Software Metrics based on Coding Standards Violations

Yasunari Takai*, Takashi Kobayashi† and Kiyoshi Agusa‡
Graduate School of Information Science, Nagoya University
Aichi, 464-8601, Japan
*takai@agusa.i.is.nagoya-u.ac.jp, †tkobaya, ‡agusa}@is.nagoya-u.ac.jp

Abstract—Software metrics is one of promise technique to capture the size and quality of products, development process in order to assess a software development. Many software metrics based on various aspects of a product and/or a process have been proposed. There is some research which discuss the relation between software metrics and faults to use these metrics as the indicator of quality. Most of these software metrics are based on structural features of products or process information related to explicit fault. In this paper, we focus on latent faults detected by static analysis techniques. The coding checker is widely used to find coding standards violations which are strongly relating to latent faults. In this paper, we propose new software metrics based on coding standards violations to capture latent faults in a development. We analyze two open source projects by using proposed metrics and discuss the effectiveness.

Keywords-Software Metrics; Coding Standard; Static Analysis;

I. INTRODUCTION

Software metrics are used to estimate the complexity, size and quality of products, projects and the process in order to manage a software development. Many software metrics such as CK metrics [1], Mood [2] have been proposed. There are many research focused on software metrics related to faults in order to improve quality of software. Binkley et al. proposed metrics to predict failure [3]. Nagappan et al. proposed a method to mine such metrics for components [4]. Misra et al. investigated the relationships between existing code and design level metrics and bug-density systematically [5]. These researches focused on structural features of products or process information related to explicit fault.

As one of other approaches to improve quality of software, static source code analysis [6], [7] has been widely used because they have better scalability [8]. However, results of static analysis tools may include many false positive answers [8], [9].

In this paper, we focus on the coding checker which is one of static source code analysis tool. A coding checker is the tool in order to capture coding standards violations on source code. Coding standards are rules which we should abide on coding. The rules includes a guide for sharing a common style and avoiding known potential problematic code.

We know that there is the relation between coding standards violations and faults empirically. Boogerd et al. have systematically investigated relationship between coding standards violation and faults [10], [11]. They reported that there were differences between effectiveness of rules of coding standards. However, they did not clarify how to use information related coding standards violations.

In this paper, we propose new software metrics which based on coding standards violations to capture latent faults in a development. Our proposed metrics are based on multiple features of coding standards violations; the number and position of violations, the transition of number of violations, the cost related to violations. By using the metrics, we can quantify the quality of products and development activities. Moreover, we can estimate the status of projects and classify the type of development activities with the metrics based on the transition of number of violations.

In order to evaluate our proposed metrics, we measure our proposed metrics for a series of revision of two real projects. We then investigate relationship between a part of our proposal metrics and development activities.

The remainder of this paper is organized as follows. In Section II, we explain coding standards and the coding checker. Section III describes the definition of our proposed software metrics. We discuss the usage of proposed metrics in Section IV and show results of preliminary assessments of them in Section V. Finally, we conclude the paper and presents future work in Section VI.

II. CODING STANDARD AND CODING CHECKER

Coding standards are rules which we should abide by when coding. Coding standards are classified into the naming rule, the rule of coding style or the rule of prohibited matter. For instance, “Do not substitute the address of the object on the stack into the other objects which has more long lifetime than that has.” If abide by these rules, source codes become more readable and maintainable. To abide by these rules prevent from fails that may occur in the future.

In this study, we use MISRA-C:1998 [12] which is the set of coding standards for C. It is formulated by MISRA (Motor Industry Software Reliability Association) and includes 127 rules of coding standards.

It is difficult to check the coding standards violations on source code by hand work. Coding checker is the tool which checks the coding standards violations automatically on source code. Coding checker gets source code and outputs the result of the checking of coding standards violations.
Both of commercial tools such as QAC, SQMlint and non commercial tools such as Checkstyle\(^1\) are widely used. CX-Checker\(^2\) is one of coding checker for C. It supports user definition of rules by using XPath and API for access CX-Model [13] which is our XML representation of C source code. Although CX-Checker is a stand alone tool, it can be work as Eclipse plug-in. Developer can check coding standard on a CDT Editor and view the position of violations which is annotated on source codes.

III. SOFTWARE METRICS BASED ON CODING STANDARDS VIOLATIONS

We define thirteen new software metrics which use the number of coding standards violations or change of it. We classified them four groups: Metrics based on the number of violations, the position of the violations, the change of the number of violations and the costs related to violations.

A. Metrics based on the number of violations

At first, we define three base metrics focused to the number of violations. NOV is the number of coding standards violations which are detected in the \(i\)-th checking. CNV is the cumulative number of coding standards violations which are detected in all checking. CNUV is the cumulative number of unique coding standards violations which are detected in all checking. Suppose check coding standard for a source code \(n\) time, NOV, CNV and CNUV can be defined as follows:

\[
\text{NOV}_i = |V_i|, \quad \text{CNV} = \sum_{i=1}^{n} \text{NOV}_i, \quad \text{CNUV} = |V|
\]

where \(V_i\) is the set of coding standards violations detected in the \(i\)-th checking. \(V\) is the set of all of unique coding standards violations which are detected in checking.

Note that if violations detected previous checking are detected again, CNV is increased however CNUV is not change.

We also define metrics for characteristics of a sequence of NOV\(_i\). ANV and MNV is the average and the maximum number of NOV\(_i\) respectively. Those metrics can be defined as follows:

\[
\text{ANV} = \frac{\text{CNV}}{n}, \quad \text{MNV} = \max(\text{NOV}_i)
\]

B. Metrics based on the position of violations

We define two metrics based on the position of violations. NVPL is the number of coding standards violations per line.

\[
\text{NVPL}_i = \frac{\text{NOV}_i}{\text{LOC}_i}
\]

where \(\text{LOC}_i\) is the number of physical lines of \(i\)-th checked source codes.

VOV (Variance of coding standards violations) represent a skew of density of violations. This can be defined as follows:

\[
\text{VOV}_i = \frac{1}{\text{NOV}_i} \sum_{v \in V_i} \left( \frac{L_i(v) - \sum_{v' \in V_i} L_i(v')}{{\text{NOV}}_i} \right)^2
\]

where \(L_i(v)\) returns physical line number of which \(v\) is detected in \(i\)-th checking.

C. Metrics based on the change of the number of violations

As metrics focused on the change of the number of violations, we define four metrics. MTOV (Mean time to occur coding standards violations) is the value in which the sum of the period between the violation is detected and the next new unique violation is detected is divided by CNUV. MTFV (Mean time to fix coding standards violations) is the value in which the sum of the existed period of all unique coding standards violations is divided by CNUV.

We exclude time to fix for unfixed violations. MTOV and MTFV can be defined as follows:

\[
\text{MTOV} = \frac{\sum_{v,v+1 \in V} (FDT(v+1) - FDT(v))}{\text{CNUV}}
\]

\[
\text{MTFV} = \frac{\sum_{v \in V} (F_n(v)(LDT(v) - FDT(v)))}{\sum_{v \in V} F_n(v)}
\]

where \(F_j(v)\) is a function to check whether unique violation \(v\) is fixed in \(j\)-th checking or not. If \(v\) is fixed in \(j\)-th checking, the function return 1 otherwise 0. \(FDT(v)\) means the first detected time of \(v\) \(LDT(v)\) means the last detected time of \(v\).

VTOV is the variance of time to occur a new coding standards violation. This can be defined as follows:

\[
\text{VTOV} = \frac{\sum_{v,v+1 \in V} ((FDT(v+1) - FDT(v)) - \text{MTOV})^2}{\text{CNUV}}
\]

VTFV is the variance of time to fix coding standards violations.

This can be defined as follows:

\[
\text{VTFV} = \frac{\sum_{v \in V} (F_n(v)(LDT(v) - FDT(v)) - \text{MTFV})^2}{\sum_{v \in V} F_n(v)}
\]

D. Metrics based on the cost related to violations

Finally we define two metrics focused cost related to violations. NVPM (Number of coding standards violations per man-hour) is the value in which CNUV is divided by cumulative man-hour until the \(i\)-th checking for the \(j\)-th checking.

\[
\text{NVPM}_i^j = \frac{|\bigcup_{k=1}^{j} V_k|}{M(i,j)}
\]

where \(M(i,j)\) means the man-hour while the \(i\)-th checking to the \(j\)-th checking.

\(^1\)http://checkstyle.sourceforge.net/
\(^2\)http://www.sapid.org/cxc/
**NFVPM** (Number of Fixed Coding Standards Violations Per Man-hour) is the value in which the cumulative number of fixed coding standards violations is divided by cumulative man-hour, until the $i$-th checking for the $j$-th checking. NFVPM can be defined as follows:

$$\text{NFVPM}_j^i = \frac{\sum_{k=1}^{j} F_j(\nu_k)}{M(i,j)}$$

### IV. Usage proposed software metrics

#### A. Evaluation of quality of products and developers

Since coding standards are used to improve readability and maintainability, we can measure a part of these quality of source code using NOV and/or NVPL. Most of coding standards is based on past defects and/or experiences of developers, coding standards violations tend to be potential defects. We can evaluate the potential of source codes to have defects with NOV, NVPL. High CNUV may also represent the potential of defects because it mean the code has a lot of violations or have often rewritten.

The productive developer is sensitive to readability and maintainability of source code. The source code which is written by productive developer must have low value of NOV, NVPL, VOV and NVPM. Since they will fix violations quickly, MTFV should be low also.

In typical organizations, coding standards violations are usually checked just only on a nightly build or code reviews before release because of the cost to introduce a coding checker and check violations. In these situation, part of proposed metrics such as VOV, MTFV, VTOV, VTFV, MTFV are not useful because most violations are fixed at the almost same timing.

It is necessary for the maximum effect of coding standard to check coding standards violations frequently. Coding checker integrated with IDE such as CX-Checker allow us to check coding standard violations with a high frequency, for example each build timing.

#### B. Evaluation of the status of the project

We can use proposed metrics in order to evaluate project status. If there is no enough time to finish the rest of the development work, programmer may not fix coding standards violations and inject new violations. As a result, CNUV, VOV, MTFV and NVPM will increase and NFVPM may decrease.

Moreover, proposed metrics have capability to classify project activity if we collect many cases with highly frequent coding standard checking. Suppose that project A and B is the same status on last checking but ANV and MNV are different. This means that NOV of project A rise sharply. Meanwhile one of project B keeps low values. NOV can be a sign of stability of a project and allow us to classify the type difference of development activities. In this case, MTOV, MTFV, VTOV and VTFV have differences.

### V. Case study

#### A. Overview

We investigate co-relation between proposed metrics and status of a project. We measured changes in our proposed metrics of revisions of TOPPERS/ASP kernel [14] which is a real-time kernel in conformity to μITORON4.0 standard and runvpn \(^3\) which is one of the project on github . We checked coding standard violations of each revisions for the subset of the MISRA-C:1998 [12].

We classified each development activities between revisions into “maintenance”, “add new features” and “delete features” by analyzing change log stored in subversion or git repository. Generally commit messages do not show the development activities of substance correctly. For this reason, it is useful to classify the type of development activities. We supposed that NVPL will decrease in maintenance activities such as debugging, refactoring because developer modify codes violated with coding standards in order to improve quality of software. Since adding new source code often causes new coding standards violations, we also supposed that NVPL will increase when new features are implemented. Conversely, NVPL will decrease when features are removed from software.

The experimental method is as follows.

1. Check out source code from the repository. In this experiment, we use version 1.0.0 to 1.6.0 of TOPPERS/ASP kernel and 11th commit(hash: 8eedd81) to 46th commit(hash: 46f1c70) of runvpn without some commits which CX-Checker can not work with. We exclude test codes and external codes of TOPPERS/ASP kernel.
2. Check coding standards violations for the subset of the MISRA-C:1998 by using CX-Checker.
3. Analyze change log descriptions manually and classify activity between revisions into “maintenance”, “add

\(^3\)https://github.com/kse/runvpn
new features” or “delete features”. Table I and Table II show type of each activity we classified.

4) Compared the type of activity with values of proposed metrics NOV, ANV, NVPL of coding standards violations.

B. Result and Discussion

1) Case Study I: TOPPERS/ASP kernel: Figure 1 shows the changes in NOV (number of coding standard violations) and LOC. We can see that NOV increased and decreased coincident with LOC in a development activities for 1.3.0 and 1.5.0. However, NOV increased significantly in 1.2.0 even though there was little change in LOC.

Figure 2 shows the changes in NVPL (number of violations per line). The value increase at revision 1.2.0 and decrease from revision 1.3.2. From Figure 1 and Figure 2, we can consider that quality of development after revision 1.3.2 increase because the density of violation (NVPL) decrease even though LOC significantly increase in revision 1.5.0.

We summarized the relation between the type of work in Table I and NVPL and LOC. Figure 3 shows the difference of NVPL and LOC from previous revision using same notation of Figure 3. We can classify correctly in the 17th, 36th and 46th commit as “fadd”. However, NVPL decreased in the 14th, 15th and 27th commit despite those activity type are also “add new features”. This is because we assume that the initial implementation of a new feature includes more dense coding standard violations than existing codes. If the NVPL of new added code is less than one of codes before commit, the NVPL of committed codes will decrease.

On the other hand, NVPL increased in the 23rd, 26th, 29th and 39th commit whose type are maintenance activities. These commit include debugging and refactoring. The LOCs decreased as a result of refactoring. However CNUV didn’t change because modified codes didn’t includes coding standard violations. Consequently, NVPL increased in these commits. The main reason why they can not classify some activities may be that runvpn is too small program. NVPL of a small program is affected by few changes of NOV.

VI. CONCLUSION

In this paper, we have proposed software metrics based on coding standards violations. We focused to the number and position of violations, changes of number of violations, cost related to violations and defined thirteen metrics. We
investigated co-relation between proposed metrics and status of a project. In case study I, experimental results show that NVPL can be used to classify development activity. However we can not classify all activities in case study II. More precise classification technique with combination of our proposed metrics is our feature work.

ACKNOWLEDGEMENTS

We would like to thank Dr. Noritoshi Atsumi and members of Agusa & Yuen laboratory for their invaluable feedback. This work is partially supported by a Grant-in-Aid for Scientific Research of MEXT Japan (#20300009, #21700027)

REFERENCES

Figure 5. The changes in ANV and NVPL of runvpn

Figure 6. The relation between the difference of LOC and NVPL of runvpn