines on duplicate code, changed lines on duplicate code, deleted lines on nonduplicate code, and deleted lines on nonduplicate code, respectively.

In this case, both the proposed method and Krinke's method regard duplicate code requiring more maintenance cost than nonduplicate code. However, there is a great difference in Krinke's method than the proposed method: in the proposed method, duplicate code is modified about 2.8 times as frequently as nonduplicate code; meanwhile, in Krinke's method, duplicate code is modified 17.8 times as frequently as nonduplicate code. This is caused by the difference of the barometers used in each method. In Krinke's method, the barometer depends on the amount of modified lines, meanwhile the barometer depends on the amount of modified places in the proposed method. This example is one of the refactorings on duplicate code. In Krinke's method, if removed duplicate code is large, duplicate code is regarded as having more influence. However, in the cases of duplicate code removal, we have to spend much effort if the number of duplicate fragments is high. Therefore, we can say that the proposed method can accurately measure the influence of duplicate code in this case.

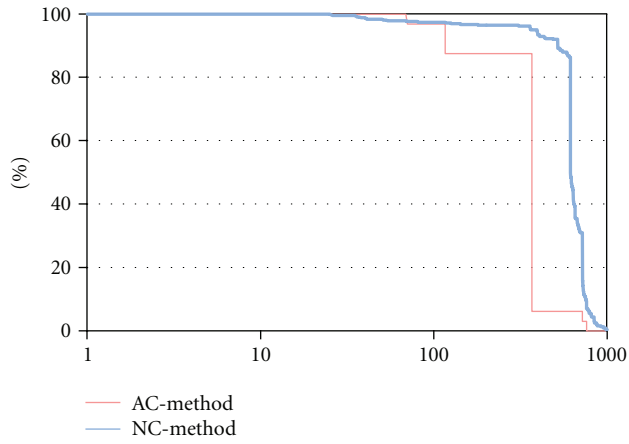


Figure 13: Result of Lozano's method on OpenYMSG with `Simian`.

This is an instance that is advantageous for the proposed method. However, we cannot investigate all the experimental data because the amount of the data is too vast to conduct manual checking for all the modifications. There is a possibility that the proposed method cannot accurately evaluate the influence of duplicate code in some situations.

In Experiment 2, we found that the different investigation methods or different detectors draw different results on the same target systems. In Experiment 1, we found that duplicate code is less frequently modified than nonduplicate code. However, the result of Experiment 2 shows that we cannot generalize the result of Experiment 1. We have to conduct more experiments and analyze the results of them in detail to gain more generic.

8. Threats to Validity

This section describes threats to validity of this study.

8.1. Features of Every Modification. In this study, we assume that cost required for every modification is equal to one another. However, the cost is different between every modification in the actual software evolution. Consequently, the comparison based on MF may not appropriately represent the cost required for modifying duplicate code and nonduplicate code.

Also, when we modify duplicate code, we have to consider maintaining the consistency between the modified duplicate code and its correspondents. If the modification lacks the consistency by error, we have to remodify them for repairing the consistency. The effort for consistency is not necessary for modifying nonduplicate code. Consequently, the average cost required for duplicate code may be different from the one required for nonduplicate code. In order to compare them more appropriately, we have to consider the cost for maintaining consistency.

Moreover, distribution of source code that should be modified are not considered. However, it differs from

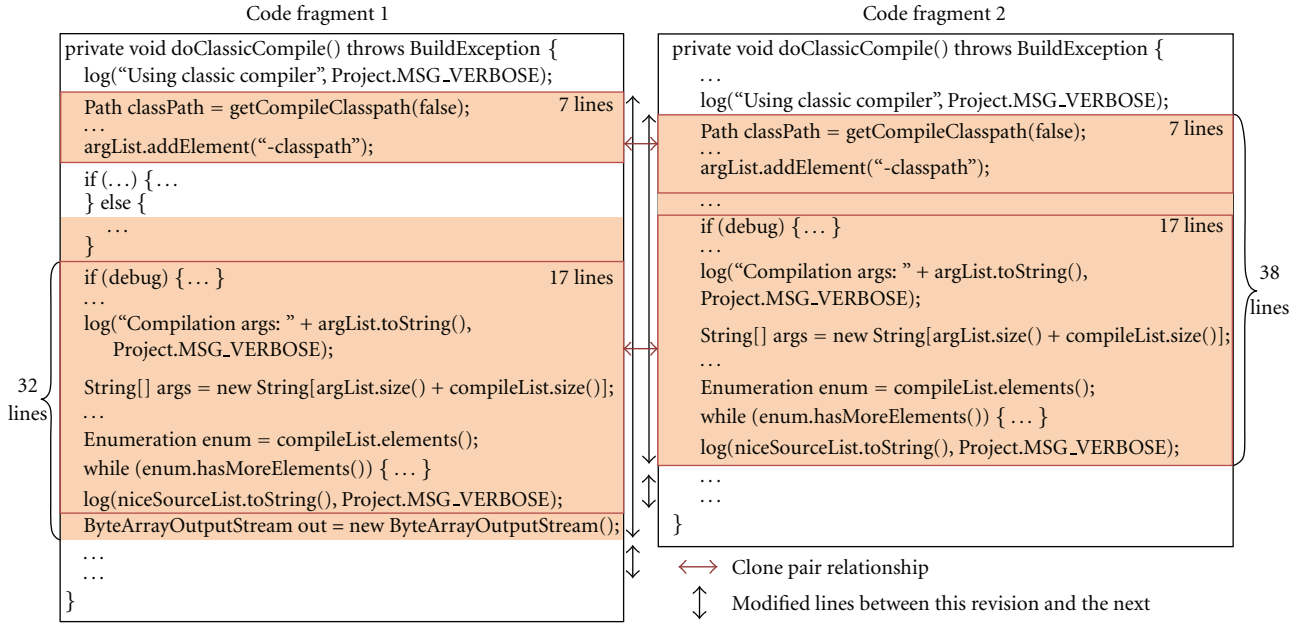


Figure 14: An Example of Modification.

every modification, thus we may get different results by considering the distribution of source code.

8.2. Identifying the Number of Modifications. In this study, modifying consecutive multiple lines are regarded as a single modification. However, it is possible that such an automatically processing identifies the incorrect number of modifications. If multiple lines that were not contiguous are modified for fixing a single bug, the proposed method presumes that multiple modifications were performed. Also, if multiple consecutive lines were modified for fixing two or more bugs by chance, the proposed method presumes that only a single modification was performed. Consequently, it is necessary to manually identify modifications if we have to use the exactly correct number of modifications.

Besides, we investigated how many the identified modifications occurred across the boundary of duplicate code and nonduplicate code. If this number is high, then the analysis suspects because such modifications increase both the counts at the same time. The investigation result is that, in the highest case, the ratio of such modifications is 4.8%. That means that almost all modifications occurred within either duplicate code or nonduplicate code.

8.3. Category of Modifications. In this study, we counted all the modifications, regardless of their categories. As a result, the number of modifications might be incorrectly increased by unimportant modifications such as format transformation. A part of unimportant modifications remained even if we had used the normalized source code described in Section 4.2.2. Consequently, manual categorization for the modifications is required for using the exactly correct number of modifications.

Also, the code normalization that we used in this study removed all the comments in the source files. If considerable cost was expended to make or change code comments on the development of the target systems, we incorrectly missed the cost.

8.4. Property of Target Software. In this study, we used only open-source software systems, so that different results may be shown with industrial software systems. Some researchers pointed out that industrial software systems include much duplicate code [24, 25]. Consequently, duplicate code may not be managed well in industrial software, which may increase MF_d. Also, properties of industrial software are quite different from ones of open source software. In order to investigate the impact of duplicate code on industrial software, we have to compare MF on industrial software itself.

8.5. Settings of Detection Tools. In this study, we used default settings for all the detection tools. If we change the settings, different results will be shown.

9. Conclusion

This paper presented an empirical study on the impact of the presence of duplicate code on software evolution. We assumed that if duplicate code is modified more frequently than nonduplicate code, the presence of duplicate code affects software evolution and compared the stability of duplicate code and nonduplicate code. To evaluate from a different standpoint from previous studies, we used a new indicator, modification frequency, which is calculated with the number of modified places of code. Also, we used 4

duplicate code detection tools to reduce the bias of duplicate code detectors. We conducted an experiment on 15 open-source software systems, and the result showed that duplicate code was less frequently modified than nonduplicate code. We also found some cases that duplicate code was intensively modified in a short period though duplicate code was stable than nonduplicate code in the whole development period.

Moreover, we compared the proposed method to other 2 investigation methods to evaluate the efficacy of the proposed method. We conducted an experiment on 5 open-source software systems, and in the cases of 2 targets, we got the opposing results to other 2 methods. We investigated the result in detail and found some instances that the proposed method could evaluate more accurately than other methods.

In this study, we found that duplicate code tends to be stable than nonduplicate code. However, more studies are required to generalize this result, because we found that different investigation methods may bring different results. As future work, we are going to conduct more studies with other settings to get the characteristics of harmful duplicate code.

Acknowledgments

This research is being conducted as a part of the Stage Project, the Development of Next Generation IT Infrastructure, supported by the Ministry of Education, Culture, Sports, Science, and Technology of Japan. This study has been supported in part by Grant-in-Aid for Scientific Research (A) (21240002) and Grant-in-Aid for Exploratory Research (23650014) from the Japan Society for the Promotion of Science, and Grant-in-Aid for Young Scientists (B) (22700031) from Ministry of Education, Science, Sports and Culture. This paper is an extended version of earlier conference papers [26, 27].

References

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 804–818, 2007.

[2] M. D. Wit, A. Zaidman, and A. V. Deursen, "Managing code clones using dynamic change tracking and resolution?" in *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09)*, pp. 169–178, September 2009.

[3] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[4] C. J. Kapser and M. W. Godfrey, "'cloning considered harmful' considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[5] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 187–196, September 2005.

[6] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Science of Computer Programming*, vol. 77, no. 6, pp. 760–776, 2012.

[7] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings of the 8th IEEE International Software Metrics Symposium*, pp. 87–94, June 2002.

[8] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: a change based experiment," in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR '07)*, May 2007.

[9] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proceedings of the 24th International Conference on Software Maintenance*, pp. 227–236, September 2008.

[10] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the relation between changeability decay and the characteristics of clones and methods," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 100–109, September 2008.

[11] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2008*, pp. 57–66, September 2008.

[12] N. Göde and J. Harder, "Clone stability," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*, pp. 65–74, March 2011.

[13] J. Krinke, "Is cloned code older than non-cloned code?" in *Proceedings of the 5th International Workshop on Software Clones (IWSC '11)*, pp. 28–33, May 2011.

[14] F. Rahman, C. Bird, and P. Devanbu, "Clones: what is that smell?" in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pp. 72–81, May 2010.

[15] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *33rd International Conference on Software Engineering (ICSE '11)*, pp. 311–320, May 2011.

[16] S. G. Eick, T. L. Graves, A. F. Karr, U. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[17] N. Göde, "Evolution of type-1 clones," in *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*, pp. 77–86, September 2009.

[18] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 36–43, October 2002.

[19] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[20] CCFinderX, http://www.ccfinder.net/ccfinderx.html/.

[21] Simian, http://www.harukizaemon.com/simian/.

[22] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*, pp. 75–84, March 2011.

[23] Scorpio, http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/Scorpio/.

[24] S. Ducasse, M. Rieger, and S. Demeyer, "Language independent approach for detecting duplicated code," in *Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM '99)*, pp. 109–118, September 1999.

[25] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. I. Matsumoto, and H. Kudo, "Software analysis by code clones in open source software," *Journal of Computer Information Systems*, vol. 45, no. 3, pp. 1–11, 2005.

[26] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software," in *Proceedings of the 4th the International Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, September 2010.

[27] Y. Sasaki, K. Hotta, Y. Higo, and S. Kusumoto, "Is duplicate code good or bad? an empirical study with multiple investigation methods and multiple detection tools," in *Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE '11)*, Hiroshima, Japan, November 2011.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 524636, 9 pages
doi:10.1155/2012/524636

Research Article

A Comparative Study of Data Transformations for Wavelet Shrinkage Estimation with Application to Software Reliability Assessment

Xiao Xiao and Tadashi Dohi

Department of Information Engineering, Graduate School of Engineering, Hiroshima University, 1-4-1 Kagamiyama, Higashi-Hiroshima 739-8527, Japan

Correspondence should be addressed to Xiao Xiao, xiaoxiao@rel.hiroshima-u.ac.jp

Received 6 January 2012; Revised 6 March 2012; Accepted 6 March 2012

Academic Editor: Chin-Yu Huang

In our previous work, we proposed wavelet shrinkage estimation (WSE) for nonhomogeneous Poisson process (NHPP)-based software reliability models (SRMs), where WSE is a data-transform-based nonparametric estimation method. Among many variance-stabilizing data transformations, the Anscombe transform and the Fisz transform were employed. We have shown that it could provide higher goodness-of-fit performance than the conventional maximum likelihood estimation (MLE) and the least squares estimation (LSE) in many cases, in spite of its non-parametric nature, through numerical experiments with real software-fault count data. With the aim of improving the estimation accuracy of WSE, in this paper we introduce other three data transformations to preprocess the software-fault count data and investigate the influence of different data transformations to the estimation accuracy of WSE through goodness-of-fit test.

1. Introduction

In the field of software reliability engineering, the quantitative assessment of software reliability has become one of the main issues of this area. Especially, people are interested in finding several *software intensity functions* from the software-fault count data observed in the software testing phases, since the software intensity function in discrete time denotes the number of software faults detected per unit time. This directly makes it possible to estimate the number of remaining software faults and the quantitative software reliability, which is defined as the probability that software system does not fail during a specified time period under a specified operational environment. Moreover, these evaluation measures can be used in the decision making such as allocation of development resources and software release scheduling. Therefore, we are interested in developing a high-accuracy estimation method for the software intensity function.

Among over hundreds of software reliability models (SRMs) [1–3], nonhomogeneous Poisson process (NHPP)-based SRMs have gained much popularity in actual software testing phases. In many cases, the NHPP-based SRM is formulated as a parametric model, where the mean value function or its difference in discrete time or derivative in continuous time, called "*software intensity function*," can be considered as a unique parameter to govern the probabilistic property. One class of parametric NHPP-based SRMs is concerned with modeling the number of software faults detected in testing phases, initiated by Goel and Okumoto [4]. Afterwards, many parametric NHPP-based SRMs were proposed in the literatures [5–8] from various points of view. However, it is well known in the software reliability engineering community that there does not exist a uniquely best parametric NHPP-based SRM which can fit every type of software-fault count data. This fact implies that nonparametric methods without assuming parametric form should be used to describe the software debugging phenomenon which is different in each testing phase.

Apart from the traditional Bayesian framework, some frequentist approaches based on non-parametric statistics were introduced to estimate the quantitative software reliability. Sofer and Miller [9] used an elementary piecewise linear estimator of the NHPP intensity function from the software-fault detection time data and proposed a smoothing technique by means of quadratic programming. Gandy and Jensen [10] applied the kernel-based estimator to estimate the NHPP intensity function in a non-parametric way. Barghout et al. [11] also proposed a kernel-based non-parametric estimation for the order statistics-based SRMs, where the likelihood cross-validation and the prequential likelihood approaches were used to estimate the bandwidth. Wang et al. [12] applied the similar kernel method to the NHPP-based SRMs, where they focused on the local likelihood method with a locally weighted log-likelihood function. By combining the non-parametric estimation with the Bayesian framework, El-Aroui and Soler [13] and Wilson and Samaniego [14] developed non-parametric Bayesian estimation methods. It should be noted that, generally, these non-parametric estimation methods require high computational cost, which, in some cases, may be almost similar to or greater than an effort on model selection in the parametric SRMs.

Another class of non-parametric estimation methods for NHPP-based SRMs is the wavelet analysis-based approach, initiated by Xiao and Dohi [15]. They proposed the wavelet shrinkage estimation (WSE), which does not require solving any optimization problem, so that the implementation of estimation algorithms is rather easy than the other non-parametric methods. They compared their method with the conventional maximum likelihood estimation (MLE) and the least squares estimation (LSE) through goodness-of-fit test. It has been shown that WSE could provide higher goodness-of-fit performance than MLE and LSE in many cases, in spite of its non-parametric nature, through numerical experiments with real software-fault count data.

The fundamental idea of WSE is to remove the noise included in the observed software-fault count data to get a noise-free estimate of the software intensity function. It is performed through the following three-step procedure. First, the noise variance is stabilized by applying the data transformation to the data. This produces a time-series data in which the noise can be treated as Gaussian white noise. Second, the noise is removed using "*Haar-wavelet*" based denoising algorithm. Third, an inverse data transformation is applied to the denoised time-series data, obtaining the estimate of the software intensity function. Among many variance-stabilizing data transformations, the Anscombe transform [16] and the Fisz transform [17] were employed in the previous work [15]. The other well-known square root data transformations are Bartlett transform [18] and Freeman transform [19]. Both Anscombe transform and Freeman transform are actually natural extensions of the Bartlett transform.

This paper focuses on the first step of WSE and aims at identifying and emphasizing the influence that the data transformation exerts on the accuracy of WSE. The remaining part of this paper is planned as follows. In Section 2, we give a preliminary on NHPP-based software reliability modeling. Section 3 is devoted to introduce data transformations. Section 4 describes the WSE for NHPP-based SRMs in details. In Section 5, we carry out the real project data analysis and illustrate numerical examples to examine the effectiveness of the proposed methods. Finally the paper is concluded with future researches in Section 6.

2. NHPP-Based Software Reliability Modeling

Suppose that the number of software faults detected through a system test is observed at discrete time $i = 0, 1, 2, \ldots$. Let Y_i and $N_i = \sum_{k=0}^{i} Y_k$ denote the number of software faults detected at testing date i and its cumulative value, where $Y_0 = N_0 = 0$ is assumed without any loss of generality. The stochastic process $\{N_i : i = 0, 1, 2, \ldots\}$ is said to be a discrete non-homogeneous Poisson process (D-NHPP) if the probability mass function at time i is given by

$$\Pr\{N_i = m\} = \frac{\{\Lambda_i\}^m}{m!} \exp\{-\Lambda_i\}, \quad m = 0, 1, 2, \ldots, \tag{1}$$

where $\Lambda_i = E[N_i]$ is called the mean value function of a D-NHPP and means the expected cumulative number of software faults detected by testing date i. The function $\lambda_i = \Lambda_i - \Lambda_{i-1}$ $(i \geq 1)$ is called the discrete intensity function and implies the expected number of faults detected at testing date i, say $\lambda_i = E[Y_i]$.

MLE is one of the most commonly used parametric estimation method. Let θ denote the vector of parameters in the mean value function $\Lambda_i = \Lambda_i(\theta)$, and $x_i(y_i)$ denote the realization of $N_i(Y_i)$. When n software faults are detected, the log-likelihood function of a D-NHPP is given by

$$\begin{aligned} \mathcal{LLF}(\theta) = &\sum_{i=1}^{n} (x_i - x_{i-1}) \ln[\Lambda_i(\theta) - \Lambda_{i-1}(\theta)] - \Lambda_n(\theta) \\ &- \sum_{i=1}^{n} \ln[(x_i - x_{i-1})!], \end{aligned} \tag{2}$$

where $x_i = \sum_{k=0}^{i} y_k$ and $y_0 = x_0 = 0$. Then, the maximum likelihood estimate of θ, say $\widehat{\theta}$, is given by the solution of $\operatorname{argmax}_{\boldsymbol{\theta}} \mathcal{LLF}(\theta)$. Therefore, the estimate of the software intensity function $\lambda_i = \lambda_i(\theta)$ $(i = 0, 1, 2, \ldots)$ can be obtained by $\widehat{\lambda}_i = \Lambda_i(\widehat{\theta}) - \Lambda_{i-1}(\widehat{\theta})$ $(i \geq 1)$ with $\lambda_0 = 0$. In the following sections, we consider the problem of estimating the software intensity function from the noise-involved observation y_i, in a non-parametric way.

3. Variance Stabilizing Data Transformation

It is very familiar to make use of data transformations (DTs) to stabilize the variance of Poisson data. By using DT, the software-fault count data which follow the D-NHPP are approximately transformed to the Gaussian data. The most fundamental data-transform tool in statistics is the *Bartlett transform* (BT) [18]. Let η_i denote the Poisson white noise, that is,

$$Y_i = \lambda_i + \eta_i, \quad i = 1, 2 \ldots, n. \tag{3}$$

Table 1: Representative data transformaions.

	Data transformation	Inverse data transformation	Distribution of random variable after DT
BT1 [18]	$B_i = 2\sqrt{Y_i}$	$Y_i = 1/4 \times \{(B_i)^2\}$	$N(2\sqrt{\lambda_i - 1/4}, 1)$
BT2 [18]	$B_i = 2\sqrt{Y_i + 1/2}$	$Y_i = 1/4 \times \{(B_i)^2 - 2\}$	$N(2\sqrt{\lambda_i + 1/4}, 1)$
AT [16]	$S_i = 2\sqrt{Y_i + 3/8}$	$Y_i = 1/4 \times \{(S_i)^2 - 3/2\}$	$N(2\sqrt{\lambda_i + 1/8}, 1)$
FT [19]	$F_i = \sqrt{Y_i + 1} + \sqrt{Y_i}$	$Y_i = 1/4 \times \{(F_i)^2 + (F_i)^{-2} - 2\}$	$N(2\sqrt{\lambda_i}, 1)$

Table 2: Representative discrete NHPP-based SRMs.

SRM	Mean value function Λ_i
Geometric (GE)	$\omega\{1 - (1-p)^i\}$
Negative Binomial (NB)	$\omega\{\sum_{k=1}^{i}((r+k-1)!/(r-1)!k!)p^r(1-p)^{k-1}\}$
Discrete Weibull (DW)	$\omega\{1 - p^{i^r}\}$

Taking the BT, the random variables:

$$B_i = 2\sqrt{Y_i}, \quad i = 1, 2, \ldots, n \tag{4}$$

can be approximately regarded as Gaussian random variables with the normal distribution $N(2\sqrt{\lambda_i - 1/4}, 1)$, so that the realizations:

$$b_i = 2\sqrt{y_i}, \quad i = 1, 2, \ldots, n \tag{5}$$

can be considered as samples from $N(2\sqrt{\lambda_i - 1/4}, 1)$. That is, the transformed realizations b_i $(i = 1, 2, \ldots, n)$ by the BT are the ones from the normally distributed random variables:

$$B_i = \lambda_i' + \nu_i, \quad i = 1, 2, \ldots, n, \tag{6}$$

where $\lambda_i' = 2\sqrt{\lambda_i - 1/4}$ is the transformed software intensity function, and ν_i is the Gaussian white noise with unit variance.

Bartlett [18] also showed that

$$b_i = 2\sqrt{y_i + \frac{1}{2}}, \quad i = 1, 2, \ldots, n \tag{7}$$

is a better transformation since it provides a constant variance more closely to 1, even when the mean of Y_i is not large.

The Anscombe transform (AT) [16] is a natural extension of BT and is employed in our previous work [15]. AT is of the following form:

$$s_i = 2\sqrt{y_i + \frac{3}{8}}, \quad i = 1, 2, \ldots, n, \tag{8}$$

where s_i can be considered as observations of Gaussian random variable $S_i = 2\sqrt{Y_i + 3/8}$ with the normal distribution $N(2\sqrt{\lambda_i + 1/8}, 1)$. Freeman and Tukey [19] proposed the following square-root transform (we call it FT), which is also an extension of BT:

$$f_i = \sqrt{y_i + 1} + \sqrt{y_i}, \quad i = 1, 2, \ldots, n. \tag{9}$$

They showed that the variance of Gaussian random variable $F_i = 2\sqrt{Y_i + 1} + \sqrt{Y_i}$ is the nearest to 1 among BT, AT, and FT if the mean of Y_i is small. Recently, these variance stabilization techniques were used to LSE of the mean value function for the NHPP-based SRMs [20]. Table 1 summaries the DTs mentioned above.

As mentioned in Section 1, the first step of WSE is to apply the normalizing and variance-stabilizing DTs to the observed software-fault count data. In this paper, we employ BT, AT, and FT in the first and the third steps of WSE. Then, the target of denoising in the second step of WSE is the transformed data b_i, s_i or f_i $(i = 1, 2, \ldots, n)$. Letting b_i', s_i', and f_i' denote the denoised b_i, s_i, and f_i, respectively, the estimate of the original software intensity function λ_i can be obtained by taking the inverse DT of b_i', s_i' and f_i', as given in Table 1.

4. Wavelet Shrinkage Estimation for NHPP-Based SRM

4.1. Haar-Wavelet-Based Denoising Procedure. The Haar-wavelet-based shrinkage technique can be used as a denoising algorithm for the second step of WSE. In general, the noise removal is performed through the following three steps: (i) expanding the transformed time-series data to obtain the empirical wavelet coefficients, (ii) removing the noise included in the empirical wavelet coefficients using thresholding method, and (iii) making use of the denoised coefficients to calculate the estimate of the transformed software intensity function.

4.1.1. Haar Wavelet Transform. The Haar scaling function and the Haar wavelet function are defined as

$$\phi(i) = \begin{cases} 1 & (0 \le i < 1) \\ 0 & (\text{otherwise}), \end{cases}$$

$$\psi(i) = \begin{cases} 1 & \left(0 \le i < \frac{1}{2}\right) \\ -1 & \left(\frac{1}{2} \le i < 1\right) \\ 0 & (\text{otherwise}), \end{cases} \tag{10}$$

Table 3: Goodness-of-fit test results for different data transformations. (Threshold level: universal threshold.)

DS1	MSE_1	MSE_2	LL
HBT1(h, ut)	4.180	0.327	−153.759
HBT2(h, ut)	2.401	0.316	−144.850
HAT(h, ut)	2.573	0.317	−145.390
HFT(h, ut)	9.842	0.403	−383.785
HBT1(s, ut)	4.180	0.327	−153.759
HBT2(s, ut)	2.401	0.316	−144.850
HAT(s, ut)	2.573	0.317	−145.390
HFT(s, ut)	3.231	0.320	−148.023
DS2	MSE_1	MSE_2	LL
HBT1(h, ut)	12.449	1.141	−197.109
HBT2(h, ut)	11.600	1.138	−196.904
HAT(h, ut)	11.725	1.138	−196.953
HFT(h, ut)	11.847	1.138	−196.806
HBT1(s, ut)	12.450	1.141	−197.109
HBT2(s, ut)	11.600	1.138	−196.904
HAT(s, ut)	11.725	1.138	−196.953
HFT(s, ut)	11.847	1.138	−196.806
DS3	MSE_1	MSE_2	LL
HBT1(h, ut)	10.159	0.719	−143.181
HBT2(h, ut)	9.055	0.791	−192.903
HAT(h, ut)	9.286	0.793	−193.178
HFT(h, ut)	9.653	0.796	−167.025
HBT1(s, ut)	12.954	1.037	−212.321
HBT2(s, ut)	10.528	1.035	−213.978
HAT(s, ut)	10.813	1.035	−213.496
HFT(s, ut)	11.496	1.037	−213.419
DS4	MSE_1	MSE_2	LL
HBT1(h, ut)	2.065	0.304	−171.605
HBT2(h, ut)	1.781	0.304	−171.491
HAT(h, ut)	1.822	0.304	−171.504
HFT(h, ut)	1.864	0.304	−171.516
HBT1(s, ut)	2.065	0.304	−171.605
HBT2(s, ut)	1.781	0.304	−171.491
HAT(s, ut)	1.822	0.304	−171.504
HFT(s, ut)	1.864	0.304	−171.516
DS5	MSE_1	MSE_2	LL
HBT1(h, ut)	14.013	0.586	−304.198
HBT2(h, ut)	12.781	0.581	−301.073
HAT(h, ut)	12.941	0.581	−301.447
HFT(h, ut)	13.200	0.582	−301.841
HBT1(s, ut)	14.054	0.601	−312.551
HBT2(s, ut)	12.804	0.596	−309.709
HAT(s, ut)	12.968	0.597	−310.005
HFT(s, ut)	13.227	0.598	−310.480
DS6	MSE_1	MSE_2	LL
HBT1(h, ut)	12.707	0.521	−378.932
HBT2(h, ut)	11.899	0.525	−380.497
HAT(h, ut)	12.053	0.526	−380.987

Table 3: Continued.

DS6	MSE_1	MSE_2	LL
HFT(h, ut)	12.186	0.526	−381.071
HBT1(s, ut)	13.546	0.585	−423.335
HBT2(s, ut)	12.505	0.584	−421.480
HAT(s, ut)	12.670	0.584	−421.729
HFT(s, ut)	12.810	0.584	−421.866

respectively. By introducing the *scaling parameter* j and *shifting parameter* k, the Haar father wavelet and the Haar mother wavelet are defined by

$$\begin{aligned}\phi_{j,k}(i) &= 2^{-j/2}\phi\left(2^{-j}i-k\right),\\ \psi_{j,k}(i) &= 2^{-j/2}\psi\left(2^{-j}i-k\right),\end{aligned} \tag{11}$$

respectively. Then the target function, transformed software intensity function λ_i' $(i = 1, 2, \ldots, n)$, can be expressed in the following equation:

$$\lambda_i' = \sum_{k=0}^{2^{j_0}-1}\alpha_{j_0,k}\phi_{j_0,k}(i) + \sum_{j=j_0}^{\infty}\sum_{k=0}^{2^j-1}\beta_{j,k}\psi_{j,k}(i), \tag{12}$$

where

$$\alpha_{j_0,k} = \sum_{i=1}^{n}\lambda_i'\phi_{j_0,k}(i), \tag{13}$$

$$\beta_{j,k} = \sum_{i=1}^{n}\lambda_i'\psi_{j,k}(i) \tag{14}$$

are called the scaling coefficients and the wavelet coefficients, respectively, for any primary resolution level $j_0(\geq 0)$. Due to the implementability, it is reasonable to set an upper limit instead of ∞ for the resolution level j. In other words, the highest resolution level must be finite in practice. We use J to denote the highest resolution level. That is, the range of j in the second term of (12) is $j \in [j_0, J]$. The mapping from function λ_i' to coefficients $(\alpha_{j_0,k}, \beta_{j,k})$ is called the Haar wavelet transform (HWT).

Since b_i, s_i or f_i $(i = 1, 2, \ldots, n)$ can be considered as the observation of λ_i', the empirical scaling coefficients $c_{j_0,k}$ and the empirical wavelet coefficients $d_{j,k}$ of λ_i' can be calculated by (13) and (14) with λ_i' replaced by b_i, s_i or f_i. The noises involved in the empirical wavelet coefficients $d_{j,k}$ can be removed by the thresholding method that we will introduce later. Finally, the estimate of λ_i' can be obtained by taking the inverse HWT with denoised empirical coefficients.

4.1.2. Thresholding. In denoising the empirical wavelet coefficients, the common choices of thresholding method are the hard thresholding:

$$\delta_\tau(u) = u1_{|u|>\tau}, \tag{15}$$

and the soft thresholding:

$$\delta_\tau(u) = \operatorname{sgn}(u)(|u| - \tau)_+, \tag{16}$$

for a fixed threshold level $\tau(> 0)$, where 1_A is the indicator function of an event A, $\operatorname{sgn}(u)$ is the sign function of u and $(u)_+ = \max(0, u)$. There are many methods to determine the threshold level τ. In this paper, we use the universal threshold [21] and the "leave-out-half" cross-validation threshold [22]:

$$\begin{aligned}\tau &= \sqrt{2\log n},\\ \tau &= \left(1-\frac{\log 2}{\log n}\right)^{-1/2}\tau\left(\frac{n}{2}\right),\end{aligned} \tag{17}$$

where n is the length of the observation s_i. Hard thresholding is a "keep" or "kill" rule, while soft thresholding is a "shrink" or "kill" rule. Both thresholding methods and both threshold levels will be employed to work on the empirical wavelet coefficients $d_{j,k}$ for denoising.

4.2. Wavelet Shrinkage Estimation for NHPP-Based SRM. Since the software-fault count data is Poisson data, the preprocessing is necessary before making use of the Haar-wavelet-based denoising procedure. Xiao and Dohi [15] combined data transformation and the standard denoising procedure to propose the wavelet shrinkage estimation (WSE) for the D-NHPP-based SRMs. They call the HWT combined with AT, the Haar-Anscombe transform (HAT). Similarly, we call the HWT combined with BT and FT, the Haar-Bartlett transform (HBT) and the Haar-Freeman transform (HFT), respectively. In the numerical study, we investigate the goodness-of-fit performance of HBT- and HFT-based WSEs and compare them with the HAT-based WSE [15].

5. Numerical Study

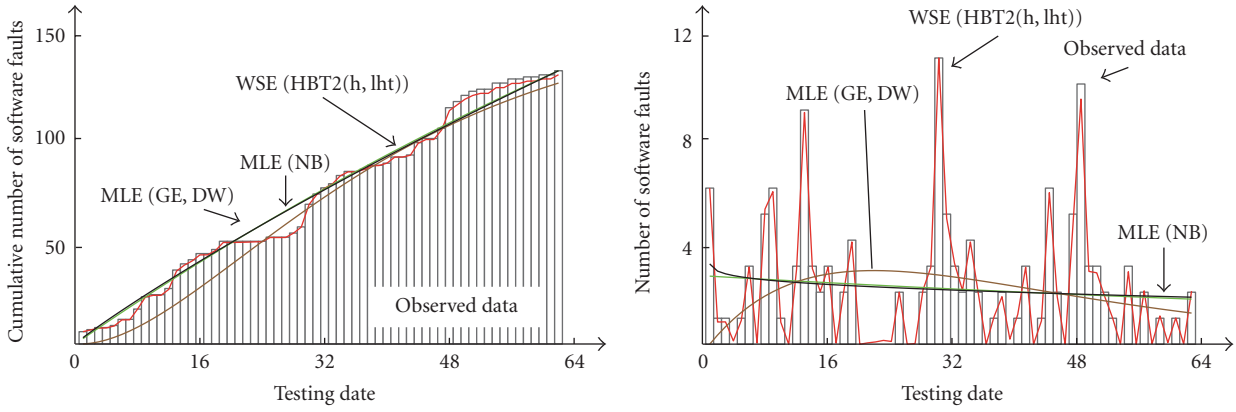
5.1. Data Sets and Measures. We use six real project data sets cited from reference [1], where they are named as J1, J3, DATA14, J5, SS1, and DATA8. These data sets are software-fault count data (group data). We rename them for convenience as DS1∼DS6 in this paper. Let (n, x_n) denote the pair of the final testing date and the total number of detected fault. Then these data sets are presented by (62, 133), (41, 351), (46, 266), (73, 367), (81, 461), and (111, 481),

Table 4: Goodness-of-fit test results for different data transformations. (Threshold level: "leave-out-half" cross-validation threshold.)

DS1	MSE_1	MSE_2	LL
HBT1(h, lht)	0.112	0.015	−59.704
HBT2(h, lht)	0.131	0.017	−60.634
HAT(h, lht)	0.114	0.014	−60.324
HFT(h, lht)	8.079	0.343	−216.320
HBT1(s, lht)	0.159	0.010	−59.581
HBT2(s, lht)	0.109	0.010	−60.195
HAT(s, lht)	0.115	0.010	−60.118
HFT(s, lht)	0.151	0.010	−59.609
DS2	MSE_1	MSE_2	LL
HBT1(h, lht)	0.477	0.102	−70.438
HBT2(h, lht)	0.225	0.072	−70.227
HAT(h, lht)	0.263	0.077	−70.663
HFT(h, lht)	0.552	0.125	−71.041
HBT1(s, lht)	0.342	0.027	−69.407
HBT2(s, lht)	0.363	0.030	−69.781
HAT(s, lht)	0.364	0.030	−69.731
HFT(s, lht)	0.351	0.028	−69.446
DS3	MSE_1	MSE_2	LL
HBT1(h, lht)	0.208	0.032	−55.829
HBT2(h, lht)	0.275	0.041	−56.767
HAT(h, lht)	0.902	0.074	−57.614
HFT(h, lht)	0.323	0.048	−56.411
HBT1(s, lht)	0.573	0.040	−55.847
HBT2(s, lht)	0.486	0.037	−56.649
HAT(s, lht)	0.496	0.037	−56.551
HFT(s, lht)	0.537	0.037	−55.922
DS4	MSE_1	MSE_2	LL
HBT1(h, lht)	0.288	0.015	−121.008
HBT2(h, lht)	0.132	0.007	−120.942
HAT(h, lht)	0.232	0.013	−121.031
HFT(h, lht)	0.281	0.014	−121.004
HBT1(s, lht)	0.097	0.005	−120.893
HBT2(s, lht)	0.046	0.003	−120.906
HAT(s, lht)	0.050	0.003	−120.907
HFT(s, lht)	0.064	0.004	−120.888
DS5	MSE_1	MSE_2	LL
HBT1(h, lht)	0.055	0.012	−120.869
HBT2(h, lht)	0.221	0.016	−121.205
HAT(h, lht)	0.228	0.017	−121.195
HFT(h, lht)	0.219	0.017	−120.890
HBT1(s, lht)	0.174	0.007	−120.805
HBT2(s, lht)	0.149	0.006	−120.913
HAT(s, lht)	0.151	0.006	−120.898
HFT(s, lht)	0.158	0.006	−120.806
DS6	MSE_1	MSE_2	LL
HBT1(h, lht)	0.778	0.025	−166.906
HBT2(h, lht)	0.558	0.017	−166.816
HAT(h, lht)	0.548	0.017	−166.767
HFT(h, lht)	0.540	0.017	−166.673

Table 4: Continued.

DS6	MSE_1	MSE_2	LL
HBT1(s, lht)	0.288	0.009	−166.432
HBT2(s, lht)	0.282	0.009	−166.532
HAT(s, lht)	0.289	0.010	−166.540
HFT(s, lht)	0.294	0.009	−166.440



(a) HBT2-based WSE(h, lht) versus MLE

(b) HBT2-based WSE(h, lht) versus MLE

Figure 1: Behavior of estimates with MLE and WSE using hard thresholding (DS1).

respectively. We employ the MSE (mean squares error) as the goodness-of-fit measures, where

$$\mathrm{MSE}_1 = \frac{\sqrt{\sum_{i=1}^{n}(\Lambda_i - x_i)^2}}{n}, \qquad \mathrm{MSE}_2 = \frac{\sqrt{\sum_{i=1}^{n}(\lambda_i - y_i)^2}}{n}. \tag{18}$$

Additionally, we calculate LL (Log Likelihood), which is defined as

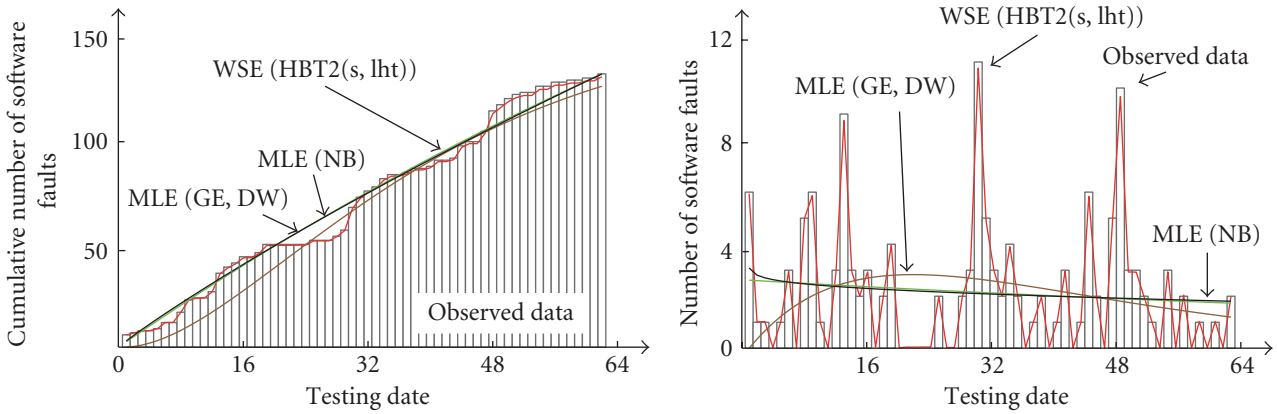
$$\mathrm{LL} = \sum_{i=1}^{n}(x_i - x_{i-1})\ln\left[\hat{\Lambda}_i - \hat{\Lambda}_{i-1}\right] - \hat{\Lambda}_n(\theta) - \sum_{i=1}^{n}\ln[(x_i - x_{i-1})!], \tag{19}$$

where $\hat{\Lambda}_i = \sum_{k=1}^{i}\hat{\lambda}_k$ $(i = 1, 2, \ldots, n)$, and $\hat{\lambda}_i$ is the WSE estimate of the software intensity function λ_i.

5.2. Goodness-of-Fit Test. A total of 16 wavelet-based estimation methods are examined in this paper since the WSE is applied with four thresholding techniques: hard thresholding (h) versus soft thresholding (s); universal threshold (ut) versus "leave-out-half," cross-validation threshold (lht). Let HBT$(\cdot,\cdot)$, HAT$(\cdot,\cdot)$ and HFT$(\cdot,\cdot)$ denote the WSEs based on Haar-Bartlett Transform, Haar-Anscombe Transform and Haar-Freeman Transform, respectively. HBT1$(\cdot,\cdot)$, and HBT2$(\cdot,\cdot)$ correspond to the transforms in (5) and (7), respectively. Additionally, the result of HAT-based WSE was introduced in [15], but they only showed the results of HAT-based WSE with hard thresholding. Here, we present comprehensively all the results of them for a further discussion.

We present the goodness-of-fit results based on different threshold levels in Tables 3 and 4, respectively. HBT2 provides smaller MSE and larger MLL than the others when "ut" is used, regardless of the thresholding method employed. It is worth mentioning that "ut" provides the same estimates with hard or soft thresholding in three data sets (DS1, DS2, and DS4). This is due to the relatively large value of "ut", since when threshold is set to be 0, the output of thresholding $\delta_\tau(d_{j,k})$ is the wavelet coefficient $d_{j,k}$ itself. In our numerical experiments, "ut" is relatively large in these 3 software-fault count data sets, which result in $\delta_{\mathrm{ut}}(d_{j,k}) = 0$ whichever hard or soft thresholding is applied. HBT2 also looks better than the others when software thresholding is applied with "lht." However, when "lht" is combined with hard thresholding, HBT1 (HAT; HFT) possesses the best results in DS3 and DS5 (DS1; DS6), respectively. Since "lht" is considered as a much more proper threshold level than "ut" in analyzing of software-fault count data, we suggest that HBT2 should be selected as an appropriate DT for the WSE.

5.3. Comparison with MLE. Our concern in this section is the comparison of WSE with MLE. We estimate the software intensity function by using three parametric D-NHPP-based SRMs listed in Table 2, where the MLE is applied to estimate the model parameters for comparison. HBT2-based WSE is selected as a representative one among the 16 wavelet-based estimation methods. Figures 1 and 2 depict the estimation behavior of cumulative number of software faults and its increment per testing date (individual number of software



(a) HBT2-based WSE(s, lht) versus MLE

(b) HBT2-based WSE(s, lht) versus MLE

Figure 2: Behavior of estimates with MLE and WSE using soft thresholding (DS1).

faults detected per testing date) with DS1. The observed data is plotted in bar graph in both figures. Looking at (i) and (iii) in these figures, it is clear that the parametric D-NHPP-based SRMs with maximum likelihood estimator can fit the real data. However, since the parametric D-NHPP-based SRMs assume the software intensity function as smooth function, they can estimate only the average tendency of the individual number of software faults detected at each testing date, but cannot follow the microscopic fluctuated behavior in (ii) and (iv) of Figures 1 and 2. In other words, the estimation accuracy based on the cumulative number of faults is embedded in "cumulative effects" in (i) and (iii). The experimental results performed here give the potential applicability of the wavelet-based estimation methods with different thresholding schemes. Our methods employed here do not need the expensive computational cost comparing with the MLE (within less than one second to get an estimate). This is a powerful advantage in applying the D-NHPP-based SRMs, in addition to the fact that practitioners do not request much time and effort to implement the wavelet-based estimation algorithms.

6. Concluding Remarks

In this paper, we have applied the wavelet-based techniques to estimate the software intensity function. Four data transformations were employed to preprocess the software-fault count data. Throughout the numerical evaluation, we could conclude that the wavelet-based estimation methods with Bartlett transform $2\sqrt{Y_i}$ have much more potential applicability than the other data transformations to the software reliability assessment practice because practitioners are not requested to carry out troublesome procedures on model selection and to take care of computational efficiency such as judgment of convergence and selecting initial guess of parameters in the general purpose of optimization algorithms. Note that, the result obtained here does not mean that the other data transformations are not good because the performance evaluation was executed only through goodness-of-fit test. Although the prediction ability of the proposed methods is out of focus of this paper at the present, the predictive performance should be considered and compared in the future.

References

[1] M. R. Lyu, Ed., *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, NY, USA, 1996.

[2] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability, Measurement, Prediction, Application*, McGraw-Hill, New York, NY, USA, 1987.

[3] H. Pham, *Software Reliability*, Springer, Singapore, Singapore, 2000.

[4] A. L. Goel and K. Okumoto, "Time-dependent error-detection rate model for software reliability and other performance measures," *IEEE Transactions on Reliability*, vol. R-28, no. 3, pp. 206–211, 1979.

[5] A. L. Goel, "Software reliability models: assumptions, limitations and applicability," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1411–1423, 1985.

[6] X. Xiao and T. Dohi, "Estimating software reliability using extreme value distribution," in *Proceedings of the International Conference on Advances in Software Engineering and Its Applications (ASEA '11)*, vol. CCIS 257, pp. 399–406, Springer, 2011.

[7] X. Xiao, H. Okamura, and T. Dohi, "NHPP-based software reliability models using equilibrium distribution," *IEICE Transactions of the Fundamentals A*, vol. E95-A, no. 5, pp. 894–902, 2012.

[8] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Transactions on Reliability*, vol. R-32, no. 5, pp. 475–484, 1983.

[9] A. Sofer and D. R. Miller, "A non-parametric software reliability growth model," *IEEE Transactions on Reliability*, vol. R-40, no. 3, pp. 329–337, 1991.

[10] A. Gandy and U. Jensen, "A non-parametric approach to software reliability," *Applied Stochastic Models in Business and Industry*, vol. 20, no. 1, pp. 3–15, 2004.

[11] M. Barghout, B. Littlewood, and A. Abdel-Ghaly, "A non-parametric order statistics software reliability model," *Software*

Testing Verification and Reliability, vol. 8, no. 3, pp. 113–132, 1998.

[12] Z. Wang, J. Wang, and X. Liang, "Non-parametric estimation for NHPP software reliability models," *Journal of Applied Statistics*, vol. 34, no. 1, pp. 107–119, 2007.

[13] M. A. El-Aroui and J. L. Soler, "A bayes nonparametric framework for software-reliability analysis," *IEEE Transactions on Reliability*, vol. 45, no. 4, pp. 652–660, 1996.

[14] S. P. Wilson and F. J. Samaniego, "Nonparametric analysis of the order-statistic model in software reliability," *IEEE Transactions on Software Engineering*, vol. 33, no. 3, pp. 198–208, 2007.

[15] X. Xiao and T. Dohi, "Wavelet-based approach for estimating software reliability," in *Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE '09)*, pp. 11–20, IEEE CS Press, November 2009.

[16] F. J. Anscombe, "The transformation of Poisson, binomial and negative binomial data," *Biometrika*, vol. 35, no. 3-4, pp. 246–254, 1948.

[17] M. Fisz, "The limiting distribution of a function of two independent random variables and its statistical application," *Colloquium Mathematicum*, vol. 3, pp. 138–146, 1955.

[18] M. S. Bartlett, "The square root transformation in the analysis of variance," *Journal of the Royal Statistical Society*, vol. 3, no. 1, pp. 68–78, 1936.

[19] M. F. Freeman and J. W. Tukey, "Transformations related to the angular and the square root," *The Annals of Mathematical Statistics*, vol. 21, no. 4, pp. 607–611, 1950.

[20] H. Ishii, T. Dohi, and H. Okamura, "Software reliability prediction based on least squares estimation," *Quality Technology and Quantitative Management Journal*. In press.

[21] D. L. Donoho and J. M. Johnstone, "Ideal spatial adaptation by wavelet shrinkage," *Biometrika*, vol. 81, no. 3, pp. 425–455, 1994.

[22] G. P. Nason, "Wavelet shrinkage using cross-validation," *Journal of the Royal Statistical Society B*, vol. 58, no. 2, pp. 463–479, 1996.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 924923, 8 pages
doi:10.1155/2012/924923

Research Article

Can Faulty Modules Be Predicted by Warning Messages of Static Code Analyzer?

Osamu Mizuno and Michi Nakai

Kyoto Institute of Technology, Matsugasaki Goshokaido-cho, Sakyo-ku, Kyoto 606-8585, Japan

Correspondence should be addressed to Osamu Mizuno, o-mizuno@kit.ac.jp

Received 5 January 2012; Accepted 24 February 2012

Academic Editor: Chin-Yu Huang

We have proposed a detection method of fault-prone modules based on the spam filtering technique, "Fault-prone filtering." Fault-prone filtering is a method which uses the text classifier (spam filter) to classify source code modules in software. In this study, we propose an extension to use warning messages of a static code analyzer instead of raw source code. Since such warnings include useful information to detect faults, it is expected to improve the accuracy of fault-prone module prediction. From the result of experiment, it is found that warning messages of a static code analyzer are a good source of fault-prone filtering as the original source code. Moreover, it is discovered that it is more effective than the conventional method (that is, without static code analyzer) to raise the coverage rate of actual faulty modules.

1. Introduction

Recently, machine learning approaches have been widely used for fault-proneness detection [1]. We have introduced a text feature-based approach to detect fault-prone modules [2]. In this approach, we extract text features from the frequency information of words in source code modules. In other words, we construct a large metrics set representing the frequency of words in source code modules. Once the text features are obtained, the Bayesian classifier is constructed from text features. In the fault-prone module detection of new modules, we also extract text features from source code modules, and Bayesian model classifies modules into either fault-prone or nonfault-prone. Since less effort or cost needed to collect text feature metrics than other software metrics, it may be applied to software development projects easily.

On the other hand, since this approach accepts any input with text files, the accuracy of prediction could be improved by selecting appropriate input other than raw source code. We then try to find another input but source code. In this study, we use warning messages of a static code analyzer. Among many static code analyzers, we used PMD in this study. By replacing the input of fault-prone filtering from raw source code to warning messages of PMD, we can get the results of prediction by PMD and fault-prone filtering.

The rest of this paper is organized as follows. Section 2 describes the objective of this research. Section 3 shows a brief summary of the fault-prone filtering technique with PMD. In Section 4, the experiments conducted in this study are described. Section 5 discusses the result of the experiments. Finally, Section 6 concludes this study.

2. Objective

2.1. Fault-Prone Module Filtering. The basic idea of fault-prone filtering is inspired by the spam mail filtering. In the spam e-mail filtering, a spam filter first trains both spam and ham e-mail messages from the training data set. Then, an incoming e-mail is classified into either ham or spam by the spam filter.

This framework is based on the fact that spam e-mail usually includes particular patterns of words or sentences. From the viewpoint of source code, a similar situation usually occurs in faulty software modules. That is similar faults may occur in similar contexts. We thus guessed that similar to spam e-mail messages, faulty software modules

have similar patterns of words or sentences. To obtain such features, we adopted a spam filter in fault-prone module prediction.

In other words, we try to introduce a new metric as a fault-prone predictor. The metric is "frequency of particular words." In detail, we do not treat a single word, but use combinations of words for the prediction. Thus, the frequency of a certain length of words is the only metric used in our approach.

From a viewpoint of effort, conventional fault-prone detection techniques require relatively much effort for application because they have to measure various metrics. Of course, metrics are useful for understanding the property of source code quantitatively. However, measuring metrics usually needs extra effort and translating the values of metrics into meaningful result also needs additional effort. Thus, easy-to-use technique that does not require much effort will be useful in software development.

We then try to apply a spam filter to identification of fault-prone modules. We named this approach as "fault-prone filtering." That is, a learner first trains both faulty and nonfaulty modules. Then, a new module can be classified into fault-prone or notfault-prone using a classifier. In this study, we define a software module as a Java class file.

Essentially, the fault-prone filtering does the text classification on the source codes. Of course, the text classification can be applied to the text information other than the source codes. We guessed that there is the other input for the text classification to achieve higher prediction accuracy. We then started seeking such information.

2.2. Static Code Analysis. The static code analysis is a method of analyzing without actually running software and finding the problem and faults in a software. By analyzing a source code structurally, we can find potential faults, violation of coding conventions, and so on. The static code analysis thus can assure the safety of software, reliability, and quality. It also reduces the cost of maintenance. In recent years, the importance of static code analysis has been emerging since finding potential faults or security hole is required at an early stage of the development. There are many kinds of tools for the static code analysis available [3]. Among them, we used the PMD (the meaning of PMD is not determined. "We have been trying to find the meaning of the letters PMD—because frankly, we do not really know. We just think the letters sound good together" [4]), since it can be applicable to the source code directly.

The PMD is one of static code analysis tools [5]. It is an open-source software and written in Java, and it is used for analyzing programs written in Java. PMD can find the code pieces that may cause the potential faults such as an unused variable and an empty catch block by analyzing the source code of Java. To do so, PMD has a variety of rule sets. According to the rule sets to be used, a broad range of purposes from the inspection of coding conventions to find potential faults can be used.

2.3. Characteristics of the Warning Messages of the Static Code Analyzer. Warning messages of a static code analyzer include rich information about potential faults in source codes. Figure 1 shows an example of warning messages. Usually, the number of warning messages generated by the static code analyzer becomes large in proportion to the length of source code. Since most of the messages are not harmful or trivial, warning messages are often ignored. It can be considered that these warning messages are quality aspects of the source code. Thus, we consider that the warning messages have less noise for fault-prone module prediction.

As mentioned in Section 2.1, applying the text information to the text classifier is an easy task. We thus implement the fault-prone filtering technique to use the warning messages of the static code analyzer. We then conduct experiments to confirm the effects of the warning messages to the performance of the fault-prone filtering approach.

2.4. Research Questions. In this study, we aim at answering the following research questions:

> RQ1: "can fault-prone modules be predicted by applying a text filter to the warning messages of a static code analyzer?"
>
> RQ2: "if RQ1 is true, is the performance of the fault-prone filtering becomes better with the warning messages of a static code analyzer?"

RQ1 tries to find a possibility to apply the warning messages to the fault-prone filtering technique. RQ2 investigates the prediction performance.

3. Fault-Prone Filtering with PMD

3.1. Applying PMD to Source Code. We used 10 rule sets of PMD in a standard rule sets: Basic, Braces, Code Size, Coupling, Design, Naming, Optimizations, Strict Exception, Strings, and Unused Code. These rule sets are frequently used for investigation of the quality of software. We apply PMD with 10 rule sets to all source code modules and get warning messages of PMD.

3.2. Classification Techniques. In this study, we used CRM114 (the controllable regex mutilator) spam filtering software [6] for its versatility and accuracy. Since CRM114 is implemented as a language to classify text files for general purpose, applying source code modules is easy. Furthermore, the classification techniques implemented in CRM114 are based mainly on Markov random field model instead of the naive Bayesian classifier.

In this experiment, we used the orthogonal sparse bigrams Markov model built in CRM114.

> Orthogonal Sparse Bigrams Markov model (OSB)
>
> Basically, CRM114 uses sparse binary polynomial Hash Markov model (SBPH). It is an extension of the Bayesian classification, and it maps features in the input text into a markov random field [7]. In this model, tokens are constructed from combinations of n words (n-grams) in a text file. Tokens are then

The class "ISynchronizerTest" has a Cyclomatic Complexity of 8 (Highest = 32).
This class has too many methods, consider refactoring it.
Avoid excessively long variable names like NUMBER_OF_PARTNERS
The field name indicates a constant but its modifiers do not
Variables should start with a lowercase character
Variables that are not final should not contain underscores (except for underscores in standard prefix/suffix).
Document empty constructor
Parameter "name" is not assigned and could be declared final
Avoid variables with short names like b1
Avoid variables with short names like b2
Parameter "b1" is not assigned and could be declared final
Parameter "b2" is not assigned and could be declared final
Parameter "message" is not assigned and could be declared final
Avoid using for statements without curly braces
Local variable "body" could be declared final
Parameter "monitor" is not assigned and could be declared final
Parameter "resource" is not assigned and could be declared final
Avoid using for statements without curly braces
...

Figure 1: A part of warning messages by PMD from a source code module of Eclipse.

mapped into a Markov random field to calculate the probability.

OSB is a simplified version of SBPH. It considers tokens as combinations of exactly 2 words created in the SBPH model. This simplification decreases both memory consumption of learning and time of classification. Furthermore, it is reported that OSB usually achieves higher accuracy than a simple word tokenization [8].

3.3. Tokenization of Inputs. In order to perform fault-prone filtering approach, inputs of fault-prone filter must be tokenized. In this study, in order to use the warning messages of PMD as input of filtering, the messages need to be tokenized. Warning messages of PMD contains English text in natural language and a part of Java code. In order to separate them, we classified them into the following kind of strings:

(i) strings that consist of alphabets and numbers;

(ii) all kinds of brackets, semicolons, commas;

(iii) operators of Java and dot;

(iv) other strings (natural language message).

Furthermore, warning messages of PMD have file names and line numbers on the top of each line. In usual, they provide useful information for debug, but for learning and classification, they may mislead the learning of faulty modules. For example, once we learn a line number of a faulty module, the same line number of the other file is wrongly considered as faulty token.

3.4. Example of Filtering. Here, we explain briefly how these classifiers work. We will show how to tokenize and classify the faulty modules in our filtering approach.

```
1:  if  x
2:   if    ==
3:   if      1
4:   if        return
```

Figure 2: Example of tokens for OSB using the source code.

```
1:  underscores  in
2:  underscores    standard
3:  underscores      prefix/suffix)
```

Figure 3: Example of tokens for OSB using the warning messages.

3.4.1. Tokenization. In OSB, tokens are generated so that these tokens include exactly 2 words. For example, a sentence "if (x == 1) return;" is tokenized as shown in Figure 2 By definition, the number of tokens drastically decreases compared to SBPH. As for the warning messages, an example of a sentence "underscores in standard prefix/suffix)." is shown in Figure 3

3.4.2. Classification. Let T_{FP} and T_{NFP} be sets of tokens included in the fault-prone (FP) and the nonfault-prone (NFP) corpuses, respectively. The probability of fault-proneness is equivalent to the probability that a given set of tokens T_x is included in either T_{FP} or T_{NFP}. In OSB, the probability that a new module m_{new} is faulty, $P(T_{\mathrm{FP}}|T_{m_{\mathrm{new}}})$, with a given set of token $T_{m_{\mathrm{new}}}$ from a new source code module m_{new} is calculated by the following Bayesian formula:

$$\frac{P(T_{m_{\mathrm{new}}}T_{\mathrm{FP}})P(T_{\mathrm{FP}})}{P(T_{m_{\mathrm{new}}}T_{\mathrm{FP}})P(T_{\mathrm{FP}})+P(T_{m_{\mathrm{new}}}T_{\mathrm{NFP}})P(T_{\mathrm{NFP}})}. \quad (1)$$

Table 1: Target project: Eclipse BIRT plugin.

Name	Eclipse BIRT plugin
Language	Java
Revision control	cvs
Type of faults	Bugs
Status of faults	Resolved; Verified; closed
Resolution of faults	Fixed
Severity	Blocker; critical; major; normal
Priority of faults	All
Total number of faults	4708

Table 2: The number of modules in Eclipse BIRT.

	Number of modules (files)
Nonfaulty	42,503
Faulty	27,641
Total	70,144

Intuitively speaking, this probability denotes that the new code is classified into FP. According to $P(T_{\mathrm{FP}} \mid T_{m_{\mathrm{new}}})$ and predefined threshold t_{FP}, classification is performed.

4. Experiment

4.1. The Outline of the Experiment. In this experiment, warning messages of PMD are used for fault-prone filtering as an input instead of a source code module, and Fault-prone module is predicted. And it is the purpose to evaluate the predictive accuracy of the proposed method. Therefore, two experiments using raw source code modules and the warning messages by the PMD as inputs are conducted. We then compare these results to each other.

4.2. Target Project. In this experiment, we use the source code module of an open source project, Eclipse BIRT (business intelligence and reporting tools). The source code module is obtained from this project by the SZZ (Śliwerski et al.) algorithm [9]. The summary of Eclipse BIRT project is shown in Table 1. All software modules in this project are used for both learning and classification by the procedure called training only errors (TOE). The number of modules is shown in Table 2.

4.3. Procedure of Filtering (Training on Errors). Experiment 1 performs the original fault-prone module prediction using the raw source code and OSB classifier by the following procedures:

(1) apply the FP classifier to a newly created software module (say, method in Java, function in C, and so on), M_i, and obtain the probability to be fault-prone;

(2) by the predetermined threshold t_{FP} ($0 < t_{\mathrm{FP}} < 1$), classify the module M_i into FP or NFP;

(3) when the actual fault-proneness of M_i is revealed by fault report, investigate whether the predicted result for M_i was correct or not;

(4) if the predicted result was correct, go to step 1; otherwise, apply FP trainer to M_i to learn actual fault-proneness and go to step 1.

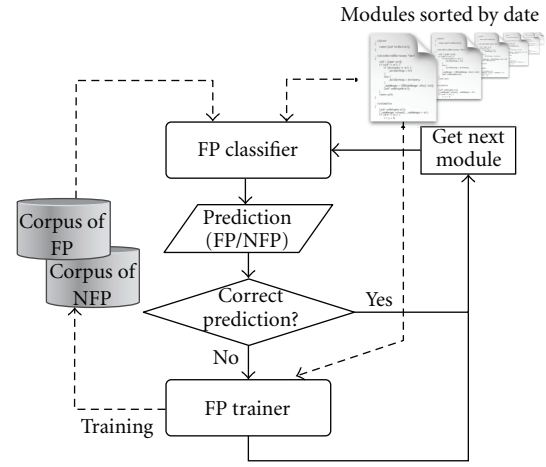


Figure 4: Outline of fault-prone filtering by training on errors.

This procedure is called "training on errors (TOE)" procedure because training process is invoked only when classification errors happen. The TOE procedure is quite similar to actual classification procedure in practice. For example, in actual e-mail filtering, e-mail messages are classified when they arrive. If some of them are misclassified, actual results (spam or nonspam) should be trained.

Figure 4 shows an outline of this approach. At this point, we consider that the fault-prone filtering can be applied to the sets of software modules which are developed in the same (or similar) project.

Experiment 2 is an extension of Experiment 1 by appending additional steps as the first step as follows:

(1) obtain warning messages W_i of PMD by applying PMD to a newly created software module M_i;

(2) apply the FP classifier to the warning messages, W_i, and obtain the probability to be fault-prone;

(3) by the predetermined threshold t_{FP} ($0 < t_{\mathrm{FP}} < 1$), classify the warning messages W_i into FP or NFP;

(4) when the actual fault-proneness of M_i is revealed by fault report, investigate whether the predicted result for W_i was correct or not;

(5) if the predicted result was correct, go to step (1); otherwise, apply FP trainer to W_i to learn actual fault-proneness and go to step (1).

Table 3: Classification result matrix.

		Prediction	
		Nonfault-prone	Fault-prone
Actual	Nonfaulty	True negative (tn)	False positive (fp)
	Faulty	False negative (fn)	True positive (tp)

4.4. Procedure of TOE Experiment. In the experiment, we have to simulate actual TOE procedure in the experimental environment. To do so, we first prepare a list of all modules found in Section 4.2. The list is sorted by the last modified date (d_i) of each module so that the first element of the list is the oldest module. We then start simulated experiment in the procedure shown in Algorithm 1. During the simulation, modules are classified by the order of date. If the predicted result s_i^p differs from actual status s_i^a, the training procedure is invoked.

4.5. Evaluation Measures. Table 3 shows a classification result matrix. True negative (tn) shows the number of modules that are classified as nonfault-prone, and are actually nonfaulty. False positive (fp) shows the number of modules that are classified as fault-prone, but are actually nonfaulty. On the contrary, false negative shows the number of modules that are classified as nonfault-prone, but are actually faulty. Finally, true positive shows the number of modules that are classified as fault-prone which are actually faulty.

In order to evaluate the results, we prepare two measures: recall and precision. Recall is the ratio of modules correctly classified as fault-prone to the number of entire faulty modules Recall is defined as $tp/(tp + fn)$. Precision is the ratio of modules correctly classified as fault-prone to the number of entire modules classified fault-prone. Precision is defined as $tp/(tp+fp)$. Accuracy is the ratio of correctly classified modules to the entire modules. Accuracy is defined as $(tp + tn)/(tn + tp + fp + fn)$. Since recall and precision are in the trade-off, F_1-measure is used to combine recall and precision [10]. F_1-measure is defined as $(2\times\text{recall}\times\text{precision})/(\text{recall}+\text{precision})$. In this definition, recall and precision are evenly weighed.

From the viewpoint of the quality assurance, it is recommended to achieve higher recall, since the coverage of actual faults is of importance. On the other hand, from the viewpoint of the project management, it is recommended to focus on the precision, since the cost of the software unit test is deeply related to the number of modules to be tested. In this study, we mainly focus on the recall from the viewpoint of the quality assurance.

4.6. Result of Experiments. Tables 4 and 5 show the result of experiment using the original approach without PMD and the approach with PMD, respectively. Table 6 summarizes the evaluation measures for these experiments.

From Table 6, we can see that the approach with PMD has almost the same capability to predict fault-prone modules as the approach without PMD. For example, F_1 for the approach without PMD is 0.779, and for the approach with PMD is 0.710. The result shows that the original approach without PMD is relatively better than the approach with PMD in precision, accuracy, and F_1 measures. The recall of the approach with PMD is better than the approach without PMD.

Table 4: Result of prediction in Experiment 1 (without PMD).

		Prediction	
		Nonfault-prone	Fault-prone
Actual	Nonfaulty	30,521	11,982
	Faulty	2,360	25,281
Total		32,821	37,263

Table 5: Result of prediction in Experiment 2 (with PMD).

		Prediction	
		Nonfault-prone	Fault-prone
Actual	Nonfaulty	22,982	19,521
	Faulty	1,674	25,967
Total		24,656	45,488

Table 6: Evaluation measures of the results of experiments.

	Precision	Recall	Accuracy	F_1
Experiment 1 (without PMD)	0.678	0.915	0.796	0.779
Experiment 2 (with PMD)	0.571	0.939	0.698	0.710

Figures 5 and 6 show the result of TOE history for the approaches without and with PMD, respectively. From this graph, we can see that evaluation measures first to decrease at the beginning of TOE procedure, then increase and become stable after learning and classification of 15,000 modules.

5. Discussions

At first, we discuss the advantage of the approach with PMD. From Table 6, we can see that the result of Experiment 2 has higher recall and lower precision than that of Experiment 1. Generally speaking, the recall is an important measure for the fault-prone module prediction because it implies how many actual faults can be detected by the prediction. Therefore, higher recall can be an advantage of the approach with PMD. However, the difference of the recalls between two experiments is rather small.

When we focus on the graphs of TOE histories shown in Figures 5 and 6, the difference between two experiments can be seen clearly. The transition of recall in Experiment 2 keeps higher than that of Experiment 1 from an early stage of the experiment. That is the recall of Experiment 2 reaches 0.90 at 10,000 modules learning. From this fact, we can say that the approach with PMD is efficient especially at an early stage of development. It can be considered as another advantage of the approach with PMD.

We discuss the reasons of the result that the approach with PMD does not shows a good evaluation measures at

```
t_FP : Threshold of probability to determine FP and NFP
s_i^p : Predicted fault status (FP or NFP) of M_i
for each M_i in list of modules sorted by d_i's
   prob = fpclassify(M_i)
   if prob > t_FP then s_i^p = FP
                  else s_i^p = NFP
   endif
   if s_i^a ≠ s_i^p then fptrain(M_i, s_i^a)
   endif
endfor
fpclassify(M) :
   if Experiment 1 then
      Generate a set of tokens T_M from source code M.
      Calculate probability P(T_FP | T_M)
         using corpuses T_FP and T_NFP.
      Return P(T_FP | T_M).
   if Experiment 2 then
      Generate a set of tokens T_W
         from warning messages W
         by applying PMD to the source code M.
      Calculate probability P(T_FP | T_W)
         using corpuses T_FP and T_NFP.
      Return P(T_FP | T_W)
fptrain(M, s^a) :
   if Experiment 1 then
      Generate a set of tokens T_M from M.
      Store tokens T_M to the corpus Ts^a.
   if Experiment 2 then
      Generate a set of tokens T_W from W
         by applying PMD to M.
      Store tokens T_W to the corpus Ts^a.
```

Algorithm 1: Procedure of TOE experiment.

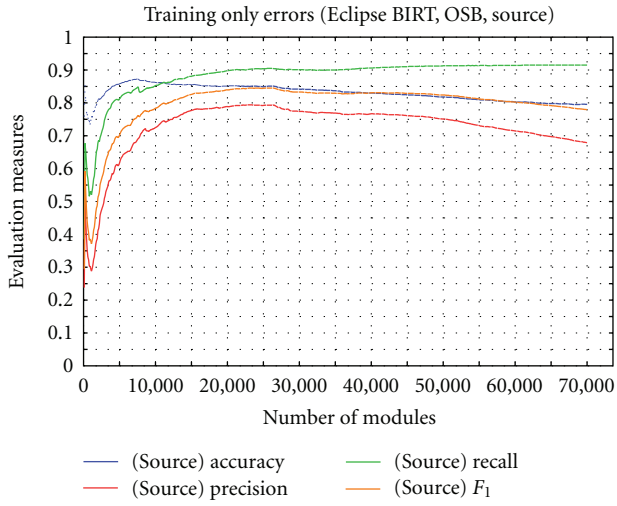


Figure 5: History of training on errors procedure in Experiment 1 (without PMD).

the end of the experiment. First, the selection of rule sets used in PMD may affect the result of experiment. Although we used 10 rule sets according to the past study, the selection of rule sets should be considered more carefully. For future research, we will investigate the effects of rule set selection to the accuracy of fault-prone filtering. Second, we need to apply this approach to more projects. We have conducted experiments on Eclipse BIRT.

Here, we investigate the details of our prediction. Table 7 shows a part of the probabilities for each token in the corpus for faulty modules. This table shows tokens with highest probabilities. The probability $P(T_x|T_{\mathrm{FP}})$ shows the conditional probability that a token T_x exists in the faulty corpus. Although these probabilities do not mean immediately that these tokens make a module fault-prone, we guess that the investigation of these probabilities helps improving accuracy.

We can see that specific identifier such as "copyInstance" and specific literals such as "994," "654," and "715" appear frequently. It can be guessed that these literals denote line number in a particular source code. These literals are effective to predict the fault-proneness of the specific source code modules, but it can be a noise for the most other modules. In order to improve the overall accuracy of the classifier, eliminating literals that describe a specific source code should be taken into account.

Finally, we answer the research questions here. We have the following research questions in Section 2.4.

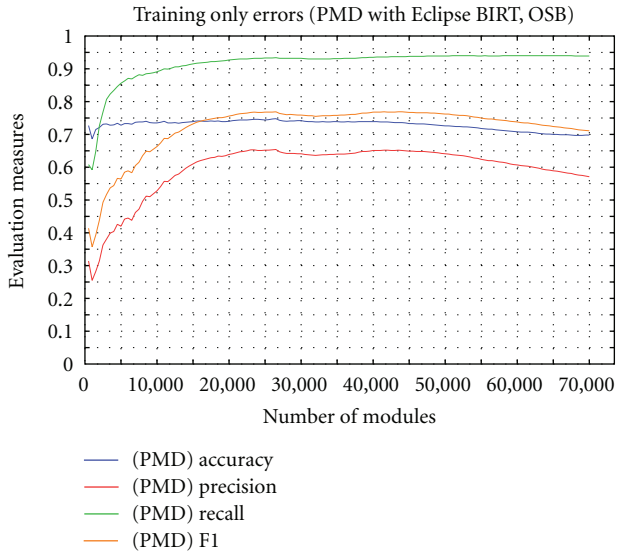


Figure 6: History of training on errors procedure in Experiment 2 (with PMD).

Table 7: Probabilities for each token in the corpus for faulty modules. The "□" denotes any words.

$P(T_x \mid T_{FP})$	T_x
0.56229	line 654
0.56229	715 Local
0.56229	"copyInstance" has
0.56229	654 Local
0.56229	instanceof The
0.56229	method copyInstance ()
0.56229	255 Parameter
0.56229	method "copyInstance"
0.56229	489 Avoid
0.56229	line 715
0.56229	copyInstance () has
0.56229	line 994
0.56229	line 474
0.56229	994 The
0.56215	715 □ variable
0.56215	blocks □ "pageNumber"
0.56215	on □ 994
0.56215	bb □ instantiating
0.56215	on □ 654
0.56215	The □ "copyInstance"
0.56215	654 □ variable
0.56215	994 □ String
0.56215	The □ copyInstance ()
0.56215	"copyInstance" □ a
0.56215	on □ 715
0.56215	on □ 474
0.56215	300 □ variables
0.56215	copyInstance () □ an

RQ1: "can fault-prone modules be predicted by applying a text filter to the warning messages of a static code analyzer?"

For this question, we can answer "yes" from the results in Table 5 and Table 6. It is obvious that the approach with PMD has prediction capability of the fault-prone modules at a certain degree.

RQ2: "if RQ1 is true, is the performance of the fault-prone filtering becomes better with the warning messages of a static code analyzer?"

For this question, we can say that the recall of the approach with PMD becomes higher and more stable during the development than the approach without PMD as shown in Table 6 and Figures 5 and 6. From the viewpoint of the quality assurance, it is a preferred property. We then conclude that the proposed approach has better performance to assure the software quality.

6. Conclusion

In this paper, we proposed an approach to predict fault-prone modules using warning messages of PMD and a text filtering technique. For the analysis, we stated two research questions: "can fault-prone modules be predicted by applying a text filter to the warning messages of static code analyzer?" and "is the performance of the fault-prone filtering becomes better with the warning messages of a static code analyzer?" We tried to answer this question by conducting experiments on the open source software. The results of experiments show that the answer to the first question is "yes." As for the second question, we can find that the recall becomes better than the original approach.

Future work includes investigating which parts of warning messages are really effective for fault-prone module prediction. Selection of rule sets of PMD is an interesting future research.

References

[1] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[2] O. Mizuno and T. Kikuno, "Training on errors experiment to detect fault-prone software modules by spam filter," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 405–414, 2007.

[3] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pp. 245–256, IEEE Computer Society, Washington, DC, USA, 2004.

[4] T. Copeland, PMD—What does "PMD" mean?, http://pmd.sourceforge.net/meaning.html .

[5] T. Copeland, *PMD Applied*, Centennial Books, Arexandria, Va, USA, 2005.

[6] W. S. Yerazunis, *CRM114—the Controllable Regex Mutilator*, http://crm114.sourceforge.net/.

[7] S. Chhabra, W. S. Yerazunis, and C. Siefkes, "Spam filtering using a markov random field model with variable weighting schemas," in *Proceedings of the 4th IEEE International Conference on Data Mining, (ICDM '04)*, pp. 347–350, Riverside, Calif, USA, November 2004.

[8] C. Siefkes, F. Assis, S. Chhabra, and W. S. Yerazunis, "Combining winnow and orthogonal sparse bigrams for incremental spam filtering," in *Proceedings of the Conference on Machine Learning/European Conference on Principles and Practice of Knowledge Discovery in Databases (ECML PKDD '04)*, 2004.

[9] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? (on Fridays)," in *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pp. 24–28, St. Louis, Mo, USA, May 2005.

[10] C. J. van Rijsbergen, *Information Retrieval*, Butterworth, Boston, Mass, USA, 2nd edition, 1979.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 949746, 28 pages
doi:10.1155/2012/949746

Research Article

Specifying Process Views for a Measurement, Evaluation, and Improvement Strategy

Pablo Becker,[1] Philip Lew,[2] and Luis Olsina[1]

[1] *GIDIS_Web, Engineering School, Universidad Nacional de La Pampa, General Pico, Argentina*
[2] *School of Software, Beihang University, Beijing, China*

Correspondence should be addressed to Luis Olsina, olsinal@ing.unlpam.edu.ar

Received 24 August 2011; Revised 12 November 2011; Accepted 8 December 2011

Academic Editor: Osamu Mizuno

Any organization that develops software strives to improve the quality of its products. To do this first requires an understanding of the quality of the current product version. Then, by iteratively making changes, the software can be improved with subsequent versions. But this must be done in a systematic and methodical way, and, for this purpose, we have developed a specific strategy called SIQinU (*Strategy for understanding and Improving Quality in Use*). SIQinU recognizes problems of quality in use through evaluation of a real system-in-use situation and proposes product improvements by understanding and making changes to the product's attributes. Then, reevaluating quality in use of the new version, improvement gains can be gauged along with the changes that led to those improvements. SIQinU aligns with GOCAME (*Goal-Oriented Context-Aware Measurement and Evaluation*), a multipurpose generic strategy previously developed for measurement and evaluation, which utilizes a conceptual framework (with ontological base), a process, and methods and tools. Since defining SIQinU relies on numerous phase and activity definitions, in this paper, we model different process views, for example, taking into account activities, interdependencies, artifacts, and roles, while illustrating them with excerpts from a real-case study.

1. Introduction

Even though software product launches now may consist of "continuous beta," users expect more and better functionality, combined with increased quality from the user's perception. Methodically improving the perceived quality, that is, its quality in use (QinU) particularly for web applications (WebApps), is not an easy job. WebApps—a kind of software applications—are no longer simple websites conveying information. Rather, they have become fully functional software applications often with complex business logic and sometimes critical to operating the business. Users, in addition, are becoming more demanding and diverse in their requirements. Consequently, WebApp quality and especially the quality in use, namely, the perceived quality by the end user has taken on increased significance as web and now cloud deployment have become mainstream delivery methods. Systematic means for evaluating QinU is important because it enables understanding the quality satisfaction level achieved by the application and provides useful information for recommendation and improvement processes in a consistent manner over time. Coincident with consistent and systematic evaluation of WebApp quality, the main goal is to ultimately improve its QinU.

This leads to our strategy with the objectives of understanding and improving the QinU—as nonfunctional requirements—of WebApps. QinU is currently redefined in the ISO 25010 standard [1], which was reused and enlarged by the 2Q2U (*internal/external Quality, Quality in use, actual Usability*, *and User experience*) quality framework—see [2] for an in-depth discussion. QinU from the actual usability standpoint (that embraces performance or "do" goals in contrast to hedonic or "be" goals [3]) is defined as the degree to which specified users can achieve specified goals with effectiveness in use, efficiency in use, learnability in use, and accessibility in use in a specified context of use [2].

Utilizing 2Q2U quality models, we developed SIQinU as an integrated means to evaluate and find possible problems in QinU which are then related to external quality (EQ) characteristics and attributes (by doing a mapping between

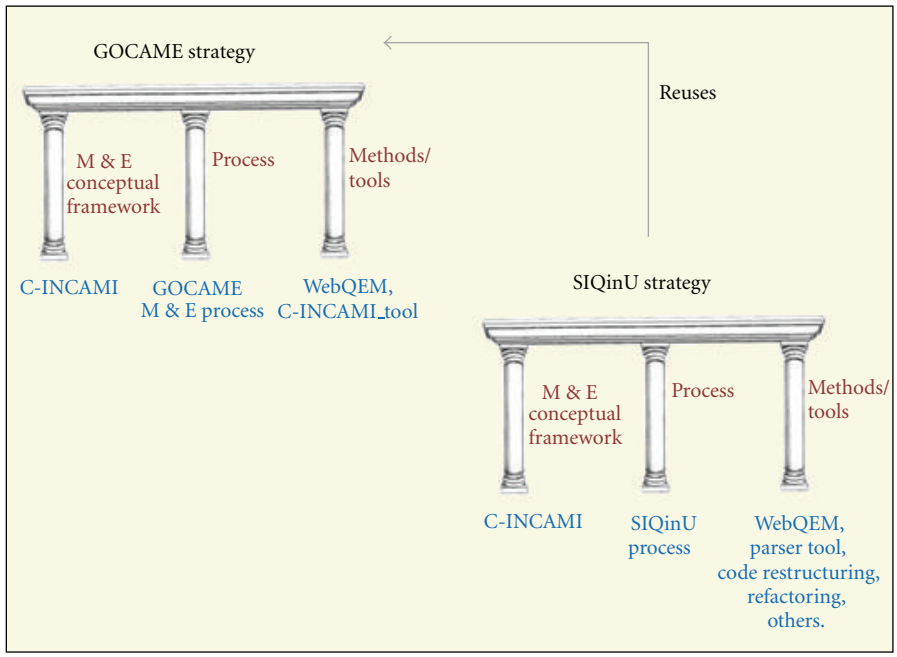


Figure 1: Allegory of the three GOCAME pillars, which are reused to a great extent by SIQinU.

QinU problems and EQ). This is followed by evaluating the application from the EQ standpoint and then making recommendations for improvements if necessary. The new version, based on recommended improvements, is reevaluated to gauge the improvement gain from both the EQ and QinU point of views. One aspect of SIQinU's uniqueness is that it collects user usage data from WebApps in a real context of use whereby code snippets are inserted (or using similar techniques) to gather data related to the task being executed by users at the subtask level enabling nonintrusive evaluations.

It is worth mentioning that SIQinU aligns with the GOCAME strategy [4]. GOCAME, a multipurpose goal-oriented strategy, was previously developed for supporting measurement and evaluation (M&E) programs and projects. Its rationale is based on three main pillars or principles, namely, (i) a conceptual framework utilizing an ontological base; (ii) a well-defined measurement and evaluation process; (iii) quality evaluation methods and tools instantiated from both the framework and process. This is allegorically depicted in Figure 1.

GOCAME's first principle is that designing and implementing a robust M&E program require a sound conceptual framework. Often times, organizations conduct start and stop measurement programs because they do not pay enough attention to the way nonfunctional requirements, contextual properties, metrics, and indicators should be designed, implemented, and analyzed. Any M&E effort requires an M&E framework built on a sound conceptual base, that is, on an ontological base, which explicitly and formally specifies the main agreed concepts, properties, relationships, and constraints for a given domain. To accomplish this, we utilize the C-INCAMI (*Contextual-Information Need, Concept model, Attribute, Metric, and Indicator*) framework and its components [4, 5] based on our metrics and indicators ontology.

GOCAME's second principle requires a well-established M&E process in order to guarantee repeatability in performing activities and consistency of results. A process prescribes a set of phases, activities, inputs and outputs, interdependencies, sequences and parallelisms, check points, and so forth. Frequently, process specifications state what to do but do not mention the particular methods and tools to perform specific activity descriptions. Thus, to provide repeatability and replicability in performing activities, a process model for GOCAME was proposed in [6], which is also compliant with both the C-INCAMI conceptual base and components. Finally, methods and tools—the third pillar in the GOCAME strategy—can be instantiated from both the conceptual framework and process, for example, the WebQEM (*Web Quality Evaluation*) methodology [7] and its tool called C-INCAMI_tool [4].

SIQinU utilizes the above three GOCAME principles while also reusing the C-INCAMI conceptual base and process. However, since SIQinU is a specific-purpose goal-oriented strategy, it has specific processes, methods, and procedures that are not specified in GOCAME. Since the process aspect is critical in specifying SIQinU, given of its numerous interrelated phases and activities, this work defines its process model in detail through illustration with excerpts of a real case study. This case study was thoroughly illustrated in [8], and also aspects of its internal and external validity were considered in [9] as well.

Note that processes can be modeled taking into account different views [10] such as (i) functional that includes the activities' structure, inputs, and outputs; (ii) informational

that includes the structure and interrelationships among artifacts produced or consumed by the activities; (iii) behavioral that models the dynamic view of processes; (iv) organizational that deals with agents, roles, and responsibilities. Additionally, a methodological view is described in [11], which is used to represent the process constructors (e.g., specific methods) that can be applied to different descriptions of activities. In order to specify all these views, different modeling languages can be used. However, no modeling language fits all needs and preferences. Each has its own strengths and weaknesses, which can make it more suitable for modeling certain views than others [12].

This paper using UML 2.0 activity diagrams [13] and the SPEM profile [14] stresses the functional, informational, organizational, and behavioral views for the SIQinU process. Modeling its process helps to (i) ease the repeatability and understandability among practitioners, (ii) integrate and formalize different activities that are interrelated in different phases, and (iii) promote the learnability and interoperability by reusing the same ontological base coming from the C-INCAMI framework. This paper is an extension of the work presented in [15] elaborating on new aspects and views (e.g., informational and organizational) for both GOCAME and SIQinU process, as we remark later on. Summarizing, the main contributions of this paper are

(i) a six-phased strategy (SIQinU) useful for understanding and improving the QinU for WebApps, which is specified and illustrated from the process viewpoint regarding activities (i.e., the functional view), interdependencies (behavioral view), artifacts (informational view), and roles (organizational view).

(ii) foundations for reusing a multipurpose goal-oriented strategy (i.e., GOCAME) to derive and integrate more specific-purpose strategies (e.g., SIQinU) regarding its conceptual M&E framework, methods, and process views.

The remainder of this paper is organized as follows. Section 2 gives an overview of the six-phased SIQinU strategy. Section 3 provides the GOCAME rationale considering its three pillars, which are to a great extent reused by SIQinU; particularly, in Section 3.4, we discuss why SIQinU is in alignment with GOCAME regarding its conceptual M&E framework (Section 3.1), its process views (Section 3.2), and its methods (Section 3.3). Section 4 models and illustrates the six-phased SIQinU process from the above-mentioned process views. Section 5 analyzes related work considering the two quoted contributions, and, finally, in Section 6, concluding remarks as well as future work are discussed.

2. Overview of the SIQinU Strategy

SIQinU is an evaluation-driven strategy to iteratively and incrementally improve a WebApp's QinU by means of mapping actual system-in-use problems—that happened while real users were performing common WebApp tasks—to measurable EQ product attributes and by then improving the current WebApp and assessing the gain both at EQ and QinU levels. SIQinU can be implemented in an economic and systematic manner that alleviates most of the problems identified with typical usability testing studies which can be expensive, subjective, nonrepeatable, time consuming, and unreliable due to users being observed in an intrusive way. This is accomplished through utilizing server-side capabilities to collect user usage data from log files adding, for example, snippets of code in the application to specifically record data used to calculate measures and indicator values for QinU in a nonintrusive way.

Note that SIQinU can apply to systems in use other than WebApps if data can be collected for analysis regarding user activities. This may be possible in client-server network environments where the developer has control over the server code and the activities of users at their client workstations can be collected. Therefore, the major constraint is in collecting easily large amounts of data in a nonintrusive manner from which to measure and evaluate the QinU serving as the basis for improvement.

The SIQinU strategy uses quality models such as those specified in the ISO 25010 standard [1] and its enhancement, that is, the 2Q2U quality framework [2]. Once the QinU model has been established, the data collected, and metrics and indicators calculated, a preliminary analysis is made. If the agreed QinU level is not met, then EQ requirements are derived considering the application's QinU problems and its tasks, subtasks, and associated screens. In turn, taking into account the derived EQ requirements, an evaluation of the WebApp attributes is performed by inspection. Thus a product analysis regarding the EQ evaluation is performed, and changes for improvement are recommended. If the improvement actions have been implemented, then the new version is reevaluated to gauge the improvement gain both from the EQ and the QinU standpoint. Ultimately, SIQinU is a useful strategy not only for understanding but also—and most importantly—for improvement purposes.

SIQinU uses the concepts for nonfunctional requirements specification, measurement, and evaluation design, and so forth, established in the C-INCAMI framework as we will see in Section 3.1. Also, SIQinU has an integrated, well-defined, and repeatable M&E process, which follows to great extent the GOCAME process as we will discuss in Sections 3.2 and 3.3. Specifically, the SIQinU process embraces six phases as shown in Figure 2, which stresses the main phases and interdependencies.

Additionally, Table 1 provides, with Phase (Ph.) reference numbers as per Figure 2, a brief description of each phase, the involved activities, and main artifacts. Section 4 thoroughly illustrates phases, activities, interdependencies, artifacts, as well as roles taking into account aspects of the functional, behavioral, informational, and organizational views.

Lastly, in the Introduction, we stated as contribution that GOCAME—a previously developed strategy—can be reused to derive and integrate more specific-purpose strategies (as is the case of SIQinU) regarding its conceptual M&E framework, process, and methods. Therefore, in the following section, the GOCAME strategy regarding these principles is outlined.

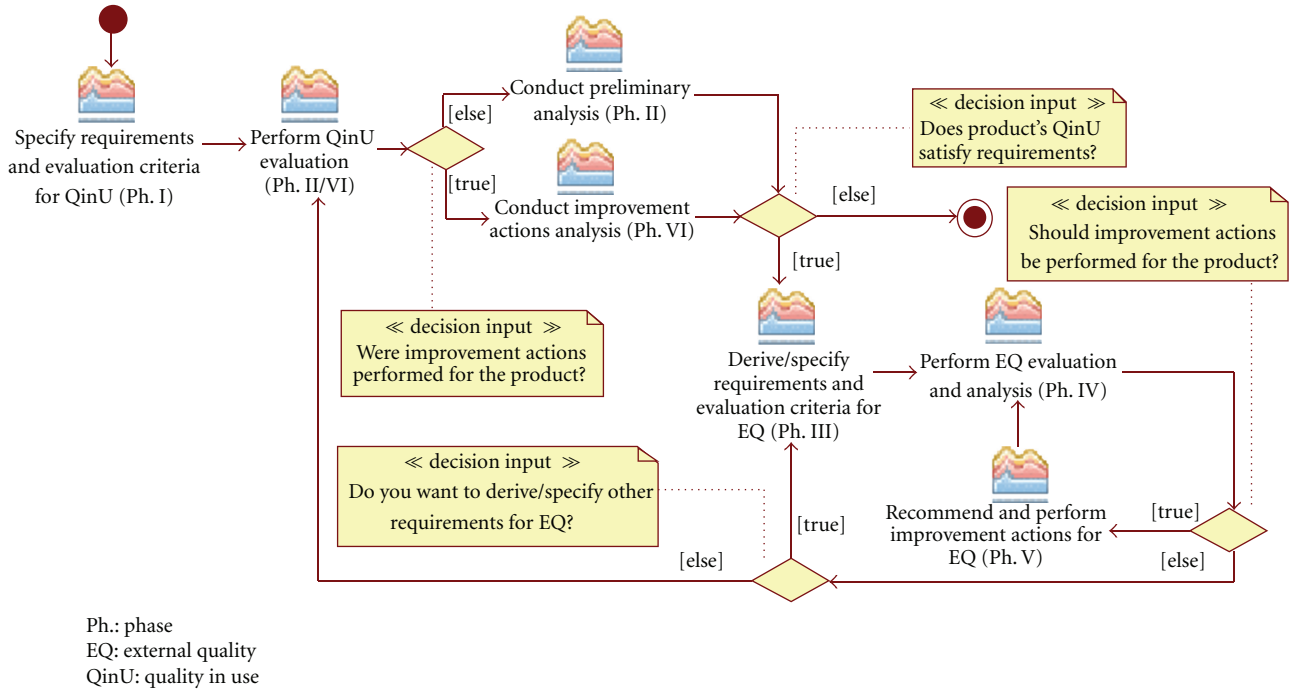


Figure 2: Process overview of SIQinU stressing phases and interdependencies.

3. GOCAME Strategy

GOCAME is a multipurpose M&E strategy that follows a goal-oriented and context-sensitive approach in defining projects. It allows the definition of M&E projects including well-specified context descriptions, providing therefore more robust evaluation interpretations among different project results at intra- and interorganization levels.

GOCAME is based on the three above-mentioned pillars, namely, a conceptual framework (described in Section 3.1); a M&E process (Section 3.2); methods and tools (Section 3.3). Finally, in Section 3.4, we discuss why SIQinU is in alignment with GOCAME regarding these capabilities.

3.1. C-INCAMI Conceptual Framework. The C-INCAMI framework provides a domain (ontological) model defining all the concepts and relationships needed to design and implement M&E processes. It is an approach in which the requirements specification, M&E design, and analysis of results are designed to satisfy a specific information need in a given context. In C-INCAMI, concepts and relationships are meant to be used along all the M&E activities. This way, a common understanding of data and metadata is shared among the organization's projects lending to more consistent analysis and results across projects.

Following the main activities of the process (shown in Section 3.2), the framework—that is, the related concepts and relationships—is structured in six components or modules, namely,

(i) *measurement and evaluation project definition*;

(ii) *nonfunctional requirements specification*;

(iii) *context specification*;

(iv) *measurement design and implementation*;

(v) *evaluation design and implementation*;

(vi) *analysis and recommendation specification*.

For illustration purposes, Figure 3 shows the main concepts and relationships for four components (i.e., from (ii) to (v)), and Table 2 defines the used terms, stressed in *italic* in the following text). The entire modeling of components can be found in [4, 5].

Briefly outlined, the GOCAME strategy follows a goal-oriented approach in which all the activities are guided by agreed *Information Needs*; these are intended to satisfy particular nonfunctional requirements of some *Entity* for a particular purpose and stakeholder's viewpoint. The nonfunctional requirements are represented by *Concept Models* including high-level *Calculable Concepts*, as in ISO 25010's quality models [1], which, in turn, measurable *Attributes* of the entity under analysis are combined. The instantiated quality models are the backbone for measurement and evaluation. *Measurement* is specified and implemented by using *Metrics*, which define how to represent and collect attributes' values; *Evaluation* is specified and implemented by using *Indicators*, which define how to interpret attributes' values and calculate higher-level calculable concepts of the quality model.

Since each *MEProject* does not occur in isolation, we therefore say that measurement and evaluation should be supported by *Context*; thus, context specifications may be

Table 1: SIQinU phases, activities, and artifacts.

Phases (Ph.)	Phase description and activities involved	Artifacts (work products)
Ph. I Specify requirements and evaluation criteria for QinU	Taking into account the recorded data of the WebApp's usage, we reengineer QinU requirements. This embraces designing tasks, defining user type, specifying usage context and characteristics. Activities include (see Figure 8) (i) establish information need; (ii) specify project context; (iii) design tasks; (iv) select QinU concept model; (v) design QinU measurement and evaluation; (vi) design Preliminary Analysis	(1) Information Need specification (2) Context specification (3) Task/subtasks specification (4) QinU NFR tree (5) QinU metrics and indicators specification (6) Analysis design
Ph. II Perform QinU evaluation and conduct preliminary analysis	As per Ph. I, data is collected purposely targeting QinU attributes for improvement. Depending on the WebApp's data collection capabilities, we collect data such as the date/time, the data is gathered, errors, task, and subtask completion and accuracy, and so forth. It includes (see Figure 12) (i) collect and parse data pertaining to tasks with their subtasks; (ii) quantify QinU attributes; (iii) calculate QinU indicators; (iv) conduct preliminary analysis	(1) Parsed data file (2) Measure and indicator values for QinU (3) QinU preliminary analysis report
Ph. III Derive/Specify Requirements and Evaluation Criteria for EQ	Based on Ph. I and II, we derive EQ requirements, that is, characteristics and attributes, with their metrics and indicators in order to understand the current WebApp's quality. Activities include (see Figure 13) (i) select EQ concept model; (ii) design EQ measurement; (iii) design EQ evaluation	(1) EQ NFR tree (2) EQ metrics and indicators specification
Ph. IV Perform EQ evaluation and analysis	Activities include (see Figure 15) (i) quantify EQ attributes; (ii) calculate EQ indicators; (iii) conduct an EQ analysis and identify parts of the WebApp that need improvement	(1) Measure and indicator values for EQ (2) EQ analysis report (and new report after reevaluation)
Ph. V Recommend, perform improvement actions, and reevaluate EQ	Using the EQ attributes that require improvement, we make improvement recommendations for modifying the WebApp, that is, version 1 to 1.1. Activities include (see Table 9) (i) recommend improvement actions; (ii) design improvement actions; (iii) perform improvement actions; (iv) evaluate improvement gain to note improvement from benchmark in Ph. IV. Note that once changes were made on the WebApp (Phase V), evaluators could detect that other EQ attributes (from problems identified in QinU) should be derived—under the premise that if further EQ improvement in these new attributes will result in greater impact on the improvement gain in QinU. This concern is taken into account in the process as shown in Figure 2	(1) EQ recommendations report (2) Improvement plan (3) New application version
Ph. VI Reevaluate QinU and analyze improvement actions	Once the new version has been used by real users, we evaluate QinU again to determine the influence of what was improved for the WebApp's EQ on QinU. This provides insight to further develop the *depends-on* and *influences* relationships [8]. Activities include (i) evaluate QinU again to determine level of improvement from Ph. II; (ii) conduct improvement action analysis, which includes developing *depends-on* and *influences* relationships between EQ improvements and QinU	(1) New measure and indicator values for QinU (2) QinU improvement analysis report (3) EQ/QinU attribute relationship table (see Table 11)

provided in order to support sounder analysis, interpretations, and recommendations. A summarized description for each component is provided below.

3.1.1. M&E Project Definition Component. This component defines and relates a set of *Project* concepts needed to articulate M&E activities, roles, and artifacts.

A clear separation of concerns among *Nonfunctional Requirements Project*, *Measurement Project,* and *Evaluation Project* concepts is made for reuse purposes as well as for easing management's role. The main concept in this component is a measurement and evaluation project (*MEProject*), which allows defining a concrete requirement project with the information need and the rest of the nonfunctional requirements information. From this requirement project, one or more measurement projects can be defined and associated; in turn, for each measurement project, one or more evaluation projects could be defined. Hence, for each measurement and evaluation project we can manage associated subprojects accordingly. Each project also has information such as responsible person's name and contact

Table 2: Some M&E terms—see [4] for more details.

Concept	Definition
Project terms	
Evaluation project	A project that allows, starting from a measurement project and a concept model of a nonfunctional requirement project, assigning indicators, and performing the calculation in an evaluation process.
Measurement project	A project that allows, starting from a nonfunctional requirements project, assigning metrics to attributes, and recording the values in a measurement process.
MEProject (i.e., measurement and evaluation project)	A project that integrates related nonfunctional requirements, measurement and evaluation projects, and then allows managing and keeping track of all related metadata and data.
Project	Planned temporal effort, which embraces the specification of activities and resources constraints performed to reach a particular goal.
Nonfunctional requirements project	A project that allows specifying nonfunctional requirements for measurement and evaluation activities.
Nonfunctional requirements terms	
Attribute (synonyms: property, feature)	A measurable physical or abstract property of an entity category.
Calculable concept (synonym: characteristic, dimension)	Abstract relationship between attributes of entities and information needs.
Concept model (synonyms: factor, feature model)	The set of subconcepts and the relationships between them, which provide the basis for specifying the concept requirement and its further evaluation or estimation.
Entity	A concrete object that belongs to an entity category.
Entity category (synonym: object)	Object category that is to be characterized by measuring its attributes.
Information need	Insight necessary to manage objectives, goals, risks, and problems.
Requirement tree	A requirement tree is a constraint to the kind of relationships among the elements of the concept model, regarding the graph theory.
Context terms	
Context	A special kind of entity representing the state of the situation of an entity, which is relevant for a particular information need. The situation of an entity involves the task, the purpose of that task, and the interaction of the entity with other entities as for that task and purpose.
Context property (synonyms: context attribute, feature)	An attribute that describes the context of a given entity; it is associated to one of the entities that participates in the described context.
Measurement terms	
Calculation method	A particular logical sequence of operations specified for allowing the realization of a formula or indicator description by a calculation.
Direct metric (synonyms: base, single metric)	A metric of an attribute that does not depend upon a metric of any other attribute.
Indirect metric (synonyms: derived, hybrid metric)	A metric of an attribute that is derived from metrics of one or more other attributes.
Measure	The number or category assigned to an attribute of an entity by making a measurement.
Measurement	An activity that uses a metric definition in order to produce a measure's value.
Measurement method (synonyms: counting rule, protocol)	The particular logical sequence of operations and possible heuristics specified for allowing the realization of a direct metric description by a measurement.
Metric	The defined measurement or calculation method and the measurement scale.
Scale	A set of values with defined properties. *Note*. The scale type depends on the nature of the relationship between values of the scale. The scale types mostly used in software engineering are classified into nominal, ordinal, interval, ratio, and absolute.
Unit	A particular quantity defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitude relative to that quantity.

Table 2: Continued.

Concept	Definition
Evaluation terms	
Decision criterion (synonym: acceptability level)	Thresholds, targets, or patterns used to determine the need for action or further investigation, or to describe the level of confidence in a given result.
Elementary indicator (synonyms: elementary preference, criterion)	An indicator that does not depend upon other indicators to evaluate or estimate a calculable concept.
Elementary model (synonym: elementary criterion function)	Algorithm or function with associated decision criteria that model an elementary indicator.
Evaluation (synonym: calculation)	Activity that uses an indicator definition in order to produce an indicator's value.
Global indicator (synonyms: global preference, criterion)	An indicator that is derived from other indicators to evaluate or estimate a calculable concept.
Global model (synonyms: scoring, aggregation model, or function)	Algorithm or function with associated decision criteria that model a global indicator.
Indicator (synonym: criterion)	The defined calculation method and scale in addition to the model and decision criteria in order to provide an estimate or evaluation of a calculable concept with respect to defined information needs.
Indicator value (synonym: preference value)	The number or category assigned to a calculable concept by making an evaluation.

information, starting and ending date, amongst other relevant information. Ultimately, this separation of concerns for each MEProject facilitates the traceability and consistency for intra- and interproject analysis.

3.1.2. Nonfunctional Requirements Specification Component. This component includes concepts and relationships needed to define the nonfunctional requirements for measurement and evaluation. One key concept is the *Information Need*, which specifies (see Figure 3)

(i) the purpose for performing the evaluation (which can be for instance "understand," "predict," "improve," "control," etc.);

(ii) the focus concept (*CalculableConcept*) to be assessed (e.g., "operability," "quality in use," "actual usability," etc.);

(iii) the category of the entity (*EntityCategory*) that will be assessed, for example, a "Web application" (which its superCategory is a "product" or "information system") and the concrete *Entities* (such as "JIRA," "Mantis" WebApps, etc.). Other super categories for entities can be "resource," "process," "information system-in-use" (e.g., as a Web application-in-use), and "project"

(iv) the userViewpoint (i.e., the intended stakeholder as "developer," "final user," etc.) from which the focus concept (and model) will be evaluated;

(v) the *Context* that characterizes the situation defined by the previous items to a particular MEProject.

The focus concept constitutes the higher-level concept of the nonfunctional requirements; in turn, a calculable concept and its subconcepts are related by means of a *Concept Model*. This model may be a tree-structured representation in terms of related mid-level calculable concepts and lower-level measurable *Attributes*, which are associated to the target entity. Predefined instances of metadata for information needs, entities, and entity categories, calculable concepts, attributes, and so forth, and its corresponding data can be obtained from an organizational repository to support reusability and consistency in the requirements specification along the organizational projects.

3.1.3. Context Specification Component. This component includes concepts and relationships dealing with the context information specification. The main concept is *Context*, which represents the relevant state of the situation of the entity to be assessed with regard to the stated information need. We consider Context as a special kind of *Entity* in which related relevant entities are involved. Consequently, the context can be quantified through its related entities. By relevant entities, we mean those that could affect how the focus concept of the assessed entity is interpreted (examples of relevant entities of the context may include resources as a network infrastructure, a working team, lifecycle types, the organization, or the project itself, among others).

In order to describe the situation, attributes of the relevant entities (involved in the context) are used. These are also Attributes called *Context Properties* and can be quantified to describe the relevant context of the entity under analysis. A context property inherits the metadata from the Attribute class such as name, definition, and objective, and also adds other information (see Figure 3). All these context properties' metadata are meant to be stored in the organizational repository, and, for each MEProject, the particular metadata and its values are stored as well. A detailed illustration of context and the relationship with other C-INCAMI components can be found in [5].

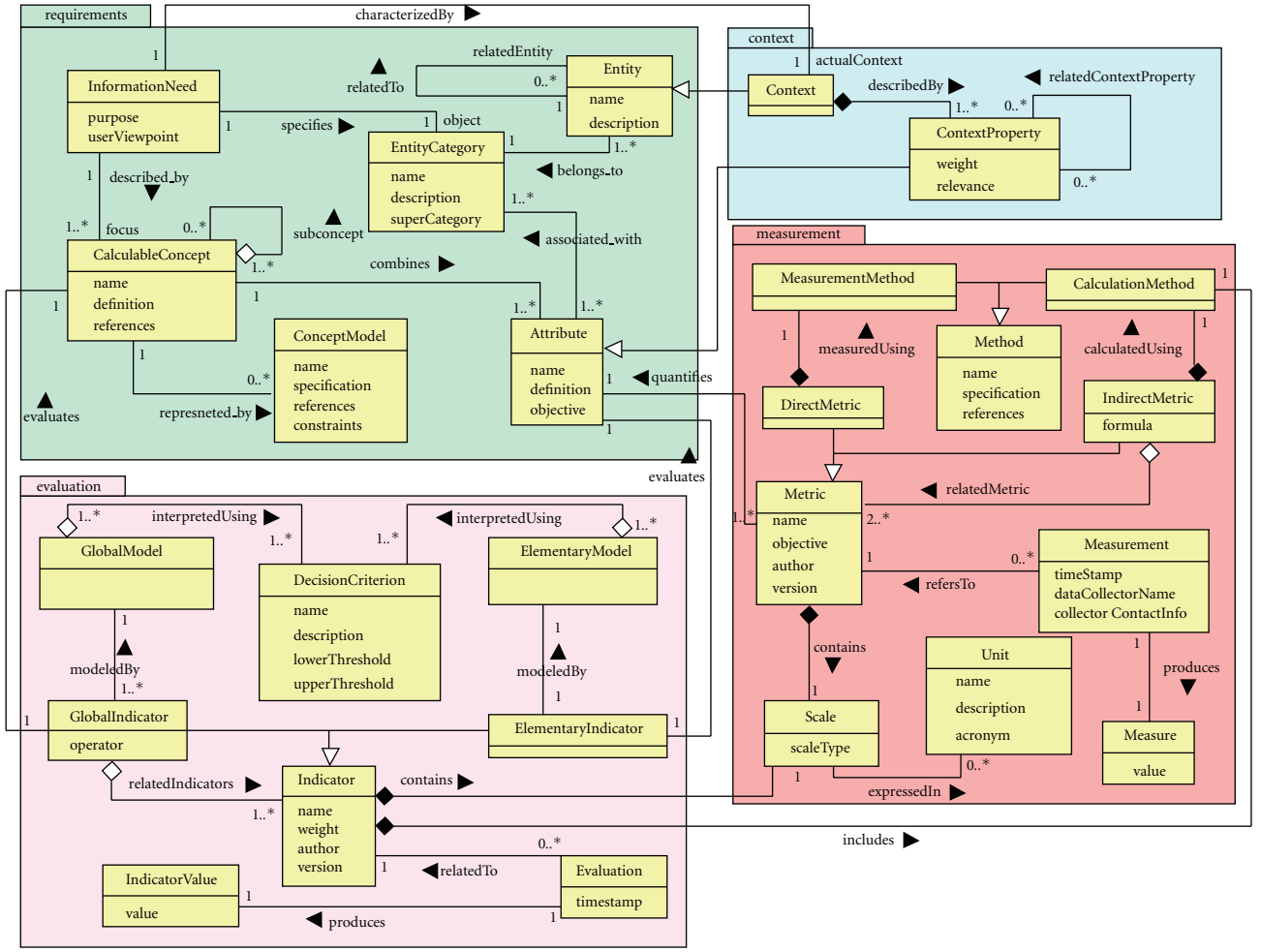


Figure 3: Main concepts and relationships of the C-INCAMI framework. Four out of six C-INCAMI components are depicted as packages, namely, nonfunctional requirements specification, context specification, measurement design and implementation, and evaluation design and implementation.

3.1.4. Measurement Design and Implementation Component. This module includes the concepts and relationships intended to specify the measurement design and implementation, for instance, the concrete Entities that will be measured, the selected *Metric* for each attribute, and so on.

Regarding measurement design, a metric provides a *Measurement* specification of how to quantify a particular attribute of an entity, using a particular *Method*, and how to represent its values, using a particular *Scale*. The properties of the measured values in the scale with regard to the allowed mathematical and statistical operations and analysis are given by the *scaleType* [16]. Two types of metrics are distinguished. *Direct Metrics* are those for which values are obtained directly from measuring the corresponding entity's attribute, by using a *Measurement Method*. On the other hand, *Indirect Metrics*' values are calculated from others direct metrics' values following a function specification and a particular *Calculation Method*.

For measurement implementation, a Measurement specifies the activity by using a particular metric description in order to produce a *Measure* value. Other associated metadata is the data collector name and the timestamp in which the measurement was performed.

3.1.5. Evaluation Design and Implementation Component. This component includes the concepts and relationships intended to specify the evaluation design and implementation. Indicator is the main term, which allows specifying how to calculate and interpret the attributes and calculable concepts of nonfunctional requirement models.

Two types of indicators are distinguished. First, *Elementary Indicators* that evaluate lower-level requirements, namely, attributes combined in a concept model. Each elementary indicator has an *Elementary Model* that provides a mapping function from the metric's measures (the domain) to the indicator's scale (the range). The new scale is interpreted using agreed *Decision Criteria*, which help analyze the level of satisfaction reached by each elementary nonfunctional requirement, that is, by each attribute. Second, *Partial/Global Indicators*, which evaluate mid-level

and higher-level requirements, that is, subcharacteristics and characteristics in a concept model. Different aggregation models (*GlobalModel*), like logic scoring of preference models, neuronal networks models, and fuzzy logic models, can be used to perform evaluations. The global indicator's value ultimately represents the global degree of satisfaction in meeting the stated information need for a given purpose and user viewpoint.

As for the implementation, an *Evaluation* represents the activity involving a single calculation, following a particular indicator specification—either elementary or global—producing an *Indicator Value*.

It is worthy to mention that the selected metrics are useful for a measurement process as long as the selected indicators are useful for an evaluation process in order to interpret the stated information need.

3.1.6. Analysis and Recommendation Specification Component. This component includes concepts and relationships dealing with analysis design and implementation as well as conclusion and recommendation. Analysis and recommendation component uses information coming from each MEProject (which includes requirements, context, measurement, and evaluation data and metadata). By storing all this information and by using different kinds of statistical techniques and visualization tools, stakeholders can analyze the assessed entities' strengths and weaknesses with regard to established information needs, and justify recommendations in a consistent way. Note this component is not shown in Figure 3. However, it is shown in Table 5 from the process specification standpoint.

3.2. GOCAME Measurement and Evaluation Process. When modeling a process, often engineers think more about what a process must do rather than how activities should be performed. In order to foster repeatability and reproducibility, a process specifies (i.e., prescribes or informs) a set of phases and activities, inputs and outputs, interdependencies, among other concerns. Also, to deal with the inherent complexity of processes, process views—also quoted in process modeling literature as perspectives—are used. A view is a particular model or approach to represent, specify, and communicate regarding the process. For instance, according to [10], a process can be modeled taking into account four views, namely, functional, behavioral, informational, and organizational.

Considering these process views, the functional perspective for GOCAME represents what activities and tasks (instead of the often-used term "task" in process modeling, which represents a fine grained or atomic activity, we will use the term "sub-activity" in the rest of the text, since, in Section 4, for QinU modeling, the term task has a very specific meaning) should be specified, what hierarchical activities structure (also known as task breakdown structure) there exists, what conditions (pre- and postconditions) should be accomplished, and what inputs and outputs (artifacts) will be required. Taking into account the terminology and components used in the C-INCAMI framework (Section 3.1), the integrated process of GOCAME embraces the following core activities: (i) *Define Non-Functional Requirements*; (ii) *Design the Measurement*; (iii) *Design the Evaluation*; (iv) *Implement the Measurement*; (v) *Implement the Evaluation*; (vi) *Analyze and Recommend* as shown in Figure 4. In addition, in Table 3, we enumerate these six activities, their involved subactivities, and the main output artifacts.

The behavioral view represents the dynamics of the process, that is, the sequencing and synchronization of activities, parallelisms, iterations, feedback loops, beginning and ending conditions, among other issues. The core GOCAME activities as well as sequences, parallelisms, main inputs, and outputs are depicted in Figure 4. The ≪*datastore*≫ stereotype shown in the figure represents repositories; for instance, the *Metrics* repository stores the metadata for the previously designed metrics. More details for the GOCAME functional and behavioral process views can be found in [6].

On the other hand, the informational view is concerned with those artifacts produced or required (consumed) by activities, the artifact breakdown structure, strategies of configuration management, and traceability models. For example, for illustration purpose, in Figure 5, the structure for the *Non-Functional Requirements Specification,* and *Metrics Specification* documents, which are outputs of *A.1* and *A.2* activities (see Table 3) is modeled. As the reader can observe in Figure 5(a), the *Non-Functional Requirements Specification* artifact is composed of the *Information Need Specification*, the *Context Specification* and the *Non-Functional Requirements Tree* documents. Besides, the *Metrics Specification* artifact (Figure 5(b)) is composed of a set of one or more *Metric Specification*, which in turn is composed of a *Scale* and a *Calculation* or *Measurement Method* descriptions. Note that, aimed at easing the communication among stakeholders these models can complement the textual specification made in the third column of Table 3.

Finally, the organizational view deals with what agents and their associated resources participate-plan-execute-control what activities; which roles (in terms of responsibilities and skills) are assigned to agents; what groups' dynamic and communication strategies are used, among other aspects. To illustrate this, Figure 6 depicts the different roles and their associated GOCAME activities. In Table 4, each role definition and its involved activities are also listed. Note that we have used italics in the definition column (in Table 4) to show the terminological correspondence between the process role definition and the C-INCAMI conceptual framework. It is important to remark that a role can be assumed by a human or an automated agent. And a human agent can be embodied by one or more persons, that is, a team.

In order to combine the above views, Table 5 presents a template which specifies just the *Analyze and Recommend* activity. The template specifies the activity name, objective and description, the subactivities and involved roles, input and output artifacts, pre- and postconditions. Also a diagram representing the *Analyze and Recommend* activity is attached as well to enable understanding and communicability.

Summarizing, the GOCAME M&E process can be described as follows. Once the *nonfunctional requirements*

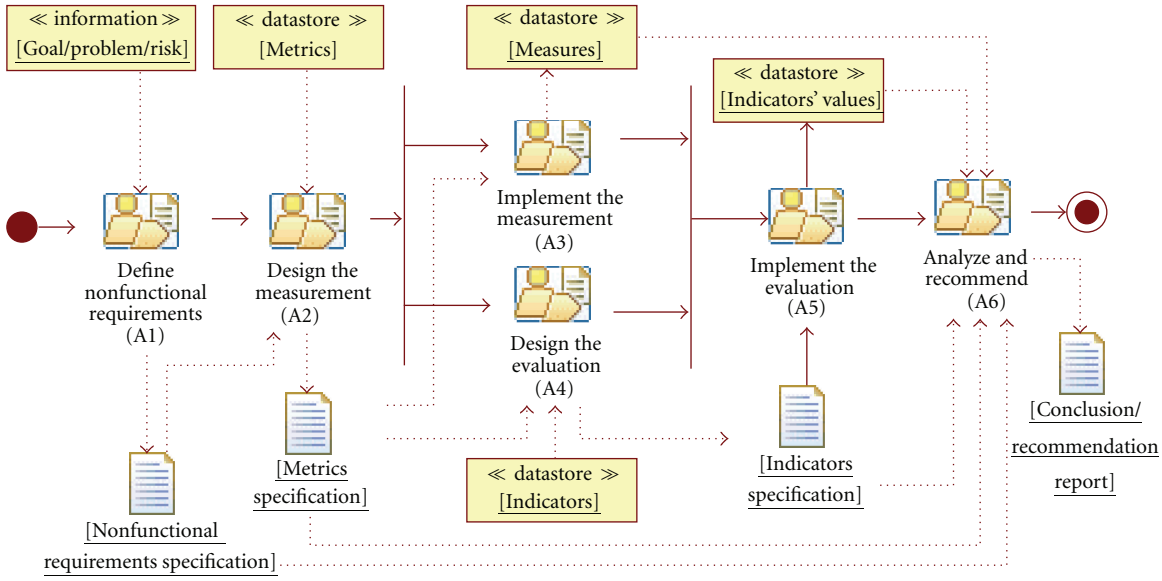


Figure 4: Overview of the GOCAME measurement and evaluation process.

Table 3: GOCAME core activities and main output artifacts.

Activities (A.)	Subactivities	Artifacts (Work Products)
A.1 Define Nonfunctional requirements	Subactivities include (i) establish information need; (ii) specify project context; (iii) select a concept model. Note that these (and below) subactivities can in turn be broken down in new ones—see [6] for more details.	Nonfunctional requirements specification (this artifact is composed of (i) information need specification; (ii) context specification; (iii) nonfunctional requirements tree)
A.2 Design the measurement	Subactivities include (i) establish entity (optional); (ii) assign one metric to each attribute.	Metrics specification
A.3 Implement the measurement	Subactivities include (i) establish entity; (ii) measure attributes	Measure values
A.4 Design the evaluation	Subactivities include (i) identify elementary indicators; (ii) identify partial and global indicators.	Indicators specification (this artifact is composed of (i) elementary indicators specification; (ii) partial/global indicators specification)
A.5 Implement the evaluation	Subactivities include (i) calculate elementary indicators; (ii) calculate partial and global indicators	Indicator values
A.6 Analyze and recommend	Subactivities include (i) design the analysis; (ii) implement the analysis; (iii) elaborate the conclusion report; (iv) perform recommendations.	Conclusion/recommendation report (this artifact is composed of (i) analysis specification; (ii) analysis report; (iii) conclusion report; (iv) recommendations report)

project has been created by the *nonfunctional requirements manager*, then, the *define non-functional requirements* activity has a specific *goal* or *problem* (agreed with the *evaluation requester*) as input and a *nonfunctional specification document* as output. Then, in the *design the measurement* activity, the *metrics expert* identifies the metrics to quantify attributes. The metrics are selected from a *metrics repository*, and the output is the *metric specification document*. Once the measurement was designed—taking into account raised issues for the evaluator requester, for example, the precision of metrics, and so forth—the *evaluation design* and the *measurement implementation* activities can be performed in any order or in parallel as shown in Figure 4. Therefore, the *design the evaluation* activity is performed by the *indicators expert* who allows identifying elementary and global indicators and their acceptability levels (agreed also with the *evaluation requester*). Both the *measurement design* and the *evaluation design* are led by the *measurement and evaluation managers* accordingly. To the *implement the measurement* activity, the *data collector* uses the specified metrics to obtain the measures, which are stored in the *measures repository*. Next, the *implement the evaluation* activity can be

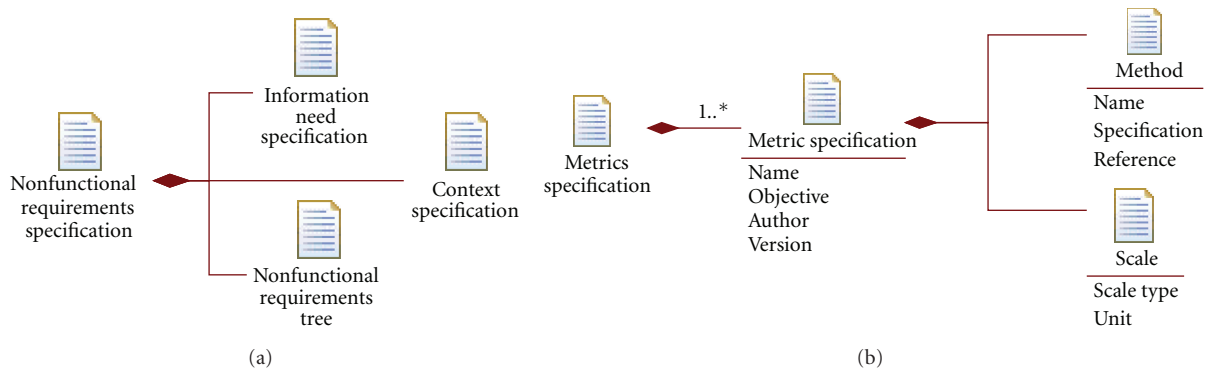


Figure 5: Excerpt of the informational view for A.1 and A.2 in Table 3, regarding artifact composition. (a) Nonfunctional requirements specification documents; (b) metrics specification document.

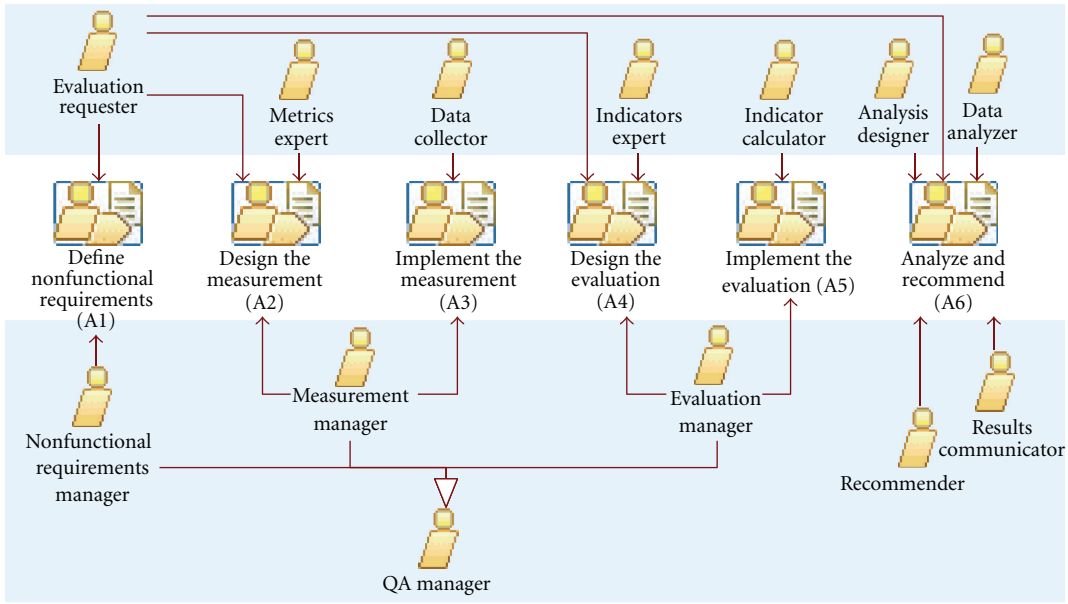


Figure 6: The organizational view: roles assigned to the GOCAME activities.

carried out by the *indicator calculator*—this role usually is enacted by a tool. Finally, *analyze and recommend* activity is performed by *analysis designer*, *data analyzer*, *recommender*, and *results communicator* roles. This activity has as inputs measures and indicators values (i.e., data), the requirements specification document, and the associated metrics and indicators specifications (i.e., metadata) in order to produce a *conclusion/recommendation report*.

3.3. GOCAME Methods and Tools: WebQEM and C-INCAMI_Tool. While activities state "what" to do, methods describe "how" to perform these activities accomplished by agents and roles, which in turn can be automated by tools. In addition, a methodology is a set of related methods. Since the above M&E process includes activities such as specify the requirements tree and identify metrics, we have envisioned a methodology that integrates all these aspects and tools that automate them; that is, a set of well-defined and cooperative methods, models, techniques, and tools that, applied consistently to the process activities, produces the different outcomes.

Particularly, the WebQEM and its associated tool the so-called C-INCAMI_Tool (see screenshots in Figure 7) were instantiated from the conceptual framework and process. The methodology supports an evaluation-driven approach, relying on experts and/or end users to evaluate and analyze different views of quality such as EQ and QinU for software and system-in-use applications. Note that GOCAME strategy and its methodology can be used to evaluate not only software/WebApps but also other entity categories, such as resources and processes.

In addition to the above-mentioned views, a methodological view is presented in [11]. This represents the process constructors to be applied to the different descriptions of activities in a given process. Note that, for a specific activity description, we can have one or more methods that give

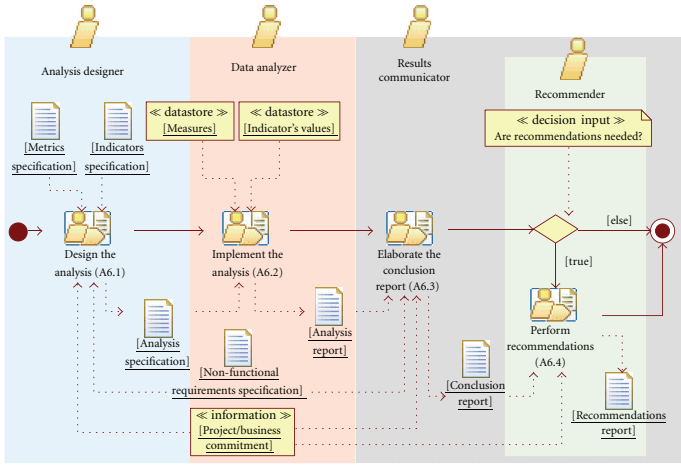
Table 4: GOCAME role definitions and involved activities.

Role name	Definition/comment	Activities (as per Figure 4)
Quality Assurance (QA) Manager	Responsible for leading a *measurement and evaluation project* (MEProject in Table 2) regarding the requester needs. Note this role is specified by three subroles as per Figure 5.	Note this role is responsible of the activities involved in three specific subroles as per Figure 5.
Nonfunctional requirements manager	Responsible for the *nonfunctional requirements project*. This role should be played by a nonfunctional requirement engineer.	(i) Define non-functional requirements
Measurement manager	Responsible for leading a *measurement project*.	(i) Design the measurement (ii) Implement the measurement
Evaluation manager	Responsible for leading an *evaluation project*.	(i) Design the evaluation (ii) Implement the evaluation
Evaluation requester	Responsible for requesting an evaluation. Note that this role can be accomplished by a human or an organization.	(i) Define nonfunctional requirements (ii) Design the measurement (iii) Design the evaluation
Metrics expert	Responsible for identifying the appropriate *metrics* from a catalogue for each *attribute* of the *requirements tree*, based on the established *information need*.	(i) Design the measurement
Data collector	Responsible for gathering *measures* of the *attributes* using the *metrics* specification. Note that the data collector role can be accomplished by either a human agent or an automatic agent.	(i) Implement the measurement
Indicators expert	Responsible for identifying the most appropriate *indicators* from a catalogue and to define *decision criteria* for each *attribute* and *calculable concept* of the *requirements tree* based on the established *information need*.	(i) Design the evaluation
Indicator calculator	Responsible for calculating the *indicators values* using the *indicators* specification. Note this role usually is accomplished by an automatic agent.	(i) Implement the evaluation
Analysis Designer	Responsible for identifying the appropriate data analysis methods and techniques to be used regarding scales, scale types, and the project/business commitment in addition to visualization and documentation techniques.	(i) Analyze and Recommend
Data analyzer	Responsible for conducting the data analysis based on the design of the analysis. Note this role can be accomplished by either a human agent or an automatic agent or both.	(i) Analyze and recommend
Recommender	Responsible for conducting the recommendations based on the conclusion report and taking into account the business commitment.	(i) Analyze and recommend
Results communicator	Responsible for communicating the evaluation results and recommendations to the evaluation requester.	(i) Analyze and recommend

support to the same activity, and, for a given method, we can have one or more tools that enact it. For instance, in Table 3, for the *A.5* activity, and particularly for the *calculate the partial/global indicators* subactivity, many methods can accomplish this such as "linear additive scoring method," "neural network method," among others.

3.4. Why Is SIQinU in Alignment with GOCAME? As we have indicated in the last paragraph of Section 2, SIQinU also relies on the three GOCAME principles above outlined and depicted in Figure 1. In fact, SIQinU utilizes the C-INCAMI conceptual base, underlying process and methods as we discuss in Section 4. However, since SIQinU is a specific-purpose goal-oriented strategy, it has specific activities, some particular methods, and procedures that are not taken into account in GOCAME. Moreover, while GOCAME is a multipurpose strategy regarding the strategy aim, SIQinU is a specific-purpose strategy. This is so, because in GOCAME the information need purpose can be "understand," "predict," "improve," "control"—as indicated in Section 3.1.2—while, in SiQinU, the purposes are just "understand" and ultimately "improve." In addition, GOCAME was designed to allow assessing different calculable concepts and entities such as the EQ or QinU of any product (including WebApps), the cost of a product, the capability quality of a resource, among others. Meanwhile, SIQinU was designed to evaluate specifically QinU of systems in-use (in a non-intrusive way) and EQ of systems, as for example, WebApps. Even more in SiQinU, from the nonfunctional requirements standpoint, QinU is evaluated from the "do goals" or pragmatic view, rather than from the "be goals" (subjective view), as thoroughly discussed in [2].

Table 5: Process template in which information and views are documented for the *Analyze and Recommend* activity.

Activity: analyze and recommend	*Code* (in Figure 4): A6
Objective: elaborate and communicate a conclusion report and (if necessary) a recommendation report for a decision-making process.	
Description: identify and select procedures, techniques and tools to be used in order to analyze data, metadata, and information, coming from metrics and indicators, for a given information need. Based on the analysis results, a conclusion report is produced, and, a recommendations report, if necessary, is yielded as well. All these reports are communicated to the evaluation requester.	
	*Subactivities*: (i) Design the analysis (A6.1) (ii) Implement the analysis (A6.2) (iii) Elaborate the conclusion report (A6.3) (iv) Perform recommendations (A6.4) *Involved roles*: (i) Analysis designer (ii) Data analyzer (iii) Recommender (iv) Results communicator
Input artifacts: (i) Nonfunctional requirements specification (ii) Metrics specification (iii) Indicators specification (iv) Measures (v) Indicators values (vi) Project/business commitment	*Output Artifacts*: Conclusion/Recommendation report. Note that this artifact is composed of (i) Analysis specification; (ii) Analysis report; (iii) Conclusion Report; and (iv) Recommendations report.
Preconditions: a MEProject must be implemented.	*Postconditions*: the MEProject finishes when the conclusion and/or recommendation report is communicated and agreed on between the QA manager and the requester of the evaluation.

Considering the process, SIQinU also reuses the GOCAME activities. For example, the GOCAME A1, A2, A4 activities, and, to some extent, the A6 activity (recall Figure 4) are included in SIQinU Ph. I and Ph. III (recall Figure 2). Likewise, the A3 and A5 activities are included in Ph. II and Ph. IV phases. However, there are particular activities in SIQinU that are not included into GOCAME. For example, in Phase V, we have activities devoted to produce WebApp improvements, as well as in Phase II, there exist activities for data filtering and collection, since SIQinU proposes utilizing server-side capabilities to gather, in a nonintrusive way, user usage data.

Considering the conceptual framework, SIQinU reuses totally C-INCAMI, that is, the ontological M&E conceptual base and its six components commented in Section 3.1. As above-mentioned SIQinU extends GOCAME activities, there are new activities (e.g., in Ph V for improvement techniques, and those related to nonintrusive data filtering in Ph. II), which lack the ontological root in C-INCAMI. Note that C-INCAMI concepts and components deal primarily with nonfunctional requirements, measurement, and evaluation issues, rather than functional aspects for design refactoring, code programming or restructuring, and so forth, which implies other domain scope and model. Note that C-INCAMI is a flexible framework that can be extended and linked with other domain models and frameworks to deal, for example, with functional aspects.

Also measurement and evaluation methods as commented in Section 3.3 are reused. However, other methods and techniques that are not included in GOCAME such as those for changing the current WebApp version (in Phase V) are needed. Finally, all the roles defined in Table 4 are totally reused as well, adding new ones for the activities of Ph V as, for example, the "Maintenance Project Manager" role (i.e., the responsible for leading a maintenance project and identifying the appropriate methods, techniques, and tools to be used for change—improve—the application) and the "Developer" role (i.e., the responsible for conducting the software/web application changes).

Despite the mentioned similarities with GOCAME, the modeling of the functional and behavioral views in SIQinU is necessary given the amount of involved phases, activities,

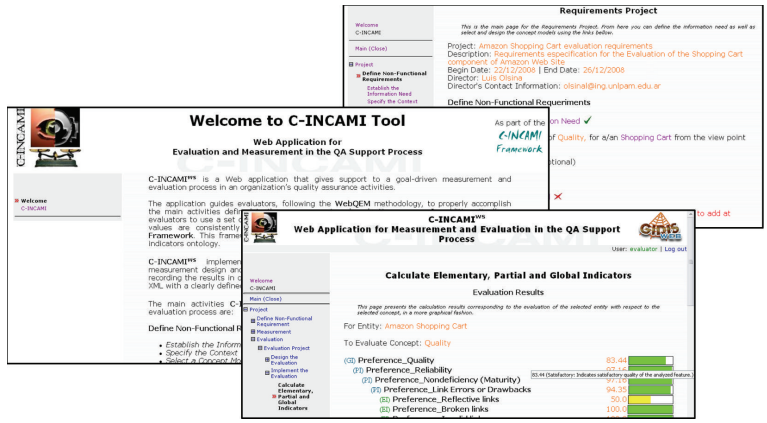


Figure 7: Snapshots of the C-INCAMI_tool.

subactivities, and their workflows. These issues will be highlighted in the next section.

4. A Process Model View for SIQinU

Process modeling is rather a complex endeavor. Given the inherent complexity of the process domain, a process can be modeled taking into account different views as analyzed in Section 3.2 for GOCAME. With the aim to model the SIQinU phases and activities, their inputs and outputs, sequences, parallelism, and iterations, we specify below using UML activity diagrams and the SPEM profile [14], the functional view taking into account behavioral concerns as well. Aspects of the organizational and informational views are to a lesser extent specified in the following diagrams, since as indicated in Section 3.4 many of the roles and artifacts are reused from the GOCAME strategy. Note that, in order to facilitate the communication, automation, and collaboration, different modeling languages and tools can be used to specify all the views. Each has its own strengths and weaknesses, which can make it more suitable for modeling certain views than others [12]. However, nowadays, SPEM is widely used and according to [17] can be adopted by researchers and practitioners for different disciplines, not just software engineering.

Also in Section 2 (Figure 2 and Table 1), we depicted the SIQinU phases, so below we concentrate on the specifications of activities and their descriptions. In order to illustrate the SIQinU process, excerpts of a case study conducted in mid-2010 are used (see details of the case study in [8]). This case study examined JIRA (http://www.atlassian.com/), a defect reporting WebApp in commercial use in over 24,000 organizations in 138 countries around the globe. JIRA's most common task, *Entering a new defect*, was evaluated in order to provide the most benefit, since entering a new defect represents a large percentage of the total usage of the application. We studied 50 beginner users in a real work environment in their daily routine of testing software and reporting defects in a software testing department of ABC, a company (with fictitious name but real one) specializing in software quality and testing. The beginner users were testers which were the majority of users. Although there are other user categories such as test managers, QA managers, and administrators, testers are the predominant user type, so we chose beginner testers as our user viewpoint.

4.1. Phase I: Specify Requirements and Evaluation Criteria for QinU. Once the requirements project has been created, using the data of the WebApp's usage recorded in log files, we reengineer and establish QinU requirements, that is, characteristics with measurable attributes, with the objective of not only understanding but also improving the system-in-use with real users. From observations of the actual WebApp, this phase embraces defining user type, designing tasks, specifying usage context and dimensions for QinU (e.g., actual usability) and their attributes. Based on these specifications, metrics (for measurement) and indicators (for evaluation) are selected. Below we describe the seven core activities (see the flow in Figure 8) involved in Ph. I.

4.1.1. Establish Information Need. This activity, according to the C-INCAMI framework (recall requirements package in Figure 3), involves *Define the purpose* and *the user viewpoint*, *establish the object* and *the entity* under study, and *identify the focus* of the evaluation (see Figure 9). These activities are accomplished by the NFR manager role considering the evaluation requester's needs. In the case study, the purpose for performing the evaluation is to "understand" and "improve" the WebApp being used from the userViewpoint of a "beginner tester." The category of the entity (i.e., the object) assessed was a "defect tracking WebApp" while the entity being studied was "JIRA" (called in our study JIRA v.1). The focus (CalculableConcept) assessed is "actual usability" and its subcharacteristics, "effectiveness in use," "efficiency in use," and "learnability in use" [2].

4.1.2. Specify Project Context. Once the information need specification document is yielded, we optionally can *Specify Project Context* as shown in Figure 8. It involves the *Select relevant Context Properties* subactivity—from the organizational repository of context properties [5], and, for each

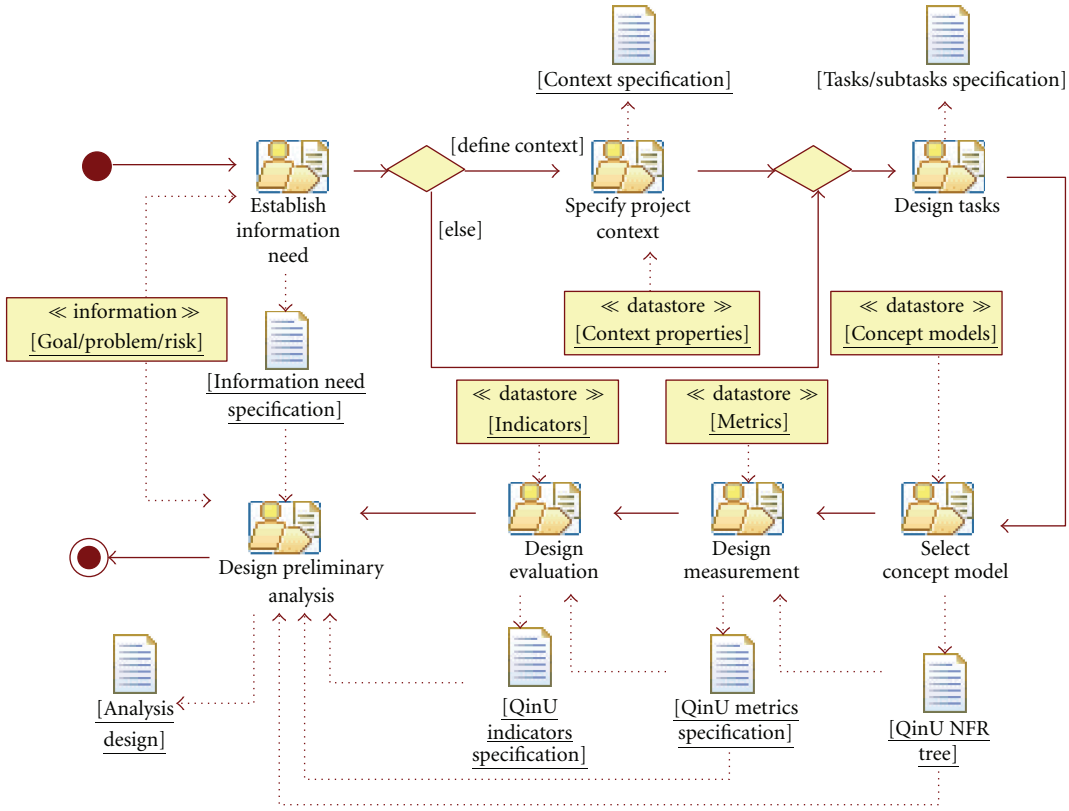


Figure 8: Overview for the specify requirements and evaluation criteria for QinU process (Ph. I).

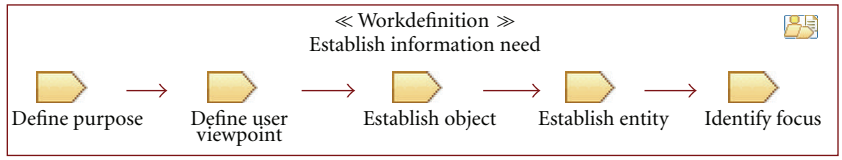


Figure 9: Activities for establish information need.

selected property, the *Quantify Context Property* activity must be performed—based on the associated metric. In the end, we get as output a context specification document for the specific project.

4.1.3. Design Tasks. In this activity, the most common and representative task or tasks should be designed. It is also important to choose a task that is performed for which sufficient data can be collected. In our case study, the selected task by the evaluation requester was "entering a new defect," as indicated above. This JIRA task included 5 subtasks specified by the task designer, namely: (i) Summary, steps, and results; (ii) Add Detail Info; (iii) Add Environment Info; (iv) Add Version Info; (v) Add Attachment (see details of tasks and screens in [8]).

4.1.4. Select QinU Concept Model. It involves both *Select a Model* and *Edit the Model* subactivities. Concept models are chosen from an organizational repository regarding the quality focus. For example, in our case study, the NFR manager based on the previously stated information need and taking into account the concept focus to evaluate actual usability, he instantiated a concept model for the "do goals" of the user [2]. Then, if the selected model is not totally suitable, for example, some subcharacteristics or attributes are missing, it is necessary to *Edit the Model*, adding or removing subconcepts, and/or attributes accordingly.

Finally, a requirements tree where attributes are the leaves and the concept focus is the root is yielded. For the selected concept model and regarding the information need and task at hand, the NFR manager instantiated the model as shown in Table 6 (attributes are in italic). Basically, the NFR manager, in the end, needs to satisfy the objectives of the sponsoring organization, that is, the evaluation requester.

4.1.5. Design QinU Measurement. For each attribute of the requirements tree—highlighted in italic in Table 6—we *Identify a Metric* to quantify them. The appropriate metrics

Table 6: Instantiated QinU NFR tree for JIRA case study.

1. Actual usability
1.1. Effectiveness in use
1.1.1. *Subtask correctness*
1.1.2. *Subtask completeness*
1.1.3. *Task successfulness*
1.2. Efficiency in use
1.2.1. *Subtask correctness efficiency*
1.2.2. *Subtask completeness efficiency*
1.2.3. *Task successfulness efficiency*
1.3. Learnability in use
1.3.1. *Subtask correctness learnability*
1.3.2. *Subtask completeness learnability*
1.3.3. *Task successfulness learnability*

are selected from a repository. In the C-INCAMI framework, two types of metrics are specified, a direct metric which applies a measurement method, that is, our data collection procedures from log files, and an indirect metric which uses a formula (based on other direct and/or indirect metrics) and calculation method (recall measurement package in Figure 3). If the metric is indirect, it is necessary *identify related metrics* and *identify attributes quantified by related Metrics* (see Figure 10). These two subactivities allow identifying the extra attributes and metrics for the indirect metric so that data collector may later gather the data accordingly.

In the JIRA case study, the metrics used to measure attributes were selected by the metrics expert from a metric catalogue which contains over 30 indirect metrics and their associated direct metrics. Below we illustrate the selected indirect metric for the subtask completeness efficiency (coded 1.2.2 in Table 6) attribute:

> Metric: average ratio of subtasks that are completed incorrectly or correctly per unit of time to do it (AvgRCput).
>
> Interpretation: 0 <= AvgRCput, more is better.
>
> Objective: calculate the overall average proportion of the subtasks that are completed, whether correct or incorrect, per time unit (usually seconds or minutes).
>
> Calculation Method (Formula): AvgRCput = AvgRC/AvgTC
>
> > AvgRC = Average ratio of subtasks that are completed incorrectly or correctly
> > AvgTC = Average time for a complete subtask, correct or incorrect
>
> Scale: numeric
>
> Type of Scale: ratio
>
> Unit (type, description): subtasks effectiveness/time, subtask completeness effectiveness per time unit (usually seconds or minutes).

As final output of these activities, we get the QinU metrics specification document.

4.1.6. Design QinU Evaluation. Once the metric specifications have been completed, we can design an indicator for each attribute and calculable concept of the requirements tree. Taking into account the C-INCAMI framework (recall evaluation package in Figure 3), there are two indicator types: elementary and global indicators. The elementary Indicators evaluate attributes and map to a new scale based on the metric's measures. The new scale is interpreted to analyze the level of satisfaction reached by each attribute. On the other hand, the global indicators (also called partial indicator if it evaluates a subcharacteristic) evaluate characteristics in a concept model and serve to analyze the level of global (or partial) satisfaction achieved.

Following the activities flow depicted in Figure 11, for each attribute of the requirements tree, the indicators expert should specify an elementary indicator by means of the next iterative activities: *establish the elementary model*, *establish the calculation method* (optional), and *identify the scale*.

The first activity (*establish the elementary model*) involves establishing a function to map between measure and indicator values and define the associated decision criteria or acceptability levels (see Section 3.1.5). In our case, the indicators expert and the evaluation requester defined three acceptability ranges in the indicator percentage scale, namely, a value within 70–90 (a marginal—bold—range) indicates a need for improvement actions; a value within 0–70 (an unsatisfactory—italic—range) means changes must take place with high priority; a score within 90–100 indicates a satisfactory level—bold italic—for the analyzed attribute. The acceptance levels in this case study were the same for all indicators, both elementary and partial/global, but could be different depending on the needs of the evaluation requester.

Note that the *establish the calculation method* activity is not mandatory because usually the model used is an easily interpreted function. In other cases, the calculation method should be specified.

Regarding to the partial/global indicators, these are specified in a similar way to the elementary indicators, as we can see in Figure 11. For example, in the JIRA case study, a global (linear additive) aggregation model to calculate the requirements tree was selected, with equal weights for their elements. This approach was used given that it was an exploratory study. Different weights would be assigned based on the requester's objectives to reflect the different levels of importance relative to one another. For example, for effectiveness in use, some organizations may weigh mistakes or correctness more heavily than completeness depending on the domain. A pharmacy or accounting application, for example, may have a higher weighting for accuracy.

The final output for the QinU evaluation design is an indicators specification document for quality in use. An artifact hierarchy (i.e., the informational view) of the indicators specification document is shown in Figure 12.

4.1.7. Design Preliminary Analysis. Taking into account the underlying SIQinU improvement objective, the specific QinU requirements for the project, the task, the metrics and indicators specifications, as well as the data properties with

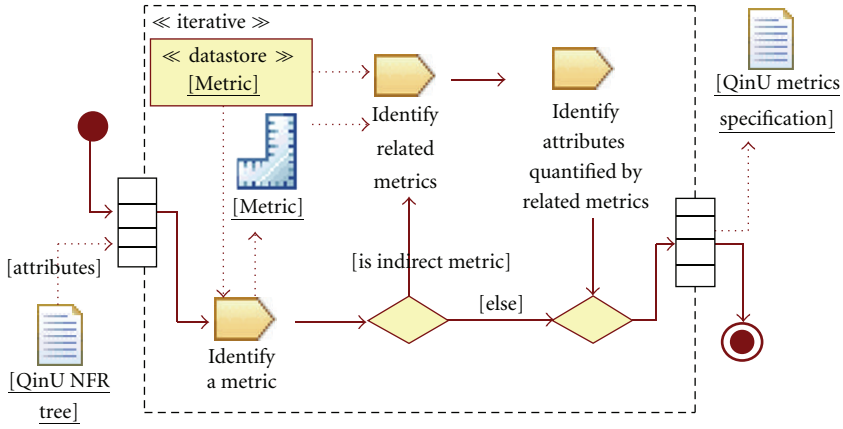


Figure 10: Design QinU Measurement activity.

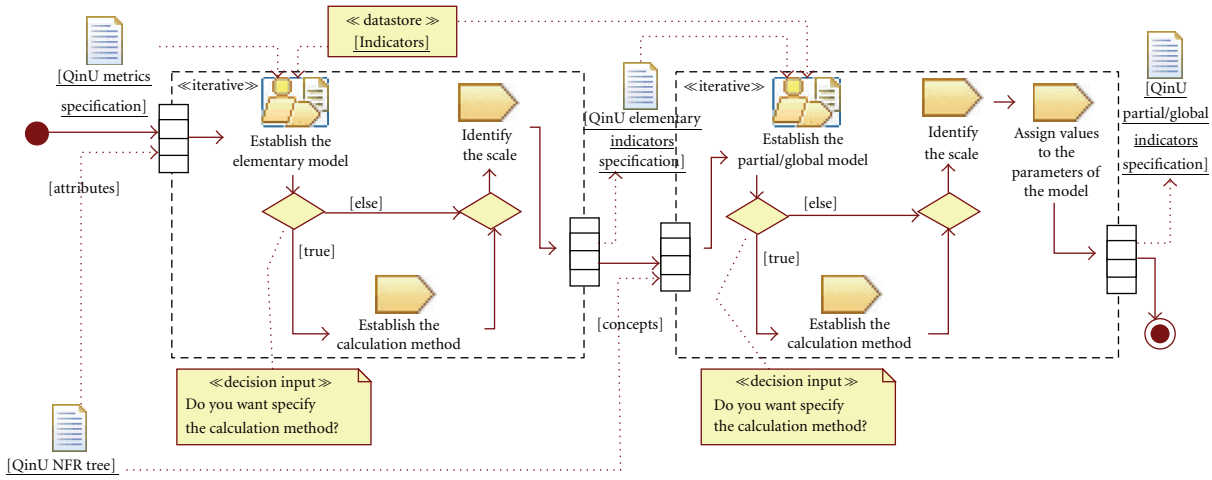


Figure 11: Design QinU evaluation activity.

regard to the scale, a preliminary analysis design should be drawn by the analysis designer role. This activity involves deciding on the allowable mathematical and statistical methods and techniques for analysis regarding the scale type, dataset properties, and so forth, the suitable tools for the kinds of analysis at hand, the presentation, and visualization mechanisms, and so forth.

4.2. Phase II: Perform QinU Evaluation and Analysis. This phase involves the basic activities to accomplish the first purpose of the SIQinU strategy, namely, understand the current QinU satisfaction level of the actual WebApp in use. To achieve this, the next four activities (see Figure 13) should be performed.

4.2.1. Collect Data. Taking into account the tasks specification, the log files with the user usage data are analyzed and the relevant data is filtered and organized to facilitate the measurement for each attribute in the next activity. Note that a tool can be used to process the log file for extracting the relevant data from user records.

4.2.2. Quantify Attributes. After collecting the data, we derive measurement values for each attribute in the QinU requirements tree. The values are obtained based on the measurement or calculation methods specified in QinU metrics specification according to the *design QinU measurement* activity (Figure 10).

4.2.3. Calculate Indicators. Taking into account the measures (values) and the indicators specification, the indicators values are calculated by the indicators calculator. The global indicator value ultimately represents the degree of satisfaction in meeting the stated information need for a concrete entity, for a given purpose, and user viewpoint. Within the *calculate indicators* activity, first, the *calculate elementary indicator* activity should be performed for each attribute of the requirements tree, and then, using these indicators' values and the specified partial or global model, the partial and global indicators are calculated by performing the *Calculate Partial/Global Indicator* activity for each calculable concept. Table 7, columns 2 and 3, shows each element of the QinU nonfunctional requirements tree evaluated at task level

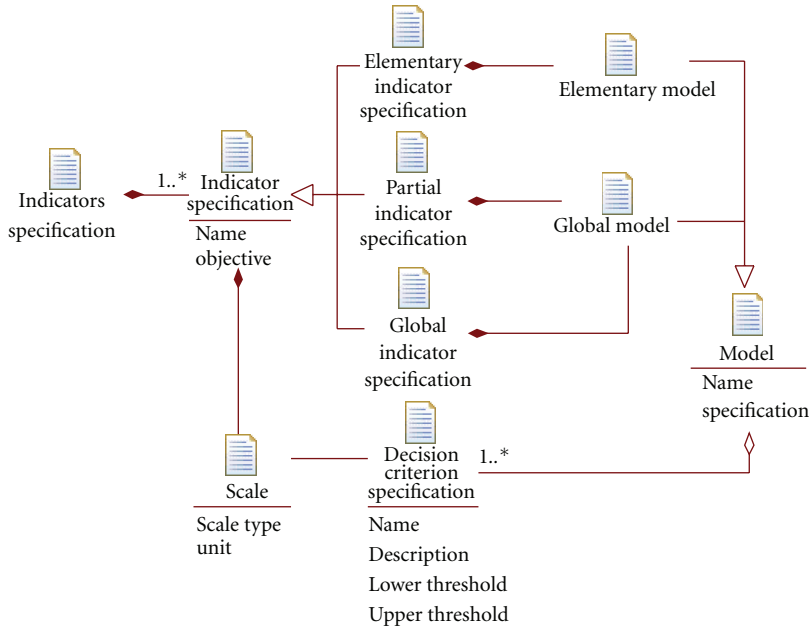


Figure 12: An informational view of the QinU indicators specification document.

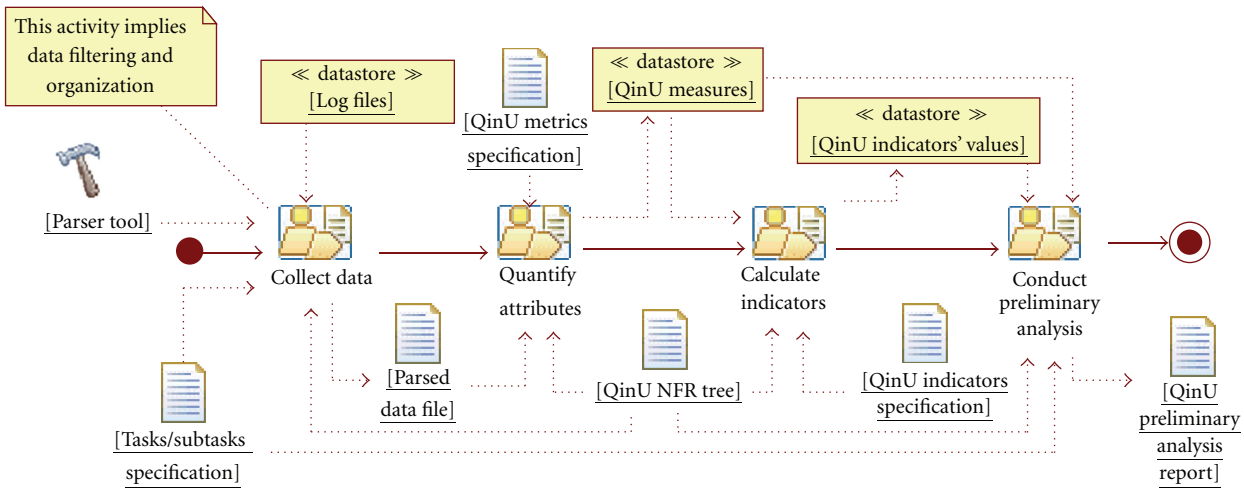


Figure 13: Overview for the perform QinU evaluation and analysis phase (Ph. II).

with elementary, partial and global indicators for the current version of JIRA (i.e., v.1).

4.2.4. Conduct Preliminary Analysis. After calculating indicators at all levels, that is, elementary, partial, and global, a preliminary analysis on the current JIRA WebApp task is conducted by the data analyzer role. Basically, it follows the analysis design (produced in the *design preliminary analysis* activity, in Ph. I), that is, implementing the designed procedures and planned tools, and storing results according to established formats in order to produce the preliminary analysis report. The analysis allows us to understand how the application performs overall (globally) and also with respect to each particular attribute for each part of the task (i.e., at subtask levels) being executed by a given user group type.

In the case study, the preliminary analysis was conducted for the above-mentioned task, its five subtasks, and their associated screens in JIRA, for the beginner user type. This allows the recommender to gauge more specifically where users had difficulty, for example, low task successfulness, low completion rate in using the application, among others.

4.3. Phase III: Derive/Specify Requirements and Evaluation Criteria for EQ. Taking into account the preliminary analysis report yielded in Phase II for QinU and the requirements tree defined in Phase I, in Phase III, a requirements tree for

Table 7: QinU evaluation of JIRA, both before (v.1) and after implementing improvements (v.1.1). EI stands for elementary indicator; P/GI stands for partial/global indicator.

Characteristics and attributes	JIRA v.1		JIRA v.1.1	
	EI	P/G I	EI	P/G I
1. Actual Usability		*53.3%*		*67.0%*
1.1. Effectiveness in use		**73.2%**		**86.7%**
1.1.1. *Subtask correctness*	**86.4%**		***91.9%***	
1.1.2. *Subtask completeness*	**87.9%**		***95.5%***	
1.1.3. *Task successfulness*	*45.5%*		**72.7%**	
1.2. Efficiency in use		*29.3%*		*42.8%*
1.2.1. *Subtask correctness efficiency*	*37.4%*		*44.3%*	
1.2.2. *Subtask completeness efficiency*	*37.5%*		*47.3%*	
1.2.3. *Task successfulness efficiency*	*13.1%*		*36.8%*	
1.3. Learnability in use		*57.3%*		**71.6%**
1.3.1. *Subtask correctness learnability*	**78.8%**		**75.1%**	
1.3.2. *Subtask completeness learnability*	*26.4%*		**77.3%**	
1.3.3. *Task successfulness learnability*	*66.7%*		*62.5%*	

EQ is derived. This requirements tree is tailored considering those product features that would need improvement with potential positive impact in QinU, mainly for those problems found in Phase II. In this phase, metrics and indicators are specified in order to evaluate the WebApp through its inspection involving three main activities (see Figure 14), namely, *select EQ concept model*, *design EQ measurement*, and *design EQ evaluation*. Note that these activities are similar to Phase I activities (recall Figure 8), but now from the EQ viewpoint.

4.3.1. Select EQ Concept Model. Given the preliminary analysis report performed in Phase II which may have reported potential problems of the actual WebApp, EQ characteristics and attributes possibly related to those QinU dimensions are identified, resulting then in a new requirements tree. The activities to be performed by the NFR manager are *select a model* and *edit the model*.

In the case study, in this activity, the requirements tree for the EQ viewpoint was established using 2Q2U (as in Phase I), instantiating the characteristics operability and information quality to determine possible effects on effectiveness in use, efficiency in use, and learnability in use. Those EQ characteristics and attributes that are possibly related to those QinU dimensions with potential problems have been instantiated resulting in the requirements tree shown on the left side of Table 8.

4.3.2. Design EQ Measurement. Following Figure 14, once the EQ model was instantiated in a requirements tree, the measurement should be designed to produce the metric specifications to perform Phase IV. As can be seen in Figure 15, this activity is similar to designing the QinU measurement (recall Figure 10 in Phase I) and is performed by the metrics expert, but now the process is executed for EQ attributes. In addition, next to *identify a metric* for an attribute from the repository and its related metrics and attributes (if the selected metric is an indirect metric), the *select a tool* activity can be performed to choose a tool that automates the metric method.

In the case study, for each attribute from the EQ requirements tree shown in Table 8, a metric was identified by the metrics expert. For instance, for the attribute navigability feedback completeness (coded 1.1.1.1), the metric is as follows.

Indirect Metric: task navigability feedback completeness (TNFC).

Objective: calculate the average of completeness considering the navigational feedback completeness level for all subtask screens for the given task.

Calculation method (formula):

$$\text{TNFC} = \sum_{j=1}^{j=n} \left(\sum_{i=1}^{i=m} \text{NFC}_{ij}/m \right)/n, \tag{1}$$

for $j = 1$ to n, where n is the number of subtasks of the given task,

for $i = 1$ to m, where m is the number of screens for subtask j.

Interpretation: 0 <= TNFC <=3, more is better.

Scale: numeric.

Scale type: ratio.

Unit: completeness level (*Note*: this metric can be converted to percentage unit, i.e., TNFC/0.03).

As this is an indirect metric, related metric and attribute were identified:

Attribute: screen navigation feedback.

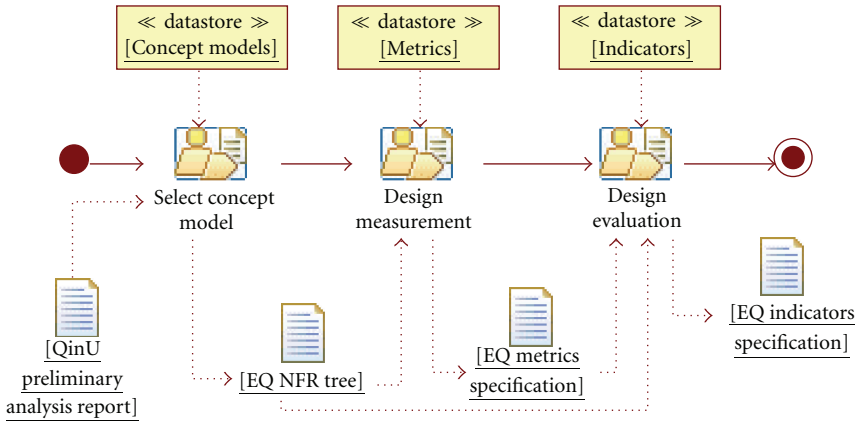


Figure 14: Overview for the derive/specify requirements and evaluation criteria for EQ process (Ph. III).

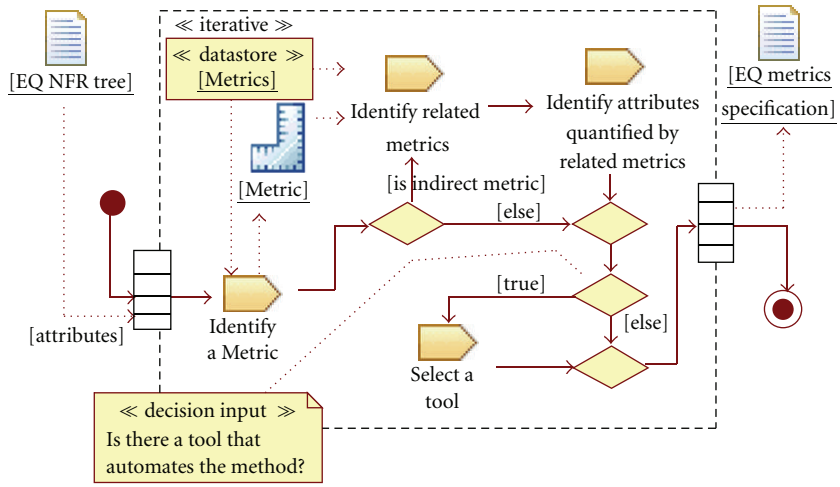


Figure 15: Activity flow involved in design external quality measurement activity.

Direct metric: navigation feedback completeness level (NFC).

Objective: determine the screen meets the criteria for navigation feedback completeness. *Note*: This metric is similar to the breadcrumb or path capability available in many WebApps.

Measurement method (type: objective): the screen is inspected to determine the rating (0–3), where evaluators through inspection observe the availability of the previous (backward), current, and next location (forward) mechanism. Screens should support the completeness of this navigation feedback.

Scale: numerical.

Scale type: ratio.

Allowed values: (0) has none; (1) has one of them; (2) has two of them; (3) has all of them.

Unit: completeness level.

4.3.3. Design EQ Evaluation. Similar to Phase I, one indicator per each attribute and concept of the EQ requirements tree should be identified by the indicators expert. In our case, when elementary, partial, and global indicators were designed, new acceptability ranges (DecisionCriterion in Figure 3) were agreed between the evaluation requester and the indicators expert. The three acceptability ranges in the indicator percentage scale were as follows: a value within 60–80 (a marginal—bold—range) indicates a need for improvement actions; a value within 0–60 (an unsatisfactory—italic—range) means changes must take place with high priority; a score within 80–100 indicates a satisfactory level—bold italic—for the analyzed attribute. Note that this indicator mapping does not necessarily have to be the same as the QinU mapping (e.g., may have different range thresholds) but rather should meet the information need and goals of the evaluation requester.

4.4. Phase IV: Perform EQ Evaluation and Analysis. Based on metric and indicator specifications obtained in Phase III, the measurement and evaluation of the EQ requirements and the analysis of the current product situation are performed. This phase is similar to Phase II, but now for EQ. The involved

activities are shown in Figure 16. Note the similarity with Phase II (recall Figure 13), but, for Phase IV, the *collect data* activity is not performed, that is, it is just carried out in Phase II to obtain user data usage from log files in a nonintrusive way. In this phase, the measurement and evaluation activities are done by inspection.

Once each attribute is measured by the data collector in *quantify attributes* activity, and all indicators are calculated in *calculate indicators* activity, the data analyzer role should *conduct EQ analysis*. The latter activity generates an EQ analysis report with information that allows us to identify, for instance, parts of the application that needs improvement from the EQ viewpoint. In Table 8 (columns 2 and 3), we can see the EQ evaluation results from JIRA (v.1) case study. Note, for example, that some attributes such as error prevention (coded 1.2.2.1.) and context-sensitive help availability (coded 1.1.2.1) need improvement with high priority. Also, we can observe that, for some elementary indicators (attributes), no improvement is needed, for example, stability of main control (coded 1.2.1.2).

4.5. Phase V: Recommend, Perform Improvement Actions, and Reevaluate EQ. Considering the previous EQ analysis report generated in *Conduct EQ Analysis* activity (Phase IV), we make recommendations to improve the application for those EQ attributes that needed improvement. After the recommended changes were completed in the current WebApp and a new version generated, we reevaluate the EQ to determine the improvement gain between both product versions. The activities for this phase are shown in Table 9 and described below.

4.5.1. Recommend Improvement Actions. Based on the EQ analysis report generated in Phase IV, the *recommend improvement actions* activity is carried out by the recommender in order to produce a recommendations report. This document has a set of recommendations for which attributes of the WebApp can be improved. For instance, a ranking of elementary indicators scored from weaker—that is, that fell in the unsatisfactory acceptability level—to stronger, but which did not fall in the satisfactory or bold italic levels can be listed.

Then, the evaluation requester can prioritize recommendations made for improvement action. Considering the case study, in this activity, some of the recommendations listed in the recommendations report were the following:

(i) for increasing the satisfaction level of defaults attribute (1.2.3.1) change fields to have default and make mandatory because they are critical defect description correctness and completeness;

(ii) for increasing the satisfaction level of Error Prevention attribute (1.2.2.1) add context sensitive help and eliminate nonvalid platform combinations.

4.5.2. Design Improvement Actions. Based on the previous recommendations report, the maintenance project manager produces an improvement plan indicating how to actually change the application. This "how" implies planning methods and techniques to be used to actually accomplish the improvement actions in the next activity (*perform improvement actions*). Methods and techniques for changing the WebApp can range from parameterized reconfigurations, code restructuring, refactoring (as made in [18]) to architectural redesign. The eventual method employed depends on the scope of the improvement recommendation as well as the resources of the evaluation requester and the desired effect. The expected effect may include an application easier to operate and learn, faster to run, more secure, among many other aspects.

For example, taking into account the two improvement recommendations listed in the above activity, the improvement plan included the following.

(i) *Recommendation*: add context sensitive help to improve the error prevention (1.2.2.1) attribute. *Action taken*: defect steps moved to next screen on add detail info, with help, examples shown to aid user.

(ii) *Recommendation*: eliminate nonvalid platform combinations to improve error prevention (1.2.2.1). *Action taken*: help provided and invalid combinations not allowed.

(iii) *Recommendation*: change fields to have default and make mandatory because they are critical defect description correctness and completeness to improve the defaults (1.2.3.1) attribute. *Action taken*: done where possible.

4.5.3. Perform Improvement Actions. With the improvement plan, the developer of the WebApp performs changes accordingly, resulting in a new application version (see the activity flow in Table 9). The ABC developer of our JIRA case study made some of the recommended changes, including those shown above, resulting in a new product version termed JIRA v.1.1. This new JIRA version had many other improvements not shown, one of which was the reduction of workload through eliminating one subtask and moving more related items together to make the overall task design more efficient. Thus, JIRA v.1.1 only has 4 subtasks, instead of 5. Because JIRA does not give access to its source code, the developer could not enact all the changes that were recommended; so only some improvements were made. Rather, through changing its configuration, they were able to perform most of the changes. Note that some recommended changes that could not be made were due to the application under study, and not due to SIQinU.

4.5.4. Evaluate Improvement Gain. Once changes were made, the WebApp can be reevaluated by inspection to determine which attributes have been improved, which have not, and get a score which can be compared to the outcomes of Phase IV. The activities involved are *quantify attributes*, *calculate indicators,* and *conduct EQ analysis*. The output is a new EQ analysis report in which the changes made

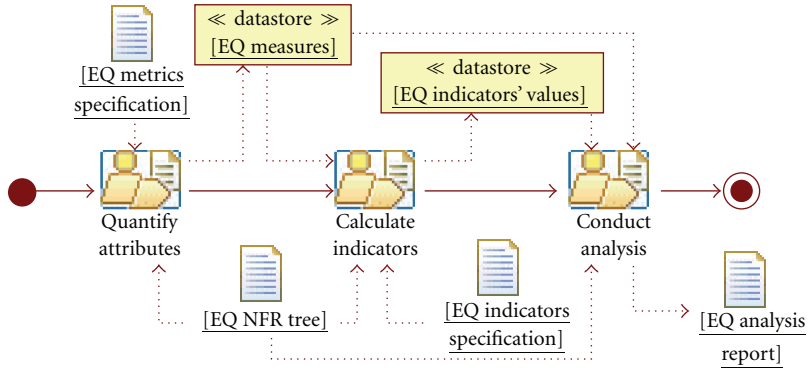


Figure 16: Overview for perform EQ evaluation and analysis phase (Ph. IV).

Table 8: EQ evaluation of JIRA, both before (v.1) and after (v.1.1) implementing improvements.

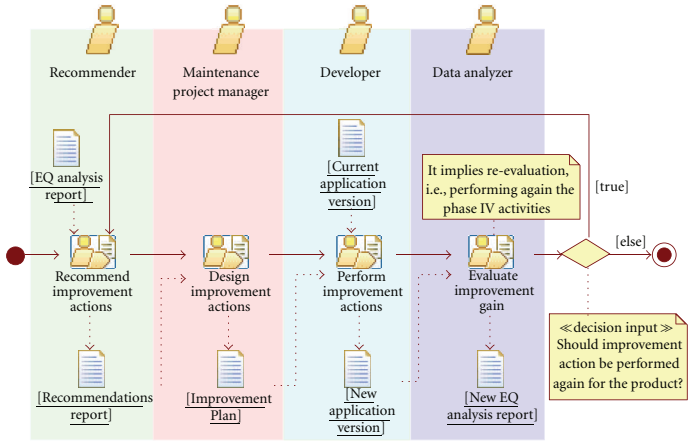
	JIRA v.1		JIRA v.1.1	
Characteristics and attributes	EI	P/G I	EI	P/G I
External Quality		*38%*		**74%**
1. Operability		*30%*		**60%**
1.1. Learnability		*26%*		*59%*
1.1.1. Feedback suitability		*38%*		*38%*
1.1.1.1. Navigability feedback completeness	*33%*		*33%*	
1.1.1.2. Task progress feedback appropriateness	*30%*		*30%*	
1.1.1.3. Entry form feedback awareness	*50%*		*50%*	
1.1.2. Helpfulness		*15%*		***80%***
1.1.2.1. Context-sensitive help availability	*20%*		***80%***	
1.1.2.2. Help completeness	*10%*		***80%***	
1.2. Ease of use		*34%*		**61%**
1.2.1. Controllability		***80%***		***80%***
1.2.1.1. Permanence of main controls	**60%**		**60%**	
1.2.1.2. Stability of main controls	***100%***		***100%***	
1.2.2. Error management		*0%*		*30%*
1.2.2.1. Error prevention	*0%*		*30%*	
1.2.3. Data entry ease		*23%*		**73%**
1.2.3.1. Defaults	*10%*		*50%*	
1.2.3.2. Mandatory entry	*10%*		***80%***	
1.2.3.3. Control appropriateness	*50%*		***90%***	
2. Information quality		*45%*		***88%***
2.1. Information suitability		*45%*		***88%***
2.1.1. Consistency		*40%*		***90%***
2.1.2. Information coverage		*50%*		***85%***
2.1.2.1. Appropriateness	*50%*		***90%***	
2.1.2.2. Completeness	*50%*		***80%***	

between the WebApp versions are compared to determine the improvement gain.

In Table 8 (columns 4 and 5), we can see the results obtained when JIRA v.1.1 was evaluated from EQ viewpoint. As can be seen from comparing the 2 evaluations (columns 2 and 3 with 4 and 5), the overall partial indicator for ease of use improved significantly from 34% to 61% with improvements in many of the individual attributes and an overall improvement in the global indicator from 38% to 74%. The next and final phase examines how these improvements in EQ affect QinU in a real context of use.

4.6. Phase VI: Reevaluate Quality in Use and Analyze Improvement Actions. Once the new application version (generated in Phase V, particularly in *perform improvement actions* activity) has been used by the same user group type in its

Table 9: Process template in which information and views are documented for the *recommend, perform improvement actions, and reevaluate EQ activities*.

Activity: recommend, perform improvement actions, and reevaluate EQ	*code* (in Figure 2): Ph. V
Objective: improve the current application version and determine the improvement gain from the EQ standpoint.	
Description: Considering the EQ analysis report generated in conduct EQ analysis activity (Phase IV), the recommender makes recommendations to improve the current application, and the maintenance project manager produces an improvement plan to enhance the current WebApp. After the recommended changes were implemented by the developer and a new version generated, a reevaluation of the EQ is performed to determine the improvement gain between both application versions.	
	*Subactivities*: (i) Recommend improvement actions (ii) Design improvement actions (iii) Perform improvement actions (iv) Evaluate improvement gain (i.e., in Phase IV) *Involved roles*: (i) Recommender (ii) Maintenance project manager (iii) Developer (iv) Data analyzer
Input artifacts: (i) EQ analysis report (ii) Current application version	*Output artifacts*: (i) EQ recommendations report (ii) Improvement plan (iii) New application version (iv) New EQ analysis report (from Phase IV)
Preconditions: there are EQ attributes with low level of satisfaction met, so improvement actions are needed to enhance the current software/web application version.	*Postconditions*: the Phase V finishes when the EQ attributes met the agreed satisfaction level.

same real context of use that the previous version, then, we are able to perform the QinU reevaluation (in a similar way to Phase II, recall Figure 13) to determine if what was improved from the EQ viewpoint had a positive quality-in-use effect using the identical tasks.

The activities involved are the same that in Phase II (see Figure 2), namely, *collect data*, *quantify attributes* and *calculate indicators*. Finally, *conduct improvement actions analysis* activity is performed to determine the improvement gain and also to hypothesize EQ/QinU relationships. These activities are described below according to the JIRA case study.

4.6.1. Collect Data, Quantify Attributes, and Calculate Indicators. When JIRA v.1.1 was evaluated from the EQ viewpoint, and its satisfaction level was achieved, then this new release was used by real users in the same real context of use as JIRA v.1. After 12 weeks (the same time period), we performed the QinU reevaluation (using the same nonfunctional requirements, metrics, and indicators designed in Phase I) to determine if what was improved from the EQ viewpoint had a positive quality-in-use effect with the same task (Entering a new defect). Following the SIQinU activities involved in Phase VI, we collect the data (i.e., the data collector using the same parser tool), quantify the attributes, and calculate all the indicators in a similar way as in Phase II. In Table 7, columns 4 and 5, we show the evaluation results for JIRA v.1.1.

4.6.2. Conduct Improvement Actions Analysis. Once all indicators were calculated, data analyzer looks at each particular attribute's change for each part of the task being executed by the user group noting the difference to calculate quantified improvement between both WebApp versions from the QinU viewpoint. Table 10 shows the attributes and indicator values regarding the QinU requirements tree depicted in Table 6 for JIRA v.1 and JIRA v.1.1 with the right most columns showing the change.

As we can see from Table 10, all attributes noted improvement with the exception of task successfulness learnability and subtask correctness learnability. A possible explanation for this is that due to metric design with data collected over a 12-week period that the learning process did not improve as much as expected. Remembering our earlier comments that temporal analysis over time is important as user behavior can change over time, these beginning users possibly were not able to ramp up their learning during this time period. And, on the other hand, if the case study had been longer, we

may have seen different behavior and hence measurements. However, their negative change was small compared to the positive changes in the other attributes resulting in an overall average change of attributes evaluation of 13.7%. While the indicators show that most of the attributes in JIRA v.1.1 still need some or significant improvement, there has been notable improvement from JIRA v.1. Again, as previously mentioned, due to the limitations in changing code in JIRA, configurations changes enabled many changes and improvements, but not all, it is possible that JIRA v.1.1 could be even better if more changes could be made. In other instances, depending on the software under evaluation, the tools/methods available to the maintenance project manager, and the time/resources available, more improvements could be made.

4.6.3. Develop "Depends-on" and "Influences" Relationships. A final action of Ph. VI (recall Table 1) is to develop *depends-on* and *influences* relationships between EQ improvements and QinU, which outputs the EQ/QinU attribute relationship table. These come also from the "influences" and "depends-on" relationships stated in the ISO 25010 standard. Table 11 summarizes the relationships found in this first case study. We discovered through our EQ evaluation, that some attributes could be improved and lead to specific improvements in QinU given the context of testers executing the specific task of entering a new defect in JIRA. In carrying out SIQinU, we were able to map EQ attributes to QinU attributes with the goal of ultimately achieving real improvement not only for JIRA but for WebApps and software design in general.

The weakness of our analysis is that we are only able to hypothesize the precise relationships between EQ and QinU attributes, but we cannot quantify the exact contribution from each because we made more than one change at a time. If we made only one change and then measured QinU for JIRA v.1.1, then we could make a more precise hypothesis for a one-to-one or one-to-many relationship. Most likely, those uncovered would be one-to-many, as one-to-one relationships probably are rare in this situation.

Regarding the hypothetical EQ/QinU relationships found in the JIRA case study, they can be further validated through additional case studies; case studies that can be carried out using our proposed SIQinU strategy grounded on a conceptual M&E framework (C-INCAMI), processes, and methods, which ensure repeatability and consistency of results. By doing this, we can generalize the conclusions of found relationships, that is, whether real improvements can be achieved not only for JIRA but for WebApps and software systems in general.

On the other hand, we are aware that this paper does not answer, from the experimental software engineering standpoint, the question of the effectiveness of the proposed strategy to accomplish similar or better objectives and results than using, for example, other strategies for improvement. The JIRA case study was made in a real context of a company, where beginner testers were performing their daily task and experts using SIQinU, as a first study. An experimental study to compare the effectiveness of SIQinU with another strategy should be performed as future work. Nevertheless, we can state that SIQinU is quite scalable in terms of amount of tasks and users performing tasks, and amount of metrics and indicators as well. Once the nonfunctional requirements are developed and data collection procedure is set, that is, the parser tool that filter and organize data from log files which serve as input to get the metrics values, then any number of metrics (and indicators) can be determined across an unlimited number of users. That is the scalability power of this nonintrusive strategy versus using usability observation and heuristic techniques.

5. Related Work

In recent years, the ISO/IEC has worked on a new project, called SQuaRE (*Software product Quality Requirements and Evaluation*), that proposes integrating many ISO standards related to quality models, M&E processes, and so forth. Although ISO 25000 [19] has guidelines for the use of the new series of standards, the documents aimed at specifying M&E processes are not issued yet. So the standards for software measurement process (ISO 15939 [20]) and the process for evaluators (ISO 14598-5 [21]) are still in effect and considered the most recent. Taking into account these standards, the process for measurement has two core activities, namely, *plan the measurement process* and *perform the measurement process* [20].The evaluation process involves five activities: *establishment of evaluation requirements*, *specification of the evaluation*, *design of the evaluation*, *execution of the evaluation plan*, and *conclusion of the evaluation* [21]. We have observed that there is so far no single ISO standard that specifies in an integrated way the M&E process and approach as a whole.

Other work, worthy to mention, is the CMMI (capability maturity model integration) [22] de facto standard, which provides support for process areas such as *measurement and analysis*, among others. It aligns information needs and measurement objectives with a focus on goal-oriented measurement—following to some extent the GQM (*goal question metric*) [23] approach and the [20] measurement process. Although CMMI specifies (specific/generic) practices to accomplish the given process area goals, a process model itself is not defined. To a certain extent, it represents practices (i.e., actions/activities) without explicitly establishing sequences, parallelisms, control points, and so forth. Some specific practices for measurement and analysis are for example, *establish measurement objectives*, *specify measures*, *obtain measurement data*, and *analyze measurement data*. However, a clear distinction between M&E processes is missing in addition to lacking a robust conceptual base for its terms.

Regarding improvement strategies for evaluation of WebApps, we can consider the work in [18], where authors present an approach for incremental EQ improvement whereby the results from EQ evaluation were used to make changes and improvements in a WebApp through WMR (*web model refactoring*) in a systematic way. But although a set of activities is considered, the underlying process is

Table 10: QinU attributes satisfaction level for JIRA v.1 and JIRA v.1.1 with improvements.

Attributes	v.1	v.1.1	Change	
1.1.1. *Subtask correctness*	**86.4**%	***91.9***%	5.5%	↑
1.1.2. *Subtask completeness*	**87.9**%	***95.5***%	7.6%	↑
1.1.3. *Task successfulness*	*45.5*%	**72.7**%	27.2%	↑
1.2.1. *Subtask correctness efficiency*	*37.4*%	*44.3*%	6.9%	↑
1.2.2. *Subtask completeness efficiency*	*37.5*%	*47.3*%	9.8%	↑
1.2.3. *Task successfulness efficiency*	*13.1*%	*36.8*%	23.7%	↑
1.3.1. *Subtask correctness learnability*	**78.8**%	**75.1**%	−3.7%	↓
1.3.2. *Subtask completeness learnability*	*26.4*%	**77.3**%	50.9%	↑
1.3.3. *Task successfulness learnability*	*66.7*%	*62.5*%	−4.2%	↓
		Average change	13.7%	

Table 11: Summary of the influence relationships between EQ and QinU attributes.

EQ attribute	QinU attribute
1.1.2.2 Operability.Learnability.Helpfulness.HelpCompleteness	Learnability in Use. Subtask completeness learnability
2.1.1 Information Quality.InfoSuitability.Consistency	
1.2.1.2 Operability.EaseOfUse.Controllability.StabilityMainControls	Effectiveness in Use. Task Successfulness
1.1.1.2 Operability.Learnability.Feedback Suitability.TaskProgressFeedbackAppropriateness	
1.1.1.3 Operability.Learnability.Feedback Suitability.EntryFormFeedbackAwareness	Efficiency in Use. Subtask completeness efficiency
1.1.2.1 Operability.Learnability.Helpfulness.Context-sensitiveHelpAvailability	
1.2.3.1 Operability.EaseOfUse.DataEntryEase.Defaults	
1.2.3.2 Operability.EaseOfUse.DataEntryEase.MandatoryEntry	
1.2.3.3 Operability.EaseOfUse.DataEntryEase.ControlAppropriateness	
2.1.1 Information Quality.InfoSuitability.Consistency	Effectiveness in Use.Subtask completeness
1.1.2.2 Operability.Learnability.Helpfulness.HelpCompleteness	Effectiveness in Use.Sub-task correctness
1.2.2.1 Operability.EaseOfUse.Error Mgmt.Error Prevention	
1.2.3.1 Operability.EaseOfUse.DataEntryEase.Defaults	
1.2.3.3 Operability.EaseOfUse.DataEntryEase.ControlAppropriateness	
2.1.2.2 Information quality.InfoSuitability.InfoCoverage.Completeness	
2.1.2.2 Information quality.InfoSuitability.InfoCoverage.Appropriateness	

neither well defined nor modeled regarding process views. On the other hand, in [24], a systematic approach to specify, measure, and evaluate QinU was discussed. However, the process used is not explicitly shown, and the outcomes were used just for understanding the current situation of the QinU for an e-learning application, without proposing any improvement strategy.

With the aim of developing quality software, there are several works that focus on improving and controlling the development process and the intermediate products because the quality of the final product is strongly dependent on the qualities of intermediate products and their respective creation processes. For example [25], deal with the use of a software project control center (SPCC) as a means for on-line collecting, interpreting, and visualizing measurement data in order to provide purpose- and role-oriented information to all involved parties (e.g., project manager, quality assurance manager) during the execution of a development project. On the other hand, in [26], the authors considered introducing usability practices into the defined software development process. With this goal in mind, authors offer to software developers a selection of human-computer interface (HCI) techniques which are appropriate to be incorporated into a defined software process. The HCI techniques are integrated into a framework organized according to the kind of software engineering activities in an iterative development where their application yields a higher usability improvement. Also, in the usability field, in [27], authors explore how some open source projects address issues of usability and describe the mechanisms, techniques, and technology used by open source communities to design and refine the interfaces to their programs. In particular, they consider how these developers cope with their distributed community, lack of domain expertise, limited resources, and separation from their users. However, in SIQinU, we start identifying problems in systems in-use, that is, with running applications

used by real users in a real context. Therefore, SIQinU is not focused on early stages of the development process but on how we can understand the current application-in-use's QinU and how we can change the attributes of a software system to improve its QinU.

Taking into account the complexity of processes, in [28], authors propose the use of an electronic process guide to provide a one-off improvement opportunity through the benefits of declaring a defined, systematic, and repeatable approach to software development. An electronic process guide offers several advantages over a printed process handbook, including easy access over the web for the most up-to-date version of the guide, electronic search facilities and hypernavigation to ease browsing information. In this work, authors combine the electronic process guide with the experience management, which refers to approaches to structure and store reusable experiences. It aims at reducing the overhead of information searching that can support software development activities. Experience management also appears to be more effective when it is process centric. Thus, the two concepts have a symbiotic relationship in that the process guide is more useful when supported by experiences and the experience base is more useful when it is process focused. The electronic process guide/experience repository (EPG/ER) is an implementation of this relationship and supports users through provision of guidance that is supplemented by task-specific experiences. However, our process specification is devoted specifically to represent the different views (as proposed in [10]) for measurement and evaluation activities rather than software development activities.

Regarding integrated strategies for M&E, it is worthy to mention the GQM$^+$ Strategies [29]—which are based on the GQM approach—as an approach that allows defining and assessing measurement goals at different organization levels, but it does not specify formal process views to conduct the evaluation and improvement lifecycle as we have shown as an integral component of SiQinU. Also, since issued, the GQM model was at different moments enlarged with proposals of processes and methods. However, [30] pointed out GQM is not intended to define metrics at a level of detail suitable to ensure that they are trustworthy, in particular, whether or not they are repeatable. Moreover, an interesting GQM enhancement, which considers indicators, has recently been issued as a technical report [31]. This approach uses both the balanced scorecard and the *goal-question-indicator-measurement* methods, in order to purposely derive the required enterprise goal-oriented indicators and metrics. It is a more robust approach for specifying enterprise-wide information needs and deriving goals and subgoals and then operationalizing questions with associated indicators and metrics. However, this approach is not based on a sound ontological conceptualization of metrics and indicators as in our research. Furthermore, the terms "measure" and "indicator" are sometimes used ambiguously, which inadvertently can result in datasets and metadata recorded inconsistently, and so it cannot assure that measurement values (and the associated metadata like metric version, scale, scale type, unit, measurement method, etc.) are trustworthy, consistent, and repeatable for further analysis among projects.

Finally, in [32], authors propose the CQA approach, consisting of a methodology (CQA-Meth) and a tool that implements it (CQA-Tool). They have applied this approach in the evaluation of the quality of UML models such as use cases, class, and statechart diagrams. Also authors have connected CQA-Tool to the different tools needed to assess the quality of models. CQA-Tool, apart from implementing the methodology, provides the capacity for building a catalogue of evaluation techniques that integrates the evaluation techniques (e.g., metrics, checklists, modeling conventions, guidelines, etc.), which are available for each software artifact. Compared with our strategies, the CQA approach lacks for instance an explicit conceptual framework from a terminological base. On the other hand, other related work in which authors try to integrate strategic management, process improvement, and quantitative measurement for managing the competitiveness of software engineering organizations is documented in [33]. In this work, a process template to specify activities from different views is considered. However, the integration of the three capabilities as made in GOCAME and SIQinU is not explicit and formalized.

6. Concluding Remarks

Ultimately, the main contribution of this paper is SIQinU, an integrated specific-purpose strategy—that is, for understanding and improving QinU—whose rationale is based on well-defined M&E processes, founded on a M&E conceptual framework backed up by an ontological base, and supported by methods and tools.

In this paper, we have specified the process of the SIQinU strategy modeled stressing the functional, informational, organizational, and behavioral views. Moreover, to illustrate the SIQinU process, excerpts from a JIRA case study were used where real users were employed to collect data and ultimately prove the usefulness of the strategy in improving the application in a process-oriented systematic means. We have also shown SIQinU, to be a derivation of GOCAME, based on three foundations, namely, the process, the conceptual framework, and methods/tools. Relying on the GOCAME foundation, SIQinU has been defined as a systematic approach with the appropriate recorded metadata of concrete projects' information needs, context properties, attributes, metrics, and indicators. This ensures that collected data are repeatable and comparable among the organization's projects. Otherwise, analysis, comparisons, and recommendations can be made in a less robust, nonconsistent, or even incorrect way.

SIQinU, although reusing the general principles of GOCAME, is a specific-purpose goal-oriented strategy with specific activities and methods that are not taken into account in GOCAME. Where GOCAME is a multipurpose strategy with general purposes such as "understand," "predict," "improve," and "control," SiQinU objectives are targeted to "understand" and ultimately "improve." In addition, as discussed in Section 3.4, SIQinU was specifically designed to evaluate QinU and EQ for WebApps, from the "do goals" perspective rather than from the "be goals."

As future work, we are planning to extend SIQinU to include processes and methods not only to gather data in a nonintrusive way (as currently it does) but also to gather data using more traditional intrusive methods such as video recording, observations, and questionnaires. This could not only help to add robustness to Phase III particularly in the derivation process from QinU problems to EQ attributes but also supplement the analysis in Phase II and VI.

Acknowledgment

This work and line of research is supported by the PAE 2007 PICT 2188 project at UNLPam, from the Science and Technology Agency, Argentina.

References

[1] ISO/IEC CD 25010.3. Systems and software engineering. Systems and software Quality Requirements and Evaluation (SQuaRE). System and software quality models, 2009.

[2] P. Lew, L. Olsina, and L. Zhang, "Quality, quality in use, actual usability and user experience as key drivers for web application evaluation," in *Proceedings of the 10th International Conference on Web Engineering (ICWE '10)*, vol. 6189 of *Lecture Notes in Computer Science*, pp. 218–232, Springer, Vienne, Austria, 2010.

[3] N. Bevan, "Extending quality in use to provide a framework for usability measurement," in *Proceedings of the 1st International Conference on Human Centered Design (HCD '09)*, vol. 5619 of *Lecture Notes in Computer Science*, pp. 13–22, Springer, San Diego, Calif, USA, 2009.

[4] L. Olsina, F. Papa, and H. Molina, "How to measure and evaluate web applications in a consistent way," in *Web Engineering: Modeling and Implementing Web Applications*, G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, Eds., HCIS, chapter 13, pp. 385–420, Springer, London, UK, 2008.

[5] H. Molina and L. Olsina, "Assessing web applications consistently: a context information approach," in *Proceedings of the 8th International Conference on Web Engineering (ICWE '08)*, pp. 224–230, Yorktown Heights, NJ, USA, July 2008.

[6] P. Becker, H. Molina, and L. Olsina, "Measurement and evaluation as quality driver," *Journal ISI (Ingénierie des Systèmes d'Information)*, vol. 15, no. 6, pp. 33–62, 2010.

[7] L. Olsina and G. Rossi, "Measuring Web application quality with WebQEM," *IEEE Multimedia*, vol. 9, no. 4, pp. 20–29, 2002.

[8] P. Lew, L. Olsina, P. Becker, and L. Zhang, "An integrated strategy to understand and manage quality in use for web applications," *Requirements Engineering Journal*, vol. 16, no. 3, 2011.

[9] E. Mendes, "The need for empirical web engineering: an Introduction," in *Web Engineering: Modelling and Implementing Web Applications*, G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, Eds., HCIS, chapter 14, pp. 421–447, Springer, London, UK, 2008.

[10] B. Curtis, M. Kellner, and J. Over, "Process modelling," *Communications of the ACM*, vol. 35, no. 9, pp. 75–90, 1992.

[11] L. Olsina, "Applying the flexible process model to build hypermedia products," in *Proceedings of the Hypertext and Hypermedia: Tools, Products, Methods (HHTPM '97)*, pp. 211–221, Hermes Ed., Paris, France, 1997.

[12] S. Acuña, N. Juristo, A. Merona, and A. Mon, *A Software Process Model Handbook for Incorporating People's Capabilities*, Springer, 1st edition, 2005.

[13] UML.Unified Modeling Language Specification, Version 2.0. Document/05-07-04, 2004.

[14] SPEM. Software Process Engineering Metamodel Specification. Doc./02-11-14., Ver.1.0, 2002.

[15] P. Becker, P. Lew, and L. Olsina, "Strategy to improve quality for software applications: a process view," in *Proceedings of the International Conference of Software and System Process (ICSSP '11)*, pp. 129–138, ACM, Honolulu, Hawaii, USA, 2011.

[16] N. E. Fenton and S. L. Pfleeger, *Software Metrics: a Rigorous and Practical Approach*, PWS Publishing Company, 2nd edition, 1997.

[17] F. García, A. Vizcaino, and C. Ebert, "Process management tools," *IEEE Software*, vol. 28, no. 2, pp. 15–18, 2011.

[18] L. Olsina, G. Rossi, A. Garrido, D. Distante, and G. Canfora, "Web applications refactoring and evaluation: a quality-oriented improvement approach," *Journal of Web Engineering, Rinton Press, US*, vol. 7, no. 4, pp. 258–280, 2008.

[19] ISO/IEC 25000. Software Engineering—Software product Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE, 2005.

[20] ISO/IEC 15939. Software Engineering—Software Measurement Process, 2002.

[21] ISO/IEC 14598-5. International Standard, Information technology—Software product evaluation—Part 5: process for evaluators, 1999.

[22] CMMI Product Team. CMMI for Development Version 1.3 (CMMI-DEV, V.1.3) CMU/SEI-2010-TR-033, SEI Carnegie-Mellon University, 2010.

[23] R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, J. J. Marciniak, Ed., vol. 1, pp. 528–532, John Wiley & Sons, 1994.

[24] G. Covella and L. Olsina, "Assessing quality in use in a consistent way," in *Proceedings of the International Conference on Web Engineering (ICWE '06)*, pp. 1–8, ACM, San Francisco, Calif, USA, July 2006.

[25] J. Munch and J. Heidrich, "Software project control centers: concepts and approaches," *Journal of Systems and Software*, vol. 70, no. 1-2, pp. 3–19, 2004.

[26] X. Ferre, N. Juriste, and A. M. Moreno, "Framework for integrating usability practices into the software process," in *Proceedings of the 6th International Conference on Product Focused Software Process Improvement (PROFES '05)*, vol. 3547 of *Lecture Notes in Computer Science*, pp. 202–215, Springer, 2005.

[27] D. M. Nichols and M. B. Twidale, "Usability processes in open source projects," *Software Process Improvement and Practice*, vol. 11, no. 2, pp. 149–162, 2006.

[28] F. Kurniawati and R. Jeffery, "The use and effects of an electronic process guide and experience repository: a longitudinal study," *Information and Software Technology*, vol. 48, no. 7, pp. 566–577, 2006.

[29] V. R. Basili, M. Lindvall, M. Regardie et al., "Linking software development and business strategy through measurement," *Computer*, vol. 43, no. 4, pp. 57–65, 2010.

[30] B. A. Kitchenham, R. T. Hughes, and S. G. Linkman, "Modeling software measurement data," *IEEE Transactions on Software Engineering*, vol. 27, no. 9, pp. 788–804, 2001.

[31] W. Goethert and M. Fisher, Deriving Enterprise-Based Measures Using the Balanced Scorecard and Goal-Driven Measurement Techniques. Software Engineering Measurement and Analysis Initiative, CMU/SEI-2003-TN-024, 2003.

[32] M. Rodrìguez, M. Genero, D. Torre, B. Blasco, and M. Piattini, "A methodology for continuos quality assessment of software artefacts," in *Proceedings of the 10th International Conference on Quality Software (QSIC '10)*, pp. 254–261, Zhangjiajie, China, July 2010.

[33] J. G. Guzmán, H. Mitre, A. Seco, and M. Velasco, "Integration of strategic management, process improvement and quantitative measurement for managing the competitiveness of software engineering organizations," *Software Quality Journal*, vol. 18, no. 3, pp. 341–359, 2010.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 642983, 12 pages
doi:10.1155/2012/642983

Research Article

Program Spectra Analysis with Theory of Evidence

Rattikorn Hewett

Department of Computer Science, Texas Tech University, Lubbock, TX 79409-3104, USA

Correspondence should be addressed to Rattikorn Hewett, rattikorn.hewett@ttu.edu

Received 30 August 2011; Revised 4 January 2012; Accepted 8 January 2012

Academic Editor: Chin-Yu Huang

This paper presents an approach to automatically analyzing *program spectra*, an execution profile of program testing results for fault localization. Using a mathematical theory of evidence for uncertainty reasoning, the proposed approach estimates the likelihood of faulty locations based on evidence from program spectra. Our approach is theoretically grounded and can be computed online. Therefore, we can predict fault locations immediately after each test execution is completed. We evaluate the approach by comparing its performance with the top three performing fault localizers using a benchmark set of real-world programs. The results show that our approach is at least as effective as others with an average *effectiveness* (the reduction of the amount of code examined to locate a fault) of 85.6% over 119 versions of the programs. We also study the quantity and quality impacts of program spectra on our approach where the quality refers to the spectra *support* in identifying that a certain unit is faulty. The results show that the effectiveness of our approach slightly improves with a larger number of failed runs but not with a larger number of passed runs. Program spectra with support quality increases from 1% to 100% improves the approach's effectiveness by 3.29%.

1. Introduction

Identifying location of faulty software is notoriously known to be among the most costly and time-consuming process in software development [1, 2]. As software gets larger and more complex, the task can be daunting even with the help of debugging tools. Over the decades, many approaches to software fault localization have been studied including diagnostic reasoning [3], program slicing [4], nearest neighbor [5], and statistical analysis [6, 7].

Recent fault localization techniques have focused on automatically analyzing program behaviors observed from the execution of a suite of test cases on the tested program called *program spectra* [8]. For each run of the test case, certain program units (e.g., statements or blocks of code) are executed and result in either a *passed test* (*run,* or *execution*) when the output of the program's execution is the same as the expected output, or a *failed test*, otherwise. A collection of program spectra contains execution profiles that indicate which part of the program is involved in each test run and whether it is passed or failed.

Spectrum-based fault localization basically tries to identify the part of the program whose activity correlates most with the resulting passed or failed test runs. Most existing spectrum-based approaches rely on similarity measures to locate faulty software units by identifying the units that most resemble the spectra error outcomes [9–15]. The technique has been used for fault localization in various applications including the Pinpoint tool [12] for large dynamic online transaction processing systems and AMPLE [16] for objected-oriented software. Spectrum-based fault localization is relatively efficient to compute and does not require modeling of the program under investigation. Therefore, it is a popular fault localization technique that can easily be integrated into testing procedures [10].

The current top three performing spectrum-based fault localizers include *Ochiai* [14], *Jaccard* [12], and *Tarantula* [13]. *Tarantula* uses a heuristic function adapted from a visualization technique while *Jaccard* and *Ochiai* employ different similarity measures, both of which are widely used in other domains such as biology and ecology. While these approaches are useful, most lack theoretical foundation and the ability to immediately incorporate new testing results into the fault localization process. Furthermore, they are not easily extensible to new findings of contributing factors. Our research aims to alleviate these shortcomings.

This paper presents a spectrum-based fault localization technique for pinpointing locations of faulty software units. The approach employs the theory of evidence called Dempster-Shafer Theory [17] for uncertainty reasoning to estimate the likelihood of faulty locations based on evidence gathered from program spectra. Our approach is theoretically grounded and computed online instead of batch. Thus, it allows the prediction of fault locations to be identified immediately as the execution of each test case is completed without having to wait to collect a set of program spectra that is large enough to be statistically valid. Our contribution also includes a study of the influences of the theory of evidence on fault localization effectiveness as well as the influences of the quantity and quality of the program spectra used in the analysis.

The rest of the paper is organized as follows: Section 2 describes preliminary concepts and terminology including basic mechanisms for software fault localization, the three spectrum-based fault localizers used in our comparison study, and the Dempster-Shafer Theory along with its fundamental elements. Section 3 presents our approach to program spectra analysis, an illustration on a small example program, and some characteristic comparisons. Section 4 evaluates the proposed technique and discusses the empirical study using a set of standard benchmarks to compare the proposed method against the other three prominent software fault localizers. Section 5 presents an empirical study to answer questions whether the performance of the proposed approach relies on the quality or quantity of the program spectra or not. Section 6 discusses the work related to fault localization and the method proposed in this paper. Section 7 concludes the paper.

2. Preliminaries

This section describes terms, concepts of spectrum-based fault localization technique, the three fault localizers and the basic foundations of the mathematical theory of evidence.

2.1. Concepts, Terms, and Notations. Following the terminology in [10], a *software failure* occurs when the actual program output, for a given input, deviates from the corresponding specified output. *Software errors*, however, define defects that may or may not cause a failure. Thus, in practice, we may not be able to detect all errors. Defects that result in failures are referred to as *software faults* (or *bugs*).

Program spectra [8] are execution profiles of the program resulting from test runs. In particular, suppose we run m test cases on a program of n units (e.g., statements, blocks, and modules). The *hit spectra* can be represented as an $m \times n$ matrix $R = (r_{ij})$, where $r_{ij} = 1$ if test i involves execution of unit j of the program, otherwise it is 0 (i.e., if test i does not involve execution of unit j). In addition, for each run of test i, we define a corresponding *error* e_i to be 1 if the test failed, and 0, otherwise (i.e., when the test was passed or successful). Program spectra include both the hit spectra and the error. To collect program spectra, we run a suite of test cases and observe execution results where the test can be performed at various levels of software units (e.g., code lines or statements, blocks of code, or modules). Much research has studied fault localization in code blocks [9, 15]. Though we illustrate our approach at a code statement (or line) level, the approach is general for application in any level of software unit.

Table 1: Different types of test run frequency count.

Software Unit	Test run	
	Passed	Failed
Not Executed	a_{00}	a_{01}
Executed	a_{10}	a_{11}

Conceptually, *fault localization* is an attempt to identify a software unit whose behaviors in all relevant test cases are most similar to the errors observed in a given program spectra. To do this, the following notations are defined. For a given test run i and a software unit j, let p and q be a binary variable signifying whether run i involves execution of unit j, and, respectively, whether run i fails or not. Here p is 1 if run i involves execution of unit j, and 0, otherwise. On the other hand, q is 1 if run i fails and 0, otherwise. Next we relate p and q to observations in the program spectra.

For each software unit j, we define the frequency of runs $a_{pq}(j) = |\{i \mid r_{ij} = p,\ e_i = q\}|$. Recall that r_{ij} represents the result of test run i on unit j (whether it executes unit j or not), and e_i represents the output error of test run i (whether the test is passed or failed). Thus, for a given software unit, we can summarize the interpretations of all possible cases of a_{pq}'s in Table 1. For example, $a_{01}(j)$ represents the number of *failed* test runs (i.e., $q = 1$) that *did not execute* (i.e., $p = 0$) unit j. Note that in general $a_{00}(j)$ is not of interest to fault localization since a successful test run that does not involve the execution of the software unit does not provide any useful information for detecting faults. It does not raise any suspicion that the unit under investigation is faulty nor does it confirm that it is not faulty. On the other hand, $a_{11}(j)$ is an important indicator for identifying faulty units.

2.2. Spectrum-Based Fault Localizers. In spectrum-based fault localization techniques, software units are ranked based on their corresponding *similarity coefficients* (e.g., see [5, 10]). A similarity coefficient of a software unit measures how closely the execution runs of the test cases that involve the considering unit resemble the errors observed [9]. The software unit that has a high similarity to the output errors is assumed to have a high probability that the unit would be the cause of such errors. Thus, a unit with a larger similarity-coefficient value would rank higher in terms of its chance to be faulty. Let S_i denote a similarity coefficient for software unit i.

Next we describe the current top three fault localizers that mainly differ by the similarity coefficient S_i. A popular Tarantula [13] has been shown to be the best performing spectrum-based fault localizer in [18]. Abreu et al. [9] have recently shown that the *Jaccard* coefficient [12] and *Ochiai* coefficient [14] marginally outperform Tarantula. Most similarity coefficients utilize the frequency of occurrences of

the result of the test execution that involves a certain code statement. We now describe them in more detail.

Tarantula. The similarity coefficient in Tarantula is adapted from a formula for displaying the color of each program statement to visualize fault locations. For a given code statement i, $\%passed(i)$ ($\%failed(i)$) represents, in percentage, a ratio of the number of passed (failed) test cases that executed i to the total number of the passed (failed) test cases in the overall test suite. Tarantula quantifies *color* value of a given code statement by the following formula:

$$\text{color} = L + \left(\frac{\%\text{passed}}{\%\text{passed} + \%\text{failed}}\right) * (H - L), \quad (1)$$

where L and H represent low end and a high end color values of the color spectrum, respectively. For example, if a program statement is executed by 10% of the passed test cases and 20% of the failed test cases, its color will be 1/3 of the way from pure red (low end color of value zero) to pure green (high end color of value 120), thus in between red and yellow making it an orange color (of value 40).

Adapting from the above color scheme, the similarity coefficient is defined as follows:

$$S_i = \frac{\%\text{failed}(i)}{\%\text{failed}(i) + \%\text{passed}(i)}. \quad (2)$$

By using the notation introduced in Section 2.1, we obtain the following:

$$\begin{aligned} \%\text{failed}(i) &= \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i)}, \\ \%\text{passed}(i) &= \frac{a_{10}(i)}{a_{10}(i) + a_{00}(i)}. \end{aligned} \quad (3)$$

The numerator in S_i is opposite from that of the color scheme because Tarantula ranks the suspicion of faulty units from those with the highest to the lowest values of $S_i's$. Thus, the higher value of S_i indicates that statement i is more likely to be faulty. Faulty likelihood is influenced by $\%failed$. On the contrary, in the color scheme, the lowest color value is represented in pure red to signify that the unit under investigation is the most likely to be faulty.

Jaccard. The Jaccard similarity coefficient [19] is a simple metric that has been used for comparing two binary data objects whose variable values are not equally important (e.g., a positive disease test result is more crucial than a negative one). Chen et al. [12] have applied the Jaccard similarity coefficient for fault localization in a pinpoint tool. The Jaccard coefficient is defined as

$$S_i = \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i) + a_{10}(i)}. \quad (4)$$

As shown in the formula, the numerator of the Jaccard similarity coefficient uses $a_{11}(i)$ to compare the similarity in frequency of the test cases of interest (i.e., failed tests that execute line i). Furthermore, the denominator omits $a_{00}(i)$, which provides no valuable information with respect to locating faulty lines. Although the Jaccard coefficient has been shown to improve fault localization performance over Tarantula [9], the difference is marginal. Additional experiments are required to evaluate and draw conclusions.

Ochiai. The Ochiai coefficient [20] has been applied to various domains including ecology and molecular biology [14]. The coefficient is defined as the following:

$$S_i = \frac{a_{11}(i)}{\sqrt{(a_{11}(i) + a_{01}(i)) * (a_{11}(i) + a_{10}(i))}}. \quad (5)$$

The Ochiai coefficient uses the same contributing factors to measure similarity as those of Jaccard's. However, the denominator has more complex computation. In fault localization application, Abreu et al.'s empirical study [9] has shown that Ochia coefficient yields more superior results than those obtained from Jaccard coefficient. However, there is no intuitive explanation given.

2.3. Mathematical Theory of Evidence. Work in spectrum-based fault localization has mostly concentrated on specifying appropriate similarity coefficients using the information extracted from program spectra. This is quite different from our approach. To provide a theoretical background of the proposed research, we describe the mathematical theory of evidence, also known as the *Dempster-Shafer (D-S) Theory* [17]. The D-S theory allows probability assignment to a set of atomic elements rather than an atomic element. Thus, the D-S theory can be viewed as a generalization of Bayesian probability theory that can explicitly represent ignorance as well as uncertainty [21].

Let $\mathcal{U}$ be a finite set of all hypotheses (atomic elements) in a problem domain. A *mass function* m provides a probability assignment to any $A \subseteq \mathcal{U}$, where $m(\varnothing) = 0$ and $\Sigma_{A\subseteq\mathcal{U}} m(A) = 1$. The mass $m(A)$ represents a belief *exactly* on A. For example, $\mathcal{U} = \{\text{faulty}, \sim\text{faulty}\}$ represents a set of two hypotheses of a suspect being faulty and nonfaulty, respectively. In such a case, the property of the mass function implies that $m(\{\text{faulty}\}) + m(\{\sim \text{faulty}\}) + m(\{\text{faulty}, \sim \text{faulty}\}) = 1$, as $m(\varnothing) = 0$. Thus, mass function is not the same as probability. When there is no information regarding $\mathcal{U}$, $m(\{\text{faulty}, \sim \text{faulty}\}) = 1$, and $m(\{\text{faulty}\}) = m(\{\sim \text{faulty}\}) = m(\varnothing) = 0$. The former (i.e., $m(\{\text{faulty}, \sim \text{faulty}\}) = 1$) deals with a state of ignorance since the hypothesis set includes all possible hypotheses and therefore, its truth is believed to be certain.

For every mass function, there are associated functions of *belief* and *plausibility*. The degree of belief on A, $\text{bel}(A)$ is defined to be $\Sigma_{X\subseteq A} m(X)$ and the plausibility of A, $\text{pl}(A)$ is $1 - \text{bel}(\sim A) = \Sigma_{X\cap A\neq\varnothing} m(X)$. For example, $\text{bel}(\{\text{faulty}\}) = m(\{\text{faulty}\}) + m(\varnothing) = m(\{\text{faulty}\})$. In general, $\text{bel}(A) = m(A)$ for any singleton set $A \subseteq \mathcal{U}$ and in such a case the computation of bel is greatly reduced. However, $\text{bel}(A)$ is not necessary the same as $m(A)$ when A is not a singleton set. Thus, m, bel and pl can be derived from one another. It can be shown that the interval $[\text{bel}(A), \text{pl}(A)]$ contains the probability of A in the classic sense (see [10]). Thus, *belief*

and *probability* are different measures. In this paper, we use the terms *likelihood* and *belief* synonymously.

A mass function can be combined using various rules including the popular *Dempster's Rule of Combination*, which is a generalization of the Bayes rule. For $X, A, B \subseteq \mathcal{U}$, a combination rule of mass functions m_1 and m_2, denoted by $m_1 \oplus m_2$ (or $m_{1,2}$) is defined as the following:

$$m_{1,2}(X) = m_1 \oplus m_2(X) = \frac{\sum_{A \cap B = X} m_1(A) m_2(B)}{1 - K}, \quad (6)$$

where $K = \Sigma_{A \cap B = \varnothing} m_1(A) m_2(B)$ and $m_1 \oplus m_2(\varnothing) = 0$.

The combination rule can be applied in pairs repeatedly to obtain a combination of multiple mass functions. The above rule strongly emphasizes the agreement between multiple sources of evidence and ignores the disagreement by the use of a normalization factor.

3. Proposed Approach

By exploiting program spectra, the proposed approach to the way an automated fault localizer builds on the concepts of Dempster-Shafer theory will be described. Section 3.1 discusses the formulation of mass functions, which are our main contributions. Section 3.2 discusses some characteristics of different fault localizers and how the proposed approach can be extended. Section 3.3 illustrates the approach in a small example along with intuitive justifications.

3.1. Mass Functions and Combination Rule. Mass functions are essential elements in estimating the likelihood of the code statement being faulty based on evidences from the program spectra described in Section 2. For any statement j of a tested program, let $\mathcal{U}_j = \{f_j, \sim f_j\}$, where f_j represents the hypothesis that j is faulty and similarly, $\sim f_j$ for nonfaulty. For each test run, we are concerned with whether the test was successful or not and what code statements were executed during the test. There are two possibilities.

Case 1 (failed test). A failed test that involves the execution of the statement under investigation is evidence that supports that the statement is likely to be faulty. In such a case, the likelihood of the statement being nonfaulty is zero. On the other hand, its likelihood of being faulty can be estimated by a ratio of one over the total number of statements involved in this test run. We can formulate this formally as follows

Recall that in the program spectra, $r_{ij} = 1$ if test i involves execution of unit j of the program, and it is 0, otherwise. Thus, a total number of units executed in test run i can be represented by $\Sigma_j r_{ij}$. We now define m_i, the mass function of failed test i for all possible nonempty subsets of the hypotheses in $\mathcal{U}_j$ as follows:

$$m_i\left(\left\{\sim f_j\right\}\right) = 0, \quad (7)$$

$$m_i\left(\left\{f_j\right\}\right) = \alpha * \left(\frac{r_{ij}}{\Sigma_j r_{ij}}\right), \quad \text{where } 0 < \alpha \leq 1, \quad (8)$$

$$m_i\left(\mathcal{U}_j\right) = 1 - m_i\left(\left\{f_j\right\}\right). \quad (9)$$

The third equation is derived from the property that $m(\varnothing) = 0$ and $\Sigma_{A \subseteq \mathcal{U}} m(A) = 1$. Based on the second equation, it should be easy to see that the likelihood of a statement being faulty can only be influenced by the (failed) test that executes that statement. The parameter α is an adjusted value that represents the strength of the property of "failed test" in determining if statement j is faulty.

Case 2 (passed test). If a test involving the execution of the statement in question is successful, it is evidence that supports that this statement behaves correctly. Thus, the likelihood of it being faulty is zero. On the other hand, the likelihood of this statement being correct (i.e., nonfaulty) can be estimated by a ratio of one over the total number of statements involved in this test run. We now define m_i, the mass function of passed test i for all possible nonempty subsets of the hypotheses in $\mathcal{U}_j$ as follows:

$$m_i\left(\left\{f_j\right\}\right) = 0, \quad (10)$$

$$m_i\left(\left\{\sim f_j\right\}\right) = \beta * \left(\frac{r_{ij}}{\Sigma_j r_{ij}}\right) \quad \text{where } 0 < \beta \leq 1, \quad (11)$$

$$m_i\left(\mathcal{U}_j\right) = 1 - m_i\left(\left\{\sim f_j\right\}\right). \quad (12)$$

It should be easy to see that in this case the likelihood of a statement being correct can only be influenced by the (successful) test that executes it. Analogous to α, the parameter β is an adjusted value that represents the strength of the property of "passed test" in determining if statement j is not faulty.

In general, the appropriate values of the adjusted values α and β are determined empirically since they are likely to depend on the size of the program, the number of tests and the ratios of failed to passed tests (see an example in Section 3.3). The more precision the values of α and β are, the more likely we can discriminate faulty belief values among a large number of software units. In this paper, α and β are estimated conservatively to one and 0.0001, respectively, in order to yield a sufficient power of discrimination for a very large program. The intuitive reason behind this is the fact that when a test fails, we can guarantee the existence of at least one faulty line. Thus, we should give very high strength to such evidence. This justifies α having the highest possible strength of one. However, when a test is successful, there is no guarantee that there is no faulty statement since the particular test may not have executed faulty statements and detected the faults. Thus, a successful test does not contribute to the belief of a statement being faulty as much as a failed test does. Nevertheless, when a statement is executed in a successful test, one may be inclined to believe that the statement is probably not likely to be faulty. As the number of such successful tests increases, we gain more confidence of such belief and thus, the successful test results have a contributing factor (although small) to the belief and cannot be ignored. In practice, the number of failed tests is typically much less than that of successful ones. Thus, each successful test should have less strength compared to that of each failed test. This explains why β takes a very small value that is as

close to zero as possible. We conjecture that the larger the size of the program and the smaller ratio of the number of failed tests to the number of successful tests would likely require much smaller β. However, to conclude such a statement requires further experiments.

Recall that a mass function of a singleton set hypothesis is the same as the degree of belief of the hypothesis. Applying one of the above two cases, a mass function is created for each of the supporting evidences (i.e., a test result, which either supports faulty or nonfaulty) of each program statement. Thus, the likelihood of a statement being faulty is estimated by combining the beliefs obtained from corresponding mass functions for each of the supporting pieces of evidence. To define the rule for combining mass functions, suppose that m_1 and m_2 are two distinct mass functions of a particular code statement i. Dempster's rule of combination can be applied as shown below. For readability, we omit i and replace $\{f_i\}$, $\{\sim f_i\}$ and $\mathcal{U}_i$ by f, $\sim f$ and $\mathcal{U}$, respectively.

$$\begin{aligned} m_{1,2}(f) &= \frac{(m_1(f)m_2(f)+m_1(f)m_2(\mathcal{U})+m_1(\mathcal{U})m_2(f))}{1-K}, \\ m_{1,2}(\sim f) &= \frac{m_1(\sim f)m_2(\sim f)+m_1(\sim f)m_2(\mathcal{U})+m_1(\mathcal{U})m_2(\sim f)}{1-K}, \\ m_{1,2}(\mathcal{U}) &= \frac{m_1(\mathcal{U})m_2(\mathcal{U})}{1-K}, \end{aligned} \tag{13}$$

where $K = m_1(f)m_2(\sim f) + m_1(\sim f)m_2(f)$.

This combination rule can be applied repeatedly pairwise until evidence from all test runs has been incorporated into the computation of the likelihood of each statement. Unlike other spectrum-based fault localizers discussed in Section 2.2, instead of ranking the lines based on the similarity coefficient values, our proposed approach ranks the lines based on the corresponding likelihood of them being faulty using the beliefs combined from all of the test evidence.

3.2. Characteristic Comparisons and Generalization. This section discusses and compares some characteristics of the proposed approach with other fault localizers: *Tarantula, Jaccard,* and *Ochiai*. Our approach is based on the principle of uncertainty reasoning, while the others are based on similarity coefficients, which share common characteristics. We now describe them below assuming that a code statement in question is given.

Suppose we classify test runs into those that *failed* and those that *executed* the code statement. The coefficient of each of the three localizers reaches its minimum value of zero when there is no test in both groups (i.e., $a_{11} = 0$). This means that there is no association between failed tests and tests that executed the code statement. On the other hand, the coefficient reflects a maximum association with a value one when all tests are in both groups. In other words, there is no failed test that did not execute the statement (i.e., $a_{01} = 0$) and no passed test that executed the statement (i.e., $a_{10} = 0$).

Unlike *Tarantula*, *Jaccard,* and *Ochiai* do not use a_{00} to compute the similarity coefficient. The denominator of the *Jaccard* coefficient represents the "sum" of all *failed* (i.e., $a_{11}+a_{01}$) and all *executed* tests (i.e., $a_{11}+a_{10}$) but not both (i.e., excluding a_{11}). On the other hand, the denominator of the *Ochiai* coefficient, which is derived from a geometric mean of a_{11}/*failed* and a_{11}/*executed,* is a square *root* of a "product" of *executed* and *failed* tests. Thus, the Ochiai coefficient amplifies the distinction between *failed* and *executed* tests more than the Jaccard coefficient. Therefore, the Ochiai coefficient is expected to provide a better discriminator for locating fault units.

Recall that the similarity coefficient of *Tarantula* is %*failed*/(%*failed* + %*passed*). This seems reasonable. However, note that when %*passed* value is zero, the coefficient value is one regardless of the %*failed* value. Suppose that the two code statements both have zero %*passed* but one was executed in one out of 100 failed tests and the other was executed in 90 out of the 100 failed tests. The chance of the latter statement being faulty should be greater than the other but *Tarantula* would rank the two statements as equally likely to be faulty. This explains why *Ochiai* and *Jaccard* can outperform *Tarantula* as reported in [10].

Our approach uses the Dempster-Shafer Theory of Evidence to account for each test run to accumulatively support the hypothesis about the statement. This makes our approach applicable for online computation. As shown in (8), each failed test adds belief to the statement being faulty. Similarly, in (10), each passed test adds belief to the statement being correct. The belief is contributed to by a probability factor that depends on the number of times that the test executed the statement and an overall number of statements executed in that test. Thus, our reasoning is in a finer grain size than the others since it focuses on a specific test (i.e., the mass functions) and the contributing factors are not necessarily expressible in terms of a_{ij}'s as with other similarity coefficients. Thus, it would be interesting to compare the performance among these approaches.

The proposed approach can be generalized to accommodate evidence of new properties. That is, we can, respectively, generalize (8) and (10) to

$$\begin{aligned} m(\{f_i\}) &= \alpha * f, \quad \text{where } 0 < \alpha \leq 1, \\ m(\{\sim f_i\}) &= \beta * g, \quad \text{where } 0 < \beta \leq 1, \end{aligned} \tag{14}$$

where function f (function g) quantifies evidential support for statement i being faulty (correct). Thus, the proposed approached is easily extensible.

3.3. Illustrated Example. To demonstrate our approach, Algorithm 1 shows a small faulty program introduced in [10]. The program is supposed to sort a list of n rational numbers using a bubble sort algorithm. There is a total of five blocks (the last block corresponding to the body of the `RationalSort` function is not shown here). Block 4 is faulty since when we swap the order of the two rational numbers, their denominators (`den`) need to be swapped as well as the numerators (`num`).

The program spectra can be constructed after running six tests with various inputs as shown in Table 2. Tests 1, 2, and 6 are already sorted and so they result in no error. Test 3 is not sorted but because the input denominators are of the same

```
void rationalsort (
  int n, int* num, int* den)
{
  /* block 1 */
  int i, j, temp;
  for (i = n - 1; i >= 0; i--)  {
    /* block 2 */
    for (j = 0; j < i; j++) {
      /* block 3 */
      if (RationalGT(num[j], den[j],
        num[j + 1], den[j + 1])) {
        /* block 4 */
        temp = num[j];
        num[j] = num[j + 1];
        num[j + 1] = temp;
      }
    }
  }
}
```

Algorithm 1: Example of a faulty program.

value, no error occurs. However, double errors occur during Test 4's execution and so errors go undetected and Test 4 is passed. Finally, Test 5 failed since it resulted in erroneous output of ⟨1/1 2/2 4/3 3/4⟩.

In this small set of program spectra we use the adjusted strength α of value one and the adjusted strength β of value 1/6 to reflect the ratio of the number of failed test to overall number of test runs. Applying our approach, since Test 1 is a passed test (i.e., error = 0), we compute the belief of each hypothesis using the mass functions (10), (11) and (12). For example, the beliefs of hypotheses related to Block 1 after Test 1 are

$$\begin{aligned} m_1(\{f_1\}) &= 0, \\ m_1(\{\sim f_1\}) &= \beta * \left(\frac{r_{11}}{\Sigma_j r_{1j}}\right) = \left(\frac{1}{6}\right) * \left(\frac{1}{1}\right) = 0.167, \\ m_1(\mathcal{U}_1) &= 1 - m_1(\{\sim f_1\}) = 0.833. \end{aligned} \quad (15)$$

Similarly, for Test 2, which is a passed test, we can apply the mass functions (10), (11) and (12) to compute the belief of each hypothesis. For example, the beliefs of hypotheses related to Block 1 after Test 2 are

$$\begin{aligned} m_2(\{f_1\}) &= 0, \\ m_2(\{\sim f_1\}) &= \beta * \left(\frac{r_{21}}{\Sigma_j r_{2j}}\right) = \left(\frac{1}{6}\right) * \left(\frac{1}{2}\right) = 0.083, \\ m_2(\mathcal{U}_1) &= 1 - m_2(\{\sim f_1\}) = 0.917. \end{aligned} \quad (16)$$

Now we can apply the Dempster's rule of combination to update the new beliefs as evidenced by the two tests. For simplicity, we omit the subscript representing software unit of Block 1. Here $K = m_1(f)m_2(\sim f) + m_1(\sim f)m_2(f) = 0$ and we have

$$\begin{aligned} m_{1,2}(f) &= m_1(f)m_2(f) + m_1(f)m_2(\mathcal{U}) \\ &\quad + m_1(\mathcal{U})m_2(f) = 0, \\ m_{1,2}(\sim f) &= m_1(\sim f)m_2(\sim f) + m_1(\sim f)m_2(\mathcal{U}) \\ &\quad + m_1(\mathcal{U})m_2(\sim f) = 0.236, \\ m_{1,2}(\mathcal{U}) &= m_1(\mathcal{U})m_2(\mathcal{U}) = 0.764. \end{aligned} \quad (17)$$

Next consider Test 3, which is again a passed test. By applying the mass functions (10), (11), and (12), we obtain the following:

$$\begin{aligned} m_3(\{f_1\}) &= 0, \\ m_3(\{\sim f_1\}) &= \beta * \left(\frac{r_{31}}{\Sigma_j r_{3j}}\right) = \left(\frac{1}{6}\right) * \left(\frac{1}{5}\right) = 0.333, \\ m_3(\mathcal{U}_1) &= 1 - m_3(\{\sim f_1\}) = 0.967. \end{aligned} \quad (18)$$

By applying the Dempster's combination rule to $m_{1,2}$ and m_3, we have the following:

$$\begin{aligned} m_{1,2,3}(f) &= m_1(f)m_2(f) + m_1(f)m_2(\mathcal{U}) \\ &\quad + m_1(\mathcal{U})m_2(f) = 0, \\ m_{1,2,3}(\sim f) &= m_1(\sim f)m_2(\sim f) + m_1(\sim f)m_2(\mathcal{U}) \\ &\quad + m_1(\mathcal{U})m_2(\sim f) = 0.262, \\ m_{1,2,3}(\mathcal{U}) &= m_1(\mathcal{U})m_2(\mathcal{U}) = 0.738. \end{aligned} \quad (19)$$

The above belief computation repeats until no more evidence from the test runs is to be considered. Thus, the belief of the hypothesis that Block 1 being faulty is calculated in accumulative fashion. For each block, the process continues for each test to accumulate the new beliefs of each hypothesis until all tests have been considered. Each new test run can be immediately integrated into the fault

Table 2: Program spectra and results.

Test input	Block 1	Block 2	Block 3	Block 4	Block 5	Error
(1)⟨⟩	1	0	0	0	0	0
(2)⟨1/4⟩	1	1	0	0	0	0
(3)⟨2/1 1/1⟩	1	1	1	1	1	0
(4)⟨4/1 2/2 0/1⟩	1	1	1	1	1	0
(5)⟨3/1 2/2 4/3 1/4⟩	1	1	1	1	1	1
(6)⟨1/4 1/3 1/2 1/1⟩	1	1	1	0	1	0
No. of matches	1	2	3	4	3	6
Fault Likelihood	0.15	0.17	0.18	0.19	0.18	

Table 3: Belief values using different adjusted strengths.

Unit	$\alpha = 0.9, \beta = 0.1$	$\alpha = 0.99, \beta = 0.01$	$\alpha = 1, \beta = 0.0001$
Block 1	0.149	0.1946	0.1999656
Block 2	0.163	0.1962	0.1999816
Block 3	0.170	0.1969	0.1999896
Block 4	0.174	0.1973	0.1999936
Block 5	0.170	0.1969	0.1999896

localization process. It is clear that our approach supports online computing. In this example, Test 5 is the only failed test to which we apply the mass functions (7), (8) and (9).

The final results obtained for the beliefs of each block being faulty are shown in the last row of Table 2. Ranking the beliefs obtained, Block 4 is identified to be the most likely faulty location. This is as expected. In fact, as shown in the second-to-bottom line of Table 2, Block 4 has the highest number of matched executions with the error results obtained from the six tests. Thus, the approach produces the result that is also in line with the concept of finding the block that corresponds most to the error results in the spectra.

As mentioned earlier that the adjusted strength values of α and β can be determined empirically for each specific program to obtain the best discriminative power among beliefs of each software unit. For example, Table 3 shows the belief values obtained for this small program using various adjusted strength values.

As shown in Table 3, all cases of the strength values are able to correctly identify Block 4 as a faulty unit. However, different strength values give different precision on the belief values, which can be critical for large-scale program spectra.

4. Evaluation and Comparison Study

To evaluate our approach, we compare the results of our approach with the top three state-of-the-art spectrum-based fault localizers using a popular benchmark data set.

4.1. Benchmark Data and Acquisition. The Siemens Program Suite [22], which has been widely used as a benchmark for testing fault localizer effectiveness [9–11]. The suite has seven programs with multiple versions that reflect real-world scenarios. Most versions contain a single fault except a few containing two faults. However, since we typically locate one fault at a time, most studies including ours focus on methods for locating a single fault. The GNU Compiler Collection (GCC) 4.4.0 compiler was used to compile the programs and the GNU Coverage (GCov) extension was used to generate code coverage information to construct the program spectra.

Table 4: Siemens program suite.

Program Name	Versions omitted	No. versions	No. lines	No. test cases
print_tokens	4, 6	7	475	4130
print_tokens2	10	10	401	4115
replace	27, 32	32	512	5542
schedule	3, 5, 6, 7, 8, 9	9	292	2649
schedule2	9	10	297	2710
tcas	38	41	135	1608
tot_info	none	23	346	1052

For data preparation, we omit 13 out of a total of 132 faulty program versions due to inappropriate changes in the header file, lack of failure test results, and crash before GCov could produce a trace file. Thus, we use the remaining total of 119 versions in the experiment. Table 4 summarizes the information of the Siemens program suite including the program names, versions excluded and their corresponding number of (faulty) versions, executable lines of code, and test cases.

4.2. Evaluation Metric. Based on the standard method for evaluating fault localizers, we use *effectiveness* [18] as our evaluation metric. Effectiveness is defined to be the percentage of unexamined code (saved efforts) when identifying fault location. More precisely, suppose $P = \langle p_1, p_2, \ldots, p_n \rangle$ is a list of program statements (units, blocks) ranking in a descending order of similarity coefficient or likelihood values. The *effectiveness*, which signifies the reduction of the amount of code examined to locate a single fault in the software, can be specified as

$$\text{effectiveness} = \left(\frac{1-k}{n}\right) * 100\%, \tag{20}$$

where $k = \min_{1 \le j \le n}\{j \mid p_j \text{ in } P \text{ is actually a faulty line}\}$. In other words, k is the first code statement found in P that

actually is faulty. This is an optimistic measure that gives a maximum amount of unexamined code or effectiveness.

Because ranking can result in different findings for fault location, an issue of how to define the effectiveness when multiple lines have the same value of similarity coefficient or belief, can be crucial. Taking the first actual faulty statement found would be too optimistic and taking the last one found would be too pessimistic. Instead, Ali et al. [11] has proposed the *midline* adjustment, which appears to be a practical compromise. The midline takes the average rank of all the statements that have the same measure, which gives

$$\text{midline effectiveness} = \left(\frac{1-\tau}{n}\right) * 100\%, \qquad (21)$$

where $\tau = (k + m - 1)/2$, where k is defined above and $m = \max_{1\le j\le n}\{j \mid p_j \text{ in } P \;\&\; p_j\text{'s measure} = p_k\text{'s measure}\}$.

Our experiments employ the *effectiveness* metric in both conventional and adjusted midline measures of effectiveness as described above to compare the performance of our approach with those of other three fault localizers, namely, *Tarantura, Jaccard,* and *Ochiai.*

4.3. Experimental Results. We implemented the four methods in C++. The experimental results show that in a total of 119 versions, our approach gives better or the same *effectiveness* as those of the other three 100% of the time. In fact, in about 40% of all cases, our results have higher effectiveness than those of the rest; specifically they are higher than those of *Tarantura, Jaccard,* and *Ochiai* in 61, 59, and 48 versions, respectively.

Figure 1 shows the results obtained for midline effectiveness, which is less optimistic than the conventional effectiveness. By ranking program versions based on their corresponding effectiveness, Figure 1 compares the average percentages, over all 119 versions, of midline effectiveness obtained by our approach to those of the others. For easy visualization, because *Tarantula* has the lowest performance in all versions, Figure 1 displays the resulting average midline effectiveness of all versions performed by *Tarantula* in increasing order.

As shown in Figure 1, our approach results in slightly higher average midline effectiveness than those of *Tarantula, Jaccard,* and *Ochiai.* In particular, compared with *Tarantula,* our approach approximately shows up to an average of 3% increase in the midline effectiveness. Thus, the proposed approach is competitive with top performing fault localizers.

To see how the midline effectiveness differs from the conventional "optimistic" effectiveness, we compare the results obtained from each metric. Figure 2 shows the average percentages of both types of effectiveness obtained by our approach in each of the programs in the Siemens set. As expected, the midline effectiveness gives slightly less optimistic effectiveness than those of the conventional ones.

Table 5 compares the average effectiveness of overall versions for each approach (in percentages). The numbers after "±" represent variances. The conventional effectiveness is higher than that of the midline effectiveness. All methods perform competitively with no more than 0.5% difference of the effectiveness and all perform well with over 88% and 86% on the average of the conventional and midline effectiveness, respectively. As shown in Table 5, the proposed method gives the highest average percentage of effectiveness of both types, followed in order by *Ochiai, Jaccard,* and *Tarantula.* The proposed method also shows the least variance, so it performs consistently. However, the differences are marginal.

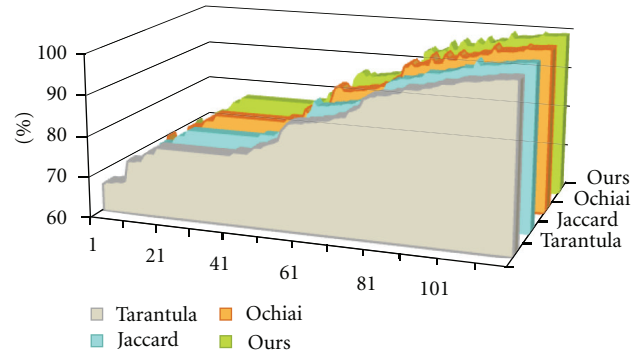


Figure 1: Comparison of average midline effectiveness.

Table 5: Average effectiveness in percentages.

Fault Localizer	% Avg. effectiveness	% Avg. midline effectiveness
Tarantula	88.10 ± 0.094	86.04 ± 0.088
Jaccard	88.14 ± 0.094	86.07 ± 0.088
Ochiai	88.34 ± 0.093	86.30 ± 0.087
Ours	88.47 ± 0.092	86.53 ± 0.086

5. Impacts of Program Spectra

This section studies the impacts of program spectra on the effectiveness of fault localization. In previous sections, although our approach shows promising and competitive results on the Siemens benchmark data, one may question whether it performs well in general, regardless of what program spectra we use. It raises the issue whether the spectra quality or quantity have any impact on the effectiveness of the proposed fault localizer. To better understand how the proposed approach performs under different conditions of program spectra, we further perform our experiments using the same concepts and methodologies introduced in [10]. The effectiveness measure in this section refers to the midline effectiveness described earlier.

5.1. Quality Measure and Impacts. In a collection of program spectra as defined in Section 2, a software unit whose column vector exactly matches with the error vector has more assurance of being faulty. This is because the program fails (represented by a value one in a corresponding error vector entry) if and only if it is executed (represented by a value one in a corresponding entry of the software unit vector).

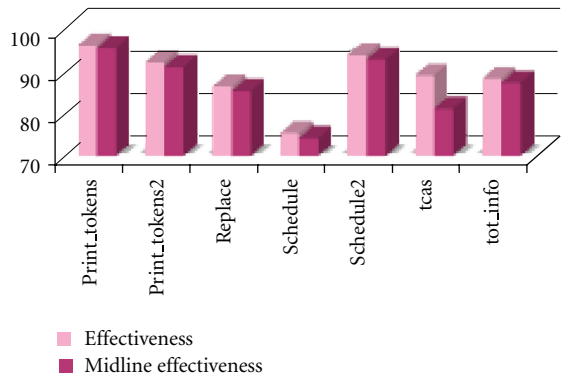


FIGURE 2: Comparison of two effectiveness measures.

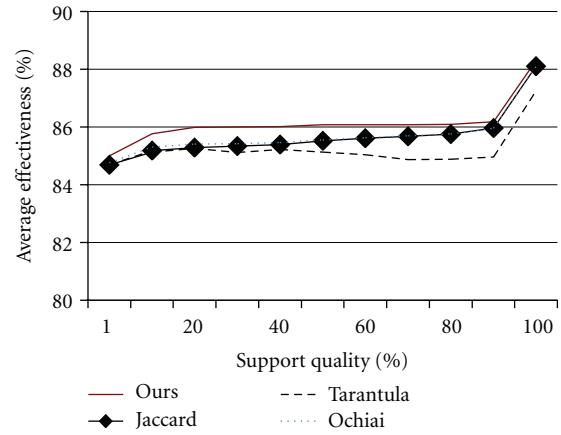


FIGURE 3: Quality impacts on effectiveness.

Locating faults in such a case can be easily achieved. However, in practice inexact matches are quite common because a program may not fail even when a faulty unit is executed. It is possible to have a faulty unit executed in a successful test run since the run may not involve a condition that introduces errors or the errors may not propagate far enough to result in a program failure. Unfortunately, the ratio between these two cases is often unknown making fault localization more complex and difficult. In fact, the higher the number of successful test runs that involved the execution of software unit i is, the less confident we are that unit i is faulty. In other words, $a_{10}(i)$ is inversely proportional to the support in locating software fault at unit i. On the other hand, as pointed out earlier, fault localization of a software unit is easy if the program fails only whenever it is executed. Thus, the number of failed test runs that involved the execution of software unit i provides support in identifying unit i as faulty. That is, $a_{11}(i)$ is proportional to the support of fault localization at unit i. Based on the above arguments, we define a *support* for locating fault at software unit i, denoted by support (i), as follows:

$$\text{support}(i) = \frac{a_{11}(i)}{a_{11}(i) + a_{10}(i)}. \tag{22}$$

In other words, the percentage of support for locating a faulty software unit is quantified by the ratio of the number of its executions in failed test runs to the total number of its executions. The support measure can be computed using information obtained from the set of program spectra used for locating faults. The higher the support value is, the more confident it is to locate the fault. Therefore, the support measure is an indicator of the quality of the program spectra in facilitating fault localization. When a fault location is known, we can compute the quality of support for locating the fault.

Each faulty version of a program in the benchmark set has an inherent *support,* whose value depends on various attributes including the type of fault and running environments of the test cases as reflected on the resulting program spectra. In the Siemens set, the support values of different faulty units range from 1.4% to 20.3%. We want to see how the quality of program spectra, as indicated by varying support values, impacts the effectiveness of the proposed fault localization approach. To do this, for each given faulty location, a different support value can be obtained by excluding runs that contribute either to a_{11} (to obtain a series of smaller support values) or a_{10} (to obtain a series of larger support values). The passed or failed runs for exclusion are randomly selected from the available set when there are too many choices. In this experiment, each controlled support value is obtained by using the maximum number of relevant runs. For example, suppose a faulty unit is executed in 10 failed and 30 passed runs. A support value of 25% would be generated from all of the available runs instead of say one passed and three failed runs. As observed in [10], although there are alternatives (e.g., setting failed runs to passed or vice versa) for controlling the support values, excluding runs is a preferred method because it maintains integrity and consistency of the data. Using the exclusion method, we obtained support values ranging from 1% to 100% (in a 10% increment).

Figure 3 shows the % of average effectiveness over all 120 faulty program versions for each approach under the study and for the support values ranging from 1% to 100%. As shown in Figure 3, the proposed approach outperformed the rest regardless of the support quality of the program spectra. As support value increases, *Jaccard*'s effectiveness moves toward that of the *Ochiai*, which consistently performed better than *Tarantura*. Each of the approaches provides the effectiveness of at least about 84% even at only 1% of quality support. Comparing these values with 100% of quality support from program spectra, the effectiveness obtained by our approach, *Ochiai, Jaccard,* and *Tarantula* increases by 3.29%, 3.35%, 3.42%, and 2.56%, respectively. The improvement of our proposed approach over *Ochiai,* the second best performing approach, ranges from 0.19% to 0.58%. The small improvements are obtained for higher quality support of 90%–100% and large improvement for lower support values of 20%–50%. This implies that the proposed approach performs more consistently and is more robust to the support quality of the program spectra than the *Ochiai* approach. As expected, the effectiveness of each

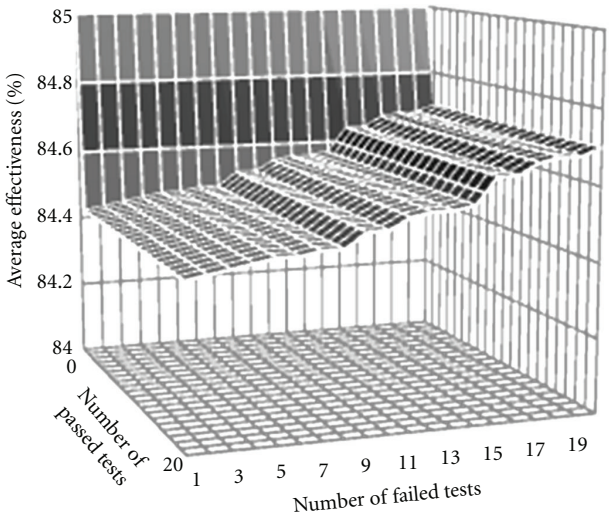


Figure 4: Quantity impact on the proposed approach.

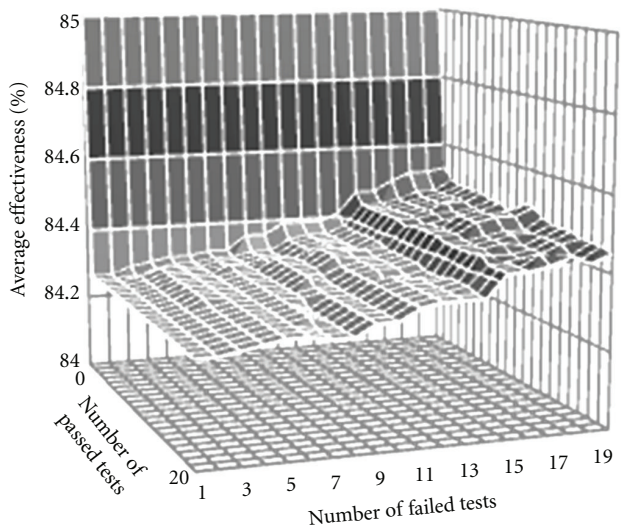


Figure 5: Quantity impact on the Ochiai fault localizer.

approach increases as the quality of the program spectra to support fault localization increases. However, for the most part, the impact does not appear to be significant.

5.2. Quantity Impacts. This section investigates the quantity impact of program spectra using our approach. We evaluate the effectiveness by varying the number of passed and failed test runs involved in fault localization across the Siemens benchmark set. The spectra set contains a large number of runs ranging from 1052 to 5542 but it has a relatively small number of failed runs ranging from one to 518. Therefore, it is not possible to have all versions representing all possible cases of passed and failed runs. In our experiments, since interesting effects tend to appear in a small number of runs [10], we have focused on the range of one to 20 of passed and failed runs to maximize the utilization of the number of versions with both types of runs. Consequently, we used 86 of a total of 119 versions in the spectra set.

Figure 4 shows the effectiveness obtained from the proposed approach for the number of passed and failed tests varying from one to 20. Each entry represents an average of effectiveness (in percentages) across 86 selected versions. As shown in Figure 4, we can see that as the number of failed test runs increases, the average effectiveness increases. Thus, adding failed runs appears to improve the effectiveness. However, adding passed runs does not seem to change the average effectiveness obtained by our approach. We also found that the average effectiveness stabilized when the number of runs is around 18.

To gain understanding of our approach compared to others in this aspect, we performed a similar experiment with the *Ochiai* fault localizer. Figure 5 shows the average effectiveness obtained in percentages within the same range of 20 passed and failed tests. As shown in Figure 5, adding more failed runs appears to improve the effectiveness of fault localization when using the *Ochiai* technique. On the other hand, adding the number of passed runs can increase or decrease the effectiveness especially when the number of passed tests runs toward 20 (from 16). These results obtained by the *Ochiai* localizer agree with those observed by Abreu et al. in [10]. Compared to the *Ochiai* localizer, our approach appears to be slightly more stable and less influenced by the size of passed runs in the benchmark set.

From our experiments on the quantity impact, it is evident that program spectra with more failed runs can slightly improve the effectiveness of fault localization using the proposed method. However, the spectra set with more passed test runs does not seem to have a great impact on its effectiveness. This is rather remarkable and distinct from the *Ochiai* method.

In general, a unit hit by a large number of passed runs should be less suspicious. Therefore, if such a unit were actually faulty, it would be difficult to detect. As a result, the effectiveness obtained by most spectrum-based fault localizers would be decreased. A similar argument applies when there is a large number of runs that can weaken the degree of similarity between the faulty unit and the errors detected from the tests. However, the total number of runs does not seem to impact the effectiveness of our approach. The reason that the proposed approach is not extremely sensitive to the number of passed runs (and thus, overall number of runs) is attributed to the strength parameters that already compensate the large difference between the number of passed and failed runs. Furthermore, performance of any fault localizer is likely to depend on individual programs and sets of program spectra collected for the analysis. Thus, by adjusting different values of strength, our approach can be easily customized to apply to the available data and enhance the fault localization effectiveness.

6. Related Work

Early automated software fault localization work can be viewed as a part of a software diagnostic system [3]. To locate

known faults, the system exploits knowledge about faults and their associated possible causes obtained from previous experience. For novel faults, the diagnostic system employs inferences based on the software model that contains program structure and functions along with heuristics to search for likely causes (and locations of faulty units). However, modeling the program behavior and functions is a major bottleneck in applying this technique in practice.

Agrawal et al. [23] introduced the fault localization approach based on *execution slices*, each of which refers to the set of a program's basic blocks executed by a test input. Weiser [4] introduced the concept of a program slice as an abstract used by programmers in locating bugs. Agrawal et al.'s approach assumes that the fault resides in the slice of a failed test and not in the slice of a successful test. By restricting the attention to the statements in the failed slice that do not appear in the successful slice, called *dice*, the fault is likely to be found. However, the technique does not fully exploit program spectra as it only uses a *single* failed (passed) test case's execution slice to locate faults [18]. Consequently, program statements that are executed by one passed test case and a different number of failed test cases would have an equal likelihood to be faulty if both were executed by the failed test case for the considered dice. Our approach, however, makes use of all the test cases in the test suite, both passed or failed, to estimate fault locations.

Recent spectrum-based fault localization techniques include *nearest neighbor queries* [5], *Tarantula* [13], *Jaccard* [12], and *Ochiai* [14]. These approaches use some measure of similarity to compare the units executed in a failed test to those of a passed test. The unit that results in the most similar behaviors to errors in all the test runs is likely to be faulty. Other fault localization techniques employ statistical [6, 7] and neural net models [24] to analyze source codes and program spectra, respectively. However, the performance of the former heavily relies on the quality of data, whereas the latter suffers from classic limitations such as local minima. Unlike these approaches, our approach is not similarity based or statistical based, but estimates the likelihoods or beliefs of units being faulty. The computation is based on a theory that is widely used for reasoning under uncertainty.

7. Discussion and Conclusions

Our approach is fundamentally different from existing methods. Although the results of the proposed approach on the benchmark data are marginally better than state-of-the-art approaches, it should be emphasized that the proposed approach provides useful aspects for software testing in practice. Its ability to do online computation to locate faults as soon as each test run is executed is particularly novel in that testing of large program can be performed more efficiently since it does not have to wait for all testing program spectral data to be completed. The intermediate results obtained can also influence the configuration or design of subsequent tests. This is useful for real-time systems where system configurations tend to be highly adaptable and it is hard to predict the program behaviors.

In summary, this paper presents a spectrum-based approach to fault localization using the Dempster-Shaffer theory of evidence. Other than its competitive performance to state-of-the-art techniques, our approach has several unique benefits. First, it is theoretically grounded and therefore it has a solid foundation for handling uncertainty in fault location. Second, it supports an online computation that allows the prediction of fault locations to be updated immediately as the execution of each test case is completed without having to wait for completion of a large enough set of program spectrum to be statistically valid. Such computation adapts well to real-time systems. Finally, the approach can be extended easily by adding new mass functions to represent additional evidence for use in the probability assignment of faulty hypotheses.

Future work includes more experiments to gain understanding of the characteristics of the proposed approach, for example, what types of program spectra on which the proposed approach would perform best or perform significantly better than the other three approaches, and how we can extend the approach so that it can deal with software with multiple faults. More experiments can be performed to see if different types of software units impact the results. These are among our ongoing and future research.

Acknowledgments

Thanks are due to Adam Jordan for his help on the experiments and to Phongphun Kijsanayothin for his helpful discussion and comments on earlier versions of this paper. The author would also like to thank the reviewers whose comments have helped improve the quality of the paper.

References

[1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] I. Vessey, "Expertise in debugging computer programs: an analysis of the content of verbal protocols," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 16, no. 5, pp. 621–637, 1986.

[3] R. Sedlmeyer, W. B. Thompson, and P. E. Johnson, "Diagnostic reasoning in software fault localization," in *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, vol. 1, pp. 29–31, Morgan Kaufmann, San Francisco, Calif, USA, 1983.

[4] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[5] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30–39, Providence, RI, USA, 2003.

[6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15–26, Association for Computing Machinery, New York, NY, USA, 2005.

[7] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[8] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proceedings of the 6th European Software Engineering Conference*, pp. 432–449, Springer, New York, NY, USA, 1997.

[9] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46, IEEE Computer Society Press, Riverside, Calif, USA, 2006.

[10] R. Abreu, P. Zoeteweij, and A. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques (MUTATION '07)*, pp. 89–98, IEEE Computer Society Press, Washington, DC, USA, 2007.

[11] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 76–87, IEEE Computer Society Press, Washington, DC, USA, 2009.

[12] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN '02)*, pp. 595–604, Berkeley, Calif, USA, 2002.

[13] J. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, pp. 467–477, Association for Computing Machinery, New York, NY, USA, 2002.

[14] A. Meyer, A. Garcia, A. Souza, and C. Souza Jr., "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L)," *Genetics and Molecular Biology*, vol. 27, no. 1, pp. 83–91, 2004.

[15] S. Nessa, M. Abedin, E. W. Wong, L. Khan, and Y. Qi, "Software fault localization using N-gram analysis," in *Proceedings of the 3rd International Conference on Wireless Algorithms, Systems, and Applications*, pp. 548–559, Springer, Berlin, Germany, 2008.

[16] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, vol. 3568 of *Lecture Notes in Computer Science*, pp. 528–550, Springer, Glasgow, UK, 2005.

[17] G. Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ, USA, 1976.

[18] J. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 273–282, Association for Computing Machinery, New York, NY, USA, 2005.

[19] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bulletin de la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[20] A. Ochiai, "Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions," *Bulletin of the Japanese Society for the Science of Fish*, vol. 22, pp. 526–530, 1957.

[21] J. Pearl, *Causality: Models, Reasoning, and Inference*, Cambridge University Press, 2000.

[22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and control flow-based test adequacy criteria," in *Proceedings of the 6th International Conference on Software Engineering*, pp. 191–200, IEEE Computer Society Press, Sorrento , Italy, 1994.

[23] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pp. 143–151, Toulouse , France, 1995.

[24] W. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 4, pp. 573–597, 2009.