Micro process analysis of maintenance effort: an open source software case study using metrics based on program slicing

Raula Gaikovina Kula*, †, Kyohei Fushida, Norihiro Yoshida and Hajimu Iida

Graduate School of Information Science, Nara Institute of Science and Technology, Japan

SUMMARY

For any software project, most experts regard the maintenance phase as the most effort and cost intensive of all phases in the software development life cycle. This is due to the high maintenance effort, time, and resources needed to effectively address issues during software maintenance (maintenance activities). Mismanagement of these efforts can lead to the degradation of software maintainability. Understanding the assessment of the related software processes can help sustain or improve maintainability during these maintenance activities. Recent studies have shown that current software process assessments are expensive, generic, and complex, especially for smaller organizations. In this paper, we investigate an alternative software process assessment approach performed by analyzing fine-grained processes (micro processes) of maintenance activities. This approach assesses maintenance efforts based on micro processes in relation to their impact on source code. The approach derives maintenance effort from the complexity and duration of micro processes and uses proposed metrics based on program slicing to measure change impact. In this paper, we investigate an alternative software process assessment approach by analysing fine-grained processes (micro processes) of maintenance activities. At statistically significant levels, results suggest that the level of the maintenance efforts correlates with its impact on source code.

Received 1 August 2011; Revised 20 July 2012; Accepted 1 August 2012

KEY WORDS: software processes; mining software repositories; program slicing; software metrics

1. INTRODUCTION

The assessment and improvement of software processes are rapidly gaining attention as an important activity in software development, with benefits seen in terms of cost-efficiency and improved business value. More specifically, these benefits include improved productivity of development, and early defect detection and maintenance, which all account for a faster time to market [24].

Several studies, however, have pointed out some issues relating to current software process quality assessment methodologies such as Capability Maturity Model Integration (CMMI) [31] and international standards (i.e., ISO 9000) [43]. Most of these issues relate to the high costs of assessment and implementation. [44]. Hall and Baddoo both pointed out that these methodologies can be rather tedious, generic, and complex as they assess all phases of the development life cycle [20, 6]. In addition, studies have shown assessments to be higher management support, training, awareness, allocation of resources, staff involvement, and experience of staff as de-motivators of software process assessments [42, 39]. Putting together all these factors, current software process assessment models pose difficulties in usage, especially for smaller software development organizations. Furthermore, process

*Correspondence to: Raula Gaikovina Kula, Software Design Laboratory, Graduate School of Information Science, NAIST, 8916–5 Takayama, Ikoma, Nara 630–0101, JAPAN.
†E-mail: raula-k@is.naist.jp

Copyright © 2012 John Wiley & Sons, Ltd.
assessment includes other aspects such as process effort, human and infrastructure management, and
the achievement of process objectives.

The purpose of this research is to investigate a simpler tailored approach for assessment of software
process quality. Because of the complications of assessments of software quality across the entire
software development life cycle, this study investigates a much easier assessment approach, focusing
primarily on the maintainability aspect of software product quality. Studies have shown that the
maintenance phase consumes a substantial amount of time and effort as compared with the other
software phases during the software development life cycle [5, 48].

The ISO/IEC 15504 Software Process Improvement and Capability Determination (SPICE)
assessment model [12] does address some of the mentioned issues of CMMI. However, in
contrast to our approach, SPICE and CMMI both have a higher level of abstraction. As well
as being cost-efficient, our approach provides results at a more fine-grained level than SPICE
and CMMI, specifically for the quantitative assessment of the effects of maintenance efforts on
source code maintainability.

Another aspect that is not present, purposely, in both SPICE and CMMI is detailed technical
methods for process assessments. Conversely, our approach uses data-mining techniques and
quantitative evaluations of the source code to assess processes, thus exploring the relationship
between process and product.

Our assessment introduces a novel approach to assess software process at the fine-grained level
(referred to as micro processes) during the maintenance phase. By mining software repositories,
sufficient data are extracted to reconstruct the micro processes related to the maintenance activities. We
exploit these micro processes to express the maintenance effort, defined as the time and resources spent
on the resolution of issues relating to defects, undesirable behavior, and enhancements, as well as on
the general maintenance of software.

We investigated the maintenance effort in relation to its change impact on the source code. The
maintenance effort, measured per issue, is expressed in terms of complexity and duration of the micro
processes. To evaluate the change impact of the maintenance effort, four code metrics based on a
program slicing technique were proposed.

In this study, we believe that maintenance effort affects the quality of the source code, which pertains to
its maintainability. We suspect that for each project, we can determine the high maintenance efforts and
relate them to having high impact on the code. To test our assumptions, we constructed the following
research questions:

- RQ1. Are we able to determine high maintenance effort?
- RQ2. How much impact does a high maintenance effort have on the source code?
- RQ3. Is there a correlation between maintenance effort and its impact to source code?

To evaluate our research questions and proposed approach, we performed a case study of three
open source projects. Mining data from a source code repository and issue tracking system
(ITS), we were able to measure the maintenance effort (based on micro process analysis) and
change impact (based on the proposed metrics). Because most micro processes follow a tailored
workflow, we developed thresholds to determine project-specific high maintenance efforts. Results
indicated that maintenance effort had statistically significant correlations with the impact on the
source code.

The paper makes the following contributions:

New approach to the assessment of software processes. At a fine-grained level, our approach focuses
on the maintainability effort and its relation to the source code.
Quantitative expression of maintenance effort in terms of micro processes. Using project-specific workflows
to resolve issues, we used the complexity and duration of the micro processes to quantitatively calculate
maintenance effort.
Proposed program slicing-based metrics to measure change impact. We introduced four
program slicing-based metrics at a more precise function level, measuring the complexity
(based on McCabe’s cyclomatic complexity (CC)) and size (the number of functions sliced)
of an issue.
Determine high maintenance efforts. We were able to identify high maintenance efforts based on the distribution of all maintenance efforts within the software project. We discovered that high maintenance efforts usually have high change impact on the source code.

Statistically significant correlations between the maintenance efforts and its change impact on source code. Using the standard t-test, our proposed metrics improved its p-values over most of the corresponding non program slicing metrics.

Application to different projects. Our approach yielded similar results across projects that differed in source code size, workflow, data management systems, and the handling of issues during the maintenance phase.

Our approach offers an alternative methodology for the assessment of micro processes. On the basis of project-tailored micro process analysis, our approach can be used to help determine where and how maintenance efforts can be reduced (i.e., the proper assignment of resources) and if the affected portions of source code are candidates for maintenance activities such as refactoring, code inspections, and reviews.

Our maintenance effort thresholds enable our approach to be specifically tailored to any project. Because only three projects are used in this study, future replication across a wider range of projects is needed to generalize our approach.

The rest of this paper is organized as follows. Section 2 discusses the related work including the literature on software processes, program slicing, and change impact analysis. Section 3 presents the background, the metrics, and showing the step-by-step of our approach. Section 4 presents the three projects as a case study and the results. Section 5 is a discussion of the results, outlining the applications of the study, revisiting the research questions, and threats to validity. Finally, Section 6 outlines the conclusion and open issues for future work.

2. RELATED WORK

The goal of our research is to provide an alternative approach to the quality assessments of software processes at the micro level. The related literature covers a wide range of fields; however, each has slightly different motivations and approaches. We divided the related works into detection and prediction metrics, program slicing, and change impact analysis. To the best of our knowledge, our objective is novel as we propose techniques to quantitatively assess software processes.

2.1. Software process analysis and assessment

Several related works have attempted to address the shortfalls of current software process assessment models. Yoo [49] suggested a model that combined CMMI and ISO models. Armbrust [4] took a different approach by treating software as manufacturing product lines, creating easier processes, however, making the processes systematic and generic. Unlike these approaches, we use the data mined from software repositories for our assessment.

2.2. Detection and prediction metrics using software repositories

Most work related to mining repositories has had objectives related to detection and prediction of fault proneness in the functions, modules, and features during maintenance [22, 28]. Zimmerman [34] used the extracted information to measure against software patterns. Fisher [14] mined bug reports and version control systems, using visualization techniques to understand features.

Like the maintenance effort in this study, similar work studied the effort spent to fix bugs [46]. Weiss and colleagues estimated the effort to fix an issue based on prior similar issues, applying time as an indicator for effort. Also, work by Kim [29] referred to the time to fix bugs as an important factor. Other related works in this area have proposed heuristic approaches to measure the impact of code changes [50, 47]. As compared with Kim, we introduce the complexity of the micro processes to further express maintenance effort.
Models commonly use product metrics for analysis. However, Kamei et al. [27] proved that process metrics outperform product metrics for bug prediction models. Our work uses combination of process (complexity of the micro processes) and product metrics (our proposed program slicing based metrics) for our analysis. We explore the process-product relationship, similar to older models such as the PROduct Focused improvement of Embedded Software processes (PROFES) improvement methodology [7].

2.3. Program slicing

Many of the metrics widely used in the field of program slicing are related to the evolution of code [21, 36]; among those, many are cohesion and coupling-based approaches. Similar research has used program slicing metrics to classify bugs using these metrics [40, 32]. Instead of the standard slicing metrics, our proposed metrics include a count of the functions affected to measure size and the CC of the code to measure complexity within the slices.

Work by Nagappan’s group is very similar to ours but with a different objective. They evaluated Windows Server 2003 and assessed the relationships between the software dependencies and churn measures with the objective of finding efficient predictors of post-release defects [38]. Our work has the objective of assessing software processes.

2.4. Change impact analysis

Program slicing is well known in the field of change impact analysis. Gallager [16] illustrated its usefulness as it assisted program comprehension, more specifically guiding developers to determine which code components were not related to a software change. Similar to this, differential symbolic execution characterized the effects of a set of program changes in terms of behavioral program differences [41]. There also has been research to predict if a software change is clean or buggy [28]. Canfora applied program slicing as well to indexing changes [10]. German, Hassan, and Robles explored the use of change impact graphs to visualize the impact of code changes to investigate real defects [19]. Hassan predicted faults using the complexity of code changes [23]. Much like our research, using information theory, Hassan deduced that code changes with complex micro processes negatively affect a program. Unlike these other efforts, we also have a different motivation and objective.

3. BACKGROUND AND APPROACH

Our motivation is to investigate a much simpler tailored assessment approach for software processes assessment related to the maintainability of source code. We particularly focus on the micro processes of the maintenance effort in relation to its impact on the source code.

Our approach involves mining repositories to analyze maintenance effort in relation to its change impact on the source code, using our proposed program slicing based metrics. Figure 1 illustrates the three-step overview of the approach taken in this research. These are (i) extracting the micro processes to express the maintenance effort, (ii) calculating proposed metrics for the impact of the maintenance effort on the source code, and (iii) determining and grouping high maintenance efforts to evaluate their impact on the source code. Because our research utilizes techniques used in both the fields of mining software repositories and program slicing, we first present a brief definition of the terminologies presented in this paper. Following this, we present our proposed approach in full detail.

3.1. Mining software repositories for micro process analysis

In our approach, we mine software artifacts (referred to as software repositories) to gather sufficient information for the reconstruction of micro processes. These processes are used to measure maintenance effort. In this paper, software repositories refer to two main management systems: the source code management system (SCM) to manage source code and the ITS to track issues related
to a software project. In recent times, the two most commonly used SCM system types have been subversion (SVN) and concurrent versioning system. In our study, we focus on SCM. An SCM stores multiple versions of the source code of a project, recording all code changes with timestamps. Issues are stored in an ITS, with each issue corresponding to a set of code changes related to resolving that issue. Each code change is stored as a revision in the SCM. As is explained in further detail in the next section, we can use program slicing techniques to analyze the change impact of the code changes to the source code.

As mentioned throughout the paper, maintenance effort refers to the time and resources used to address the demands in resolving defects, undesirable behavior, enhancements, and for the general maintenance of the software. These demands are referred to as issues. The ITS manages the processing of all issues. It also records all interactions between the developers and reporters of an issue. The fine-grained processes taken to resolve these issues are known as micro processes. A micro process begins with a user reporting the issue. Next, the issue is confirmed and assigned to a developer for bug fixing (code changes, testing, and verification). Finally, after bug fixing, the issue is resolved and closed. Each project has tailored their micro processes to suit their organizations. This is defined as the workflow of resolving an issue. In this study, the workflow is derived from how an issue changes state. A state change is regarded as a fragment of the micro processes used to resolve an issue. In addition to the workflow, the time to fix an issue is important. Using both the state changes and time-to-fix properties of an issue, we can measure the maintenance effort. Much of our previous work Morisaki [37], Kula [30] was based on the analysis of micro processes. However, in this paper, we extend this work; our metrics are at the function level.

With recent improvements of historical information software artifacts, software repositories have rapidly become a major source of information on software projects. Like this work, data mining of these repositories has led to extraction of fine-grained processes [18]. However, mining software repositories still has its flaws. Howison and Crowston [26] illustrated these perils, especially with open source software (OSS) projects, showing the importance of the quality and reliability of good change logs. In this work, we preprocess our data to ensure that the data collected is of high quality. This is explained in detail later in this section.
3.2. Program slicing in change impact analysis

First proposed by Weiser [45], program slicing refers to a subset of a program's behavior, reducing a program to its minimal form that still produces that behavior. On the basis of a slicing criterion, program slicing can isolate interprocedural dependencies at the module, file, or function level. Program slicing is a well-known technique applied in the field of change impact analysis [16]. In this study, we propose metrics to measure attributes of the slices, thereby measuring the impact of a change in code. The slicing criterion is performed at the function level, so assume improved slicing precision as compared with file level slicing.

To evaluate the slices, we applied two simple metrics to measure complexity and size. McCabe's CC is a well-known software metric that indicates the logical complexity of a program [35]. To compute the CC, the program control flow is examined to measure the number of independent paths in the source code. A metric usually used to measure size is the lines of code (LoC). However, because our slicing is at the function level, we found it more appropriate to express size by counting the number of functions per slice as the function count (FC). First developed by Albrecht [1], several process standards based on functions such as the ISO/IEC 20968 and ISO/IEC 14143 have used functions to determine software size [17]. Using the slices, we propose metrics that measure CC and FC to evaluate the complexity of code based on its impact at the function level.

3.3. Proposed approach

Given this background, we proposed a three-step approach as shown in Figure 1. The first step is the extraction of the micro processes from both the SCM and the ITS. The extraction provides sufficient data to reconstruct the micro processes of the maintenance effort. In the second step, we introduce the proposed program slicing metrics to identify impact in relation to the maintenance effort. In the third step, we propose grouping parameters to evaluate each maintenance effort. On the basis of the micro processes, we determine and group the high maintenance efforts. These steps are explained in the succeeding text.

1. Extraction of micro processes. Our goal of mining the software repositories was to extract sufficient information for each issue to be able to reconstruct the related micro processes and measure the maintenance for an issue. The micro process involves all the processes from when an issue is first opened until it is closed.

Figure 2 illustrates the base micro process model used in this paper. The model consists of three steps. In the first step, the issue is detected and reported into the system as a new state. In the second step, the issue undergoes various states until it is resolved. For instance, most issues usually state change to confirmed and/or accepted before developers begin committing code changes. The second step is concluded once all code changes needed to resolve the issue are committed to the source code. Moving into the third step, the issue changes state to closed, thus marking the issue as being resolved. There are some cases when the solution is not sufficient, so the issue changes
state to reopened, reverting the micro process to the second step. Figure 3 shows screenshots of the software repositories of an issue that has been resolved. The example shows a typical change log, with an issue stored using the TRAC ITS and a source code revision retrieved from an SVN system. To reconstruct this micro process, the highlighted information needs to be extracted from the software repositories. In addition, Table I contains a description of the data extracted that is needed for our approach. To extract data, we specifically designed our tool to extract data from the TRAC ITS and SVN systems.

Figure 3. Example illustrating issue 3780 and corresponding rev. 2710. This shows the data needed to reconstruct the micro process (issue report) and related code change impact (source code).

Table I. Information extracted from the software repositories.

<table>
<thead>
<tr>
<th>Software repository</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue management system</td>
<td>Issue ID (bug ID)</td>
<td>Identification of the issue</td>
</tr>
<tr>
<td></td>
<td>Date open</td>
<td>Date when the issues was reported into the system</td>
</tr>
<tr>
<td></td>
<td>Date closed</td>
<td>Date when the issues was resolved</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>Transition of states of the issue (state changes)</td>
</tr>
<tr>
<td>Source code repository</td>
<td>Revision ID</td>
<td>Identification to track changes made to source code</td>
</tr>
<tr>
<td></td>
<td>Issue ID (bug ID)</td>
<td>Reference linking issue to the code change</td>
</tr>
<tr>
<td></td>
<td>Edit date</td>
<td>Refers to the date when the latest code change was performed</td>
</tr>
<tr>
<td></td>
<td>Edited functions</td>
<td>Function(s) where code was edited</td>
</tr>
</tbody>
</table>

(Compared with previous revision)
As seen in Figure 3, the extraction method is based on the identification of the linkage between the issue and the revision in which the code changes were committed. Our extraction method is based on the two main approaches used in the field. The two well-known methods for extraction of bug-fix data are by Fisher et al. [15] and Chen et al. [11]. The first involves searching the change logs for keywords such as bug or fix to extract the bug data, whereas the other manually compares the correctness of change logs. In this research, we applied a combination of both methods. Additionally, we chose a project that had been studied previously, which was known to have high quality change logs. Our approach searches for code changes based on keywords such as bug or fix as well as linkages (i.e., referred to as either bug ID or issue ID) from within change logs of the source code repository. As seen in Figure 3, extraction of the functions is carried out by comparing the changes to the previous revision. We implemented this methodology using our data mining tool. This tool combined simple web scripts to download, parse, and extract the required data from online open source repositories into a relational database for analysis.

2. Proposed metrics to measure change impact. As described earlier, to measure the impact of the maintenance effort, the study used metrics based on the behavioral properties of the program. We applied two metrics: (i) McCabe’s CC to measure the complexity of the changes and (ii) FC to measure the size of the changes. We selected these two parameters as they are two widely accepted and relatively simple analytical metrics [13]. In addition, we introduced non program slicing counterparts Eqs (1) and (4) to evaluate the effectiveness of using the program slicing technique.

Figure 4 illustrates how our approach applied program slicing. For each issue, we assume that each file edited during a code change is stored in the SCM as a revision. Therefore, for each affected revision, we identify all the functions modified during the code change. Refer to Figure 3 for an example of the code changes during an issue resolution. We refer to these functions as edited functions. For every edited function, we then calculate the backward slices and the forward slices. The backward slice is the set of functions that the edited function depends on. The forward slice refers to the set of functions that depend on the edited function. Program slicing ensures that only the source code related to the edited functions is analyzed.

Formally, we define code changes as a sequence of revisions $\mathcal{R} = \{r_1, r_2, \ldots\}$. For each revision $r$, we define $F_{E,r} = \{f_1, f_2, \ldots\}$ as a set of edited functions in revision $r$. Given the edited function $f$, edited in revision $r$, we define the $S_B(f)$ and $S_F(f)$ as backward and forward slices of $f$. Note that we can assume that $S_B(f)$ is a set of functions that affect $f$, and $S_F(f)$ is a set of functions affected by $f$.

![Figure 4. Example of backward slice and forward slice for a program slice of an edited function.](image-url)
To define the following metrics, we introduce $S_{B,r}$ as a backward slice and $S_{F,r}$ as a forward slice in revision $r$. Trivially, $S_{B,r} = \cup_{f \in \mathbb{F}_r} S_B(f)$ and $S_{F,r} = \cup_{f \in \mathbb{F}_r} S_F(f)$.

A) CC-based metrics Given a function $f$, we define the function $C(f)$ that gets the CC of $f$. These proposed metrics shown in Eqs (1)–(3) are used to measure the total complexity of all functions in the slice at a certain revision.

- **EditedFunctionCC.** This is the summation of the CC for functions edited during a code change, that is,
  \[ \sum_{f \in \mathbb{F}_r} C(f) \]  
  (1)

- **Rationale:** This is the non program slicing metric for comparison against both Eqs (2) and (3).

- **BackwardSliceFunctionCC.** This is the summation of the CC for each function in $S_{B,r}$, that is,
  \[ \sum_{f \in S_{B,r}} C(f) \]  
  (2)

- **Rationale:** This metric computes the combined total CC of the functions that $\mathbb{F}_r$ is dependent on.

- **ForwardSliceFunctionCC.** This is the summation of the CC for each function in $S_{F,r}$, that is,
  \[ \sum_{f \in S_{F,r}} C(f) \]  
  (3)

- **Rationale:** This metric computes combined total CC of the functions depending on functions in $\mathbb{F}_r$.

B) FC-based metrics To measure the size of the code change, we introduce three metrics shown in Eqs (4)–(6) based on the number of functions affected by the code change in revision $r$.

- **EditedFC.** The number of functions edited during a code change, that is,
  \[ |\mathbb{F}_r| \]  
  (4)

- **Rationale:** This is the non program slicing metric for comparison against both Eqs (5) and (6).

- **BackwardSliceFC.** The number of functions in $S_{B,r}$, that is,
  \[ |S_{B,r}| \]  
  (5)

- **Rationale:** This metric computes the number of functions that $\mathbb{F}_r$ is dependent on.

- **ForwardSliceFC.** The number of functions in $S_{F,r}$, that is,
  \[ |S_{F,r}| \]  
  (6)

- **Rationale:** This metric computes the number of functions depending on functions in $\mathbb{F}_r$.

To generate our metrics, the software analysis tool by GrammaTech called CodeSurfer [3], a sophisticated and widely used tool for interprocedural slicing, was used [40, 2, 36, 9]. Customized scripts within CodeSurfer were used to calculate the CC and FC within the slices.

3. **Grouping code changes.** Grouping of issues based on their maintenance effort involves two steps: first is preprocessing to improve the quality of the data (known as cleansing), and second is calculating the effort threshold to determine and group high maintenance efforts. Once the threshold is defined, evaluations on the two groupings to test correlations are performed.
Preprocessing procedures (cleansing of the data). To improve the quality of our results, we applied filters to the datasets in the experiment. This process removed data that could negatively affect the results and gave confidence in the quality of the data collected. From our collected code changes, we removed issues having the following criteria:

- **Open issues**: Code changes related to issues not yet resolved (i.e., not closed status)
- **Non code related changes**: Code changes related to documentation or images (changelog.txt, pic.jpg... etc)
- **Build-related issues**: Code changes related to compiling or build errors
- **Revisions with no linkages**: Code changes with missing issue tracking information (i.e., no Bug ID references in the revision change log and vice versa)

Preprocessing was a two-step process: (i) our tool parsed the data extracted from the software repositories to remove code changes meeting the criteria mentioned earlier. To remove false positives, we then (ii) manually checked all remaining code changes against these filters to ensure high quality results. Using the aforementioned criteria, all the cleansing of the data could be performed automatically; however, because of time constraints and the current limitations of our tool, the current implementation provides a semi automatic verification of the datasets.

**Grouping using effort thresholds.** Referring back to Figure 2, we propose that the complexity of the micro processes for each issue contributes to its maintenance effort. This implies that more complex issues usually have more state changes or takes more time to resolve. Each micro process fragment is defined as the state change for an issue related to a set of code changes. For instance, an issue has a new state when first reported and finally ends up with a closed state when resolved. Duration, which is measured in the number of days, is defined as the time from when the code change is first requested in the issue change log (has a new state), until the time when the issue is closed (closed state).

Maintenance effort depends on both the complexity of the micro process as well as the duration until the resolution of the issue. As seen in Figure 2, the complexity of the micro process can be measured by the combination of either the number of state changes and/or the duration of the issue before it was resolved. It seems likely that there is a mutual dependency between complexity and duration. For instance, we are able to differentiate complex issues that were quickly resolved. We suggest that such issues could be prone to be reopened.

We formulated effort thresholds, designed to determine maintenance efforts that require more than the normal state changes and duration. Formally, given an issue i, S(i) is the number of state changes, and D(i) is the duration. We introduced two effort thresholds: $\mathcal{S}$ to represent the state change threshold and $\mathcal{D}$ for the duration threshold. Analysis of the distribution of S(i) and D(i) allows us to identify the $\mathcal{S}$ and $\mathcal{D}$ thresholds specific for any project. We propose to set the effort thresholds based on the outliers of these distributions.

On the basis of the thresholds, we divided the issues into two groupings, those that require more than the normal maintenance effort (higher effort needed to resolve) and the rest of the issues (requiring normal or less effort to resolve). **High maintenance efforts** refer to issues identified as being higher/above the effort thresholds. **Normal maintenance efforts** refer to issues identified as being less than/below the effort thresholds.

Formally, given the maintenance effort $\mathcal{M}E$ of any issue, we define high maintenance efforts as Eq. (7) and normal maintenance effort as Eq. (8).

$$\mathcal{M}E_{\text{High}} = \{i | i \in \mathcal{M}E \land (S(i) \geq \mathcal{S} \lor D(i) \geq \mathcal{D})\}$$  \hspace{1cm} (7)

$$\mathcal{M}E_{\text{Normal}} = \{i | i \in \mathcal{M}E \land (S(i) < \mathcal{S} \land D(i) < \mathcal{D})\}$$  \hspace{1cm} (8)
4. CASE STUDY AND RESULTS

4.1. Experiment setup

To test our approach, we chose three OSS projects for analysis. The chosen projects are Filezilla, which is a file transfer protocol application, WxWidgets, a C++ library that lets developers create GUI applications for major OS as well as mobile OS and embedded GTK+ architectures, and Lighttpd, a lightweight open-source web server. At the time when the study was conducted, the latest release of Filezilla (up to Ver. 3.3.1) had 210,629 LoC, WxWidgets, the largest (up to Ver. 2.9.0), had 409,148 LoC, whereas Lighttpd, the smallest (up to Version 1.5.0), had 40,712 LoC.

For the experiment, we randomly selected two versions from each project as datasets for analysis. These datasets, as shown in Table II, are the basis on which the program slicing-based metrics were calculated. Then, using the code changes from issue reports from the ITS, we calculated the proposed metrics of the code related to these versions. The time window shows the duration period in which the issues were resolved. Note that Lighttpd generally has a longer time window, especially Lighttpd Dataset 2 with almost 4 years to resolve an issue.

Both Filezilla and WxWidgets use the TRAC Management System, whereas Lighttpd uses Redmine as their ITS. Figure 5 shows the workflow of each project. For Filezilla and WxWidgets, the workflows were constructed based on the workflow guidelines available at each TRAC repository. As shown in Figure 5, WxWidgets has a more complicated workflow in comparison with Filezilla. Offering more choices for the developers, the WxWidgets workflow offers more guidance. However, its implementation is questionable. On the other hand, Filezilla has a more simplistic approach. Therefore, in the case of Filezilla, the duration of an issue rather than the complexity of the micro process could be a better indicator of maintenance effort. Unlike the other two, Lighttpd had no documented workflow. Instead, as shown in Figure 2, it had a set of selectable states. Using the extracted issues, we were able to construct a simple workflow.

For WxWidgets, we extracted data from the WxWidgets TRAC system (ITS) and WxWidgets SCM inspecting 64,005 revision changes. For Filezilla, we extracted 3611 revision changes from the Filezilla TRAC (ITS) and Filezilla SCM. In the case of Lighttpd, its Redmine-based system holds both the ITS and SCM for the project, giving us access to 2815 revision changes.

Because our extraction is automated, we included preprocessing to ensure high quality representations of each project, for instance, by removing duplicates. This can be seen as in the code change sets column of Table II, where the remaining code change sets after preprocessing are shown in the brackets. Although, this preprocessing further reduced the datasets, it also gave us greater confidence in our data by removing false positives.

As shown in Table II, the number of code change sets extracted is relatively low compared with the inspected revision changes. We attribute these discrepancies to missing linkages from the ITS to the revision change log by our automated tool. This is well known as one of the perils of mining particularly OSS projects, as is also explained by Howison and Crowston [26]. Another reason for missing code changes may be that the revision changes are not related to our dataset versions of code, as the functions do not yet exist or have been modified to a point that it is no longer recognizable by our parsing tool.

4.2. Determining effort thresholds

To identify high maintenance efforts, we determined effort thresholds by analyzing the distribution of the state changes and the duration of the datasets. Because Figure 6 suggests that these distributions follow a standard distribution, on the basis of Tukey’s outlier filter [25], we used the formula

\[ \text{Tukey’s outlier filter} \]

\[ Q1 - 1.5 \times (Q3 - Q1) \]

\[ Q3 + 1.5 \times (Q3 - Q1) \]

where \( Q1 \) and \( Q3 \) are the first and third quartiles, respectively.
Q3 + 1.5 × IQR to determine the effort thresholds as the outliers for $\mathcal{S}$ and $\mathcal{D}$, where Q3 is upper quartile of state changes and duration and IQR being the interquartile range.

Using the grouping thresholds Eqs (7) and (8), we are able to separate the issues into a set of high and normal issues. Table III shows the size of each set, expressed as the cardinality as well as the thresholds for $\mathcal{S}$ and $\mathcal{D}$ for the three projects. As expected, normal maintenance effort is the larger of the two sets. It is interesting to note that there are few high maintenance issues for the Lighttpd project. This could be caused by the relatively longer durations to resolve issues and the lack of state changes of issues as noticed in Figure 6.

On the basis of the graphs shown in Figure 6 and Table III, the following observations can be made about the duration and the state changes.

- The density distribution (curve on the graphs) suggests that both parameters (state changes and duration) are standard distributions, justifying our methodology for determining effort thresholds.
- As seen in the box plots for the state change distributions, the state change is constant for all datasets of the same project. This suggests that state change could be constant for each project.
- As shown in the duration distribution for each project, the duration of the maintenance effort varies between projects and even between datasets within a project. This is also evident for $\mathcal{D}$ of the WxWidgets datasets in Table III.

To further understand and validate the state changes, we took a closer look at the state changes lying within the effort thresholds for each project. For both Filezilla and WxWidgets, we had a closer look as shown in the box plots of Figure 6 where Filezilla has 1–3 state changes and WxWidgets 1–2. To validate that the changing states did represent the workflow, we took a closer look at the actual state changes. Figure 7 shows the number of state changes as it represents the workflow for each project. In addition, most state changes are from new to closed, suggesting a simple state change. However, it is interesting that in WxWidgets, there was some relatively moderate use of the confirmed and accepted status. Also, in Filezilla, the moreinfo status was relatively commonly used. The analysis

Table II. Software project overview.

<table>
<thead>
<tr>
<th>Project</th>
<th>Code change sets (after preprocessing)</th>
<th>Number of functions</th>
<th>Time window</th>
<th>Release version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filezilla dataset 1</td>
<td>156 (100)</td>
<td>8959</td>
<td>August 07–July 08</td>
<td>3.1</td>
</tr>
<tr>
<td>Filezilla dataset 2</td>
<td>173 (136)</td>
<td>10,488</td>
<td>January 09–December 09</td>
<td>3.3</td>
</tr>
<tr>
<td>WxWidgets dataset 1</td>
<td>358 (304)</td>
<td>17,836</td>
<td>December 06–November 07</td>
<td>2.6.4</td>
</tr>
<tr>
<td>WxWidgets dataset 2</td>
<td>347 (303)</td>
<td>23,203</td>
<td>March 07–February 08</td>
<td>2.8.0</td>
</tr>
<tr>
<td>Lighttpd dataset 1</td>
<td>121 (58)</td>
<td>856</td>
<td>August 07–September 08</td>
<td>1.4.21</td>
</tr>
<tr>
<td>Lighttpd dataset 2</td>
<td>101 (83)</td>
<td>840</td>
<td>March 05–August 10</td>
<td>1.4.28</td>
</tr>
</tbody>
</table>

Figure 5. Workflow for all projects: (a) Filezilla, (b) WxWidgets, and (c) Lighttpd workflow.
Table III. Issue sets (high/normal maintenance effort) and effort thresholds.

<table>
<thead>
<tr>
<th>Project</th>
<th>High effort (set size)</th>
<th>Normal effort (set size)</th>
<th>$S$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filezilla dataset 1</td>
<td>31</td>
<td>69</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>Filezilla dataset 2</td>
<td>30</td>
<td>106</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>WxWidgets dataset 1</td>
<td>96</td>
<td>208</td>
<td>2</td>
<td>190</td>
</tr>
<tr>
<td>WxWidgets dataset 2</td>
<td>99</td>
<td>204</td>
<td>2</td>
<td>283</td>
</tr>
<tr>
<td>Lighttpd dataset 1</td>
<td>5</td>
<td>53</td>
<td>4</td>
<td>1630</td>
</tr>
<tr>
<td>Lighttpd dataset 2</td>
<td>10</td>
<td>73</td>
<td>4</td>
<td>1290</td>
</tr>
</tbody>
</table>

Figure 6. Distribution of the datasets.
confirmed that the issues usually followed the workflow procedures (starting at \textit{new} and ending with \textit{closed}). It is interesting that some issues in the WxWidgets’ project start from \textit{closed} and \textit{assigned} status, causing slight concern in the workflow.

We found that in Lighttpd, the state changes were rarely used with almost 80–90\% of issues extracted with the \textit{new} to \textit{fixed} state. There were some instances of other state changes, as shown in Figure 2. Our results confirmed that most of the issues do not require a complex process for maintenance effort for the projects, as all three projects mostly use only one state change. However, use of the state changes indicates additional maintenance effort, such as reopening, further information needed, or the reassignment of an issue.

4.3. Metrics evaluation

Using the effort thresholds specific for each project, each maintenance effort was determined to be either high or normal. Figure 8 shows the matrix of the maintenance effort, expressed in issues, grouped against each metric. Results suggest that high maintenance effort exhibited higher CC and FC, especially using the backward and forward slice metrics.

To prove statistical significance, the student \textit{t}-test was applied to the groupings. The results shown in Table IV further prove that in all projects, the differences in the groupings were significant and that generally the program slicing-based metrics had improved \textit{p}-values compared with the non program slicing metrics. The program slicing-based metrics were enhanced compared with the \textit{EditedFunction CC} and \textit{EditedFunction FC}. \textit{BackwardSlice CC} and \textit{BackwardSlice FC} had the best results, which outperformed all other metrics.

On the basis of the results of the \textit{t}-test as well as the visual representation shown in Figure 8, we can conclude that issues that required high maintenance effort exhibited higher CC and FC values and that the program slicing metrics enhanced this relationship, even in cases where the non program slicing metrics were not significant.

4.4. Other observations of the micro processes

During the preprocessing and cleansing procedures of the datasets, there were a couple of notable differences between all projects. Apart from the obvious workflow and project size differences, under closer examination, we noticed that all the projects had different methodologies in handling issues.

Lighttpd exhibited very high thresholds, especially for duration. For instance, looking at one of the issues, we realized that most fixes are not applied to the source code until the next main version; therefore, a normal fix could be resolved, but not committed to code until months later, nearing the next release. More complex issues were being carried over into new revisions before resolved. We also noticed that state changes were being used mostly to identify duplicates and invalid issues rather than track issue states.
Figure 8. Matrix showing the comparison of normal and high maintenance efforts against the metrics.

Table IV. p-values for the classic student t-test for statistical significance (p-value less than 0.05).

<table>
<thead>
<tr>
<th>Project</th>
<th>Edited function CC</th>
<th>Backward slice CC</th>
<th>Forward slice CC</th>
<th>Edited function FC</th>
<th>Backward slice FC</th>
<th>Forward slice FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filezilla dataset 1</td>
<td>0.001</td>
<td>0.000</td>
<td>0.003</td>
<td>0.001</td>
<td>0.000</td>
<td>0.004</td>
</tr>
<tr>
<td>Filezilla dataset 2</td>
<td>0.009</td>
<td>0.000</td>
<td>0.039</td>
<td>0.016</td>
<td>0.000</td>
<td>0.051</td>
</tr>
<tr>
<td>WxWidgets dataset 1</td>
<td>0.209</td>
<td>0.008</td>
<td>0.024</td>
<td>0.005</td>
<td>0.017</td>
<td>0.013</td>
</tr>
<tr>
<td>WxWidgets dataset 2</td>
<td>0.111</td>
<td>0.032</td>
<td>0.019</td>
<td>0.135</td>
<td>0.056</td>
<td>0.009</td>
</tr>
<tr>
<td>Lighttpd dataset 1</td>
<td>0.052</td>
<td>0.000</td>
<td>0.128</td>
<td>0.004</td>
<td>0.000</td>
<td>0.006</td>
</tr>
<tr>
<td>Lighttpd dataset 2</td>
<td>0.055</td>
<td>0.000</td>
<td>0.196</td>
<td>0.000</td>
<td>0.000</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Nonsignificant values are in bold.  
CC, cyclomatic complexity; FC, function count.
For both Filezilla and WxWidgets, an issue is usually submitted with a description of the issue that may also include a screenshot and/or error-log attached. However, it was noticed that almost 90% of WxWidgets issues were accompanied with a potential solution as a patch. Therefore, the WxWidgets developer, in addition to verifying that the solution is correct, assess the impact of the solution on the stability of the current system operation and design. Filezilla has a more traditional approach, with the developer creating the solution based only on information provided by the submitter.

These observations illustrate that even if the projects’ organization and manner in which to resolve issues are different, our proposed approach was successfully applied.

5. DISCUSSION

In this section, we evaluate our approach and how the results could be applied. We then revisit our research questions. Finally, we present the threats to the validity of the study.

5.1. Generalizability of our approach

Our research focused on three OSS projects, all very different in terms of tools, size, workflow, and even the micro processes of handing issues as explained in Section 4.3. Yet, our approach and metrics yielded consistent results, suggesting that our approach is feasible across OSS projects. Our project-specific effort threshold method enables us to expand our approach to various projects. Also, our study proved that this approach can be applied to projects ranging up to 400 kLoC.

From the study, we identified two factors that could influence the effectiveness of our approach. The first one is the availability of the data. Smaller projects or newer projects may not have sufficient data for analysis. Also, we assume that more mature projects would have more data for analysis. Furthermore, although we have determined that this approach can be applied to projects up to 400 kLoC, this study did not determine the lower limit needed for reliable results. The second factor is the linkages between the issue management system and its SCM. Detailed documentation and linkages from the SCM and ITS are essential for our approach. It is envisioned that as technologies improve the documentation and their tracking of linkages between SCM and ITS, our approach will become more feasible.

5.2. Applications of the study

As an empirical study, our work has shown that the proposed methodology is able to determine the micro processes requiring high maintenance effort, on the basis of the combination of duration and complexity of the micro processes. There are two major viewpoints where we can utilize these results, from both a source code maintainability perspective as well as the project process management point of view.

5.2.1. Code-based application. Identification of high maintenance efforts as well as the affected code, especially those with relatively high CC and FC characteristics, could be beneficial from a source code maintenance point of view. These code portions could be candidates for source code maintenance activities such as refactoring, code reviews, and code inspections. These activities would improve the quality of the source code, thus ensuring that the complexity of code is kept low. Because there is a correlation between complexity and maintenance effort, these activities have potential to reduce the maintenance efforts. However, it is important to note that we are not suggesting causation. Investigation of a causative link is viewed as future work. In addition, the affected code could be marked as complex, so that a developer can take extra care when modification of these portions of code is required.

5.2.2. Project management-based application. From a project team standpoint, the ideal scenario is to assign the best developers and resources to the code changes that require the most effort. To do this, a team first identifies high maintenance efforts at any point and on a particular version and compares this to maintenance efforts of previous versions. The team then assesses if their current
workflow is still suitable for the project. Through this, the team can be reorganized so that the most experienced developers and resources are assigned to high maintenance effort portions of code.

More specifically, in projects similar to WxWidgets’ method of handling issues (i.e., in which a potential solution accompanies the issue), the developer can already identify which functions will be edited and consequently assess what level of effort (more experienced developer) should be assigned to this issue beforehand. In regard to projects like Filezilla and Lighttpd, there could be an additional verification micro process, in which functions that have potentially high CC and FC can be validated by the more experienced members of the project team.

Our approach identifies issues that require high maintenance effort and also have considerable change impact on source code. However, high effort due to longer durations to resolve issues can be influenced by other factors, including back burners (referring to issues that are of low priority because they do not greatly affect the system) or simply poor documentation creating a delay in assignment or breakdowns in the micro processes. Still, this approach can inform project managers of the current quality of the implemented software processes, giving useful insights to improve the micro processes. Also, analysis of the changes in the effort thresholds over versions, although not in the scope of this study, could prove useful.

5.3. Research questions revisited

Before this study, we constructed three research questions in relation to our assumption that high maintenance effort is related to complex source code. In this section, we revisit each question against the results of the study.

• RQ1. Are we able to determine high maintenance effort? By analyzing micro processes during the maintenance phase, we were able to use an effort threshold to differentiate and group issues that consumed high maintenance effort. Our approach proved that this threshold is project-specific.

• RQ2. How much impact does high maintenance effort have on the source code? Results of the study indicate a correlation between high maintenance effort and code changes with high complexity metrics (proposed FC and CC based metrics).

• RQ3. Is there a correlation between maintenance effort and its impact to source code? Applying the standard student t-test, results proved a significant statistical correlation between maintenance effort and the impact of its changes to the source code. In some cases, our proposed metrics outperformed the non program slicing metrics.

Relative to one of Lehman’s laws of evolution [33], we can logically conclude that higher maintenance effort would most likely be caused by complex code. This may seem trivial and common sense; however, in this study, our novel contribution is that we have quantitatively expressed this established phenomenon.

5.4. Threats to validity

5.4.1. Internal. The major internal threats would be the accuracy of the tool used for extraction and the measurement of the data because the extraction process is automated. To mitigate this, a combination of manual verification and preprocessed data was used to ensure the quality of the data collected. It was noticed that the amount of data extracted is significantly lower than the expected amount of code changes available. This can be attributed to the missing linkages between ITS and SCM as well as the lack of proper documentation.

Another threat could be that the preprocessing of the data has an influence on the results. In a real world scenario, for example, data are usually not preprocessed. We believe, however, that our preprocessing was based on rules that can be applied automatically in the real world, as they are not based on human judgment.

Research by Binkley et al. [8] indicated that as data structures increases, so does the size of the slice, causing accuracy problems. In this study, we did not address this issue, as most large slices (having large FC and/or CC) were identified as having valid complex interprocedural dependencies. Also, we believe our case study datasets are under one million LoC, therefore not considered large scale projects. However, we see this as an important issue, especially when looking at large-scale programs and regard this as future work.
5.4.2. External. The major external threat to validity is the generalization of our approach to different projects, under different software repository systems, and because the micro processes and resolution workflows differ from project to project. Our effort threshold method offers project-specific measurements to take into account these differences. In addition, because these are only three projects, we are uncertain if this is a true representation of an OSS project. Our data are all from OSS projects; hence, we cannot assume that results may be the same for commercial projects. Generalization of our techniques is seen as the most important priority for future work.

In this research, we focused solely on change impacts in source code, as we believe maintenance of software mainly revolves around source code changes. Our definition of maintenance effort also investigated micro processes during maintenance. However, other aspects such as software architecture issues were not taken into account. This could be interesting avenues for future work.

Another threat is that our tool used to extract data was specifically made for the TRAC and Redmine (ITS) as well as SVN SCM systems. We would like to expand to other tracking systems such as Bugzilla ITS and concurrent versioning system SCM systems. We plan to further develop our tool to handle other systems so that we can apply our approach to a wider set of projects.

As mentioned previously, there are not many available projects with sufficient linking information of the software repositories to date. We, however, believe that the tools and technologies to manage documentation of software repositories are steadily improving. This makes our approach more practical in the near future.

6. CONCLUSION AND OPEN ISSUES

Software process assessment and improvement can be a rather complicated and costly exercise that is only suited for larger organizations. Measuring software quality is also complicated and covers a wide range. In this paper, we present a simpler alternative: to focus on the micro processes of maintainability of the source code.

To the best of our knowledge, this work is the first to express maintenance effort in terms of the complexity of the micro process. By using effort thresholds, we were able to determine which micro processes required high maintenance effort. Also, using novel program slicing-based metrics, we were able to measure the impact of these efforts on the source code. We concluded that there is a statistically significant relationship between maintenance effort and its change impact on the source code.

Although the results are promising, there are still outstanding issues for future work, including the following:

- **Generalization of our approach.** Replication of the study with more OSS projects is needed.
- **Explore the correlation between maintenance effort and impact on source code further.** Currently, we have identified a correlation but whether this is causative is another avenue for research.
- **Explore the change impact to handle large-scale systems.** Our program slicing-based metrics outperformed the non program slicing metrics; however, we need to address the issue what to do when program slices become too large. We plan to study strategies to break down these structures so that program slicing is more manageable.
- **Study the effort threshold behavior over a project’s lifetime.** The change in effort thresholds over different releases of a project could potentially measure the state of the maintenance effort during the duration of the project.
- **Assessment model for the assessment of the maintainability of software.** Although we are only in the early stages of validating the generalizability of our approach, the final goal would be to create an assessment framework and prediction model for this assessment.

In this study, we explore the relationship between process and product. An interesting caveat is that as model-driven software development approaches evolve and the automation in software evolves along, model elements instead of the source code will be automated. In such a scenario, the relationship between the process and product will be a promising research avenue.
MICRO PROCESS ANALYSIS OF MAINTENANCE EFFORT

As data management processes, tools, and techniques improve, we envision this line of research to prosper. We see this study as a step towards a viable quantitative alternative for software process assessment of the maintainability of software.

ACKNOWLEDGEMENTS

First and foremost, the authors would like to thank the anonymous reviewers for their valuable insights, guidance towards this paper. Special thanks also to Shinji Kawaguchi of JAXA for his contribution in the Micro Process Analysis research and to Mike Barker of NAIST for his editorial efforts. This work is also supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (No.22500027).

REFERENCES


**AUTHORS’ BIOGRAPHIES:**

**Raula Gaikovina Kula** is currently a PhD candidate at Nara Institute of Science and Technology. He obtained a BSc degree in Information Technology with Advanced Studies from Griffith University, Australia, in 2004. In 2010, he obtained his MS degree from Nara Institute of Science and Technology, Japan. He is a member of the IEEE Computer Society.

**Kyohei Fushida** received his ME and DE degrees in Information Science from Nara Institute of Science and Technology in 2007 and 2010, respectively. He was a postdoctoral fellow from 2010–2011 and an assistant professor from 2011–2012 at Nara Institute of Science and Technology. At present, he works for NTT DATA Corporation. His research interests include software process, software metrics, and project management methods. He is a member of the IEEE, IPSJ, IEICE, and JSSST.
Norihiro Yoshida received his BE from the Kyushu Institute of Technology in 2004 and his Master and PhD from Osaka University in 2006 and 2009, respectively. He is an assistant professor at the Nara Institute of Science and Technology since 2010. His research interests include program analysis and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Hajimu Iida received his BE, ME, and Dr of Eng. degrees from Osaka University in 1988, 1990, and 1993, respectively. From 1991 to 1995, he worked for the Department of Information and Computer Science, Faculty of Engineering Science, Osaka University as a research associate. Since 1995, he has been with the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His current position is a professor of the laboratory of Software Design and Analysis. His research interests include modeling and analysis of software and development process.