Proceedings

International Workshop on
Empirical Software Engineering in Practice 2010
(IWESEP 2010)

Nara, Japan, December 7-8, 2010

Sponsored by
StagE Project, MEXT Japan
Foundation for Nara Institute of Science and Technology
Microsoft Research
Osaka University
Nara Institute of Science and Technology (NAIST)

In cooperation with
SIG Software Science, Information and Systems Society, IEICE
SIG Software Engineering, IPSJ

Table of Contents

Process Analysis

Preface

It is our great pleasure to welcome everyone to the International Workshop on Empirical Software Engineering in Practice 2010 (IWESEP 2010). Our workshop is to foster the development of the area by providing a forum where young researchers and practitioners can report on and discuss their new research results and applications in the area of empirical software engineering. The workshop encourages the exchange of ideas within the international community so as to be able to understand, from an empirical viewpoint, strengths and weaknesses of technology in use and new technologies, with the expectation of furthering the more generic field of software engineering.

IWESEP has received 13 submission including 12 regular papers and 1 tool demonstration proposal. After the careful evaluations of the program committee, 8 regular papers and 1 tool demonstration have been accepted to be presented at the workshop. The papers cover a variety of topics, including project management, fault prediction, formal verification and analysis of software development process. In addition, the program includes a keynote speech by Dr. Thomas Zimmermann. We hope that these proceedings will serve as a valuable reference for researchers and developers.

In addition, we hold the MSR School in Asia as the tutorial at IWESEP 2010. The MSR School in Asia provides an overview of the Mining Software Repositories (MSR) filed and an opportunity to learn the techniques used in this field. MSR is a rapidly growing field that holds an annual tutorial and a co-located working conference at the International Conference on Software Engineering (ICSE) every year. We have invited some of the top researchers, Dr. Ahmed Hassan, Dr. Sunghun Kim, and Dr. Thomas Zimmermann, in the MSR field to give the tutorial.

Finally, on behalf of the program committee and the organizing committee, we thank you for attending IWESEP 2010 and hope that you will we enjoy the workshop. We would like to take this opportunity to thank the organizers who spend considerable time reviewing publications and preparations for this workshop.

We hope you will have a great time and an unforgettable experience at the IWESEP2010.

Akinori Ihara, Nara Institute of Science and Technology, Japan
IWESEP 2010 General Chair

Takashi Ishio, Osaka University, Japan
IWESEP2010 Program Chair

Organization

General Chair

Akinori Ihara (Nara Institute of Science and Technology, Japan)

Program Chair

Takashi Ishio (Osaka University, Japan)

Publication Chair

Kyohei Fushida (Nara Institute of Science and Technology, Japan)

Publicity Co-Chair

Sunghun Kim (Hong Kong University of Science and Technology, China)

Masataka Nagura (Hitachi, Ltd., Japan)

Emad Shihab (Queen's University, Canada)

Registration Chair

Masateru Tsunoda (Nara Institute of Science and Technology, Japan)

Local Arrangements Chair

Norihiro Yoshida (Nara Institute of Science and Technology, Japan)

Program Committee

Bram Adams (Queen's University, Canada)
Sousuke Amasaki (Okayama Prefectural University, Japan)
Christian Bird (Microsoft Research, USA)
Ahmed E. Hassan (Queen's University, Canada)
Hideaki Hata (Osaka University, Japan)
Shinpei Hayashi (Tokyo Institute of Technology, Japan)
Israel Herraiz (University Alfonso X el Sabio, Spain)
Yasutaka Kamei (Queen's University, Canada)
Masa Katahira (JAXA, Japan)
Shinji Kawaguchi (JAMSS, Japan)
Jacky Keung (NICTA, Australia)
Hua Jie Lee (University of Melbourne, Australia)
Shinsuke Matsumoto (Kobe University, Japan)
Koji Toda (Nara Institute of Science and Technology, Japan)
Hidetake Uwano (Nara National College of Technology, Japan)
Rodrigo Vivanco (University of Manitoba, Canada)
Thomas Zimmermann (Microsoft Research, USA)

Keynote Address

Keynote Address

Empirical Software Engineering and Measurement at Microsoft

Thomas Zimmermann (*Microsoft Research / University of Calgary*)

Abstract:

Software engineering is an data rich activity: changes to source code are recorded in version archives, bugs are reported to issue tracking systems, and communications are archived in e-mails and newsgroups. The Empirical Software Engineering and Measurement (ESM) group at Microsoft Research analyzes such data to better understand various software development issues from an empirical perspective. In this talk, I will highlight our research themes and activities using examples from our research on socio technical congruence, bug reporting and triaging, and data-driven software engineering. I will highlight our unique ability to leverage industrial data and developers and the ability to make near term impact on Microsoft via the results of our studies. The work presented in this talk has been done by Chris Bird, Brendan Murphy, Nachi Nagappan, myself, and many others who have visited our group over the past years.

Project Management

Process Fragment Based Process Complexity with Workflow Management Tables

Masaki Obana[1] , Noriko Hanakawa[2] , Norihiro Yoshida[1] , Hajimu Iida[1]
[1]Nara Institute Science and Technology
[2]Hannan University
masaki-o@is.naist.jp, hanakawa@hannan-u.ac.jp, yoshida@is.naist.jp, iida@itc.naist.jp

ABSTRACT
The actual software development processes deviate from initial model planned at early stage. One of the reasons is that additional processes get triggered by urgent changes of software specifications. Such additional processes increase the complexity of the whole development process, and possibly decrease the product quality. In this paper, we propose a novel complexity measure of software process, which based on the number of process fragments, the number of simultaneous process fragments, and the number of developers' groups. Proposed process complexity is applied to two industrial projects in order to show the usefulness of this measure. As a result, we found that the higher value of process complexity indicates higher risk of products' faults.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management: software process models

General Terms
Management, Measurement.

Keywords
Process complexity, risk management, a workflow management table, change of customers' demand.

1. Introduction
In software development projects, large gaps between planned development processes and actual executed development processes exist. Even if a development team has originally selected the waterfall model unplanned small processes are often triggered as shown in the following examples.

i. One activity of waterfall-based process is changed into new iterative process because of urgent specification changes.

ii. Developers design GUI using a new prototype development process at design phase.

iii. In an incremental development process, developers correct defects in the previous release while developers are implementing new functions in current release.

That is, the waterfall-based process at planning time is often gradually transformed into the combination of the original waterfall-based process and several unplanned small processes (hereinafter referred to as process fragments). In consequence, actual development processes are more complicated than planned one (see also Figure 1).

In this paper, we firstly assume that complicated process decreases product quality, and then propose a new metric for process complexity based on the number of unplanned process fragments, the number of simultaneous execution processes, and the number of developers' groups. It can be used to visualize how process complexity increases as actual development process proceeds.

An aim of measuring process complexity is to prevent software from becoming low quality product. A process complexity is derived from a base process and process fragments. A base process means an original process that was planned at the beginning of a project. Process fragments mean additional and piecemeal processes that are added to the base process on the way of a project. Process fragment occurs by urgent changes of customers' requirement, or sudden occurrence of debugging faults. Process fragments can be extracted from actual workflow management tables which are popular in Japanese industrial projects. We especially focus on simultaneous execution of multiple processes to model the process complexity. Simultaneous execution of multiple processes is caused by adding process fragments on the way of a project. Finally, we perform an industrial case study in order to show the usefulness of process complexity. In this case study, we found that process complexity indicated the degree of risk of post-release faults.

Section 2 shows related work about process metrics, and risk management. The process complexity is proposed in section 3. Section 3 also describes how the proposed complexity would be used in industrial projects. Case studies of two projects are shown in section 4. In section 5, we discuss a way of risk management using the process complexity. Summary and future works are described in Section 6.

2. Related Work
Many software development process measurement techniques have been proposed. CMM [1] is a process maturity model by Humphrey. Five maturity levels of organizations have been proposed in CMM. When a maturity level is determined, various values of parameters (faults rate in total test, test density, review density) are collected. In addition, Sakamoto et al. proposed a metrics for measuring process improvement levels [2]. The metrics were applied to a project based on a waterfall process model. Theses process measurement metrics' parameters include the number of times of review execution and the number of faults in the reviewed documents. The aim of these process measurement techniques is improvement of process maturity of an organization, while our research aims to measure process complexity of a project, not organization. Especially, changes of process complexity in a project clearly are presented by our process complexity.

Many researches of process modeling techniques have been ever proposed. Cugola et al. proposed a process modeling language that describes easily additional tasks [3]. Extra tasks are easily added to a normal development process model using the modeling language. Fuggetta et al. proposed investigated problems about software development environments and tools oriented on various process models [4]. These process modeling techniques are useful to simulate process models in order to manage projects. However, these process modeling techniques make no mention of process complexity in a project.

In a field of industrial practices, Rational Unified Process (RUP) has been proposed [5]. The RUP has evolved in integrating several practical development processes. The RUP includes Spiral process model, use-case oriented process model, and risk oriented process model. Moreover, the RUP can correspond to the latest development techniques such as agile software development, and .NET framework development. Although problems of management and scalability exist, the RUP is an efficient integrated process for practical fields. The concept of the various processes integration is similar to our process fragments integration, while the RUP is a pre-planned integration processes. The concept of our process complexity is based on more flexible model considering changes of development processes during a project execution. Our process complexity focuses on changes of an original development process by process fragments, and regarded as a development processes change.

Garcia et al. evaluated maintainability and modifiability of process models using new metrics based on GQM [6]. They focus on additional task as modifiability. The focus of additional task is similar to our concept of process complexity. Although their research target is theoretical process models, and our research target is practical development processes. In this way , there was no studies for measuring complexity of process during a project as long as we examined it. Therefore, the originality of our proposed complexity may be high.

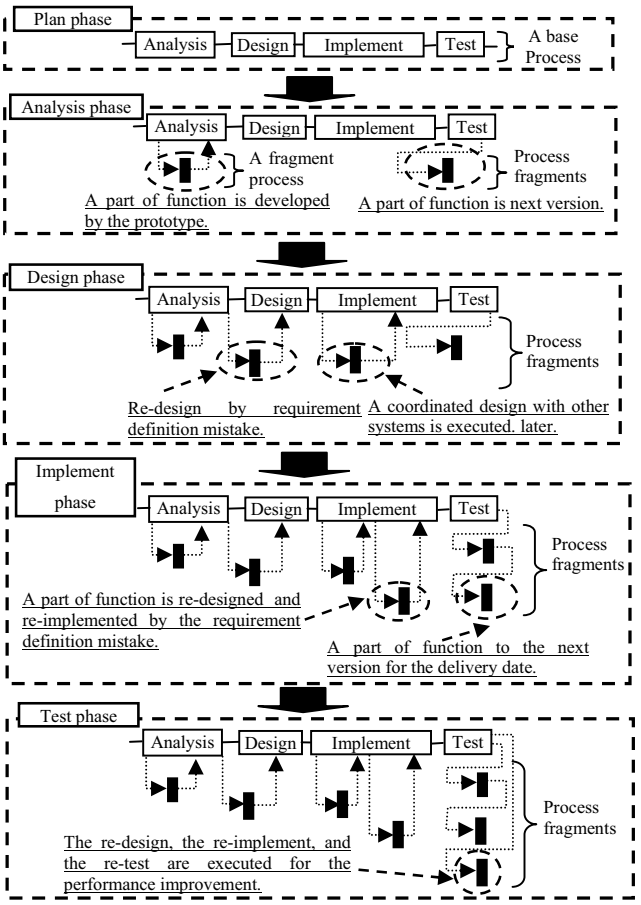


Figure 1 A concept of process fragments

3. Proposed Metrics Based on Process fragment

3.1 Process fragment

In software development, a manager makes a plan of a single development process like a waterfall process model at the beginning of a project. However, the planned single process usually continues to change until the end of the project. For example, at the beginning of a project, a manager makes a plan based on a waterfall process model. However the original process is changed to an incremental process model because several functions' development is shifted to next version's development. Moreover, at the requirement analysis phase, prototyping process may be added to an original process in order to satisfy customers' demands. If multiple releases like an incremental process model exist, developers have to implement new functions while developers correct detects that were caused in the previous version's development. In this paper, we call the original process "a base process", we call the additional process "a process fragment". While original process is a process that was planned at the beginning of a project, a process fragment is a process that is added to the original process on the way of a project. "Fragment" means piecemeal process. Process fragments are simultaneously executed with a base process. Process fragment does not mean simple refinement of a base process, but rather may separately executes from a base process execution.

Figure1 shows an example of process fragment. At the planning phase, it is a simple development process that consists of analysis activity, design activity, implementation activity, and testing activity. In many cases, because of insufficient information, a manager often makes a rough simple process (macro process) rather than a detailed process (micro process) [7]. However, information about software increases as a project progresses, and the original simple process changes to more complicated processes. In the case of Figure 1, at the analysis phase, an unplanned prototype process was added to the original process because of customers' requests. As a result of analysis phase, implementations of some functions were shifted to next version development because of constraints of resources such as time, cost, and human. The process fragments were shown at the Figure1 as small black boxes. In the design phase, because customers detected miss-definitions of system specifications that were determined in the previous analysis phase, a process for reworking of requirement analysis was added to the development process. Moreover, the manager shifted the development of a combination function with the outside system when the outside system was completed. During the implementation phase, several reworks of designs occurred. In the test phase, reworks of designs occurred because of low performance of several functions.

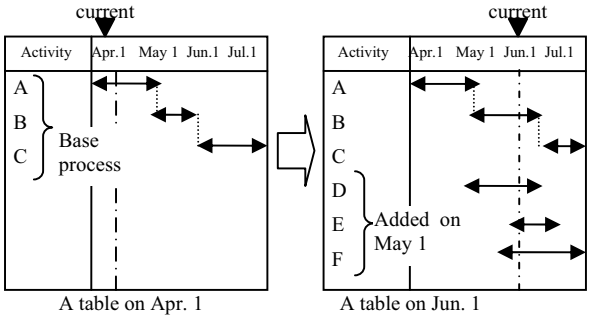


Figure 2 Extracting process fragments from configuration of a workflow management table

Process fragments are caused by urgent customers' requests, design errors, combination with outside systems on the way of development. Various sizes of process fragments exist. A small process fragment includes only an action such as document revision. A large process fragment may include more activities for example, a series of developing activities; design activity, implementation activity, and test activity.

3.2 Calculation of process complexity

3.2.1 Extracting process fragments

Process complexity is calculated based on process fragments. Process fragments are identified from a series of workflow management table. That is a continuator revised along the project. Figure 2 shows two versions of a workflow management table. Each column means a date, each row means an activity. A series of A, B, C activities is a base process. D, E, F activities are added to the base process on May 1. Therefore, D, E, F activities are process fragments. On Jun. 1, the base process and three process fragments are simultaneously executed. In this way, process fragments can be identified from configuration management of a workflow management table. Difference between current version and previous version of a workflow management table means process fragments. Of course, the proposed complexity is available in various development processes such as an agile process as long as managers manage process fragments in various management charts.

3.2.2 Definition of process complexity

Process complexity is defined by the following equation.

$$PC_{(t)} = \sum_{i=1}^{N_{(t)}} (Num\_dev_{(t)i} \times L_{(t)i} \times term_{(t)i}) \quad \text{.....(1)}$$

$PC_{(t)}$: process complexity on time t

$N_{(t)}$: the total number of process on time t

$Num\_dev_{(t)i}$: the number of group of developers of the *i-th* process fragment on time t

$L_{(t)i}$: the number of simultaneous processes of the *i-th* process fragment on time t. But the i-th fragment is eliminated from these multiplications in L(t).

$term_{(t)i}$: ratio of an executing period of the *i-th* process fragment for the whole period of the project on time t, that is, if $term_{(t)i}$ is near 1, a executing period of the process fragment is near the whole period of the project.

Basically, proposed process complexity is the accumulation of all process fragments including finished processes. The reason of the accumulation is that the proposed complexity's target is a whole project, not a spot timing of a project. For example, many process fragments occur at the first half of a project. Even if the process fragments have finished, the fragments' executions may harmfully influence products and process at the latter half of the project. Therefore, the proposed complexity is accumulation of all process fragments.

The process complexity basically depends on three elements; the number of group of the *i-th* process fragment on time *t*: $Num\_dev_{(t)i}$, the number of simultaneous processes of the *i-th* process fragment on time *t*: $L_{(t)i}$, and ratio of an executing period of the *i-th* process fragment for the whole period of project time *t*: $term_{(t)i}$. Granularity of group of developer($Num\_dev_{(t)i}$) depends on the scale of process fragments. If a process fragment is in detail of every hour, a group simply correspond to a person. If process fragment is large such as design phase, a group unit will be an organization such as SE group and programmer group. The group of developers will be carefully discussed in future research. The ratio of an executing period of the *i-th* process fragment for the whole period of project time *t* ($term_{(t)i}$) means impact scale of a process fragment. If a process fragment is very large, for example an executing period of the process fragment is almost same as the whole period of a project, the process fragment will influence largely the project. In contrast, if a process fragment is very small, for example an executing period is only one day, the process fragment will not influence a project so much.

In short, when more and larger scale process fragments are simultaneously executed, a value of process complexity becomes larger. When fewer and smaller scale process fragments simultaneously are executed, a value of process complexity becomes smaller. Values of the parameters of equation (1) are easily extracted from configuration management data of a workflow management table.

3.3 Setting a base process and extracting process fragments

At the beginning of a project, a base process is determined. If a planned schedule is based on a typical waterfall process model such as the base process in Figure 1, the parameters' values of process complexity are $t=0$, $N_{(0)}=1$, $Num\_dev_{(0)}=3$ (SE group developer group, customer group), $L_{(0)}=1$, *and* $term_{(0)}=1$. Therefore the value of process complexity $PC_{(0)}=3$.

As a project progresses, unplanned process fragments are occasionally added to the base process at time *t1*. A manager registers the process fragments as new activities to the workflow management table. The manager also assigns developers to the new activities. Here, we assume that the planned period of a base process is 180 days. A manager adds two activities to the workflow management table. The period of each additional activity is planned as 10 days. Therefore the total number of process $N_{(t1)} = 3$, and the process complexity is calculated as follows;

(1) for $i = 1$ (a base process)
- $Num\_dev_{(t1)} = 3$
- $L_{(t1)1} = 3$
- $term_{(t1)1} = 180/180 = 1.0$

(2) for $i = 2$ (the first process fragment)
- $Num\_dev_{(t1)2} = 1$
- $L_{(t1)2} = 3$
- $term_{(t1)2} = 10/180 = 0.056$

(3) for $i = 3$ (the second process fragment)
- $Num\_dev_{(t1)3} = 1$
- $L_{(t1)3} = 3$
- $term_{(t1)3} = 10/180 = 0.056$

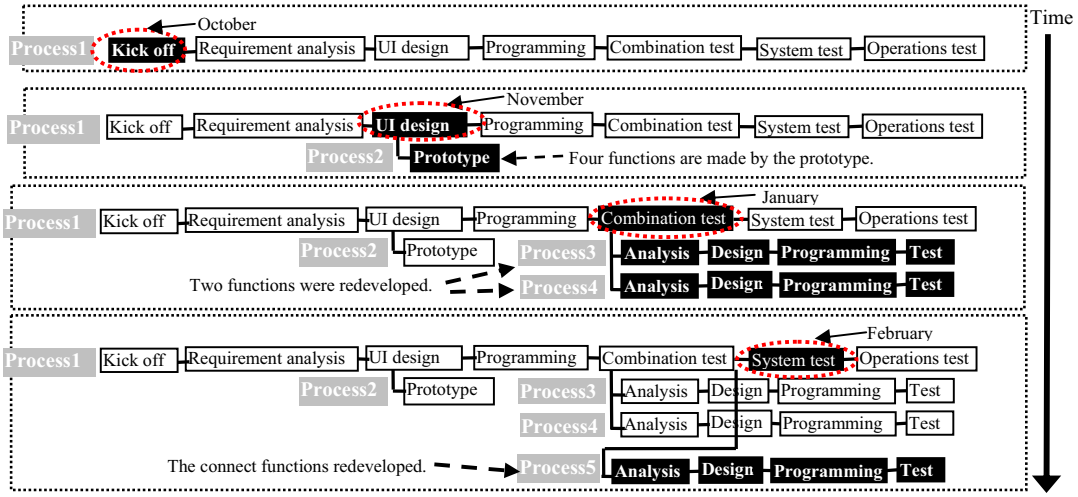


Figure 3 A variation of development process of the HInT project

Finally, a value of process complexity at $t=t1$ can be calculated as $PC_{(t1)} = 9.000 + 0.168 + 0.168 = 9.336$.

In this way, the value of $PC_{(t)}$ at any value of time t can be calculated based on workflow management table.

4. Application to Two Industrial Projects

The process complexity has been applied to two practical projects.

4.1 The HInT project

The first project's name is HInT (Hannan Internet communication Tool). The HInT project developed a web-based educational portal system. The development began from October 2007, the release of the HInT was April 2008. Because the workflow management table was updated every week, 20 versions of the workflow management table are obtained. At the beginning of the project, the number of activities in the workflow table was 20. At the end of the project, the number of activities reached to 123. Each activity had a planned schedule and a practice result of execution. Figure 3 shows a rough variation of development process of the project. At the beginning of the project, the shape of the development process was completely a waterfall process. However, at the UI design phase, a prototype process was added to the waterfall process. In the prototype process, four trial versions were presented to customers. At the combination test phase, developers and customers found significant errors of specifications and two process fragments were added to the development process in haste. Reworks such as re-design, re-implement, and re-test for the specification errors continued until the operation test phase. At the system test phase, an error of a function connecting with an outside system occurred and a new process fragment was introduced to the development process. The introduced process fragment consists of various activities such as investigating network environments, investigating specification of the outside system, and revising the programs. These activities continued until the delivery timing.

Figure 4 shows process complexity of the HInT project. At the beginning of the project, the value of process complexity was very low while the value of the process complexity increased as the project progresses. In particular, after the system test phase, four processes were simultaneously executed. Then process complexity increased to 44.167.

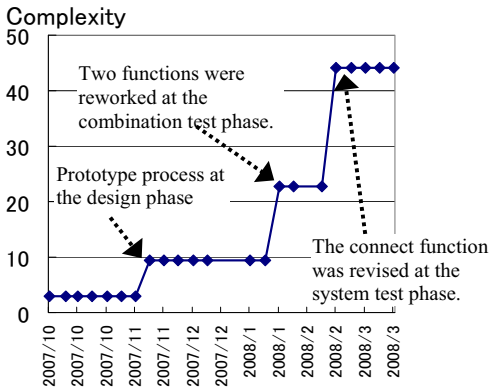


Figure 4 Process complexity of the HInT project

4.2 The p-HInT project

The p-HInT is a lecture support system for large-scale lectures with mobile terminals [8][9]. A base process of the p-HInT project was an incremental development process. We released the p-HInT product four times from April 2008 through April 2010. A development process of the each release included analysis phase, design phase, implement phase, and test phase. At the first version development, basic functions and infrastructure implementation were planned in detail. However, the manager determined only rough design of next versions' functions. At the second version development, new four functions were developed while the developers fixed defects that had been introduced at the previous version's development. Of course, the debugging works were not planned in the original schedule. At the third version development, product refactoring was executed because the design of the product became too complicated for introduction of new functions and also for debugging works. At the fourth version development, customers requested new functions that were not discussed at the requirement analysis phase. The customers said that the requested new functions were more important than the functions that were originally planned at the beginning of the project. At same time, performance improvement work for some functions in the previous releases was made at the fourth version development.

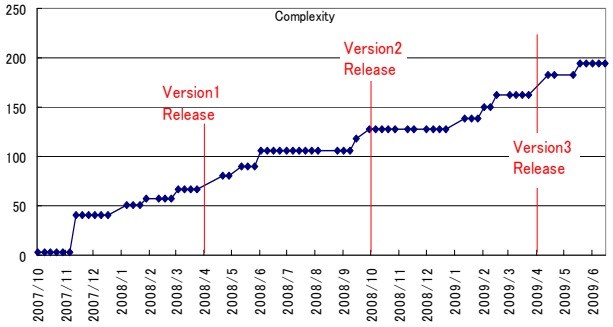


Figure 5 Process complexity of the p-HInT project

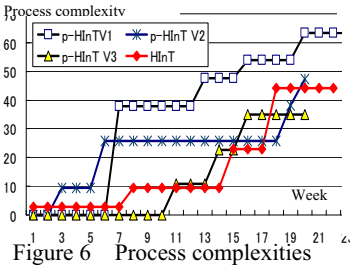


Figure 6 Process complexities of the four project

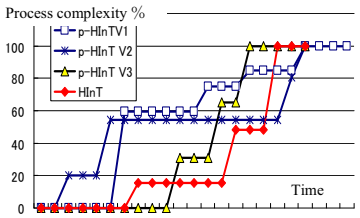


Figure 7 Patterns of growth of process complexity

The process complexity of the p-HInT project was measured until June 2009. The number of items listed in the workflow table was 31. The items were divided into activities that were executable on each week. Figure 5 shows a result of the measurement of process complexity.

At the begging of the project, the value of process complexity was low (3.0) because the original development process is a simple incremental process. At November 2007, because the original process was divided into each process for each function, the value of the process complexity increases to about 45.0.

At the release time of the first version, the value was 63.48. In the second version development, processes of improving performance for the released functions were added to the original process. That is, developers had to revise the released functions while the developers had to implement new functions as originally planned at the beginning of the project. The value of the process complexity increased to 127.56.

At the test phase of the second version, some specification errors came to light. For fixing the specification errors, several process fragments were added and the value of process complexity increased again.

On the first half of the third version development, the value of process complexity did not increase because of a product refactoring activity. Developers concentrated to the product refactoring work. However, on the latter half of the third version, problems such as low performance of the released functions and miss data connection with outside systems occurred. Because process fragments for fixing the problems were added, the value of process complexity increased to 182.26.

5. Discussion

We discuss usefulness of process complexity. Process complexity has a role of capturing dynamic processes as a project progresses. If the values of process complexity for the beginning of a project and the end of a project don't differ so much, we can interpret that the project was a stable because the project may have been executed smoothly without large changes. In contrast, if a value of process complexity largely increased, we can interpret that the project was unstable due to large changes. Figure 6 shows changes of the values of process complexity of the two projects. We regard each version of the p-HInT as one project. The fourth version development of the p-HInT is excluded because all data extracting the workflow management table was not prepared.

It became clear change of process complexity of each project in Figure 6. Projects with large increase of process complexity are version 1 and version 2 of the p-HInT project, and the HInT project. Each increase value is 63.48, 47.28, and 41.17. In contrast, in the version 3 of the p-HInT, the value of process complexity was 35.10.

In addition, each project has a feature of a process complexity growth pattern. Figure 7 shows the patterns of the projects. A maximum value of process complexity of each project is set to 100%. The value of process complexity of the version 1 of the p-HInT increased suddenly on the first half of the project. In version 2 of the p-HInT, the value of complexity largely increased at two timings; at the first half of the project, and at the end of the project. In addition, in version 3 of the p-HInT, the value of complexity gradually increased. The value of complexity of the HInT project increased at the latter half of the project. We call these patterns as "growth patter on early stage type", "growth patter on early and late stage type, "growth patter smoothly type", and "growth patter on late stage type". The "growth patter on early stage type" often occurs when a concept of development and development methods are changed on the first half of a project. In the version 2 of the p-HInT, debugging activities for the previous version's faults were set to highest priority activities on the first half of the project. The process of the version 2 largely changed. Moreover, the unreasonable executions of the debugging activities caused occurrences of new faults on the latter half of the project. Therefore, addition of the debugging activities for the new faults caused the increase of the value of process complexity on the latter half of the project. On the other hand, in the HInT project, developers presented the product demonstration to customers at the system test phase. As a result, several specification errors and customers' demand changes were clarified. Therefore, many activities such as re-design, re-implement, re-test were added to the development process. The process fragments such as the re-work activities caused the increase of process complexity on the latter half of the project.

Therefore, we propose a way of evaluating project risk using the changes of process complexity. The project risk is as follows;

$$Risk = Rank \times Variation \qquad \text{.....(2)}$$

Risk: Project Risk
Rank: growth patter patterns of process complexity:
"growth patter smoothly type": 1,
"growth patter on early stage type" : 2,
"growth patter on late stage type" : 3,
"growth patter on early and late stage type" : 4.
Variation: gap of values of process complexity between at the beginning of a project and at the end of a project.

If a value of *Risk* is large, we can judge the project high risk. *Rank* shows growth patter patterns of process complexity. The pattern of "growth patter smoothly type" is most stable. The pattern of "growth patter on early stage type" is relatively stable. The reason is that developers can cope with the process changes on the first half of a project. Because the changes occur on the early stage,

developers have time for arranging their works. In contrast, changes on the latter half of a project such as "growth patter on late stage type" have high risk. Because developers have little time for arranging their works, developers are confused in simultaneous executions of processes for the changes and an original process. The pattern of "growth patter on early and late stage type" is worst. Because the additional works on the first half of a project influence the works on the latter half of a project. Managers may add the additional works in unplanned on the first half of a project. In addition, *Variation* means a gap value of process complexity between at the beginning of a project and at the end of a project. Of course, a small value of *Variation* is better. In this way, by calculating a value of *Risk*, we evaluate whether a project is complicated or not. A detailed way of determining a value of *Rank* will be discussed in future.

Figure 8 shows results of project risks of the four projects. The project risk of the version 2 of the p-HInT is highest. The version 3 of the p-HInT has lowest risk. Figure 9 shows specific gravity of failures occurred after release. The specific gravity failure is calculated by the number of failure and importance level of failure. The importance level means strength of impact on a product. As Table 1, we classified the failures into five important levels; SS, S, A, B, C. SS means most strong impact like system-down. C means weakest impact like GUI's improvement. The specific gravity of the failures is a value that multiplied the number of failures and the importance level. In the calculation, we set up that a constant of SS is 5, a constant of S is 4, a constant of A is 3, a constant of B is 2, and a constant of C is 1. For example, a value of specific gravity of the p-HInT version 1 is calculated by "5*2 + 4*3 + 3*19 +2*1 + 1*1". The value is 82. In the same way, the specific gravity of failure of the p-HInT version 2 is 88, one of p-HInT version 3 is 37, and one of HInT is 156.

Comparing Figure 8 with Figure 9, Spearman rank correlation coefficient is 0.4. Therefore, we confirmed a weak relationship between the values of project risk and the value of specific gravity of failures. The version 3 of the p-HInT is not only lowest risk but also lowest specific gravity of failure. In addition, if a value of the project risk is high, the specific gravity of failure will be high. That is, possibility of occurrence of strong impact failures becomes high as a value of project risk is more. In the HInT project, because customers suddenly requested new functions, many process fragments were added to the process on the latter half of the project. A value of the project risk increased at the latter half of the project. As a result, many significant failures with SS rank occurred after release.

In this way, by the project risk we can predict possibility of failure occurrences after release. A most useful feature of the project risk is early prediction before start of executions of processes. The prediction is possible when a workflow management table is revised by adding process fragments. The possibility of the early prediction in the project risk is most different from product metrics. Product metrics is calculated by the already complicated product. However, the project risk can prevent a project from becoming high risk. It is useful for managers to judge whether new process fragments are added.

Table 1 Failures after releases of the four projects

	SS	S	A	B	C
p-HInT Version1	2	3	19	1	1
p-HInT Version2	10	6	4	1	0
p-HInT Version3	3	3	2	1	2
HInT	14	12	12	1	0

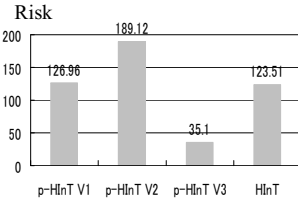


Figure 8 Project risks of the four projects

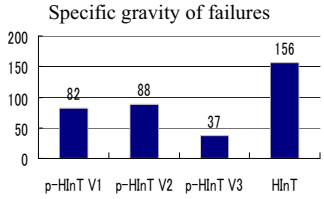


Figure 9 Specific gravity of failures of the four projects

6. Summary

We proposed process complexity model for risk management in software development. The process complexity is calculated by process fragments derived from a workflow management table. In addition, project risk metric has been also proposed. The project risk is calculated by changes of process complexity during a project. We applied the process complexity and the project risk metrics to two industrial projects. As a result, the project risk values have a weak relation with specific gravity of failures. Therefore, managers can predict the quality of released product when the workflow management table is revised.

In future, we will apply the process complexity and the project risk metrics to many industrial projects. We will clarify the relationships between project risk and specific gravity of failures. Then, we will determine value of thresholds of the project risk.

7. REFERENCES

[1] Humphrey Watts S.1989. *Managing the software process.* Addison-Wesley Professional, USA.

[2] Sakamoto, K., Tanaka, T., Kusumoto, S., Matsumoto, K., and Kikuno T. 2000. *An Improvement of Software Process Based on Benefit Estimation.* IEICE Trans. Inf. & Syst.(in Japanese), J83-D-I(7), pp.740-748.

[3] Cugola, G. 1998. *Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models*, IEEE Transaction of Software Engineering, Vol. 24, No. 11, pp.982-1001.

[4] Fuggetta, A. and Ghezzi, C. 1994. *State of the art and open issues in process-centered software engineering environments*, Journal of Systems and Software, Vol. 26, No. 1, pp.53-60.

[5] Kruchten, R. 2000. *The Rational Unified Process,* Addison-Wesley Professional, USA.

[6] Garcia, F., Ruiz, F., Piattini, M. 2004. *Definition and empirical validation of metrics for software process models,* Proceedings of the 5th International Conference Product Focused Software Process Improvement (PROFES'2004), pp.146-158.

[7] Obana, M., Hanakawa, N., Iida, H. 2010. Process complexity metrics based on fragment process on workflow management tables, Proceeding of the Software Engineering Symposium SES2010 (in Japanese), pp.89-96.

[8] Hanakawa, N., Yamamoto, G., Tashiro, K., Tagami, H., Hamada, S. 2008. *p-HInT: Interactive Educational environment for improving large-scale lecture with mobile game terminals*, Proceedings of the16th International Conference on Computers in Education (ICCE'2008), pp.629-634.

[9] Hanakawa, N., Obana, M. 2010. *Mobile game terminal based interactive education environment for large-scale lectures*, Proceeding of the Eighth IASTED International Conference on Web-based Education (WBE2010)

Applying Outlier Deletion to Analogy Based Cost Estimation

Masateru Tsunoda
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
masate-t@is.naist.jp

Akito Monden
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
akito-m@is.naist.jp

Mizuho Watanabe[1]
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
mizuho.watanabe01@is.naist.jp

Takeshi Kakimoto
Kagawa National College of Technology
355 Chokushicho, Takamatsu-shi, Kagawa 761-8058 Japan
kakimoto@t.kagawa-nct.ac.jp

Ken-ichi Matsumoto
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
matumoto@is.naist.jp

ABSTRACT
In this research, we apply outlier deletion methods to analogy based software effort estimation to evaluate their effects. We employed existing deletion methods (Cook's distance based deletion, and Mantel's correlation based deletion) and new deletion method proposed in this research. While existing deletion methods eliminates outliers from entire dataset before estimation, our method identifies and eliminates outliers from neighborhood projects of estimation target. Our method treats a project as an outlier when the effort of the project is extremely higher or lower than other neighborhood projects. In the experiment, our method showed highest performance among applied deletion methods, and average *BRE* (Balanced Relative Error) indicated 20.8% improvement by our method.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *Cost estimation*, K.6.1 [**Computing Milieux**]: Project and People Management – *Staffing*

General Terms
Management, Measurement, Economics, Experimentation.

Keywords
Case based reasoning, effort prediction, abnormal value, project management, productivity.

1. INTRODUCTION
To achieve success of software development project, it is important to estimate development effort accurately, and therefore many quantitative estimation methods have been proposed [2][17][20]. Recently, analogy based estimation [19] gets attention, and many proposals and case studies have been reported [6][7][13][21][22]. Analogy based estimation selects projects (neighborhood projects) which are similar to the estimated project from past project dataset, and estimates effort based on similar projects' effort. One of the advantages of analogy based estimation is that estimation results are comprehensible for estimators such as project managers [22], because they can confirm neighborhood projects used for estimation. Although ordinary estimation models like linear regression model estimate various target projects' effort by one model, analogy based estimation does not make such a model, and estimates effort by neighborhood projects' effort. So analogy based estimation can reflect individuality of each target project in estimation.

Past project dataset sometimes includes project data which should not be used for estimation [18]. For example, projects where exceptional amount of reworks were occurred have larger effort than other same scale projects. Additionally, when effort was inaccurately collected or recorded, recorded effort is different from actual effort. These projects should be eliminated from dataset before estimation, because they lessen estimation accuracy. However, to identify these projects is not easy because inside details of each project are usually not recorded in the dataset. Even if such details can be grasped, it is difficult to settle elimination criteria (For example, to settle criterion of abnormal amount of reworks).

Thus, such projects are often eliminated by statistical outlier deletion methods. Outlier deletion methods identify projects as outliers when specific variants' values are extremely large or combination of variants' values (effort, system size, or duration) is fairly different from other projects' one, and remove them from dataset. Cook's distance is widely used as outlier deletion method when applying linear regression analysis. In addition to Cook's distance, some outlier deletion methods for effort estimation [6][18] have been proposed. However, there are very few case studies which apply outlier deletion methods to analogy based estimation, and evaluate effect of outlier deletion methods toward analogy based estimation.

In this research, we apply outlier deletion methods to analogy based estimation to evaluate their effects. Two types of outlier deletion methods (Cook's distance based deletion, and Mantel's

[1] Presently with IBM Japan, Ltd.

correlation based deletion) are applied to ISBSG dataset [5], and development effort is estimated by analogy based estimation. ISBSG dataset includes many project data which are collected from software development companies, and it is widely used in many researches.

Also, we propose new outlier deletion method considering characteristics of analogy based estimation, and compare its effect to others. In analogy based estimation, when variance of actual effort of neighborhood projects is large, estimation accuracy gets low [16]. Our method identifies a project as an outlier when the effort of the project is extremely higher or lower than other neighborhood projects, and excludes it from computation of estimated effort. Actual effort of neighborhood projects is normalized by Z-score computation [9], and when the normalized value is larger than threshold, the project is identified as an outlier. While existing deletion methods eliminates outliers from entire dataset before estimation, our method does from neighborhood projects.

In what follows, Section 2 explains analogy based estimation. Section 3 explains outlier deletion methods, and Section 4 describes experimental settings. Section 5 shows results of the experiment and discusses it, and Section 6 concludes the paper with a summary.

2. ANALOGY BASED ESTIMATION

The origin of analogy based estimation is CBR (case based reasoning), which is studied in artificial intelligence field. Shepperd et al. [19] applied CBR to software development effort estimation. CBR selects a case similar to current issue from accumulated past cases, and applies solution of the case to the issue. CBR assumes similar issues can be solved by similar solution. Analogy based estimation assumes neighborhood (similar) projects (For example, development size and used programming language is similar) have similar effort, and estimates effort based on neighborhood projects' effort. Although ready-made estimation models such as COCOMO [2] can make estimation without stored software project dataset, analogy based estimation cannot estimate without it. It is a weak point of analogy based estimation, but it can be overcome by using public dataset.

Analogy based estimation uses $m \times n$ matrix shown in Table 1. In the matrix, $Proj_i$ is *i-th* project, $Metric_j$ is *j-th* variable, x_{ij} is a value of $Metric_j$ of $Proj_i$, fp_i is the development size (e.g. function point) of $Proj_i$, and y_i is the actual effort of $Proj_i$. We presume $Proj_a$ is estimated project, and $\hat{y}_a$ is the estimated value of y_a. Procedures of analogy based estimation consist of the three steps described below.

Step 1: Since each variable $Metric_j$ has different range of value, this step makes the ranges [0, 1]. The value x'_{ij}, normalized the value of x_{ij} is calculated by:

$$x'_{ij} = \frac{x_{ij} - \min(Mertic_j)}{\max(Metric_j) - \min(Metric_j)} \quad (1)$$

In the equation, $\max(Metric_j)$ and $\min(Metric_j)$ denote the maximum and minimum value of $Metric_j$ respectively.

Step 2: To find projects which are similar to estimated project $Proj_a$ (i.e. identifying neighborhood projects), distance between $Proj_a$ and other projects $Proj_i$ is calculated. Although various measures (e.g. a measure directly handling nominal variables) are proposed [1], we applied Euclidean distance measure because it is widely used [21]. In the measure, short distance indicates two projects are similar. Distance $\mathrm{Dist}(Proj_a, Proj_i)$ between $Proj_a$ and $Proj_i$ is calculated by:

$$\mathrm{Dist}(Proj_a, Proj_i) = \sqrt{\sum_{h=1}^{m} (x'_{ah} - x'_{ih})^2} \quad (2)$$

Table 1. Dataset used by analogy based estimation

	Effort	*Size*	$Metric_1$	$Metric_2$	…	$Metric_j$	…	$Metric_n$
$Proj_1$	y_1	fp_1	x_{11}	x_{12}	…	x_{1j}	…	x_{1n}
$Proj_2$	y_2	fp_2	x_{21}	x_{22}	…	x_{2j}	…	x_{2n}
…	…	…	…	…		…		…
$Proj_i$	y_i	fp_i	x_{i1}	x_{i2}	…	x_{ij}	…	x_{in}
…	…	…	…	…		…		…
$Proj_m$	y_m	fp_m	x_{m1}	x_{m2}	…	x_{mj}	…	x_{mn}

Step 3: The estimated effort $\hat{y}_a$ of project $Proj_a$ is calculated by actual effort y_i of k neighborhood projects. While average of neighborhood projects' effort is generally used, we adopt size adjustment method, which showed high estimation accuracy in some researches [7][13][22]. Size adjustment method assumes effort y_i is s times (s is real number greater than 0) larger when development size fp_i is s times larger, and the method adjusts effort y_i based on ratio of estimated project's size fp_a and neighborhood project's size fp_i. Adjusted effort $adjy_i$ is calculated by equation (3), and estimated effort $\hat{y}_a$ is calculated by equation (4). In the equation, *Simprojects* denotes the set of k neighborhood projects which have top similarity with $Proj_a$.

$$adjy_i = y_i \times \frac{fp_a}{fp_i} \quad (3)$$

$$\hat{y}_a = \frac{\sum_{h \in Simproject\,s} adjy_h}{k} \quad (4)$$

3. OUTLIER DELETION METHOD

Outlier deletion method examines whether a case (project) in dataset is an outlier or not, and eliminates it from dataset when it is identified as an outlier. When software development effort is estimated, Cook's distance based deletion is widely applied before building a linear regression model to eliminate outliers (e.g., [12]). However, there are few researches which analyzed effects of outlier deletion methods [18], or proposed outlier deletion method suitable for analogy based estimation [6].

Seo et al. [18] proposed that LTS (least trimmed squares) based deletion and k-means based deletion are applied before effort estimation, and evaluated their effects by estimating development effort with linear regression model, neural network, and Bayesian network. However, they did not use analogy based estimation with a deletion method. Keung et al. [6] proposed Mantel's correlation based deletion. Although they analyzed which projects were elim-

inated, they did not compare it with other deletion method, estimating development effort with cross validation. Outlier deletion methods used in our research are explained below.

3.1 Cook's Distance Based Deletion

Cook's distance based deletion is used with multiple linear regression analysis, and identifies an outlier when the case greatly varies coefficient of the regression model. Cook's distance indicates how much residual of all cases varies when a certain case is omitted from model building. Large Cook's distance means the case greatly affects the model. A case is eliminated from dataset when Cook's distance is larger than 4 / n (n is the number of cases in the dataset). Although Cook's distance based deletion is used when linear regression model is built, we applied it to analogy based estimation, because it is widely used in many effort estimation researches.

3.2 Mantel's Correlation Based Deletion

Mantel's correlation based deletion identifies an outlier when a set of independent variables' values is similar, but dependent variable's value is not similar to other cases. The method is originally proposed in Analogy-X method [6] designed for analogy based estimation. Analogy-X method is (1) delivering a statistical basis, (2) detecting a statistically significant relationship and reject non-significant relationships, (3) providing simple mechanism for variable selection, (4) identifying abnormal data point (project) within a dataset, and (5) supporting sensitivity analysis that can detect spurious correlations in a dataset. We applied function (3) as outlier deletion method.

While ordinary correlation coefficient like Pearson's correlation denotes strength of relationship between two variables, Mantel's correlation does between two set of variables (i.e. a set of independent variables and a dependent variable). Mantel's correlation clarifies whether development effort (dependent variable) is similar or not, when project attributes like duration or development size (a set of independent variable) is similar. To settle Mantel's correlation, Euclidean distance based on independent variables and Euclidean distance based on dependent variable is calculated, and then correlation coefficient of them is calculated.

Mantel's correlation based deletion identifies outliers by the following procedure.

1. For all projects, Mantel's correlation r_i is calculated by excluding *i-th* project.
2. Average of r_i ($\bar{r}$), and standard deviation of r_i (*rs*) are calculated.
3. Leverage metric lm_i, impact of *i-th* project on $\bar{r}$ is calculated by the following equation:

$$lm_i = r_i - \bar{r} \tag{5}$$

4. lm_i is divided by *rs*, and when the value (standard score) is larger than 4, the project is eliminated from dataset.

3.3 Neighborhood's effort based deletion

Our method, neighborhood's effort based deletion identifies an outlier when effort of a project is extremely higher or lower than other neighborhood projects. As stated in section 2, procedure of analogy based estimation consists of range normalization (step 1), neighborhood projects selection (step 2), and estimated effort computation (step 3). When effort of neighborhood projects is not homogeneous in step 2, estimation accuracy gets low [16]. Focusing on the issue, our method identifies an outlier in step 2. Analogy based estimation assumes when characteristics (independent variables' values) of project is similar, effort (dependent variable's value) is also similar. Our method treats a project as an outlier when the project is not fit to the assumption. While existing deletion methods eliminates outliers from entire dataset before estimation, our method eliminates outliers after selecting neighborhood projects.

To identify an outlier, effort of each neighborhood project is compared to average of neighborhood projects' effort. However, when variance of neighborhood projects' effort is large, each project's deviation from the average effort is also large. So Z-score computation [9] is applied to standardize each neighborhood's effort before the comparison. In more detail, our method eliminates outliers from k neighborhood projects as follows. Note that although neighborhood project's effort is denoted by y_i, y_i signifies size adjusted effort, not actual effort when using size adjustment method.

1. Average of y_i ($\bar{y}$) and standard deviation of y_i (*ys*) is calculated.
2. Standardized effort y'_i is calculated by the following equation (Z-score):

$$y'_i = \frac{y_i - \bar{y}}{ys} \tag{6}$$

3. A project is identified as an outlier and eliminated when absolute value of y'_i is greater than threshold *th* (i.e. when deviation from $\bar{y}$ is greater than *th* times *ys*). Note that if all neighborhood projects are identified as outliers, no project is eliminated. Estimated effort $\hat{y}_a$ is calculated by the following equation:

$$\hat{y}_a = \frac{\sum_{h \in Eliminated\,projects} y_h}{k - d} \tag{7}$$

In the equation, *Eliminatedprojects* denotes a set of k neighborhood projects excluded d outlier projects. We set *th* as 1.65 (one sided 5% of standard normal distribution).

Some researches pointed out that when productivity (development size / effort) of neighborhood projects is not homogeneous, estimation accuracy with size adjustment method gets low [7][13]. Our method with size adjustment is regarded to eliminate outliers based on productivity. When size adjustment is used, our method eliminates a project whose adjusted effort $adjy_i$ is extremely higher or lower. From equation (3), $adjy_i$ is calculated by multiplying estimated project's size fp_a by the reciprocal productivity y_i / fp_i (y_i is effort of a neighborhood project, and fp_i is development size of its). fp_a is same for all neighborhood projects, and therefore it is regarded as a constant. Therefore, $adjy_i$ is regarded as productivity in our method.

Our method is totally different from neighborhood selection such as distance-based neighborhood selection. When neighborhoods are selected, only independent variables are used and dependent variable is not used, because the value of dependent variable of the target project is unknown. On the contrary, our method is used independent variable.

4. EXPERIMENT

4.1 Dataset

To evaluate existing deletion methods and our method, we compared estimation accuracy of analogy based estimation when each outlier deletion method is applied. We used ISBSG dataset [5], which is provided by International Software Benchmark Standard Group (ISBSG). It includes project data collected from software development companies in 20 countries, and the projects were carried out between 1989 and 2004.

We assumed estimation point is the end of project plan phase. So, only variables whose values were fixed at the point were used as independent variables, although 99 variables are recorded in the dataset. The independent variables are same as the previous study [11] (unadjusted function point, development type, programming language, and development platform). Development type, programming language, and development platform were transformed into dummy variables, because they are nominal scale variables.

ISBSG dataset includes low quality project data (Data quality ratings are also included in the dataset). So we extracted projects based on the previous study [11] (Data quality rating is A or B, and function point was recorded by IFUPG method, and so on). Also, we excluded projects which included missing values. As a result, we used 593 projects.

4.2 Evaluation criteria

To evaluate accuracy of effort estimation, we used average and median of *AE* (Absolute Error), *MRE* (Magnitude of Relative Error) [4], *MER* (Magnitude of Error Relative to the estimate) [8], and *BRE* (Balanced Relative Error) [14].

When x denotes actual effort, and $\hat{x}$ denotes estimated effort, each criterion is calculated by the following equations:

$$AE = |x - \hat{x}| \tag{8}$$

$$MRE = \frac{|x - \hat{x}|}{x} \tag{9}$$

$$MER = \frac{|x - \hat{x}|}{\hat{x}} \tag{10}$$

$$BRE = \begin{cases} \dfrac{|x - \hat{x}|}{\hat{x}}, x - \hat{x} \geq 0 \\ \dfrac{|x - \hat{x}|}{\hat{x}}, x - \hat{x} < 0 \end{cases} \tag{11}$$

Lower value of each criterion indicates higher estimation accuracy. Intuitively, *MRE* means relative error to actual effort, and *MER* means relative error to estimated value. However, *MRE* and *MER* have biases for evaluating under and over estimation [3][10]. Maximum *MRE* is 1 even if terrible underestimate is occurred (For instance, when actual effort is 1000 person-hour, and estimated effort is 0 person-hour, *MRE* is 1). Similarly, maximum *MER* is smaller than 1 when overestimate is occurred. So in addition to *MRE* and *MER*, we adopted *BRE* whose evaluation is not biased both *MRE* and *MER* [15]. We did not use Pred(25) [4] which is sometimes used as an evaluation criterion, because Pred(25) is based on *MRE* and it has also bias for evaluating under estimation.

4.3 Experimental Procedure

Experimental procedure for existing deletion methods is follows:

1. Dataset is randomly divided into two equal set. One is treated as fit dataset, and the other is treated as test dataset. Fit dataset is used to compute estimated effort (regarded as past projects), and test dataset is used as estimation target (regarded as ongoing projects).
2. Outlier deletion method is applied to fit dataset, to eliminate outliers from fit dataset.
3. To decide neighborhood size *k*, estimation for fit dataset is performed, changing *k* from 1 to 20. After estimation, residual sum of square (It is same as sum of squares of *AE*, and widely used for estimation model selection [10]) is calculated, and *k* which shows smallest residual sum of square is adopted.
4. Estimation for test dataset is performed. *k* which is settled at step 3 is used.
5. Evaluation criteria are calculated by actual effort of test dataset and estimated effort.
6. Step 1 to 5 is repeated 10 times (As a result, 10 sets of fit dataset, test dataset, and evaluation criteria are made).

Experimental procedure for our method is follows:

1. Dataset is randomly divided into two equal set. One is treated as fit dataset, and the other is treated as test dataset.
2. Estimation for fit dataset is performed, changing *k* from 1 to 20. After estimation, residual sum of square is calculated, and *k* which shows smallest residual sum of square is adopted.
3. Estimation for test dataset is performed with our method. *k* which is settled at step 2 is used.
4. Evaluation criteria are calculated by actual effort of test dataset and estimated effort.
5. Step 1 to 4 is repeated 10 times.

5. RESULTS AND DISCUSSION

Experimental results are shown in Table 2 and Table 3. Table 2 denotes the values of evaluation criteria and deletion ratio when each outlier method is applied. Deletion ratio is defined as the number of deleted projects / the number of all projects. When our method is applied, it is defined as average of the number of deleted projects / neighborhood size. The values of evaluation criteria are average for 10 test dataset. Table 3 denotes differences of evaluation criteria between when each outlier deletion method is applied and not applied. Negative values mean evaluation criteria got worse by applying outlier deletion method. Table 3 also shows statistical test results for the difference by Wilcoxon signed-rank test (The values of evaluation criteria did not have normal distribution). Significant level was set as 5%, and italicized figures in the table signify there were significant differences.

When our method was applied, average *MER* got worse, but other criteria got better (Average *AE*, median *AE*, and median *MRE* showed significant difference). Especially, average *BRE* showed 20.8% improvement and median *BRE* did 5.3%. In case of Man-

Table 2. Evaluation criteria of each outlier deletion method

Outlier deletion method	Average *AE*	Median *AE*	Average *MRE*	Median *MRE*	Average *MER*	Median *MER*	Average *BRE*	Median *BRE*	Deletion ration
Not applied	3680	1607	166.93%	59.59%	91.27%	58.05%	210.36%	91.69%	0.00%
Neighborhood's effort	3252	1374	138.07%	55.98%	97.35%	56.67%	189.56%	86.36%	7.75%
Mantel's correlation	3549	1603	162.40%	60.40%	92.61%	58.48%	207.17%	92.01%	1.21%
Cook's distance	3371	1611	154.69%	62.50%	110.35%	58.80%	216.35%	99.05%	4.65%

Table 3. Difference of evaluation criteria of each outlier deletion method

Outlier deletion method		Average *AE*	Median *AE*	Average *MRE*	Median *MRE*	Average *MER*	Median *MER*	Average *BRE*	Median *BRE*
Neighborhood's effort	Difference	428	234	28.9%	3.6%	-6.1%	1.4%	20.8%	5.3%
	p-value	*0.03*	*0.00*	0.06	*0.03*	0.43	0.70	0.08	0.06
Mantel's correlation	Difference	131	4	4.5%	-0.8%	-1.3%	-0.4%	3.2%	-0.3%
	p-value	0.08	0.58	*0.01*	0.25	0.43	0.74	0.19	0.84
Cook's distance	Difference	309	-4	12.2%	-2.9%	-19.1%	-0.8%	-6.0%	-7.4%
	p-value	*0.01*	0.70	0.06	0.20	0.06	0.85	0.43	0.77

tel's correlation based deletion, four evaluation criteria out of eight were improved, and lowering of the other criteria were smaller than 1.3%. However, the extent of improvement was 3.2% on average *BRE*, and it was very small on median *BRE* (Only average *MRE* showed significant difference). When Cook's distance based deletion was applied, only two evaluation criteria out of eight were improved, and moreover, degradation of average *BRE* and median *BRE* were more than 6%.

We should carefully understand the results because we used only one dataset, but at least, our method shows high performance to eliminate outliers toward a certain kind of dataset (i.e. ISBSG dataset), and we could say that our method is promising. The effect of Mantel's correlation based deletion is not very strong when applied to ISBSG dataset. The result does not means Mantel's correlation based deletion is always not effective to eliminate outliers, but means it is not very effective to a certain kind of dataset. Applying Cook's distance based deletion made estimation accuracy lower. Cook's distance based deletion may be effective to other dataset, but it is not fit to a certain kind of dataset, and therefore we should consider whether Cook's distance based deletion apply or not before using analogy based estimation.

While existing deletion methods identifies outliers from whole dataset, our method does from neighborhood projects of the estimated project. The characteristics of our method may be effective especially when it is applied to dataset like ISBSG dataset which includes various projects.

6. CONCLUSIONS

In this research, we applied outlier deletion methods to analogy based software development effort estimation, and evaluated their effects. Also, we propose new outlier deletion method for analogy based estimation. While existing deletion methods eliminates outliers from entire dataset before estimation, our method does after neighborhood projects are selected by analogy based estimation. In our method, when the effort of the project is extremely higher or lower than other neighborhood projects, it is not used for effort estimation. In the experiment, we estimated development effort using ISBSG dataset, and in the results, our method is most effective, Mantel's correlation based deletion is not very effective, and Cook's distance based deletion made estimation accuracy lower. As future work, we will apply other deletion methods to other dataset and compare their effects to enhance reliability of our research.

7. ACKNOWLEDGMENTS

This work is being conducted as a part of the StagE project, The Development of Next-Generation IT Infrastructure, and Grant-in-aid for Young Scientists (B), 22700034, 2010, supported by the Ministry of Education, Culture, Sports, Science and Technology. We would like to thank Dr. Naoki Ohsugi for offering the collaborative filtering tool.

8. REFERENCES

[1] Angelis, L., and Stamelos, I. 2000. A Simulation Tool for Efficient Analogy Based Cost Estimation, *Empirical Software Engineering*, 5, 1, 35-68.

[2] Boehm, B. 1981. *Software Engineering Economics*. Prentice Hall.

[3] Burgess, C., and Lefley, M. 2001. Can genetic programming improve software effort estimation? A comparative evaluation. *Journal of Information and Software Technology*, 43, 14, 863-873.

[4] Conte, S., Dunsmore, H., and Shen, V. 1986. *Software Engineering, Metrics and Models*. Benjamin/Cummings.

[5] International Software Benchmarking Standards Group (ISBSG). 2004. ISBSG Estimating: Benchmarking and research suite. ISBSG.

[6] Keung, J., Kitchenham, B., and Jeffery, R. 2008. Analogy-X: Providing Statistical Inference to Analogy-Based Software Cost Estimation. *IEEE Trans. on Software Eng*. 34, 4, 471-484.

[7] Kirsopp, C., Mendes, E., Premraj, R., and Shepperd, M. 2003. An Empirical Analysis of Linear Adaptation Techniques for Case-Based Prediction. In *Proc. of International Conference Case-Based Reasoning*, Trondheim, Norway, June 2003, 231-245.

[8] Kitchenham, B., MacDonell, S., Pickard, L., and Shepperd, M. 2001. What Accuracy Statistics Really Measure. In *Proc. of IEE Software*, 148, 3, 81-85.

[9] Larsen, R., and Marx, M. 2000. *An Introduction to Mathematical Statistics and Its Applications*. Prentice Hall.

[10] Lokan, C. 2005. What Should You Optimize When Building an Estimation Model?, In *Proc. of International Software Metrics Symposium (METRICS)*, Como, Italy, Sep. 2005, 34.

[11] Lokan, C., and Mendes, E. 2006. Cross-company and single-company effort models using the ISBSG Database: a further replicated study, In *Proc. of the International Symposium on Empirical Software Engineering (ISESE)*, Rio de Janeiro, Brazil, Sep. 2006, 75-84.

[12] Mendes, E., Martino, S., Ferrucci, F., and Gravino, C. 2008. Cross-company vs. single-company web effort models using the Tukutuku database: An extended study. *The Journal of Systems and Software*, 81, 5, 673-690.

[13] Mendes, E., Mosley, N., and Counsell, S. 2003. A Replicated Assessment of the Use of Adaptation Rules to Improve Web Cost Estimation. In *Proc. of the International Symposium on Empirical Software Engineering (ISESE)*, Rome, Italy, September 2003, 100-109.

[14] Miyazaki, Y., Terakado, M., Ozaki, K., and Nozaki, H. 1994. Robust Regression for Developing Software Estimation Models. *Journal of Systems and Software*, 27, 1, 3-16.

[15] Mølokken-Østvold, K., and Jørgensen, M. 2005. A Comparison of Software Project Overruns-Flexible versus Sequential Development Models, *IEEE Trans. on Software Eng*. 31, 9, 754-766.

[16] Ohsugi, N., Monden, A., Kikuchi, N., Barker, M., Tsunoda, M., Kakimoto, T., and Matsumoto, K. 2007. Is This Cost Estimate Reliable? - the Relationship between Homogeneity of Analogues and Estimation Reliability. In *Proc. of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Madrid, Spain, September 2007, 384-392.

[17] Selby, R., and Porter, A. 1988, Learning from examples: generation and evaluation of decision trees for software resource analysis. *IEEE Trans. on Software Eng*. 14, 12, 743-757.

[18] Seo, Y., Yoon, K., and Bae, D. 2008. An Empirical Analysis of Software Effort Estimation with Outlier Elimination. In *proc. of the international workshop on Predictor models in software engineering (PROMISE)*, Leipzig, Germany, May 2008, 25-32.

[19] Shepperd, M., and Schofield, C. 1997. Estimating software project effort using analogies. *IEEE Trans. on Software Eng*. 23, 12, 736-743.

[20] Srinivasan, K., and Fisher, D. 1995. Machine Learning Approaches to Estimating Software Development Effort. *IEEE Trans. on Software Eng*. 21, 2, 126-137.

[21] Tosun, A., Turhan, B., and Bener, A. 2009. Feature weighting heuristics for analogy-based effort estimation models. *Expert Systems with Applications*, 36, 7, 10325-10333.

[22] Walkerden, F., and R. Jeffery. 1999. An Empirical Study of Analogy-based Software Effort Estimation. *Empirical Software Engineering*, 4, 2, 135-158.

A Survey of Public Datasets for Comparative Effort Prediction Studies

Sousuke Amasaki
Okayama Prefectural University
111 Kuboki, Soja
Okayama, Japan
amasaki@cse.oka-pu.ac.jp

Tomoyuki Yokogawa
Okayama Prefectural University
111 Kuboki, Soja
Okayama, Japan
t-yokoga@cse.oka-pu.ac.jp

ABSTRACT

Background: In the past survey, available public datasets for effort prediction study were listed up. In addition, using sufficient number of datasets and statistical tests was recommended for valid comparative effort prediction study. **Aim:** This paper aims to identify a set of useful public datasets in terms of comparative effort prediction study. **Method:** We sieved 38 public datasets listed in the past study and PROMISE repository with respect to their availability, variety of features, and applicability of statistical practice. **Results:** Only 12 public datasets were found to be available and useful for simple models. For complex models 8 of 12 datasets were specified. Among 8 datasets, only 3 datasets could keep original sample size and feature scale. This is because newly proposed effort prediction methods usually try to improve performance by using a multiple predictors and many datasets included categorical features with a level which was not found frequently. To obtain more datasets, it was needed to adapt remained 5 datasets by case reduction, feature reduction, or numeric conversion. **Conclusions:** It is difficult to compare multiple effort prediction methods with a large number of public datasets along with good statistical practice. However, selected datasets must be used for valid study at least.

Categories and Subject Descriptors

D.29 [**Software Engineering**]: Cost Estimation

General Terms

Economics

1. INTRODUCTION

Software cost estimation is still popular and important research area. To evaluate performance of a proposed software estimation model construction method, datasets from real projects are used in an experiment.

Although it is usually difficult to obtain a dataset from real projects, we can find public datasets in the past study. There are many public datasets available to researchers for model evaluation. PROMISE repository[3] now serves 18 downloadable datasets. In [14], the authors identified 31 datasets freely available to researchers from publications.

One of criticisms for cost estimation papers is the small number of datasets used for evaluation. In [11], it is thus recommended to use more public datasets shown in [14] and [3]. However, examination of that listing is needed because comparative effort prediction study was not considered in [14] and [3]. For example, existence of size-related metric such as lines of code(LOC), Function Points(FP), and effort record was not checked while those are essential information for models.

In this paper, we thus examined datasets listed in PROMISE repository[3] and [14] in order to specify common datasets available and useful for comparative effort prediction study. As a result of examination, 12 datasets were found to be available and useful for simple models. For complex models 8 of 12 datasets were specified. For valid study, these datasets must be used at least.

2. COMPARATIVE STUDY DESIGN

In comparative study, the followings must be considered.

- Simple and Complex models
- Benchmark
- Evaluation procedure

Most of newly proposed software estimation model construction methods utilized predictors in order to improve predictive performance. For example, feature weighting method[1] for Estimation by Analogy(EbA)[18] was proposed to improve EbA by weighing effects of predictors differently. We called models with multiple features complex models in this paper. On the other hand, simple models including only effort and size-related metric are also popular because it is easy to evaluate new methods and to use in case of small sample size. Both types of models are important but their requirements for dataset are different. We thus selected datasets for simple and complex models differently.

Benchmark is important for comparative study to evaluate how well performance of a newly proposed model is. For this purpose, conventional models such as linear regression and EbA are often used. In fact, most comparative study used one of these methods as benchmark. Kitchenham recommended to confirm whether a new proposed model outperformed linear regression at least[11]. Both models can be used for simple and complex models. In this

study, we thus supported that linear regression and EbA are used as benchmarks.

In comparative study, two types of cross-validation[7] were often used as evaluation procedure: leave-one-out and K-fold cross-validation (CV). Leave-one-out CV can be used for smaller datasets and it is preferable to K-fold CV in terms of variety of datasets. It was also recommended in [11] because of its deterministic property.

On the other hand, K-fold CV is preferable to leave-one-out CV in terms of evaluation reliability because an estimate obtained using leave-one-out CV has high variance, leading unreliable estimates[5]. Furthermore, some performance measures for effort estimation study cannot be differentiated when leave-one-out CV is adopted. In each round of leave-one-out CV, one test case is used for calculating performance measures. That is, *MMRE* and *MdMRE* [17] of each test set always have identical value. In case of K-fold CV, *MMRE* and *MdMRE* have different values if each test set has more than 2 cases.

In [11], bootstrapping[6] was also recommended for performance comparison based on statistical tests. On the other hand, Kohavi recommended 10-fold CV after comparing to bootstrap[13]. Furthermore, property of resampling with replacement used in bootstrapping seems unsuitable for EbA. When a replicated project is placed in both training and test sets, identical project is always selected as the nearest neighbor in EbA. If the number of neighbors is set to 1 for EbA, EbA can estimate exact effort for replicated projects.

In this study, we considered 10×10-fold CV followed with t-test as evaluation procedure in order to avoid inflated Type I error[4].

3. DATASETS

In this study, we examined published datasets served on PROMISE repository[3] at November 2010 or listed in [14]. PROMISE repository serves datasets as readable text file which can include comments. From this repository, we selected datasets which contain a comment regarding information of projects or a citation they were used. Datasets from [14] were collected from three journals: Transactions on Software Engineering, Information & Software Technology, and Journal of Systems & Software. After removing duplicates, we prepared an initial set of 38 datasets shown in Table 1.

We then checked these datasets in terms of availability. First and second columns of Table 1 show whether a dataset can be obtained actually from referenced papers in [14] or PROMISE repository. If both columns of a dataset were marked as 'N', this dataset is not easily available.

We found that 7 datasets were not easily obtained from referenced papers and PROMISE repository. ID 3, 4, and 30 were not shown in an original paper[2] cited in [14] and this paper said that an old book published in 1986 contains these datasets. We assessed ID 12 as unavailable though it was registered in PROMISE repository. This is because the number of features in this dataset was very fewer than that described in a referenced paper[18]. The number of features was 29 according to [18] while dataset held on PROMISE repository included only 6 features (excluding logarithm of effort and size.) Referenced papers of the others did not have a dataset, a pointer to a paper, nor a book which may include corresponding datasets.

In the following sections, remained 31 datasets were examined.

4. SELECTION FOR SIMPLE MODELS

Simple models usually estimate effort from single size-related metric such as KSLOC(kilo source LOC) and FP. Thus, size-related metric must be included in a dataset. Sample size is also important for evaluation procedure and reliability of results.

Datasets were selected by evaluation criteria related to these points.

4.1 Effort and Size Records

Third and fourth columns of Table 1 show whether a dataset includes effort and size records. Here, unavailable 7 datasets in the previous section were ignored and blanks were placed.

We found that 6 datasets did not include effort or size metric. ID 25, 26, and 27 did not include effort records. These datasets only included KSLOC and size related-metrics such as the number of screens and were used to construct sizing models[15]. ID 5 included 3 features: actual effort, expert estimates, and company labels. ID 18 and 19 only included actual effort and experts estimates. These datasets were not suitable for comparative effort estimation study because an expert estimate was rarely used as the only predictor in a software estimation model.

4.2 Sample Size

In many situations, a fitted regression model is likely to be reliable when the number of predictors p (the number of candidate predictors if using variable selection) is less than $m/10$ or $m/20$, where m is total sample size[7]. Following this rule, at least, a dataset has to include more than 10 projects in case of simple models. Furthermore, the size of training subsets becomes smaller when the size of training set in evaluation procedure is considered. In case of 10-fold CV, sample size N must be $N > 12$ because $9/10 \cdot N > 10$ must be held.

To make performance measures such as *MMRE* and *PRED(25)*[17] reliable, the size of test set must be larger as described in Section 2. If the size of test set must be always equal or larger than 3, though performance measure based on 3 results is still less precise and less fine-grained, sample size must be equal or larger than 30 because $1/10 \cdot N \geq 3$ must be held.

Fifth column of Table 1 shows whether a dataset has the sufficient number of projects. Here, we marked as 'Y' if a dataset had 30 or more projects; otherwise we marked as 'N' with its size. This column indicated that 13 datasets were too small to be used in comparative effort prediction study.

Finally, 12 public datasets were remained for comparative effort prediction study of simple models.

5. SELECTION FOR COMPLEX MODELS

Complex models assume multiple predictors. For instance, CART selects and combines predictors so that a software effort estimation model achieves higher prediction accuracy. Sample size is more important for reliable evaluation of complex models. Furthermore, types of candidate predictors are also important. If there is discrete variables, using k-fold CV may be difficult.

Table 1: Public datasets from [14] and [3]

ID	Name	Paper	PROMISE	Effort	Size	Sample Size	Multiple Features	Categorical
1	Abran-Robillard	Y	N	Y	Y	N(21)		
2	Albrecht-Gaffney	Y	Y	Y	Y	N(24)		
3	Bailey-Basili	N	N					
4	Belady-Lehmann	N	N					
5	Shepperd-Cartwright	Y	N	Y	N			
6	BT-software-houses	Y	N	Y	Y	N(10)		
7	BT-systemX	Y	N	Y	Y	N(10)		
8	COCOMO	N	Y	Y	Y	Y	Y(16)	Y(16)
9	CSC	Y	N	Y	Y	Y	Y(2)	Y(2)
10	Desharnais	N	Y	Y	Y	Y	Y(7)	Y(1)
11	Dolado	Y	N	Y	Y	N(24)		
12	Finnish	N	N					
13	Hastings-Sajeev	Y	N	Y	Y	N(8)		
14	Heiat-Heiat	Y	N	Y	Y	Y	Y(1)	N
15	ICL	Y	N	Y	Y	N(10)		
16	Jørgensen97-A	Y	N	Y	Y	N(16)		
17	Jørgensen97-B	Y	N	Y	Y	N(20)		
18	Jørgensen04-X	Y	N	Y	N			
19	Jørgensen04-Y	Y	N	Y	N			
20	Kemerer	Y	Y	Y	Y	N(15)		
21	MERMAID1	N	N					
22	MERMAID2	Y	N	Y	Y	Y	Y(1)	Y(1)
23	Misic-Tesic	Y	N	Y	Y	N(7)		
24	Miyazaki et al.1	Y	Y	Y	Y	Y	Y(7)	Y(1)
25	Miyazaki et al.2	Y	N	N	Y			
26	Miyazaki et al.3	Y	N	N	Y			
27	Miyazaki et al.5	Y	N	N	Y			
28	Moser-etal	Y	N	Y	Y	Y	Y(1)	Y(1)
29	Telecom	Y	Y	Y	Y	N(18)		
30	Wingfield	N	N					
31	WSD1	N	N					
32	WSD2	N	N					
33	cocomonasa_v1		Y	Y	Y	Y	Y(15)	Y(15)
34	cocomo_sdr		Y	Y	Y	N(12)		
35	Maxwell		Y	Y	Y	Y	Y(22)	Y(22)
36	NASA93		Y	Y	Y	Y	Y(20)	Y(20)
37	usp05		Y	Y	Y	Y	Y(14)	Y(14)
38	usp05ft		Y	Y	Y	Y	Y(12)	Y(12)

We evaluated 12 datasets selected in the previous section by criteria related to these points.

5.1 Multiple Features

Complex models utilize multiple features. Therefore datasets must include multiple features for comparative effort prediction study.

Seventh column of Table 1 shows whether a dataset includes at least one feature other than KSLOC and FP-variants. The number following 'Y' indicates the number of features. For example, Desharnais (ID 10) has 7 features because unadjusted and adjusted FP were ignored and elements such as Transactions were counted. Inappropriate features such as duration were also ignored.

As a result, all 12 datasets were remained.

5.2 Sample Size

Usually the size of datasets for complex models must be larger than that for simple models. For instance, if linear regression models using size-metric and one feature as predictors are considered as benchmark, a dataset must include more than 23 projects according to the rule described in subsection 4.2.

Here, we left all 12 public datasets because they all have equal or more than 30 projects and complex models based on linear regression with two predictors can be constructed at least. Although additional samples might be needed for other models, we did not consider that case here.

5.3 Categorical Features

Ten-fold CV can be applied to a dataset with categorical features if each training subset includes all levels of each categorical feature included in corresponding testing subset. To satisfy this condition, each level of a categorical feature must be found in more than $N/10$ records. If there is categorical feature with a level which was found less than $N/10$ times, case reduction, feature reduction, or numeric conversion is needed.

Eighth column in Table 1 shows whether a dataset includes categorical feature(s) and the number of them. Datasets having categorical features can be classified into two groups:

- a few categorical variable(s) are included(5 datasets) and
- many categorical variables are included(6 datasets).

In the first group, Desharnais(ID 10) and MERMAID2(ID 22) have one categorical feature which satisfies the condition for levels. A categorical feature of Desharnais dataset has three levels each of which was found in more than one-eighths of records. A categorical feature of MERMAID2 dataset also has two levels each of which was found in at least one-thirds of records. Thus, they can be used for comparative study as is.

CSC(ID 9), Miyazaki1(ID 24), and Moser-etal(ID 28) datasets have one or two categorical features with a level which was found less than $N/10$ times. These categorical features were not used as predictor in original papers[12, 15, 16].

Miyazaki1 has one categorical feature representing companies. For this dataset, case reduction is useless because there is no level with more than $N/10$ records. Concatenation of levels or numeric conversion cannot be applied because this feature is nominal scale and there is no knowledge of relations among companies. Thus, we concluded that feature reduction is the best way for comparative study. We think that this feature has too many levels for sample size and is useless for estimation.

Moser-etal has one categorical feature representing project characteristics. If case reduction is applied, the size of this dataset becomes lower than 30. Concatenation of levels or numeric conversion cannot be applied because it is nominal scale. Feature reduction makes this dataset inappropriate for complex models because it has only one feature.

CSC has two categorical features representing clients and project types. Case reduction decreased the size of this dataset to 106. It also reduced the numbers of client types and project types to 1 and 2, respectively. Thus, one categorical feature of project types was remained. In [12], the authors considered simple models and used several types of partitioning based on categorical features. On the other hand, we considered complex models and thus case reduction was prefer to feature reduction.

In the second group, all datasets have multiple categorical features with a level which was found less than $N/10$ times. It is difficult to adapt these datasets to comparative study by case reduction. Because there are many categorical features and case reduction based on a categorical feature may influence on the other features. Furthermore, large portion of cases may be reduced for satisfying equal to or more than $N/10$ cases for all levels. Concatenation and feature reduction are also difficult because there is no general way for all possible effort estimation models. Thus, numeric conversion was first considered for this group.

Most of categorical features in COCOMO(ID 8), cocomonasa_v1 (ID 33), and NASA93(ID 36) were COCOMO or COCOMO II attributes. For these attributes, adjustment factors were often used in the past study. With adjustment factors, 10-fold CV can be applied to COCOMO and cocomonasa_v1.

NASA93(ID 36) dataset included some nominal scale features having a level which was found less than $N/10$ times. For development mode among these features, case reduction is helpful because this feature can satisfy $N/10$ rule by removing one level with 3 projects. Other nominal features were looked as less important for software estimation models. In fact, they were not used in the past study. Thus, we concluded that NASA93 dataset can be used after numeric conversion, small case reduction, and feature reduction are applied.

Maxwell(ID 35) dataset contains ordinal features and nominal features. Numeric conversion can be applied to ordinal features. Four out of 7 nominal features have a level which was found less than $N/10$ times. All of them were looked important and feature reduction was not preferable to case reduction. Case reduction based on these features reduced the size of this dataset from 62 to 46. Three nominal features were also removed because they have only one level in this reduced dataset.

Many features in usp05 and usp05ft datasets(ID 37 and 38) were categorical with many levels and it is difficult to determine the way for adapting it to comparative study. Thus, we concluded that 10-fold CV cannot be applied for these datasets.

As a result, 9 datasets were remained.

5.4 Detailed Examination

Desharnais, Heiat-Heiat, MERMAID2, and Miyazaki1 are preferable to the other datasets. Because they could keep original sample size and feature scale. Here, we examined 4 datasets selected in terms of usefulness and reliability. Since Desharnais has already been examined in [11], we examined other 3 datasets.

5.4.1 Heiat-Heiat dataset[8]

This dataset came from projects completed for private and public organizations in Billings, Montana. There was no missing value. These projects were selected on the basis of 6 attributes so that the dataset consists of homogeneous projects.

Figure 1 shows a scatterplot of FP and Effort for all 35 projects. There are no apparent skewness, no heteroscedacity, and no outlier.

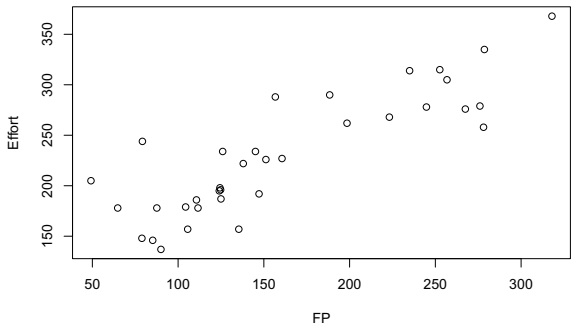


Figure 1: Heiat-Heiat dataset

This dataset included a feature representing the sum of total number of corrected relationships at data stores and total number of

data flows which connect the system to external entities. Heiat et al. proposed this metric, called REIO, as alternative measure of size such as LOC[8]. In fact, REIO and LOC were highly correlated (Spearman's ρ is 0.93.) These facts imply that the difference between simple and complex models will be subtle.

Thus, we concluded that this dataset was inappropriate for comparative effort prediction study in terms of variety of features.

5.4.2 Miyazaki1 dataset[15]

This dataset came from 48 projects in 20 companies. There is no missing value. All systems were newly developed except for one project, in which more than half of its code was adapted from an existing system[15].

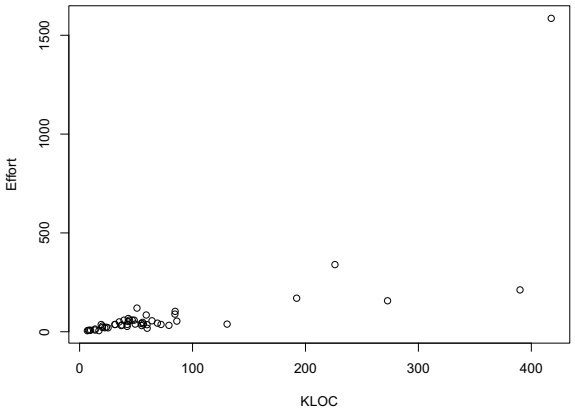


Figure 2: Miyazaki1 dataset

Figure 2 shows a scatterplot of KLOC and Effort. There is one outlier. There is skewness and small heteroscedacity. It looks as if all projects except for the outlier have similar relationships between size and effort.

Six continuous features in this dataset can be classified into three pairs with respect to their semantic similarity. Features of each pair were also highly correlated. Even if a half of features were dropped by a software estimation model construction method, the others still remain.

In this study, we concluded that the dataset may be useful for comparative effort prediction study because there is room for model selection and optimization.

5.4.3 MERMAID2 dataset[10]

This dataset has one categorical feature representing new projects and enhancement projects. Here, dividing the dataset into two project types is not appropriate approach because the number of new projects is less than 10.

There is two missing values on a categorical value. If raw FP is used instead of adjusted FP, other 2 projects must also be removed because of missing values. Figure 3 shows a scatterplot of adjusted FP and Effort of 28 projects. The dataset has two outliers[10]. There is skewness and heteroscedacity.

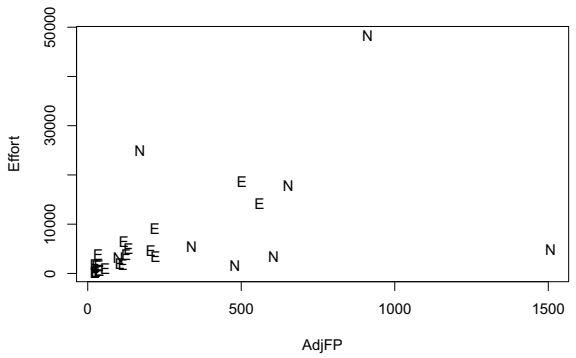


Figure 3: Mermaid2 dataset (N = New projects, E = Enhancement Projects)

From these characteristics, we concluded this dataset was sometimes inappropriate for comparative effort prediction study. Because this dataset is too small to perform 10-fold CV when missing values must be removed. We determined to leave this dataset.

Table 2 shows remained 8 datasets. Size and the number of features are from reduced datasets. Marks 'FR', 'CR', and 'NC' indicate feature reduction, case reduction, and numeric conversion, respectively.

As a result of examination, available 38 public datasets were decreased to 12 datasets for simple models and to 8 datasets for complex models.

6. DISCUSSION

In subsection 5.3, several types of preprocessing was applied to datasets in order to adapt them to comparative effort prediction study. These preprocessing have drawbacks.

The problems of case reduction are decreasing of size and change of population. In this study all selected datasets have sufficient number of projects and the size was not problem. Case reduction in this study was based on categorical feature having a level which was found less than $N/10$ times. Thus, population was changed but it becomes more homogeneous. Homogeneity of dataset is considered as important property for effort estimation.

Most of reduced features in this study were due to case reduction. Thus, dropped features seemed less important at least for reduced homogeneous datasets.

Although using adjustment factors is like a calibration, not construction of software estimation model, it was popular method for dealing COCOMO drivers. Numeric conversion of ordinal scales was also popular though this conversion implicitly assumes linear relationship among levels. To manage these problems, feature subset selection may be useful. When all selected categorical features satisfy $N/10$ rule, these features can be used as is.

This study did not use technique concatenating levels though using it instead of case reduction may also be useful. Because it is difficult to automate concatenation process because human decision

Table 2: Selected Datasets for Complex Models

No.	Name	Size	Features	FR	CR	NC
8	COCOMO	63	16			X
9	CSC	106	1	X	X	
10	Desharnais	77	7			
22	MERMAID2	30	1			
24	Miyazaki1	48	6	X		
33	cocomonasa_v1	60	15			X
35	Maxwell	46	19	X	X	X
36	NASA93	90	16	X	X	X

plays an important role as shown in [9].

7. CONCLUSION

In this paper, we examined the list of public datasets in [14] and PROMISE repository. As a result, we found 12 of 38 datasets were only appropriate for comparative effort prediction study of simple models. For complex models, this number was decreased to 8. Among 8 datasets, only 3 datasets could keep original sample size and feature scale. We thus concluded that using one or two public datasets is insufficient but there is not so many datasets suitable for comparative study.

To make effort prediction study of complex models more comparable, we recommend to use those 3 datasets at first. In practical view, CSC and MERMAID2 has too few features to evaluate complex models such as CART though. In this case, it is recommended to consider remained 5 datasets next to Desharnais. The fact that many datasets need to be changed for use implies that repositories like PROMISE should provide detailed and recommend preprocessing sequence for removing heterogeneity among comparative studies.

8. REFERENCES

[1] M. Auer and S. Biffl. Increasing the accuracy and reliability of analogy-based cost estimation with extensive project feature dimension weighting. In *Proc. of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*, pages 147–155, 2004.

[2] R. D. Banker, H. Chang, and C. F. Kemerer. Evidence on economies of scale in software development. *Information and Software Technology*, 36:275–282, 1994.

[3] G. Boetticher, T. Menzies, and T. Ostrand. PROMISE repository of empirical software engineering data. `http://promisedata.org/repository`, West Virginia University, Department of Computer Science, 2007.

[4] R. R. Bouckaert. Choosing between two learning algorithms based on calibrated tests. In *Proc. of 20th International Conference on Machine Learning*, pages 51–58, 2003.

[5] B. Efron. Estimating the error rate of a prediction rule: Improvement of cross-validation. *Journal of the American Statistical Association*, 78:316–330, 1983.

[6] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, 1993.

[7] F. E. Harrel. *Regression Modeling Strategies*. Springer, 2001.

[8] A. Heiat and N. Heiat. A model for estimating efforts required for developing small-scale business applications. *Journal of Systems and Software*, 39:7–14, 1997.

[9] B. Kitchenham. A procedure for analyzing unbalanced datasets. *IEEE Trans. on Software Engineering*, 24(4):278–301, 1998.

[10] B. Kitchenham. The question of scale economies in software — why cannot researchers agree? *Information and Software Technology*, 44(1):13–24, 2002.

[11] B. Kitchenham and E. Mendes. Why comparative effort prediction studies may be invalid. In *Proc. of PROMISE Workshop 2009*, 2009.

[12] B. Kitchenham, S. Pfleeger, B. McColl, and S. Eagan. An empirical study of maintenance and development estimation accuracy. *Journal of Systems and Software*, 64:57–77, 2002.

[13] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of International Joint Conference on AI*, pages 1137–1145, 1995.

[14] C. Mair, M. Shepperd, and M. Jørgensen. An analysis of data sets used to train and validate cost. In *Proc. of 1st International Workshop on Predictor Models in Software Engineering(PROMISE'05)*, 2005.

[15] Y. Miyazaki and M. Terakado. Robust regression for developing software estimation models. *Journal of Systems and Software*, 27(1):3–16, 1994.

[16] S. Moser, B. Henderson-Sellers, and V. B. Mišić. Cost estimation based on business models. *Journal of Systems and Software*, 49:33–42, 1999.

[17] D. Port and M. Korte. Comparative studies of the model evaluation criterions MMRE and PRED in software cost estimation research. In *Proc. of 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08)*, 2008.

[18] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Trans. on Software Engineering*, 23(11):736–743, 1997.

Faults and Verification

Reconstructing Fine-Grained Versioning Repositories with Git for Method-Level Bug Prediction

Hideaki Hata
Dept.of Information Systems Engineering
Osaka University
Osaka, Japan
h-hata@ist.osaka-u.ac.jp

Osamu Mizuno
Dept. of Information Science
Kyoto Institute of Technology
Kyoto, Japan
o-mizuno@kit.ac.jp

Tohru Kikuno
Dept.of Information Systems Engineering
Osaka University
Osaka, Japan
kikuno@ist.osaka-u.ac.jp

ABSTRACT

Change metrics derived from software repositories are known to be effective for bug prediction. However, bug prediction based on change metrics is limited to file-level because existing SCM repositories stores file histories. To tackle fine-grained (method-level) bug prediction, we try to establish fine-grained versioning for repositories. This paper presents an automatic technique constructing software repositories that can manage method histories. We utilize Git, one software configuration management (SCM) systems, and reconstruct fine-grained versioning repositories from existing repositories. We utilize these method-level versioning repositories for method-level bug prediction with one eclipse related project. The result shows that method-level change metrics are effective for method-level bug prediction.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.8 [**Software Engineering**]: Metrics; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Management

Keywords

fine-grained versioning, software repository, bug prediction, method-level, Git, software evolution

1. INTRODUCTION

Software configuration management (SCM) repositories have been widely targeted for many research areas because of their rich product and process related data. Bug prediction is one of those research areas. Many studies reported that change metrics derived from software repositories are effective for bug prediction [9, 11, 13, 20, 23, 25].

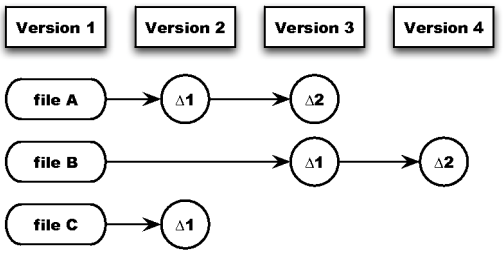
SCM systems can track the history of files and record, for example, existence periods, previous operations, and the reasons of the changes. Therefore, change metrics, such as age, number of changes, bug fixes are easily available from software repositories. Previous studies conducted file-level bug prediction based on file histories. In this paper, we tackle method-level bug prediction based on method histories because we think it is easy to localize bugs if the size of predicted modules is smaller.

From existing software repositories, seeing method histories is not so easy. It is reported that though there are lots of refactorings, which include the file relocations and method signature changes, developers do not always leave in commit logs enough information describing the changes they did in commit logs [21]. Since general repositories do not control method histories explicitly, we cannot obtain rich information for methods as for files. In this paper, we prepare method-level version control repositories before mining change metrics.

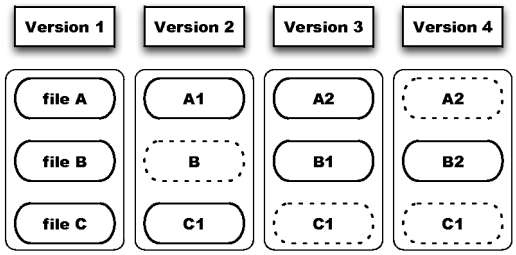
The concept of fine-grained version control can be seen in Orwell [27] as method-level version control in object-oriented programming. Though several tools have been proposed to support fine-grained versioning, no such a tool has been actually integrated within widely used SCM systems [6]. Since existing repositories remain file-level versioning, what we have to do is converting existing repositories into fine-grained versioning repositories. Though there are some studies tackling extraction of method histories from repositories, we try not only to extract method histories but also to controlling method histories. For the purpose of converting, we utilize Git, one SCM system.

Herraiz et al. presented software repository evolution from CVS, Subversion to Git, and insisted that those improvements made repositories research friendly [16]. Bird et al. studied both promise and peril of Git [3]. Though Git is known for the decentralization of source code management, we found that Git architecture is also useful for our purpose. "Directed acyclic graph (DAG)," "origin analysis," and "revisionist history," which are Git features explained by Bird et al. [3], are also desirable for our purpose. In addition, "snapshots" is also an important feature.

The rest of the paper is structured as follows. Section 2 introduces four Git features, which is important for our purpose of reconstruction. Then, Section 3 presents a technique of reconstructing repositories from existing repositories based on Git. Section 4 shows usefulness of our fine-grained versioning repositories with a case study with an open source project. In addition, we utilize fine-grained versioning for method-level bug prediction. Section 5 introduces



(a) Difference-based storing (Subversion)



(b) Storing snapshots (Git)

Figure 1: Differences in storing version data

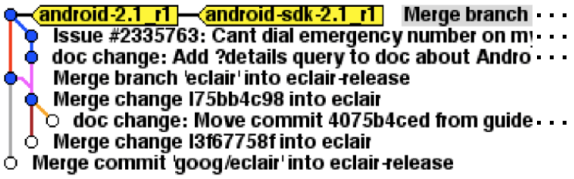


Figure 2: A subgraph of the Git DAG of commits leading up to 2.1 release 1 of Android

related work. Finally, we concluded this paper in Section 6.

2. GIT FEATURES

Recently Git attracted the attention of some researchers [3, 14, 16]. Though decentralization is one of the main characteristics compared to Subversion, there are some other distinctions. In this section, we introduce four Git features that are desirable for reconstructing fine-grained versioning repositories.

2.1 Snapshots

One of the major differences between Git and Subversion (or CVS) is the way it stores version data. Figure 1 shows an example of version control of three files. The file A is changed in version 2 and version 3, the file B is changed in version 3 and 4, and the file C is changed in version 2. Subversion stores a set of files and the changes made to each file over time, as illustrated in Figure 1 (a). Figure 1 (b) shows how Git stores data. Git stores each snapshot of what all files look like at each commit. To be efficient, if files have not changed, Git does not store the files but stores links to the previous files that have been already stored. What is import is that each file is stored without begin connected to files in the previous version.

2.2 DAG

As explained in [3], parent-child relation in commits is represented as a graph in Git. A commit have more than or equal to one parent except for the initial commit. Since no commit cannot be its own ancestor, the graph is a directed acyclic graph (DAG). Figure 2 shows an example of commit DAG. As shown in the example, merging commits have more than one parent commit. As seen in Figure 1 (b), each version of one file is stored in each snapshot separately. Each commit knows its snapshot. The orders or relations of file versions are kept as DAG of commits relation.

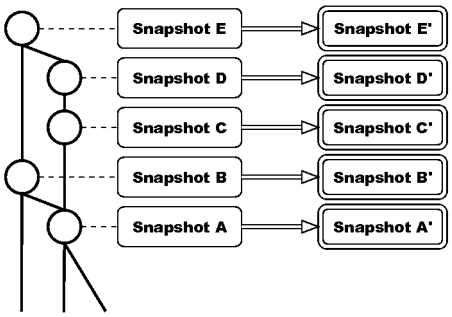


Figure 3: Retaking snapshots without changing commit DAG

2.3 Origin analysis

As explained by Bird et al., Git tracks the file content history [3]. Therefore, files with the same content but different names or locations in the parent and child commits are found as a renamed or moved file. This origin analysis is not limited to identical content but also similar content. When file paths are changed, often files content are also modified. Even in such cases, Git is able to detect relationships between changes if the file contents are similar enough. Origin analysis is conducted by checking that the amount of deletion of the original content and insertion of new content is larger than the threshold, which is set to 50% of the size of smaller files (original or modified). Therefore, if deletion or insertion is less than 50%, two files in parent and child commits are detected as moving or renaming. The threshold value can be changed. With this feature, developer can recognize the renaming and moving.

2.4 Rewriting histories

Bird et al. introduced rewritable Git histories from the aspect of DAG changing [3]. Git allows a repository owner to rearrange DAG retaining each snapshot. This rewriting is commonly conducted to simplify commit relation. In addition to that rewriting, it is also possible to change snapshots by calling *filter-branch* command[1]. As shown in Figure 3, each snapshot can be changed retaining its DAG. One of the uses of this rewriting is getting rid of a subset of unnecessary files.

3. REPOSITORY RECONSTRUCTION

As seen in Section 2, there are some features in Git different from Subversion. We found that these features are desirable for our purpose of reconstructing fine-grained versioning repositories. In this Section, first we show the fundamental idea of our approach, and then we explain our technique and its implementation.

3.1 Idea

For fine-grained (method-level) version control, what we have to do is making each method controllable in repositories. Since repositories are able to control file histories, it is possible to control method

[1]This command is not limited to changing snapshots retaining DAG. It is also possible to change DAG.

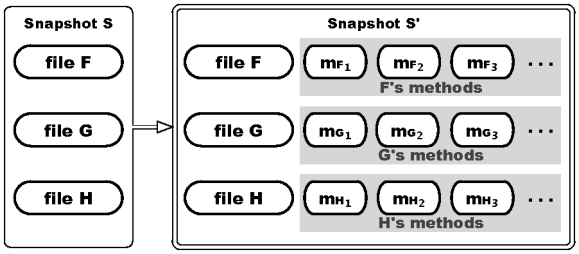


Figure 4: Retaking a snapshot for method storing

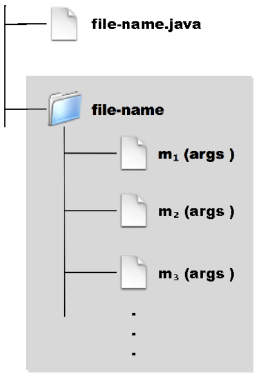


Figure 5: File structure

histories if each method is stored as a file. As explained in Section 2.1, Git stores existing files at each commit as a snapshot. In addition, it is possible to retake snapshots without changing commit history relation as stated in Section 2.4.

Figure 4 shows a concept of our approach. We retake each snapshot and replace it with a snapshot that includes additional method files. This replacement enables us to control method histories like traditional file histories. In original snapshot S, there exist original files F, G, and H. To retake snapshot S', methods exist in each file are copied and stored as files. Each snapshot is retaken separately without retaining file-level and method-level relations. Relations between methods will be detected by Git origin analysis. If methods are moved or renamed, the corresponding methods should be detected automatically, and method histories are managed by Git.

3.2 Technique

To retake the snapshots, we have to obtain the list of Java files that exist in the original snapshots. From each Java file, exiting methods are extracted and copied to individual files.

For storing method files, file structures are designed as Figure 5. Here, we explain the case of Java language. Each java file is saved as it is (file-name.java in Figure 5); if there are methods in the file, a directory named as the file's name (file-name) is created; files of methods are saved in the directory; these files are named by their method signatures. For simplicity, we limited to methods in top-level classes in this paper. Methods in inner classes or anonymous classes are ignored. In addition, abstract methods are not stored. Directories and files in gray space of Figure 5 are newly prepared for new snapshots.

Changes of method signatures correspond to file name changes and moving of methods correspond to moving files. If a method is deleted in a commit and reappear in later commit, Git can output its history including disappearing periods. As described in Section 2.3, the origin analysis is conducted based on file content similarity. Based on the threshold of content similarity, corresponding methods are identified. If two methods are highly similar, it is rational to detect them as corresponding methods. However, they might be corresponding methods in spite of not high similarity. In Section 4.1, we manually investigate whether found method origins are correct or not.

3.3 Implementation

Though existing repositories are not in Git system, it is possible to convert them to Git repositories for most SCM systems, as explained as ninth promise of Git in [3]. Therefore we can concentrate on reconstruction of Git repositories. In this paper, we target repositories of software written in Java language. As stated in Section 2.4, *filter-branch* command can be used for our purpose with a *–tree-filter* option. With this option, each snapshot appears in working directory and we can retake a snapshot after conducting some procedure. For our purpose, we copy the methods before retaking the snapshots. For locating the methods in Java files, we used the source code analysis tool MASU[2], which is an open source tool. It utilizes ANTLR for parsing source code [1].

4. ANALYSIS

In this Section, we investigate the usefulness of fine-grained versioning repositories. First, we examine whether Git origin analysis is useful for method-level origin analysis. Next, we conducted method-level bug prediction based on method-level change metrics. We targeted the WTP incubator project in Eclipse[3] for the analysis. This project is written in Java and is maintained with Git[4]. We cloned the Git repository on the 15th November, 2010.

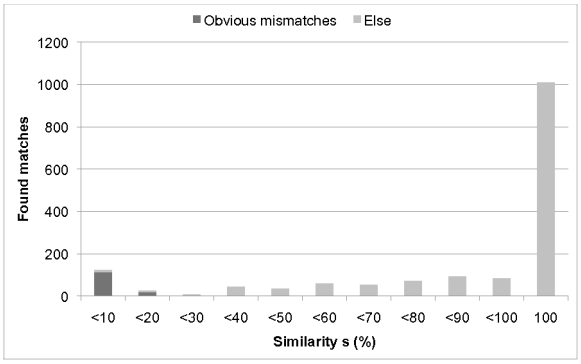
4.1 Origin analysis evaluation

Git detects origins of methods after changing names or location of the methods. When contents of methods are modified, detection of corresponding methods is conducted by calculating similarity of potential method pairs. If similarity is greater than or equal to threshold, the potential method pairs are detected as renamed methods as stated in Section 2.3. Users can change the threshold.

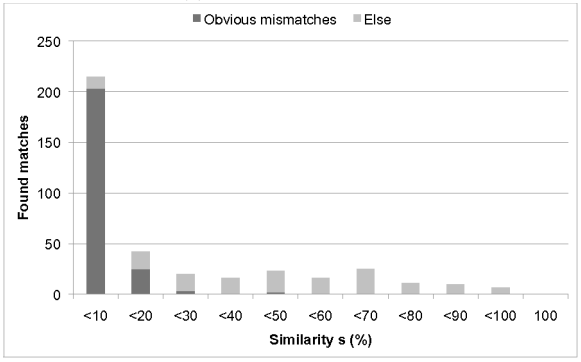
In this Section, we classify the method pairs according to similarity values to see the impact of the threshold and investigate whether detected method pairs seem to be correct or not. To know whether it is correct matching or not, we investigate every matching methods based on their method signatures and method paths. Since it is difficult to distinguish correct matches from mismatches perfectly, we count the number of obvious mismatches.

As shown in Figure 6, the found method matches are classified based on similarity values from 0% to 100% at 10% intervals. To see the impact of the text size for the similarity calculation, the result is summarized separately, (a) different lines (deletion lines + insertion lines) is less than 10, and (b) the others. Dark gray bars represent the numbers of detected obvious mismatches, and the numbers of other matches are represented as light gray bars.

[2] `http://sourceforge.net/projects/masu/`
[3] `http://www.eclipse.org/webtools/incubator/`
[4] `http://git.eclipse.org/c/webtools/org.eclipse.webtools.incubator.git/`



(a) different lines < 10



(b) different lines $>= 10$

Figure 6: Method matches based on a threshold

After checking manually, we found that automatic method matching based on Git works relatively well. There is a large number of method pairs whose similarity value is 100%, which means these methods have been renamed without being modified. There are few obvious mismatches if the similarity is greater than 20% whether the number of different lines is less than 10 or not.

Though the results of origin analysis evaluation may depend on projects or applications, line-based matching works relatively well as shown by Kim [19]. We think this is because there should be few cases where a newly created method is similar in content to a deleted method when the created and the deleted methods are independent of each other. We think that developers tend to commit simple changes, that is, the number of matching candidates is small and matching pairs are easy to find, not to confuse module histories.

4.2 Bug Prediction Based on Method History

Table 1 shows metrics we collected in this paper. The threshold of origin analysis is the default value (50%). This means that methods are regarded as newly created if most lines (>50%) are changed when names or locations of the methods are changed. In this paper, file-level bug prediction is also conducted to evaluate method-level bug prediction. For file-level bug prediction, the same metrics in Table 1 are collected except for NCHG_SIG.

Whether modules are faulty or not is decided by using the SZZ algorithm [26]. We obtained bug reports from the bug databases (Bugzilla) under the following conditions. The type of these faults is "bugs"; therefore, these faults do not include any enhancements or functional patches. The status of faults is either "resolved", "verified", or "closed", and the resolution of faults is "fixed". This

Table 1: Change metrics collected in this paper

Metrics	Description
NCHG	Number of individual changes
NFIX	Number of bug fixes
NAUTH	Number of authors
AGE	Existing period
ITVAL_{MAX, MIN}	Period of {maximum, minimum} interval between commits
LOC_{ADD, DEL}	Some of {added, deleted} lines of code between first and last revisions
NCHG_BUG_INTRO	Number of changes that is in commits that include bug-introducing changes to other methods
NCHG_BUG_COPL	Number of changes that is in commits that include changes to other methods (change coupling)
NCHG_SIG	Number of method signature changes
NCHG_FILE	Number of file name changes
NCHG_PCK	Number of package changes
IS_FAULTY	Faulty or not. Dependent variable in prediction models

Table 2: Results of the 10-fold cross validation

(a) Method-level

Model	Recall	Precision	F_1
Naive Bayes	0.704	0.030	0.057
Logistic regression	0.556	0.750	0.638
J48	0.593	1	0.744

(b) File-level

Model	Recall	Precision	F_1
Naive Bayes	0.778	0.073	0.134
Logistic regression	0.111	0.286	0.160
J48	0.389	0.875	0.538

means that the collected faults have already been fixed and have been resolved, and thus fixed revisions should be included in the entire repository. The severity of the faults is either "BLOCKER", "CRITICAL", or "MAJOR" in order to remove trivial bugs. Herraiz categorized these severity categories as important and the others without ENHANCEMENT as non-important [15].

We conducted a 10-fold cross validation on a snapshot with the tag, *v20090510*. There are $4,733$ methods including 27 faulty methods, and there exist 698 files including 18 faulty files in this snapshot. We adopted the WEKA data mining toolkit [29] for building bug prediction models. We built the following three models: naive Bayes, logistic regression, and J48. For the evaluation of the experiment, we use three measures: Recall ($\frac{TP}{TP+FN}$), Precision($\frac{TP}{TP+FP}$), and F_1($\frac{2\times\text{recall}\times\text{precision}}{\text{recall}+\text{precision}}$)[5].

Table 2 shows the method-level and file-level results. The precision values of naive Bayes are low because of lots of false positive. As a result, F_1 values of naive Bayes are low. Though the ratio of faulty modules per entire modules in method-level is lower than file-level, F_1 values in method-level are superior to file-level from logistic regression and J48 models. Those two models achieved more than 0.63 F_1 values. Though bug prediction is conducted on only one project, it seems that method-level change metrics may be effective for method-level bug prediction.

5. RELATED WORK

[5]TP (true positive), TN (true negative), FP (false positive), FN (false negative)

Table 3: Comparison of method history analysis

Method/Tool/Authors	Objective	Relationship analysis target	Storing
JDiff [2]	detection of structural changes	name	-
C-REX [12]	assessment of change propagation tools	calls, tokens	XML
Origin analysis [8]	detection of merging & splitting	name, code metrics, calls	BEAGLE repository (relational database)
S. Kim et al. [19]	detection of renaming	name, code metrics, calls, text	-
UMLDiff [31]	detection of structural changes	name, structure	-
RefactoringCrawler [7]	detection of refactorings	tokens, structure	XML
Weißgerber and Diel [28]	detection of refactorings	name, structure, text	-
SemDiff [5]	recommending adaptive changes	name, calls, structure	-
LSdiff [17]	detection & representation of systematic changes	name, calls, structure	-
AURA [30]	detection of structural changes	calls, text	-
This paper	fine-grained versioning	text	Git

Bug prediction based on change metrics. Graves et al. have studied software change history and found that if modules were changed many times, the modules tended to contain faults [9]. In addition, they found that if modules had not changed for one year, the rate of the modules containing faults would be low. Nagappan and Ball examined code churn, which is a measure of the amount of code change, and showed that relative code churn is highly predictive of defect density [22]. Ostrand et al. showed that fault and modification history of the file from previous release could be predictive of faults in the next release of a system [23]. Schröter et al. have studied the correlation with past failure component history and failure-prone components [25]. They analyzed SCM repositories and bug-tracking systems to extract the failure component's usage pattern, and applied some prediction models to compare the results. Hassan and Holt computed the ten most fault-prone modules after evaluating four heuristics: most frequently modified, most recently modified, most frequently fixed, and most recently fixed [11]. Kim et al. have tried to predict the fault density of entities using previous faults localities based on the observation that most faults do not occur uniformly [20]. Ratzinger et al. investigated the interrelationship between previous refactoring and future software defects [24]. Hassan predicted faults using the entropy of code changes [10].

Method history analysis. Table 3 shows the related work about method history analysis. JDiff is a tool of identifying differences of Java structure between changes [2]. In method level, it identifies insertion and deletions. C-REX is a tool for C [12]. Its architecture is similar to Git-base system in the point of origin analysis timing. Method origins are found between snapshots. Merging and splitting are detected by origin analysis [8]. Detection of renames is tackled in [19, 31]. Refactorings are detected in [7, 28]. SemDiff detects adaptations to client programs [5]. LSdiff detects method relations and infers change rules [17]. AURA can detect one-to-many and many-to-one change rules [30].

The main objective of the proposed system is not the identification of method histories but the controlling the method histories. We intend to make the system a research platform like BEAGLE in origin analysis [8]. Our Git-based system has the following advantages:

- Method histories can be seen as SCM logs since data are stored in Git repositories.
- Incremental reconstruction is possible. Additional new commits can be stored in reconstructed repositories.
- Relations between methods are not stored explicitly. Git does not store relations between files, but stores snapshots and commit DAGs. This enables us to see necessary method histories when we need. If we do not trace method histories beyond renames, Git show us the method histories before and after renaming separately. This is important for a research platform because there should not be an unique relation between methods in two sequential commit. A method, which is identified as merged method by origin analysis, may be identified as many-to-one changed method by AURA, and might be a clone of other methods. Though this paper evaluate the performance of Git origin analysis, it is possible to identify different relations by different technique if we replace Git origin analysis with them. Consequently, code clone genealogies [18] and documenting program changes [4], for example, can be conducted on the system.

6. CONCLUSION

Lots of studies examined and investigated the power of change metrics mined from software repositories. File-level change metrics are widely studied and those effectiveness is shown so far. To proceed to method-level bug prediction, we choose to enrich repositories by converting them to fine-grained versioning repositories. This paper contributes to the following:

- A technique of reconstructing method-level version control repositories from existing repositories is presented.
- Method-level change metrics are collected and used for bug prediction on a real software project.

Since lots of SCM repositories can be converted to Git repositories, lots of existing repositories can be converted to fine-grained versioning repositories. Since reconstructed repositories are also Git repositories, there is no additional labor to mine change metrics compared to natural Git repositories.

With reconstructed repositories, method-level change metrics can be easily collected. In addition, bug location using the SZZ algorithm can be conducted on method-level. As a result, we can realize method-level bug prediction. The result of one real software project shows that method-level change metrics are effective for method-level bug prediction.

7. REFERENCES

[1] Antlr. `http://www.antlr.org/`.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. of 19th IEEE International Conference on Automated Softw. Engg.*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proc. of 6th International workshop on Mining software repositories*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[4] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proc. of 25th IEEE/ACM International Conference on Automated Softw. Engg.*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.

[5] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. of 30th International Conference on Softw. Engg.*, pages 481–490, New York, NY, USA, 2008. ACM.

[6] A. De Lucia, F. Fasano, R. Oliveto, and D. Santonicola. Improving context awareness in subversion through fine-grained versioning of java code. In *Proc. of 9th international workshop on Principles of software evolution*, pages 110–113, New York, NY, USA, 2007. ACM.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proc. of 20th European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[8] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.

[9] T. L. Graves, A. F. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[10] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. of 31st International Conference on Softw. Engg.*, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[11] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proc. of 21st International Conference on Softw. Maintenance*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[12] A. E. Hassan and R. C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg.*, 11:335–367, September 2006.

[13] H. Hata, O. Mizuno, and T. Kikuno. Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Softw. Engg.*, 15(2):147–165, 2010.

[14] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *Proc. of 6th International workshop on Mining software repositories*, pages 141–150, Washington, DC, USA, 2009. IEEE Computer Society.

[15] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proc. of 5th International workshop on Mining software repositories*, pages 145–148. ACM, 2008.

[16] I. Herraiz, G. Robles, and J. M. Gonzalez-Barahona. Research friendly software repositories. In *Proc. of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 19–24, New York, NY, USA, 2009. ACM.

[17] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proc. of 31st International Conference on Softw. Engg.*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[18] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM.

[19] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proc. of 12th Working Conference on Reverse Engineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[20] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proc. of 29th International Conference on Softw. Engg.*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[21] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proc. of 31st International Conference on Softw. Engg.*, pages 287–297, 2009.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of 27th International Conference on Softw. Engg.*, pages 284–292, 2005.

[23] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[24] J. Ratzinger, T. Sigmund, and H. Gall. On the relation of refactorings and software defect prediction. In *Proc. of 5th International workshop on Mining software repositories*, pages 35–38. ACM New York, NY, USA, 2008.

[25] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proc. of ACM/IEEE international symposium on Empirical software engineering*, pages 18–27, New York, NY, USA, 2006. ACM.

[26] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? (on Fridays.). In *Proc. of 2nd International workshop on Mining software repositories*, pages 24–28, 2005.

[27] D. Thomas and K. Johnson. Orwell: A configuration management system for team programming. In *Proc. of 3rd conference on object oriented programming languages and applications*, pages 135–141. ACM, 1988.

[28] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. of 21st IEEE/ACM International Conference on Automated Softw. Engg.*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[29] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2 edition, 2005.

[30] W. Wu, Y.-g. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proc. of 32nd International Conference on Softw. Engg.*, 2010.

[31] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proc. of 20th IEEE/ACM International Conference on Automated Softw. Engg.*, pages 54–65, New York, NY, USA, 2005. ACM.

Reachability Analysis of Probabilistic Timed Automata Based on an Abstraction Refinement Technique

Takeshi Nagaoka, Akihiko Ito, Toshiaki Tanaka, Kozo Okano, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University
{t-nagaok,a-ito,tstanaka,okano,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Model checking techniques are considered as promising techniques for verification of information systems due to their ability of exhaustive checking. Well-known state explosion, however, might occur in model checking of large systems. In order to avoid it, several abstraction techniques have been proposed. Some of them are based on CounterExample-Guided Abstraction Refinement (CEGAR) technique proposed by E. Clarke *et al.*. This paper proposes a reachability analysis technique for probabilistic timed automata. In the technique, we abstract time attributes of probabilistic timed automata by our abstraction technique proposed in our previous work. Then, we apply probabilistic model checking to the generated abstract model which is just a markov decision process (MDP) with no time attributes. Also, our technique can produce a counter example as a set of paths when a given model does not satisfy a specification. The paper also provides some experimental results on applying our method to IEEE 1394, FireWire protocol. Experimental results show our algorithm can reduce the number of states and total execution time dramatically compared to one of existing approaches.

Keywords

Probabilistic Timed Automaton, CEGAR, Model Checking, Real-time System, Formal Verification

1. INTRODUCTION

Model checking[1] techniques are considered as promising techniques for verification of information systems due to their ability of exhaustive checking. For verification of real-time systems such as embedded systems, timed automata are often used. On the other hand, probabilistic model checking[2, 3, 4] can evaluate performance, dependability and stability of information processing systems with random behaviors. In recent years, probabilistic models with real-time behaviors, called probabilistic timed automata (PTA) attract attentions. As well as traditional model checking techniques, however, state explosion is thought to be a major hurdle for verification of probabilistic timed automata.

Clarke *et al.* proposed an abstraction technique called CEGAR (CounterExample-Guided Abstraction Refinement)[5]. In the CEGAR technique, we use a counter example (CE) produced by a model checker as a guide to refine abstracted models. In [6], we have proposed an abstraction algorithm for timed automata based on CEGAR. In this algorithm, we generate finite transition systems as abstract models where all time attributes are removed. The refinement modifies the transition relations of the abstract model so that the model behaves correctly even if we don't consider the clock constraints.

This paper proposes a reachability analysis technique for probabilistic timed automata. In the technique, we abstract time attributes of probabilistic timed automata by applying our abstraction technique for timed automata proposed in [6]. Then, we apply probabilistic model checking to the generated abstract model which is just a markov decision process (MDP) with no time attributes. The probabilistic model checking algorithm calculates summation of occurrence probability of all paths which reach to a target state for reachability analysis. For probabilistic timed automata, however, we have to consider required clock constraints for such paths, and choose the paths whose required constraints are compatible. Since our abstract model does not consider the clock constraints, we add a new flow where we check whether all paths used for probability calculation are compatible. Also, if they are not compatible, we transform the model so that we do not accept such incompatible paths simultaneously.

This paper also provides some experimental results on applying our method to some examples. Experimental results show our algorithm can reduce the number of states and total execution time dramatically compared to one of existing approaches.

Several papers including Paper [2, 3, 4] have proposed probabilistic model checking algorithms. These algorithms, however, don't provide CEs when properties are not satisfied. Our proposed method provides a CE as a set of paths based on k-shortest paths search. This is a major contribution of our method. The proposed method also performs model checking considering compatibility problem. Few approaches resolve the compatibility problem. Our approach also shows the efficiency via performing experiments.

The organization of the rest paper is as follows. Sec.2 provides some definitions and lemmas as preliminaries. Sec.3 describes our proposed abstraction technique for the probabilistic timed automaton. Sec.4 gives some experimental results. Finally, Sec.5 concludes the paper and gives future works.

2. PRELIMINARY

2.1 Clock and Zone

Let C be a finite set of clock variables which take non-negative real values ($\mathbb{R}_{\geq 0}$). A map $\nu : C \to \mathbb{R}_{\geq 0}$ is called a clock assignment. The set of all clock assignments is denoted by $\mathbb{R}^C_{\geq 0}$. For any $\nu \in \mathbb{R}^C_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$ we use $(\nu + d)$ to denote the clock assignment defined as $(\nu + d)(x) = \nu(x) + d$ for all $x \in C$. Also, we use $r(\nu)$ to denote the clock assignment obtained from ν by resetting all of the clocks in $r \subseteq C$ to zero.

DEFINITION 2.1. *Syntax and semantics of a differential inequality E on a finite set C of clocks is given as follows:*
$E ::= x - y \sim a \mid x \sim a$,
where $x, y \in C$, a is a literal of a real number constant, and $\sim \in \{\leq, \geq, <, >\}$. Semantics of a differential inequality is the same as the ordinal inequality.

DEFINITION 2.2. *Clock constraints $c(C)$ on a finite set C of clocks is defined as follows: A differential inequality in on C is an element of $c(C)$. Let in_1 and in_2 be elements of $c(C)$, $in_1 \wedge in_2$ is also an element of $c(C)$.*

A zone $D \in c(C)$ is described as a product of finite differential inequalities on clock set C, which represents a set of clock assignments that satisfy all the inequalities. In this paper, we treat a zone D as a set of clock assignments $\nu \in \mathbb{R}^C_{\geq 0}$ (For a zone D, $\nu \in D$ means the assignment ν satisfies all the inequalities in D).

2.2 Probability Distribution

A discrete probability distribution on a finite set Q is given as the function $\mu : Q \to [0, 1]$ such that $\Sigma_{q \in Q} \mu(q) = 1$. Also, $support(\mu)$ is a subset of Q such that $\forall q \in support(\mu).\mu(q) > 0$ holds.

2.3 Markov Decision Process

A Markov Decision Process (MDP)[7] is a markov chain with non-deterministic choices.

DEFINITION 2.3. *A markov decision process MDP is 3-tuple $(S, s_0, Steps)$, where S is a finite set of states, $s_0 \in S$ is an initial state, and $Steps \subseteq S \times A \times Dist(S)$ is a probabilistic transition relation where $Dist(S)$ is a probability distribution over S.*

In our reachability analysis procedure, we transform a given PTA into a finite MDP, and perform probabilistic verification based on the Value Iteration[8] technique.

2.3.1 Adversary

An MDP has non-deterministic transitions called action. To resolve the non-determinism, an adversary is used. The adversary requires a finite path on an MDP, and decides a transition to be chosen at the next step.

2.3.2 Value Iteration

A representative technique of model checking for an MDP is Value Iteration[8]. The Value Iteration technique can obtain both of maximum and minimum probabilities of reachability and safety properties, respectively. At each state, Value Iteration can select an appropriate action according to the property to be checked. Therefore, the technique can obtain the adversary as well as the probability.

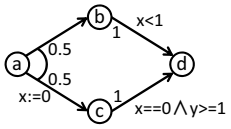


Figure 1: An Example of a PTA

2.4 Timed Automaton

DEFINITION 2.4. *A timed automaton $\mathscr{A}$ is a 6-tuple (A, L, l_0, C, I, T), where A is a finite set of actions, L is a finite set of locations, $l_0 \in L$ is an initial location, C is a finite set of clocks, $I \subset (L \to c(C))$ is a mapping from locations to clock constraints, called a location invariant, and $T \subset L \times A \times c(C) \times \mathscr{R} \times L$ is a set of transitions, where $c(C)$ is a clock constraint, called guards and $\mathscr{R} = 2^C$ is a set of clocks to reset.*

DEFINITION 2.5. *Given a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, let $S \subseteq L \times \mathbb{R}^C_{\geq 0}$ be a set of whole states of $\mathscr{A}$. The initial state of $\mathscr{A}$ shall be given as $(l_0, 0^C) \in S$. For a transition $(l_1, a, g, r, l_2) \in T$, the following two transitions are semantically defined. The former one is called an action transition, while the latter one is called a delay transition.*

$$\frac{l_1 \xrightarrow{a,g,r} l_2, g(\nu), I(l_2)(r(\nu))}{(l_1, \nu) \xRightarrow{a} (l_2, r(\nu))}, \qquad \frac{\forall d' \leq d \quad I(l_1)(\nu + d')}{(l_1, \nu) \xRightarrow{d} (l_1, \nu + d)}$$

DEFINITION 2.6. *For timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, an infinite transition system is defined according to the semantics of $\mathscr{A}$, where the model begins with the initial state.*

2.5 Probabilistic Timed Automaton

A PTA is a kind of a timed automaton extended with probabilistic behavior. In the PTA, a set of probabilistic distributions is used instead of a set T of discrete transitions on the timed automaton.

DEFINITION 2.7. *A probabilistic timed automaton PTA is a 6-tuple $(A, L, l_0, C, I, prov)$, where A is a finite set of actions, L is a finite set of locations, $l_0 \in L$ is an initial location, C is a finite set of clocks, $I \subset (L \to c(C))$ is a location invariant and $prob \subseteq L \times A \times c(C) \times Dist(2^C \times L)$ is a finite set of probabilistic transitions, where $c(C)$ represents a guard condition, and $Dist(2^C \times L)$ represents a finite set of probability distributions p. The Distribution $p(r, l) \in Dist(2^C \times L)$ represents the probability of resetting clock variables in r and also moving to the location l;*

Figure 1 shows an example of a PTA. In the figure, from the location a, the control moves to the location b with the probability 0.5 and also moves to the location c letting the value of the clock x reset to zero with the probability 0.5.

DEFINITION 2.8. *Semantics of a probabilistic timed automaton $PTA = (A, L, l_0, C, I, prob)$ is given as a timed probabilistic system $TPS_{PTA} = (S, s_0, TSteps)$ where, $S \subseteq L \times \mathbb{R}^C$ is a set of states, $s_0 = (l_0, 0^C)$ is an initial state, and $TSteps \subseteq S \times A \cup \mathbb{R}_{\geq 0} \times Dist(S)$ is composed of action transitions and delay transitions, where*

a) *action transition*

if $a \in A$ and there exists $(l, a, g, p) \in prob$ such that $g(\nu)$ and $I(l')(r(\nu))$ for all $(r, l') \in support(p)$, $((l, \nu), a, \mu) \in TSteps$ where for all $(l', \nu') \in S$

$$\mu(l', \nu') = \sum_{r \subseteq C \wedge \nu' = r(\nu)} p(r, l').$$

b) *delay transition*

if $d \in \mathbb{R}_{\geq 0}$, and for all $d' \leq d$, $I(l)(\nu + d')$, $((l, \nu), d, \mu) \in TSteps$ where $\mu(l, \nu + d) = 1$.

In this paper, using a location l and a zone D, we describe a set of semantic states as $(l, D) = \{(l, \nu) \mid \nu \in D\}$.

DEFINITION 2.9. *A path ω with length of n on a timed probabilistic system $TPS_{PTA} = (S, s_0, TSteps, L')$ is denoted as follows.*

$$\omega = (l_0, \nu_0) \xrightarrow{d_0, \mu_0} (l_1, \nu_1) \xrightarrow{d_1, \mu_1} \cdots \xrightarrow{d_{n-1}, \mu_{n-1}} (l_n, \nu_n)$$

, where $(l_i, \nu_i) \in S$ for $0 \leq i \leq n$ and $((l_i, \nu_i), d_i, \mu) \in TSteps \wedge ((l_i, \nu_i + d_i), 0, \mu_i) \in TSteps \wedge (l_{i+1}, \nu_{i+1}) \in support(\mu_i)$ for $0 \leq i \leq n - 1$.

For model checking of a probabilistic timed automaton, we extract a number of paths and calculate a summation of their occurrence probabilities in order to check the probability of satisfying a given property. The important point is that we have to choose a set of paths which are compatible with respect to time elapsing.

LEMMA 2.1. *If a set Ω of paths on a timed probabilistic system TPS_{PTA} satisfies the following predicate $isCompatible$, then all of the paths over Ω are said to be compatible.*

$isCompatible(\Omega) =$

$$\begin{cases} \textit{true}, & \textit{if } \forall i \leq min(\Omega) \bigwedge_{\substack{\omega^\alpha, \omega^\beta \in \Omega \\ \wedge \omega^\alpha \neq \omega^\beta}} (l_i^\alpha = l_i^\beta \wedge d_i^\alpha = d_i^\beta) \\ & \quad \textit{or there exists } i \leq min(\Omega) \textit{ such that} \\ & \quad \bigwedge_{\substack{\omega^\alpha \, \omega^\beta \in \Omega \\ \wedge \omega^\alpha \neq \omega^\beta}} (l_i^\alpha \neq l_i^\beta \wedge d_i^\alpha = d_i^\beta \wedge \bigwedge_{j \leq i} (l_j^\alpha = l_j^\beta \wedge d_j^\alpha = d_j^\beta)), \\ & \quad \textit{and also} \bigwedge_{\substack{\Omega' \in 2^\Omega \wedge \\ \Omega' \neq \Omega \wedge |\Omega'| \leq 2}} isCompatible(\Omega') \\ \textit{false}, & \textit{otherwise}. \end{cases}$$

In Lemma2.1, we give the predicate $isCompatible$ for a set Ω of paths on a timed probabilistic system. In the lemma, we let paths in Ω be compatible if there is no contradiction with respect to time elapsing at the branching point of all the paths in Ω, and also if the compatibility is kept for every subset of Ω which contains more than two paths.

2.6 CounterExample-Guided Abstraction Refinement

2.6.1 General CEGAR Technique

Since model abstraction sometimes over-approximates an original model, we may obtain spurious CEs which are infeasible on the original model. Paper[5] gives an abstraction refinement framework called CEGAR (CounterExample-Guided Abstraction Refinement) (Fig.2).

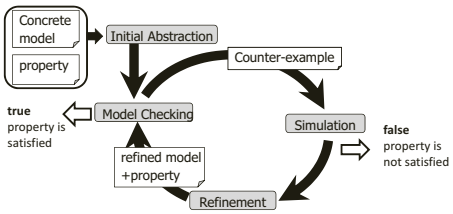


Figure 2: A General CEGAR Technique

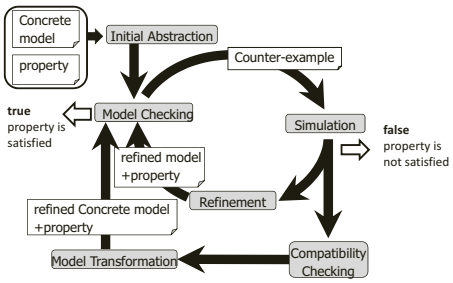


Figure 3: Our CEGAR Technique for Reachability Analysis of a Probabilistic Timed Automaton

In the algorithm, at the first step (called Initial Abstraction), it generates an initial abstract model. Next, it performs model checking on the abstract model. In this step, if the model checker reports that the model satisfies a given specification, we can conclude that the original model also satisfies the specification, because the abstract model is an over-approximation of the original model. If the model checker reports that the model does not satisfy the specification, however, we have to check whether the CE detected is spurious or not in the next step (called Simulation). In the Simulation step, if we find that the CE is valid, we stop the loop. Otherwise, we have to refine the abstract model to eliminate the spurious CE, and repeat these steps until valid output is obtained.

2.6.2 CEGAR Technique for a Timed Automaton

In [6], we have proposed the abstraction refinement technique for a timed automaton based on the framework of CEGAR. In this approach, we remove all the clock attributes from a timed automaton. If a spurious CE is detected by model checking on an abstract model, we transform the transition relation on the abstract model so that the model behaves correctly even if we don't consider the clock constraints. Such transformation obviously represents the difference of behavior caused by the clock attributes. Therefore, the finite number of application of the refinement algorithm enables us to check the given property without the clock attributes. Since our approach does not restore the clock attributes at the refinement step, the abstract model is always a finite transition system without the clock attributes.

3. PROPOSED APPROACH

In this section, we will present our abstraction refinement technique for a probabilistic timed automaton. In the technique, we use the abstraction refinement technique for a timed automaton proposed

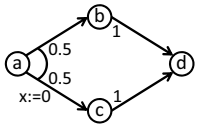


Figure 4: An Initial Abstract Model

in [6]. In addition, we resolve the compatibility problem shown in Sec.2.5 by performing a backward simulation technique and generating additional location to distinguish the required condition for every incompatible path. Figure 3 shows our abstraction refinement framework. As shown in the figure, we add another flow where we resolve the compatibility problem.

Our abstraction requires a probabilistic timed automaton PTA and a property to be checked as its inputs. The property is limited by the PCTL formula $P_{<p}[\text{true } \mathbf{U}\ err]$. The formula represents a property that the probability of reaching to states where err (which means an error condition in general) is satisfied, is less than p.

3.1 Initial Abstraction

The initial abstraction removes all the clock attributes from a given probabilistic timed automaton as well as the technique in The generated abstract model over-approximates the original probabilistic timed automaton.

DEFINITION 3.1. *For a given probabilistic timed automaton $PTA = (A, L, l_0, C, I, prob)$, a markov decision process $M\hat{D}P_{PTA} = (\hat{S}, \hat{s}_0, \hat{Steps})$ is produced as its abstract model, where $\hat{S} = L$, $\hat{s}_0 = l_0$, and $\hat{Steps} = \{\ (s, a, p)\ |\ (s, a, g, p) \in prob\ \}$*

Figure 4 shows an initial abstract model for the PTA shown in Fig.1 As shown in the figure, the abstract model is just an MDP where all of the clock constraints are removed though we keep a set of clock reset as a label of transitions.

3.2 Model Checking

In model checking, we apply Value Iteration[8] into the markov decision process obtained by abstraction and calculate a maximum reachability probability. Also, it decides an action to be chosen at every state as an adversary. If the obtained probability is less than p, we can terminate the CEGAR loop and conclude that the property is satisfied.

Although Value Iteration can calculate a maximum reachability probability, it cannot produce concrete paths used for the probability calculation. To obtain the concrete paths, we use an approach proposed in [9] which can produce CE paths for PCTL formulas. The approach translates a probabilistic automaton into a weighted digraph. And we can obtain at most k paths by performing k-shortest paths search on the graph.

DEFINITION 3.2. *A path $\hat{\omega}$ on an abstract model $M\hat{D}P_{PTA} = (\hat{S}, \hat{s}_0, \hat{Steps})$ for $PTA = (A, L, l_0, C, I, prob)$ is given as follows,*

$$\hat{\omega} = \hat{s}_0 \xrightarrow{a_0, p_0, r_0} \hat{s}_1 \xrightarrow{a_1, p_1, r_1} \cdots \xrightarrow{a_{n-1}, p_{n-1}, r_{n-1}} \hat{s}_n$$

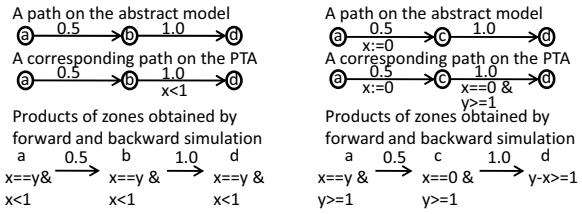


Figure 5: Products of zones obtained by Forward and Backward Simulation

, *where $\hat{s}_i \in \hat{S}$ for $0 \le i \le n$ and $(\hat{s}_i, a_i, p_i) \in \hat{Steps} \wedge (r_i, \hat{s}_{i+1}) \in support(p_i)$ for $0 \le i \le n-1$.*

As defined in Def. 3.2, we associate a set r of clock reset with a path on an abstract model in order to show the difference of r over the probabilistic distribution p.

For the abstract model shown in Fig.4, Value Iteration outputs 1.0 as the probability that it reaches to the state d from the state a. On the other hand, k-shortest paths search ($k \ge 2$) detects two paths $\hat{\omega}^\alpha = a \xrightarrow{\tau, 0.5, \{\}} b \xrightarrow{\tau, 1.0, \{\}} d$ and $\hat{\omega}^\beta = a \xrightarrow{\tau, 0.5, \{x:=0\}} c \xrightarrow{\tau, 1.0, \{\}} d$, where τ represents a label for transitions with no label in the figure.

3.3 Simulation

Simulation checks whether all the paths obtained by k-shortest paths search are feasible or not on the original probabilistic timed automaton. We use the simulation algorithm proposed in [6] If there is at least one path which is infeasible on the original PTA, we proceed to the abstraction refinement step.

3.4 Abstraction Refinement

In this step, we refine the abstract model so that the given spurious CE also becomes infeasible on the refined abstract model. We can use the algorithm proposed in [6]. Since the algorithm of [6] performs some operations on transitions of a timed automaton, we replace such operations by those on probability distributions of a probabilistic timed automaton.

3.5 Compatibility Checking

When all the paths obtained by k-shortest paths search are feasible and a summation of occurrence probabilities of them is greater than p, we also have to check whether all the paths are compatible or not. In this compatibility checking step, at each location of the paths, we have to obtain a condition (zone) which is reachable from the initial state and also reachable to the last state along with the path. Next, we check the compatibility of such conditions among all paths. To obtain such conditions, we have to perform both forward simulation shown in Sec. 3.3 and backward simulation for each path, and merge the results. For the result of forward simulation, we can reuse the result obtained in the Simulation step. Then we check the compatibility based on Lemma 2.1. Paper[10] shows the algorithm of compatibility checking in detail. In the algorithm, we check the compatibility of such products of zones at every location of the paths.

Figure 5 shows the products of zones obtained by both forward and backward simulation for two paths $\hat{\omega}^\alpha$ and $\hat{\omega}^\beta$ in Sec.3.2. For the path $\hat{\omega}^\alpha$, the product zone at a is given as $D^{\hat{\omega}^\alpha}_{c,0} = (x == y \wedge x <$

Table 1: Experimental Result

		Digital Clocks[3]				Proposed Approach				
$D(\mu s)$	p	$Result$	$Time(s)$	$State$	$MEM(MB)$	$Result$	$Time(s)$	$Loop$	$State$	$Heap(MB)$
5	1.09×10^{-1}	false	20.90	297,232	10.2	false	4.19	10	37	8.0
	3.28×10^{-1}	true	20.89	297,232	10.2	true	3.60	9	36	8.0
10	1.26×10^{-2}	false	54.80	685,232	21.7	false	8.16	19	134	8.0
	3.79×10^{-2}	true	54.82	685,232	21.7	true	6.57	15	115	8.0
20	1.85×10^{-4}	false	176.93	1,461,232	41.0	false	1186.08	47	477	64.0
	5.56×10^{-4}	true	177.46	1,461,232	41.0	true	31.32	32	435	8.0

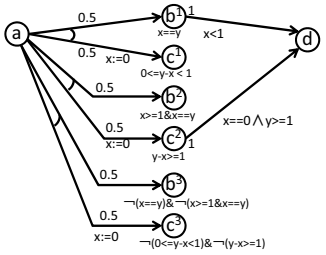


Figure 6: A Transformed PTA

1), which means a zone which is reachable from the initial state and also can move to d. Similarly, for the path $\hat{\omega}^{\beta}$, the product zone is given as $D_{c,0}^{\hat{\omega}^{\beta}} = (x == y \wedge y > 1)$. Since $D_{c,0}^{\hat{\omega}^{\alpha}}$ and $D_{c,0}^{\hat{\omega}^{\beta}}$ contradict each other, we can conclude that the paths $\hat{\omega}^{\alpha}$ and $\hat{\omega}^{\beta}$ are incompatible each other.

3.6 Model Transformation

When the compatibility check procedure decides a given set $\hat{\Omega}$ of paths is incompatible at i-th location, our proposed algorithm resolves the incompatibility by refining behaviors from the i-th location. Our algorithm uses $D_c^{\hat{\omega}}$ which is a product of results of forward and backward simulation for a path $\hat{\omega} \in \hat{\Omega}$. It duplicates locations which are reachable from the zone $D_{c,i}^{\hat{\omega}}$ by an action associated with the i-th distribution p_i. Also it constructs transition relations so that the transformation becomes equivalent transformation. For example, transition relations from a duplicated location are duplicated if the relations are executable from the invariant associated with the duplicated location. Detailed Algorithms to transform the model are given in [10].

Figure 6 shows the transformed PTA by applying the model transformation procedure for the paths $\hat{\omega}^{\alpha}$ and $\hat{\omega}^{\beta}$. The locations b^1 and c^1 are duplicated locations based on the path $\hat{\omega}^{\alpha}$ and the zone $D_{c,0}^{\hat{\omega}^{\beta}} = (x == y \wedge x < 1)$ on the location a. We associate invariants to b^1 and c^1 based on zones which are reachable from $D_{c,0}^{\hat{\omega}^{\beta}}$ through transitions from a to b, and from a to c, respectively. Also, we duplicate a transition from b to d as the transition from b^1 to d because the transition is feasible from the invariant of b^1. On the other hand, we do not duplicate a transition from c to d because the transition is not feasible from the invariant of c^1. Similarly, locations b^2 and c^2 are duplicated locations based on the path $\hat{\omega}^{\beta}$ and the zone $D_{c,0}^{\hat{\omega}^{\beta}}$. Locations b^3 and c^3 are generated as complements of the invariant associated with each duplicated location in order to preserve the equivalence.

By transforming the original PTA in such a way, if we remove all clock constraints from the model in Fig.6, Value Iteration on its abstract model outputs 0.5 as the maximum probability.

4. EXPERIMENTS

We have implemented a prototype of our proposed approach with Java, and performed some experiments. Though the prototype can check the compatibility of a given set of paths, currently it cannot deal with the model transformation.

The prototype performs k-shortest paths search and simulation concurrently in order to reduce execution time. By implementing the algorithms concurrently, we have not to wait until all of k paths are detected, i.e. if a path is detected by the k-shortest paths search algorithm, we can immediately apply simulation and (if needed) abstraction refinement procedures. Also, our prototype continues the k-shortest search algorithm when a spurious CE is detected and the refinement algorithm is applied. If other paths which do not overlap with the previous spurious CEs, are detected, we can apply simulation and refinement algorithms to it again. This helps us reduce the number of CEGAR loop.

4.1 Goal of the Experiments

The goal of this experiment is to check that our approach, which also searches a CE, is performed within realistic time. In this experiment, we evaluated the performance of our proposed approach with regard to execution time, memory consumption, and qualities of obtained results. As a target for comparison, we chose the approach of Digital Clocks[3], which is considered as a basic approach, where they approximate clock evaluations of a PTA by integer values.

4.2 Example

We used a case study of the FireWire Root Contention Protocol[11] as an example for this experiment. This case study concerns the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus (called "FireWire") which takes place when a node is added or removed from the network. In the experiment, we checked the probability that a leader is not selected within a given deadline. The probabilistic timed automaton for the example is composed of two clock variables, 11 locations, and 24 transitions.

4.3 Procedure of the Experiments

In this experiment, we checked the property that "the probability that a leader cannot be elected within a given $deadline$ is less than p." We considered three scenarios where the parameter $deadline$ is 5, 10, 20 μs, respectively. Also, for each scenario, we conducted two experiments where the value of p is 1.5 times as an approximate value of the maximum probability obtained by the Digital Clocks approach[3] and a half of it, respectively. In the proposed approach, we searched at most 5000 paths by letting the parameter k of the k-shortest paths search algorithm be 5000. For evaluation of existing approach, we used the probabilistic model checker PRISM[12].

Table 2: Analysis of Counter Example Paths

$D(\mu s)$	p	$Path$	$Probability$	$CC(ms)$
5	1.0938×10^{-1}	7	1.2500×10^{-1}	0.7
10	1.2635×10^{-2}	43	1.2695×10^{-2}	5.9
20	1.8500×10^{-4}	2534	1.8501×10^{-4}	296.9

The experiments were performed under Intel Core2 Duo 2.33 GHz, 2GB RAM, and Fedora 12 (64bit).

4.4 Results of the Experiments

The results are shown in Table 1. The column of D means the value of $deadline$. For each approach, columns of $Results$, $Time$, and $States$ show the results of model checking, execution time of whole process, and the number of states constructed, respectively. The column MEM in the columns of the Digital Clocks shows the memory consumption of PRISM. The columns $Loop$ and $Heap$ in the columns of the proposed approach show the number of CEGAR loops executed and the maximum heap size of the Java Virtual Machine (JVM) which executes our prototype, respectively.

Table 1 shows that for all cases we can dramatically reduce the number of states and obtain correct results. Moreover, we can reduce the execution time more than 80 percent except for the case when $deadline = 20\mu s$ and $p = 1.85 \times 10^{-4}$. In this case, however, the execution time drastically increases.

Table 2 shows the results of analysis of CE paths obtained when the results of model checking are false. The columns of $Path$, $Probability$ and CC show the number of CE paths, the summation of occurrence probability of them, and execution time for compatibility checking, respectively. For this example, the obtained sets of CE paths are compatible in every case.

4.5 Discussion

From the results shown in Table 1, we can see that our proposed approach is efficient with regard to both execution time and the number of states. Especially, the number of states decrease dramatically. The execution time is also decreased even though we perform model checking several times shown in the column of $Loop$.

On the other hand, in the case when $deadline = 20\mu s$ and $p = 1.85 \times 10^{-4}$, the execution time increases drastically. We think that as shown in Table 2 we have to search 2534 paths and this causes the increase of execution time especially for k-shortest paths search. A more detailed analysis shows that the execution time for k-shortest paths search accounts for 1123 seconds of total execution time of 1186 seconds. Also, the results shows that the JVM needs 64MB as its heap size in this case. This is because compatibility checking for 2534 of paths needs a large amount of the memory. From the results, we have to resolve a problem of the scalability when the number of candidate paths for a CE becomes large.

5. CONCLUSION

This paper proposed the abstraction refinement technique for a probabilistic timed automaton by extending the existing abstraction refinement technique for a timed automaton. The main contribution of this work is generation of a CE. Also, the experimental results show the efficiency of our technique compared to one of existing approaches.

Future work includes completion of implementation. General DBM does not support not operator[13]; so we have to investigate efficient algorithms for the not operator.

Acknowledgments

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology, as well as Grant-in-Aid for Scientific Research C(21500036), as well as grant from The Telecommunications Advancement Foundation.

6. REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, editors. *Model checking*. MIT Press, 1999.

[2] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):1027–1077, 7 2007.

[3] M. Kwiatkowska, G. Norman, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Int. Journal on Formal Methods in System Design*, 29(1):33–78, 7 2006.

[4] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic games for verification of probabilistic timed automata. In *Proc. of the 7th Int. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS'09)*, volume 5813 of *LNCS*, pages 212–227, 9 2009.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. Helmut. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[6] T. Nagaoka, K. Okano, and S. Kusumoto. An abstraction refinement technique for timed automata based on counterexample-guided abstraction refinement loop. *IEICE Transactions on Information and Systems*, E93-D(5):994–1005, 5 2010.

[7] C. Derman, editor. *Finite-State Markovian Decision Processes*. New York: Academic Press, 1970.

[8] D. P. Bertsekas. *Dynamic programming and optimal control*. Athena Scientific, 1995.

[9] H. Aljazzar and S. Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Transactions on Software Engineering*, 36(1):37–60, 1 2010.

[10] A. Ito, T. Nagaoka, K. Okano, and S. Kusumoto. Reachability analysis for probabilistic timed system based on timed abstraction refinement technique (in japanese). In *IEICE Technical Report*, volume 109, pages 85–90, 3 2010.

[11] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the ieee1394 firewire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 4 2003.

[12] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker:. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. of the 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920, pages 441–444, 2006.

[13] A. David, J. Hakansson, K G. Larsen, and P. pettersson. Model checking timed automata with priorities using dbm subtraction. In *Proc. of the 4th Int. Conf. on Formal Modelling and Analysis of Timed Systems*, pages 128–142, 2006.

Fault-prone Module Prediction Using Contents of Comment Lines

Osamu Mizuno
Kyoto Institute of Technology
Matsugasaki Goshokaido-cho, Sakyo-ku
Kyoto 606-8585, Japan
o-mizuno@kit.ac.jp

Yukinao Hirata
Kyoto Institute of Technology
Matsugasaki Goshokaido-cho, Sakyo-ku
Kyoto 606-8585, Japan
y-hirata@se.is.kit.ac.jp

ABSTRACT

Comment lines in the software source code include descriptions of codes, usage of codes, copyrights, unused codes, comments, and so on. From the viewpoint of fault-prone module prediction, comment lines may have useful information for faulty modules as well as in code lines. In fault-proneness filtering approach, which we have been proposed based on text filtering technique, comment lines and code lines in a source code modules are regarded as text without any distinction. For better prediction results of fault-prone modules, the effects of the comment lines on prediction should be investigated. This study conducts experiments using Eclipse data sets to reveal the effects of comment lines on the fault-proneness filtering. The result of experiments was somehow unexpected: prediction using comment lines shows better recall and precision than that of code lines.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management

General Terms

Measurement

1. INTRODUCTION

Fault-prone prediction is one of the most mature areas of software engineering. The prediction of faulty software modules is important for both the reduction of development cost and the assurance of software quality. Much research has been conducted so far [1, 3, 4, 6–13, 16]. Most research used some kind of software metrics, such as program complexity, size of modules, object-oriented metrics, etc., and constructed mathematical models to calculate fault-proneness. This approach is usually based on a hypothesis that the more complex module the more bugs. However, such a hypothesis is not always true. For example, although many if-sentences increase program complexity,

We thus tried to break through the conventional fault-prone prediction by introducing a text-mining technique. This paper introduces a new idea for fault-prone module detection. The idea is inspired from a spam e-mail filtering technique. Spam filters are usually implemented as a generic text discriminator. We thus tried to apply a generic text discriminator to the fault-prone detection. We call our approach "fault-prone filtering." In fault-prone filtering, we consider a software module as an e-mail message, and assume that all of the software modules belong to either fault-prone(FP) modules or not-fault-prone(NFP) modules. After learning of existing FP and NFP modules, we can classify a new module into either FP or NFP by applying a spam filter. One advantage of such a statistical approach is that we do not have to investigate source code modules in detail. We do not measure any metrics explicitly, but implicitly our approach measures only one metric: frequency of tokens found in the source code.

Although we have confirmed usefulness of our approach, there are several remaining issues. One issue is effect of comment lines. Previous implementation of fault-prone filtering did not distinguish the comment lines and code lines. This is because the source code module is passed into text filter without any modification. However, since code lines and comment lines have different roles in source code modules, we need to consider such difference in the fault-prone filtering. We expect to improve the prediction accuracy of fault-prone filtering by using information obtained from comment lines.

To do so, we conducted an experiment to confirm the effect of comment lines in the source code modules to the fault-prone filtering. The result of experiment showed that comment lines have almost the same capability as code lines to predict fault-prone modules.

2. OBJECTIVE

2.1 Fault-prone module filtering

The basic idea of fault-prone filtering is inspired by spam mail filtering. In spam e-mail filtering, the 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.

Intuitively speaking, 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.

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 non-faulty modules. Then, a new module can be classified into fault-prone or not-fault-prone using a classifier.

In this study, we define a software module as a Java class file.

2.2 Effect of comment lines

In this paper, we call "comments" in the source code module as comment lines, and call other lines as "code lines".

Code lines describe a list of operations that developers would like to realize on computers. Comment lines include descriptions of code lines, usage of methods or modules. Code lines are written in a specific programming language, but comment lines are written in a free form.

Previous implementation of fault-prone filtering did not distinguish the comment lines and code lines. This is because the source code module is passed into text filter without any modification. However, since code lines and comment lines have different roles in source code modules, we need to consider such difference in the fault-prone filtering.

For example, comments are usually placed near the difficult codes. Therefore, learning the contents of comment lines may be useful to identify the bug-related part in modules. We expect to improve the prediction accuracy of fault-prone filtering by using comment lines effectively.

3. IMPLEMENTATION OF FAULT-PRONE FILTER

In the previous studies [14, 15], we used a ready-made open-source text classifier, CRM114 [19]. However, use of ready-made software hides essential information in experiments. We thus implement a text classifier from scratch for this experiment.

3.1 Tokenization

First of all, inputs of fault-prone filter must be tokenized. Code lines are tokenized by the Java syntax scanner as follows:

1. Strings with alphabets, numeric characters, and period
2. Operators in Java language
3. Strings with single quotations
4. Strings with double quotations

Other characters, spaces, commas, braces, colons, semicolons, and so on, are used as a separator to make tokens.

As for the comment lines, tokenization is done by similar manner with the case of code lines. However, since it is difficult to determine the end of quoted strings in comment lines, we treat single and double quotations as a separator in the comment lines. Consequently, a code piece `output("Hello, world!");` is tokenized into 2 tokens, "`output`" and "`"Hello, world"`", in code lines, but is tokenized into 3 tokens, "`output`", "`Hello`", and "`world`" in comment lines.

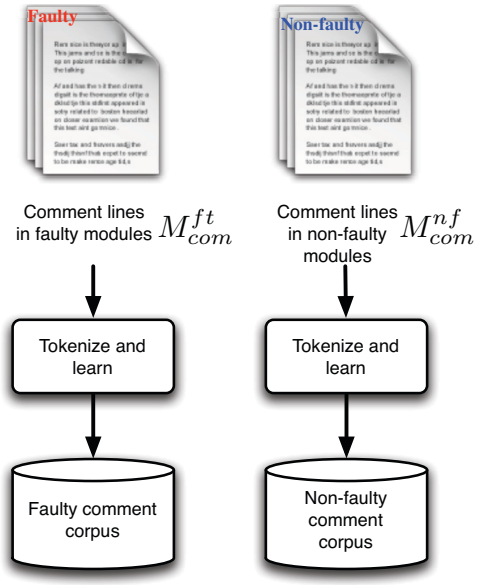


Figure 1: Learning procedure

3.2 Learning algorithm

The algorithm of learning and classification is the same as that of [5].

Step 1 Tokens are extracted from faulty and non-faulty modules. The number of occurrence for each token in faulty and non-faulty modules are counted. For a given token t, the following two hash functions can be defined:

- $nonfaulty(t)$: The occurrence of token t in all non-faulty modules.
- $faulty(t)$: The occurrence of token t in all faulty modules.

Step 2 We construct a hash table to obtain the probability, $P_{ft|t}$, by equation (1) that a module that includes the token t is faulty. Here, N_{nf} denotes the number of non-faulty modules and N_{ft} denotes the number of faulty modules. We call this hash table as a corpus.

$$\begin{aligned} r_{nf} &= \min\left(1, \frac{2 \times nonfaulty(t)}{N_{nf}}\right) \\ r_{ft} &= \min\left(1, \frac{faulty(t)}{N_{ft}}\right) \\ P_{ft|t} &= \max\left(0.01, \min\left(0.99, \frac{r_{ft}}{r_{nf} + r_{ft}}\right)\right) \end{aligned} \quad (1)$$

3.3 Classification algorithm

The classification process is also based on Graham's algorithm [5].

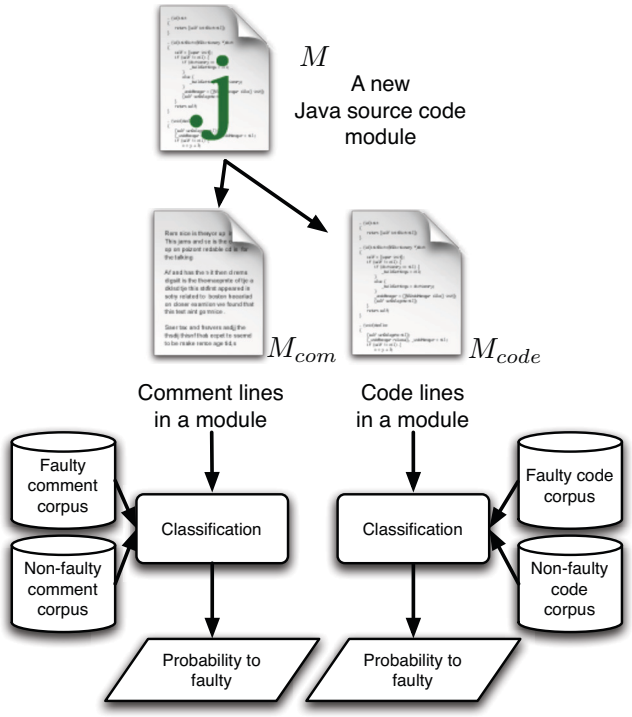


Figure 2: Classification procedure

Step 1 For a new module, distinct tokens are extracted. In this study, we use all extracted tokens. Here, we count the number of tokens, n, for the next step[1].

Step 2 According to the equation (2), the probability P_{ft} is calculated. We determine whether a module is fault-prone or not. If $P_{ft} \geq 0.9$, the module is determined as fault-prone.

$$P_{ft} = \frac{\prod_{i=1}^{n} P_{ft|t_i}}{\prod_{i=1}^{n} P_{ft|t_i} + \prod_{i=1}^{n}(1 - P_{ft|t_i})} \quad (2)$$

To both comment lines and code lines, we separately apply this classification algorithm. Intuitive description of the classification procedure is shown in Figure 2.

4. EXPERIMENT

4.1 Target project

In order to conduct experiments, we prepare a data set obtained from Eclipse project by Zimmermann [2,20], which is called *promise-2.0a* data set. In the data set, 31 complexity metrics as well as the number of pre- and post-release faults are defined and collected. Although *promise-2.0a* includes metrics collected from both files and packages, we used the metrics from files for this study. One of advantages to use *promise-2.0a* from Eclipse is that it is publicly available on the Web. Many researchers can use the same data set and compare their approaches. The data set includes the values of software complexity metrics shown and fault data collected by the SZZ algorithm [18] for each class file, that is, software modules. The data is obtained from Eclipse version 2.0, 2.1, and 3.0. The number of modules in Eclipse 2.0, 2.1 and 3.0 are 6,729, 7,888 and 10,593, respectively.

[1]We can use a limited number of tokens here. To do so, among those extracted tokens, most characteristic n tokens, $t_1 \cdots t_n$ are determined. In more detail, $C_x = abs(0.5 - P_{ft|t_x})$ is calculated for each token t_x, and we select n tokens with the largest C_xs by descending order. For calculation of C_x, if a token t is a never seen one, we assume that $P_{ft|t} = 0.4$. Based on the pre-experiment, we found that the prediction performance does not differ if we use, for example, 20 characteristic tokens. But for simplicity, we used all tokens in this experiment.

Table 1: Target projects

Project	Eclipse		
Version	2.0	2.1	3.0
Number of modules	6,729	7,888	10,593
Number of faulty modules	975 (14.5%)	854 (10.8%)	1,568 (14.8%)
Number of non-faulty modules	5,754 (85.5%)	7,034 (89.2%)	9,025 (85.2%)
Total lines of code	1,267,350	1,564,872	2,098,034
Comment lines	475,486 (37.5%)	577,033 (36.9%)	788,055 (37.6%)
Code lines	791,864 (62.5%)	987,839 (63.1%)	1,309.979 (62.4%)

Table 2: Classification result matrix

		Classified	
		non-fault-prone	fault-prone
Actual	non-faulty	True negative (TN)	False positive (FP)
	faulty	False negative (FN)	True positive (TP)

4.2 Evaluation measures

Table 2 shows a classification result matrix. True negative (TN) shows the number of modules that are classified as non-fault-prone, and are actually non-faulty. False positive (FP) shows the number of modules that are classified as fault-prone, but are actually non-faulty. On the contrary, false negative shows the number of modules that are classified as non-fault-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, precision. Recall is the ratio of modules correctly classified as fault-prone to the number of entire faulty modules Recall is defined by equation (3).

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3)$$

Precision is the ratio of modules correctly classified as fault-prone to the number of entire modules classified fault-prone. Precision is defined by equation (4).

$$\text{Precision} = \frac{TP}{TP + FP} \quad (4)$$

Accuracy is the ratio of correctly classified modules to the entire modules. Accuracy is defined by equation (5).

$$\text{Accuracy} = \frac{TP + TN}{TN + TP + FP + FN} \quad (5)$$

Since recall and precision are in the trade-off, F_1-measure is used to combine recall and precision. F_1-measure is defined by equation

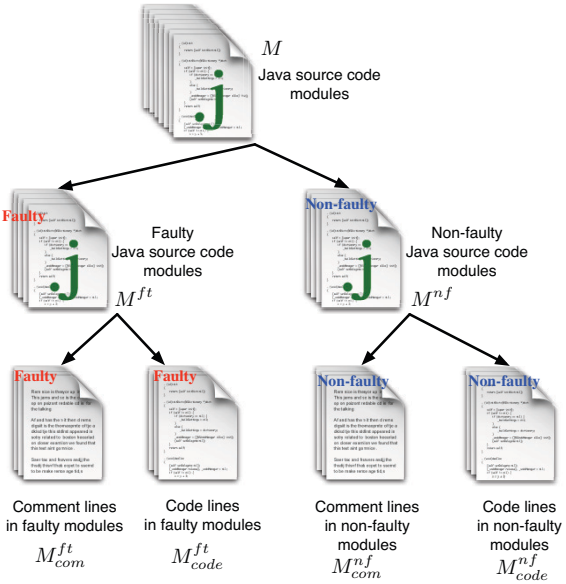


Figure 3: Getting modules

(6).

$$F_1 = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} \quad (6)$$

In this definition, recall and precision are evenly weighed.

4.3 Application to Eclipse

In this section, we describe a procedure of experiments.

We obtained the source code modules from tar archives of corresponding Eclipse versions. We can define faulty and non-faulty source code modules by the metric "post-release failure (*post*)" in *promise-2.0a*. If $post > 0$, a module is faulty; otherwise non-faulty. Based on the faulty/non-faulty information, we divide source code modules into "faulty" and "non-faulty", represented ft and nf, respectively. Moreover, we separate a module into code and comment lines for each source code module. This procedure is shown in Figure 3.

Here, let M_{code} and M_{com} be code lines and comment lines extracted from a module M, respectively. In the classification with code lines, we use only M_{code} as training and test data. Similarly, we use only M_{com} in the classification with comment lines.

The following procedure was conducted in this experiment:

1. For the learning step, we first pick up a version of Eclipse project as a training data set. We then divide the training data set into faulty and non-faulty. For each faulty module, M^{ft}_{code} is extracted and conducted learning for a faulty code corpus. Similarly, M^{ft}_{com} is extracted and learnt in a faulty comment corpus. For non-faulty modules, the same procedure for M^{nf}_{code} and M^{nf}_{com} is performed.
2. For the classification step, we pick up the other version of Eclipse project as a testing data set. The, the testing data set is applied to the classification procedure. For each module in the testing data set, M_{code} and M_{com} are extracted.

Table 3: Result of prediction: (comment, 2.0, 2.1)

		Prediction	
		non-fault-prone	fault-prone
Actual	not faulty	5,952	1,082
	faulty	508	346

Table 4: Result of prediction: (code, 2.0, 2.1)

		Prediction	
		non-fault-prone	fault-prone
Actual	not faulty	5,711	1,323
	faulty	492	362

Table 5: Result of prediction: (comment, 2.0, 3.0)

		Prediction	
		non-fault-prone	fault-prone
Actual	not faulty	7,625	1,400
	faulty	835	733

Table 6: Result of prediction: (code, 2.0, 3.0)

		Prediction	
		non-fault-prone	fault-prone
Actual	not faulty	7,477	1,548
	faulty	934	634

By using code and comment corpuses, M_{code} and M_{com} are classified.

In this experiment, we use versions 2.0 and 2.1 for training and versions 2.1 and 3.0 for testing, since it is natural to use older version for training and newer version for testing. These training pairs are used for different targets: code lines or comment lines. We thus prepare six experiment 3-tuples denoted by (target, training, testing): (code, 2.0, 2.1), (comment, 2.0, 2.1), (code, 2.0, 3.0), (comment, 2.0, 3.0), (code, 2.1, 3.0), and (comment, 2.1, 3.0).

The results of six prediction experiments are shown in Tables 3 – 8.

4.4 Result of experiment

A summary of experimental results are shown in Table 9. Bold figures show the best result. We can find following three findings from Table 9:

1. Predicted results with comment lines have better F_1's than that of code lines.

Table 7: Result of prediction: (comment, 2.1, 3.0)

		Prediction	
		non-fault-prone	fault-prone
Actual	not faulty	7,563	1,462
	faulty	986	582

Table 8: Result of prediction: (code, 2.1, 3.0)

		Prediction	
		non-fault-prone	fault-prone
Actual	not faulty	7,883	1,142
	faulty	1,133	435

Table 9: Summary of experimental results (Threshold = 0.9)

Train	Test	Target	Acc.	Rec.	Prec.	F_1
2.0	2.1	Comment	**0.798**	0.405	**0.242**	**0.303**
		Code	0.770	**0.424**	0.215	0.285
		All	0.782	0.419	0.228	0.294
2.0	3.0	Comment	**0.789**	**0.467**	**0.344**	**0.396**
		Code	0.766	0.404	0.291	0.338
		All	0.777	0.420	0.312	0.358
2.1	3.0	Comment	0.769	**0.371**	**0.285**	**0.322**
		Code	**0.785**	0.277	0.276	0.277
		All	0.781	0.289	0.274	0.281

Table 10: Summary of experimental results (Threshold = 0.5)

Train	Test	Target	Acc.	Rec.	Prec.	F_1
2.0	2.1	Comment	**0.789**	0.419	**0.234**	**0.300**
		Code	0.755	**0.438**	0.204	0.279
		All	0.778	0.426	0.224	0.294
2.0	3.0	Comment	**0.776**	**0.492**	**0.328**	**0.394**
		Code	0.747	0.415	0.270	0.327
		All	0.770	0.430	0.304	0.356
2.1	3.0	Comment	0.738	**0.416**	**0.260**	**0.320**
		Code	**0.768**	0.292	0.254	0.272
		All	**0.769**	0.302	0.259	0.279

2. Recalls are usually higher than precisions.

In Table 9, we show results of regular fault-prone filtering, that is, predictions using both comment lines and code lines. Rows "All" show the results.

As we described in subsection 3.3, the threshold of probability to determine as fault-prone is 0.9. Since we guessed that this threshold is too high, we conducted another experiments using threshold of 0.5. The summarized result is shown in Table 10. Because we decrease the threshold to determine fault-prone, the number of fault-prone modules is increased. Consequently, recalls are relatively higher than that of Table 9.

5. DISCUSSIONS

The result in Tables 9 and 10 are unexpected for us. The result indicates that using only comment lines can achieve a certain extent of prediction accuracy. Furthermore, it is better than that of code lines from the viewpoint of F_1. We can also see that prediction results using both code and comment show the intermediate result between that of comment lines and code lines. However, the evaluation measures are very close between comment lines and code lines. We expected that there is more difference between them. We guess a reason for this result as follows: We used only one revision for training. This may decrease the accuracy of fault-prone filtering. The fault-prone filtering originally aims to utilize historical information of texts in the source code modules. When we adopt such data, we may get different result with this. In order to investigate the effects of comment lines in more detail, we will conduct further research using historical data.

One concern on this experiment is the number of modules that do not include comments. We cannot deal with modules without comments correctly in the prediction with comment lines. Table 11 shows the number of modules without comment lines for each version. We can see that there is few modules without comment lines.

Table 11: Number of modules without comments

version	faulty	non-faulty
2.0	3 (0.31%)	34 (0.59%)
2.1	0 (0%)	0 (0%)
3.0	0 (0%)	3 (0.03%)

Table 12: Comparison with Zimmermann's result [20]

Train	Test	Target	Acc.	Rec.	Prec.	F_1
2.0	2.1	Comment	0.789	0.419	0.234	**0.300**
		Code	0.755	**0.438**	0.204	0.279
		Zimmermann	**0.890**	0.191	**0.478**	0.273
2.0	3.0	Comment	0.776	**0.492**	0.328	**0.394**
		Code	0.747	0.415	0.270	0.327
		Zimmermann	**0.861**	0.171	**0.613**	0.267
2.1	3.0	Comment	0.738	**0.416**	0.260	**0.320**
		Code	0.768	0.292	0.254	0.272
		Zimmermann	**0.864**	0.139	**0.717**	0.233

On the other hand, we need to compare with the other fault-prone prediction approach. The most appropriate subject of comparison is Zimmermann's result [20] since the data we used in this study is quoted their paper. The result of comparison is summarized in Table 12. Table 12 shows that Zimmermann's approach has higher precision than ours, but our approach has higher recall. At least, from the viewpoint of F_1, we can say that our approach is as accurate as that of [20]. For another comparison, we compare with the results obtained by Shihab et al. [17]. Table 13 shows the results. We can see that their result is similar to the result of Zimmermann's. That is, we guess that software metrics based approaches tend to achieve high precision, whereas spam filter based approach high recall. In further research, we need to analyze why our approach shows high recall.

6. CONCLUSIONS

In this study, we tried to investigate the effects of comment lines in the source code modules on the accuracy of fault-prone filtering, which is a text-classification based fault-prone module prediction method. We conducted experiments using Eclipse data sets to reveal the effects of comment lines on the fault-prone filtering. The result of experiments was somehow unexpected: prediction using comment lines shows better recall and precision than that of code lines.

Future work includes more experiments using entire historical data in the repository. In this study, we only used comment lines only, but logs in SCM can be a good candidate of input for fault-prone filtering. On the other hand, analysis to identify important tokens for fault-prone modules is an interesting future research. We are currently tackling this issue.

7. ACKNOWLEDGMENTS

Table 13: Prediction results obtained by Shihab et al. [17]

Version	Acc.	Rec.	Prec.
2.0	0.875	0.285	0.663
2.1	0.898	0.158	0.600
3.0	0.869	0.257	0.641

This research is partially supported by the Japan Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 20700025, 2008–2010.

8. REFERENCES

[1] P. Bellini, I. Bruno, P. Nesi, and D. Rogai. Comparing fault-proneness estimation models. In *Proc. of 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 205–214, 2005.

[2] G. Boetticher, T. Menzies, and T. Ostrand. *PROMISE Repository of empirical software engineering data repository*. http://promisedata.org/, West Virginia University, Department of Computer Science, 2007.

[3] L. C. Briand, W. L. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. on Software Engineering*, 28(7):706–720, 2002.

[4] G. Denaro and M. Pezze. An empirical evaluation of fault-proneness models. In *Proc. of 24th International Conference on Software Engineering*, pages 241–251, 2002.

[5] P. Graham. *Hackers and Painters: Big Ideas from the Computer Age*, chapter 8, pages 121–129. O'Reilly Media, 2004.

[6] L. Guo, B. Cukic, and H. Singh. Predicting fault prone modules by the dempster-shafer belief networks. In *Proc. of 18st International Conference on Automated Software Engineering*, pages 249–252, 2003.

[7] T. M. Khoshgoftaar and E. B. Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 6(4):303–317, 1999.

[8] T. M. Khoshgoftaar and E. B. Allen. Controlling overfitting in classification tree models of software quality. *Empirical Software Engineering*, 6(1):59–79, 2001.

[9] T. M. Khoshgoftaar, E. B. Allen, and J. Deng. Using regressin trees to classify fault-prone software modules. *IEEE Trans. on Reliability*, 51(4):455–462, 2002.

[10] T. M. Khoshgoftaar and N. Seliya. Software quality classification modeling using SPRINT decision tree algorithm. In *Proc. of 14th International Conference on Tools with Artificial Intelligence*, pages 365–374, 2002.

[11] T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical study. *Empirical Software Engineering*, 9:229–257, 2004.

[12] T. M. Khoshgoftaar, R. Shan, and E. B. Allen. Using product, process, and execution metrics to predict fault-prone software modules with classification trees. In *Proc. of 5th IEEE International Symposium on High Assurance Systems Engineering*, pages 301–310, 2000.

[13] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. on Software Engineering*, 33(1):2–13, January 2007.

[14] O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *Proc. of 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 405–414, 2007.

[15] O. Mizuno and T. Kikuno. Prediction of fault-prone software modules using a generic text discriminator. *IEICE Trans. on Information and Systems*, E91-D(4):888–896, 2008.

[16] N. Seliya, T. M. Khoshgoftaar, and S. Zhong. Analyzing software quality with limited fault-proneness defect data. In *Proc. of 9th IEEE International Symposium on High-Assurance Systems Engineering*, pages 89–98, 2005.

[17] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[18] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? (on Fridays.). In *Proc. of 2nd International workshop on Mining software repositories*, pages 24–28, 2005.

[19] W. S. Yerazunis. *CRM114 – the Controllable Regex Mutilator*. http://crm114.sourceforge.net/.

[20] T. Zimmermann, R. Premrai, and A. Zeller. Predicting defects for eclipse. In *Proc. of 3rd International Workshop on Predictor models in Software Engineering*, 2007.

Process Analysis

A Preliminary Study on Impact of Software Licenses on Copy-and-Paste Reuse

Yu Kashima
Graduate School of Information Science and Technology, Osaka University
y-kasima@ist.osaka-u.ac.jp

Yasuhiro Hayase
Faculty of Information Sciences and Arts, Toyo University
hayase@toyo.jp

Norihiro Yoshida
Graduate School of Information Science, Nara Institute of Science and Technology
yoshida@is.naist.jp

Yuki Manabe
Graduate School of Information Science and Technology, Osaka University
y-manabe@ist.osaka-u.ac.jp

Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
inoue@ist.osaka-u.ac.jp

ABSTRACT

Source code of open-source software is permitted to be reused when and only when the conditions of its license are satisfied. There are many different conditions for reusing, since various open-source licenses are used. Therefore, the license of the source code may affect the frequency of reusing or the property of the software for which the source is reused. To identify the relationship between software license and reusing, we are planning to classify copy-and-pasted code fragments based on the license of the fragments. This paper presents a preliminary and manual investigation on a small source file set. The result indicates that the license of a fragment affects the quantity and the license of copied fragments.

Categories and Subject Descriptors

K.5.1 [**LEGAL ASPECTS OF COMPUTING**]: Hardware / Software Protection—*Licensing*
; K.6.3 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Software Management—*Software Selection*

General Terms

Legal Aspects, Experimentation

Keywords

Software License, Open Source Software, Reuse, Copy and Paste

1. INTRODUCTION

Source code of open source software (i.e. OSS) is available for anyone to modify or redistribute. According to the growth of OSS development [16], software developers can reuse huge amount of OSS source code nowadays.

Everyone has to adhere to the license of a software product when he/she obtains or uses the product. The license of a product expresses the intent of the right holder; Several OSS licenses require the derivative works to apply the same license of the original product, i.e. copyleft[1]. Since different right holders have different intents, many OSS licenses are used.

Software reuse is recognized as a practice for reducing the development cost and improving the product quality. Software reuse happens at several levels of granularity; from simple copy and paste of code snippet, to whole the inclusion, to subsystem reuse.

Software reuse must adhere the license of the reused product. Furthermore, developers must pay attention not to violate the licenses of the product they are developing by reusing other software.

OSS licenses have different attitudes toward reuse.[14] For example, the GNU General Public Licenses (GPL)[5] requires derivative works, including a product that contains a code fragment copied from a GPL product, to be distributed under the GPL. On the other hand, the BSD license[13] only requires that the copyright notice, the license text and the disclaimer are retained. Thus, the reused product must be carefully selected to not conflict with the license of a product under development.

Consequently, software license may affect the frequency of code reuse or the variety of the derived products. For instance, the source code distributed under a copyleft license may be reused less frequently and reused in narrow a variety of software products, compared to non-copyleft license

[1] `http://www.gnu.org/copyleft/` (accessed Oct 2010)

source code. To the knowledge of the available, there is no quantitative study of CnP reuses from the point of view of software license.

This paper presents a preliminary study on the impact of the software licenses on CnP reuse on a small data set. The preliminary experiment intends to assess the process of evaluating CnP source code based on the software license. The target of the experiment is Java source code distributed in Debian GNU/Linux lenny[3]. We investigated copy-and-pasted code fragments that are distributed under three major licenses; the 3-clause BSD license (BSD3), the GPL Version 2 or later (GPLv2+) and the Apache License 2.0 (Apachev2)[1]. To detect copy-and-pasted code fragments, CCFinder[10] and LNR filtering criterion are used.

Result of the experiment shows that source code distributed under the BSD3 or the Apachev2 is more frequently reused than the GPLv2 code. On the other hand, Apachev2 and GPL code has a trend to be reused in code distributed under specific licenses. Through the preliminary experiment, we confirmed that the process of evaluation is effective and applicable for large data set.

The rest of this paper is organized as follows. Section 2 explains background of this study. Section 3 illustrates design and result of the experiment, and Section 4 interprets the result of the experiment. Section 5 discusses about the validity of the experiment, finally, Section 6 shows the conclusion and future remarks.

2. SOFTWARE LICENSE AND REUSING

Nowadays, many open-source licenses are used; for instance, the Open Source Initiative officially approves 67 licenses [2]. This section illustrates three of most major licenses, 3-clause BSD license (BSD3), Apache License 2.0 (Apachev2) and GNU General Public License version 2 (GPLv2) from the perspective of a developer who makes a derivation product. From the point of view of a developer, different licenses mean different restrictions.

When a developer makes a derivative work from a BSD3 product, the former should retain the copyright notice and the full text of the license.

If a developer makes a derivative work from an Apachev2 product, all copyrights, patents, trademarks, and attribution notices should be retained in the new product. Moreover, changed file also should have notices of the modification.

When a developer makes a derivation from a GPLv2 product, the whole derivation must be distributed under the GPLv2 and, changed file should have notices of the modification.

2.1 OSS License and Reuse

Software reuse is recognized as a practical method for developing high quality software with low cost. A developer can reuse source code of OSS products since the source code is easily available.

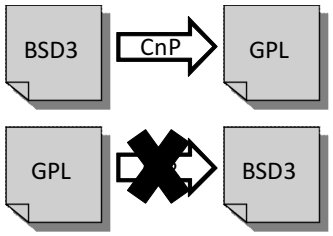


Figure 1: Reusing source code in a different license product

Copy-and-paste (CnP) is one of the most frequently performed methods of reuse. In CnP reuse, source code is copied, modified if needed, and finally, used as a part of new product.[11, 14]

When reusing existing software, both the license of the product being reused and of the developing product must be satisfied. In case the two licenses are incompatible, both cannot be satisfied simultaneously. For example, Apachev2 products cannot be incorporated into the GPLv2 products, since several requirements in the Apachev2 conflict with a clause in GPLv2.[3]

Furthermore, even if the licenses do not conflict, an OSS product cannot be reused if license of the developing product cannot be changed. For example, GPLv2 source code cannot be incorporated into a BSD3 product. In contrast, BSD3 source code can be incorporated into a GPLv2 product because the restrictions of the BSD3 are included in the GPLv2. (Figure 1)

2.2 Impact of Software License on Copy and Paste Reuse

As described above, the license of the reused product is the main concern to software reuse. If a license of source code doesn't match a developing product, the source code cannot be reused in the developing product unless the license of the developing product is changed. Therefore, it is clear that the reusability of software depends on not only functionality or quality, but also license of the software.

Whether reusing source code by CnP is allowed or not depends on its license. As previously explained, when a developer reuses source code, the developer must observe the source codes license. Licenses with very restrictive terms (i.e. GPL) might make it difficult to satisfy their condition, while licenses with more relaxed conditions have easier to meet requirements.

We make the following assumptions:

- source code with a relaxed license is reused under various licenses.

[2] `http://www.opensource.org/licenses` (accessed Oct 2010)

[3] `http://www.gnu.org/licenses/license-list.html` (accessed Oct 2010)

Table 1: kept packages and excluded packages

kept packages	excluded packages
antlr3	antlr
asm3	asm, asm2
db4.6	db4.2, db4.3
junit4	junit
tomcat6	tomcat5.5

- source code with a relaxed license is reused more frequently than source code under a strict license.

3. EXPERIMENT

The goal of this study is an investigation of the impact of licenses on CnP reuse in OSS. To achieve the goal, we need to analyze actual OSS to identify the relationship between licenses and CnP reuse. In this paper, we analyzed the source files of OSS in order to validate our method.

This study focused only source code reuse across applications. We believe that the impact of license on source code reuse within an application is small. Because, even though there are CnP between files under different license, a license problem occurred by these files must be resolved.

3.1 Analysed Code

We selected a part of the source files of Debian/GNU Linux as our analysis target for the following reasons:

- Various licenses are used by it.
- It includes different type of software.

First, we downloaded the packages contained in the main section of Debian/GNU Linux; second, we extracted the content of the ".tar.gz" files; finally, we selected the source code written in Java(.java) the target of our analysis. Consequently, the analysis target consisted of 77452 files (8530896 LOC) from 452 packages.

Note that we kept one of several packages having different versions respectively and excluded the others in the analysis target. For example, between asm, asm2 and asm3, we kept asm3 and excluded asm and asm2 from the analysis target. Table 1 shows the kept packages and the excluded packages.

3.2 Experiment Method

Figure 2 shows an overview of the method of this experiment.

Step 1 We identified the licenses of each file by analyzing the description specifying its licenses in the file. We used Ninka[7] to identify the licenses. Ninka is a tool which analyzes descriptions specifying licenses in a source file and identifies specified licenses. The reason why Ninka was used in this step is that Ninka is a state-of-art license identification tool and more precise than other existing license identification tools such as FOSSology[8]. Note that not only a source file but also a package can have a license. According to the Debian Policy[4], Each package has a 'copyright' file specifying the license of the package. However, this step didn't use this 'copyright' file because the license of source files in the package does not always correspond to the license of the package[6, 8].

Step 2 We extracted the clone sets[10] created by copy-and-paste. A clone set is an equivalence class of the clone-relation. The clone-relation is defined as an equivalence relation. The clone-relation holds between two code portions if (and only if) they are the same sequence. A code fragment which is similar to another is called code-clone. We give a detailed description of the method for extracting clone sets in Section 3.3.

Step 3 We extracted clone sets including a code fragment under specific license A, and we classified and counted the code fragments in these clone set based on their license. Since the source of the CnP cannot be retrieved from code clones, the direction of CnP was not considered in this experiment.

Table 2: Distribution of licenses in all files of the analysis target

License Name	#File	
Apachev2	16350	✓
GPLv2+	8160	✓
LesserGPLv2.1+	6534	
GPLnoVersion,GPLv2+,LinkException	5887	
GPLv2	3222	
BSD3	2181	✓
GPLv2,ClassPathException	1498	
No description specifying license in the file	15813	
Fail to analyze the description	6862	
SeeFile	2786	

Due to time limitation, we investigated only some licenses that are widely used and differ from each other in the condition on CnP reuse. Table 2 shows the abbreviated names of the licenses ranked in descending order by the number of files under it in the analysis target. Licenses marked with a check are the ones we investigated. Table 3 shows the full names of the licenses in Table 2. When we show a version of license, "v<number>" follows. In addition, if users can choose this version or any later, "+" follows. Furthermore, if user can choose several licenses, usable license names follow after ",".

We analyzed Apachev2, GPLv2+ and BSD3 as Table 2 shows. Apachev2 is used by the largest number of files in the analysis target; GPLv2+ and GPL derivatives are counted in the second. BSD3 is used by the largest number of files except files under Apachev2 or licenses which conditions like GPLv2+ such as "LesserGPLv2.1+", "GPLnoVersion, GPLv2+, LinkException", "GPLv2+".

3.3 CnP Detection

We used CCFinder[10] for detecting code fragments created by CnP. CCFinder is a code-clone detection tool. We can

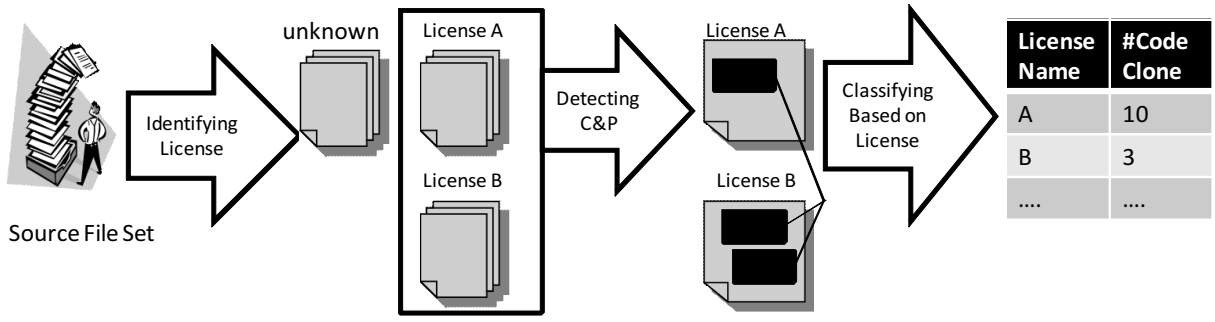


Figure 2: Overview of evaluation method

Table 3: License name abbreviations

Abbreviation	Name
Apache	Apache Public License
BSD3	Original BSD minus advertisement clause
ClassPathException	GNU Classpath License
CPL	Common Public License
GPL	General Public License
LesserGPL	Lesser General Public License
LibraryGPL	Library General Public License
LinkException	GPL linking exception
MITX11noNotice	MIT License/X11License
MPL	Mozilla Public License
MX4jLicense	MX4J License
publicDomain	Public Domain
SeeFile	File Points to another where the its license is
subversion	Subversion License

extract code fragments copied and pasted by detecting code-clones.

In this experiment, when we detect CnP from the analysis target, we detected code-clone ignoring the identifier names, because we wanted to detect code fragments in which identifier names changed after the CnP.

Additionally, we used LNR to filter clone set involving code fragment created by CnP from extracted clone set. LNR is the number of tokens of non repeated elements in a code fragment. A code fragment which LNR is small might includes only variable declarations, assignments or getter/setter declarations. These code fragments are called language specific clones. By contrast, if the LNR of a code fragment is large, the code fragment has higher classes to be created by CnP. Therefore, in this experiment, we presumed that a clone set is created by CnP if the average of LNR of code fragments is over some specific value. This value was set at 50, because our experience shown that 50 is an appropriate value to exclude language specific clones.

3.4 Results

Table 4 shows the number of code fragments under each license; Table 5 shows the case of Apachev2; Table 6 shows the case of GPLv2+.

Table 7 shows the number of code fragments, files and the ratio of code fragments to files of BSD3, Apachev2 and GPLv2+. We can see from Table 7 that the order arranged in descending order of number of code fragments compared to number of files is BSD3, Apachev2, GPLv2+.

Table 4: Distribution of code fragments having clone-relation to files under BSD3

License Name	#Fragments
BSD3	613
GPLv2+	20
Apachev2	16
LibraryGPLv2+	14
GPLv2,ClassPathException	1
LesserGPLv2.1+	1

Table 5: Distribution of code fragments having clone-relation to files under Apachev2

License Name	#Fragments
Apachev2	1533
Apachev1.1	316
LesserGPLv2.1+	42
MPLv1_1	33
BSD3	29
MX4JLicensev1	16
GPLv2+	4
LibraryGPLv2+	3
MPLv1_0	2
MITX11noNotice	2
publicDomain	1
subversion+	1
EPLv1	1

We found that CnP fragments tend to have the same license. In the case of BSD3, code fragments under BSD3 account for 92% of all code fragments. Similarly, in the case of Apachev2, code fragments under Apachev2 account for 77% of all. In the case of GPLv2+, code fragments under GPLv2+ account for 48% of all.

In the case of Apachev2, code fragments under Apachev1.1 account for 16% of all. Similarly, in the result of GPLv2+, "GPLnoVersion, GPLv2+, LinkException" account for 41% of all.

If we evaluate the results from the point of view of the number of licenses, Apachev2 has CnP relationship to the largest number of licenses. Apachev2 has CnP relationship to 13 licenses. BSD3 and also GPLv2+ have CnP relationship to 6 licenses.

Table 6: Distribution of code fragments having clone-relation to files under GPLv2+

License Name	#Fragments
GPLv2+	268
GPLnoVersion,GPLv2+,LinkException	225
BSD3	28
LibraryGPLv2+	20
Apachev2	4
LesserGPLv2.1+	4

Table 7: Number of code fragments and files

	#Fragments	#File	#Fragments/#File
BSD3	665	2181	0.304906
Apachev2	1983	16350	0.121284
GPLv2+	549	8160	0.067279

4. DISCUSSION

In the result of Apachev2, there is large number of code fragments having CnP relationship to code fragments under Apachev1.1. We believe that Apachev1.1 has been changing to Apachev2 currently because Apachev1.1 is an old version of Apachev2.

GPLv2+ has the smallest number of #Fragments/#File in table 7. This result shows that GPLv2+ is reused less frequently than BSD3 and Apachev2. We believe that code fragments under other licenses are copied into code under GPLv2+, however, there is little case that a code fragment under GPLv2+ is copied into code under another license not in the GPL family. We believe the reasons are:

- Copy-and-Pasting a code fragment under GPLv2+ to code under another license except GPLv2+ and GPLv3 violates the condition of GPLv2+.
- Copy-and-Pasting a code fragment under Apachev2 or BSD3 in code under GPLv2+ is permitted.
- Copy-and-Pasting a code fragment under GPL family such as LesserGPL or "GPLnoVersion, GPLv2+, LinkException" and changing their license to GPL is permitted.

Code fragments under BSD3 or Apachev2 are reused more frequently than code fragments under GPLv2+. We believe that it is easier to satisfy the conditions of the BSD3 or the Apachev2 than these of the GPLv2+. In fact, code fragments under the Apachev2 have CnP relationship to code fragments under more licenses than code fragments under the GPLv2+. Therefore, we suggest that code fragments under Apachev2 are copy-and-pasted frequently into code under another license.

In section 3.4, we shown that all licenses share the common characteristic that code fragments under each of the own license have the highest proportion in each result of investigations. We suggest that there are many cases that code fragments are copy-and-pasted to code fragments created by a developer in same development organization.

There are many open source licenses as described in Section 2, the developers of the OSS have to select a license from the many available. The result of the investigation may be contributory to license selection.

This experiment is a preliminary study. Therefore, we plan to perform an experiment on a larger analysis target. It is possible to detect CnP in larger source code because we can split a set of source code to decrease their size as possible as to analyze by CCFinder, and merge these results. On the other hand, identifying license of code fragment in large object is also possible. Because Ninka analyzes files one by one.

5. THREAT TO VALIDITY

It is possible that the result of the experiment depends on the CnP detection capability of code clone detection tool CCFinder[10]. We are motivated to use CCFinder in the experiment because one of main application of code clone detection tool is CnP detection, and the usefulness of CCFinder is shown in Bellon's benchmark[2]. As future work, comparative experiments with other available code clone detection tools (e.g., CP-Miner[12]) should be performed.

The result of the experiment also depends on the characteristic of metric LNR and its threshold value. We use the metric LNR to exclude language-dependent clones (e.g., repeated setter/getter invocations in Java programs). As future work, we should perform the experiments to show the effect of threshold values of metric LNR. RNR[9] is another metric to exclude language-dependent clones. Currently, we are planning to perform the comparative experiment with LNR and RNR.

Ninka is employed to identify the licenses of source files. We believe that using Ninka is valid because the accuracy of Ninka is good; In [7], recall was 82% and the precision was 96%.

However, Ninka cannot detect a license if a source file contains no description about license. In addition, Ninka also cannot detect a license if the license is not registered in the database. In this experiment, source files that Ninka couldn't detect their license are removed from the target of detecting CnP. Hence, source files which were not detected their license didn't influence results.

The analysis target is a small portion of source files in Debian GNU/Linux. Therefore, this result may include a lot of sampling error. For this reason, we believe that applying this result to general OSS is not acceptable.

6. CONCLUSION AND FUTURE WORK

This paper performed a preliminary study on impact of software licenses on CnP reuse. Open-source Java source code in Debian/GNU Linux was analyzed and classified based on their license. In particular, copied code fragments that relate to the source files distributed under 3-clause BSD license, Apache License version 2.0 or GPL version2 or lator were analyzed.

The result of the experiment shows that most of the code

fragments are copied to files distributed under same license or relative licenses that is designed by same organization. On the other hand, comparing to BSD3 and GPLv2+, Apachev2 code fragments are copied into files that are distributed under a various licenses. By contrast, almost all GPLv2+ code are copied between the files distributed under the GPL family licenses.

We are planning a large scale experiment. Especially, the all source code and licenses in OSS products in Debian GNU/Linux is a target of the experiment.

This paper focused on clone sets created by CnP. Thus, the direction of copying is not identified. To clarify the true impact of the license to copy-and-pasting, origin analysis is required to know the direction.

To identify the developing organization or the developer who copied a fragment is also important to clarify the impact.

In this experiment, we didn't discriminate between code clone generated by copy-and-pasted and code clone generated by including library in source form. Discriminating these code clone, it will be clear that difference of an impact of license on their cases. Therefore, we plan to retrieve code clone generated by including library's source code using with FCFinder[15].

7. ACKNOWLEDGMENTS

This work has been supported by Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No.21700031), and Grant-in-Aid for Research Activity start-up (No.22800040).

8. REFERENCES

[1] Apache Software Foundation. Apache license, version 2.0. `http://www.apache.org/licenses/LICENSE-2.0` Accessed Oct 2010.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577 –591, Sept. 2007.

[3] Debian Project. Debian gnu/linux. `http://www.debian.org/` Accessed Oct 2010.

[4] Debian Project. Debian policy manual. `http://www.debian.org/doc/debian-policy/` Accessed Oct 2010.

[5] Free Software Foundation. GNU general public license. `http://www.gnu.org/licenses/gpl.html` Accessed Oct 2010.

[6] D. M. German, M. Di Penta, and J. Davies. Understanding and auditing the licensing of open source software distributions. ICPC '10, pages 84–93, 2010.

[7] D. M. German, Y. Manabe, and K. Inoue. A sentence-matching method for automatic license identification of source code files. In *ASE 2010*, pages 437–446, 2010.

[8] R. Gobeille. The fossology project. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 47–50, New York, NY, USA, 2008. ACM.

[9] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Information & Software Technology*, 49(9-10):985–998, 2007.

[10] 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*, 28:654–670, 2002.

[11] J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio. Development with off-the-shelf components: 10 facts. *IEEE Software*, 26:80–87, 2009.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Soft. Eng.*, 32(3):176 – 192, March 2006.

[13] Open Source Initiative. The BSD license. `http://www.opensource.org/licenses/bsd-license.php` Accessed Oct 2010.

[14] M. Ruffin and C. Ebert. Using open source software in product development: A primer. *IEEE Software*, 21(1):82–86, 2004.

[15] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in freebsd ports collection. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 102 –105, 2010.

[16] W. Scacchi. Free/open source software development: recent research results and emerging opportunities. In *ESEC/FSE 2007*, pages 459–468, 2007.

Using Program Slicing Metrics for the Analysis of Code Change Processes

Raula Gaikovina Kula, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida
Graduate School of Information Science, Nara Institute of Science and Technology
Takayamacho 8916-5, Ikoma, Nara 630-0101, JAPAN
{raula-k,kyohei-f,yoshida}@is.naist.jp, iida@itc.naist.jp

ABSTRACT

Software Process Improvement is increasingly becoming a major activity for most software development organizations due to the benefits seen in the cost and business value. Software process assessments, however, can be tedious, complicated and costly. This research explores alternative assessments by analysis of fine-grained processes with its related source code characteristics.

This approach attempts to propose program slicing metrics for code changes in a software project. Using the wxWidgets software project as the case study, through project artifacts such as issue tracking system and change logs, this research explored relationships between code characteristics and its fined grained processes.

This research contributes to the development of assessment tools to assist with Software Process Improvement. It opens possibilities for assessment tools for fine-grained software processes.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]

General Terms

Experimentation

Keywords

Program Slicing, Code Change Process, Micro Process Analysis

1. INTRODUCTION

1.1 Background and Related Works

The related works are divided into two parts 1) Software Process Assessment and 2) Program Slicing.

Improvement of software processes in software development is seen as a major activity for larger software organizations as benefits are seen in the cost and business value of improvement efforts, as well as the yearly improvement in productivity of development, early defect detection and maintenance, and faster time to market [1].

A number of studies have outlined issues relating common software process improvement assessment methodologies such as CMMI[2] and ISO 9000 [3] [4] [5] [6]. Much of the issues relate to cost of assessment and implementation, especially for small software organizations [7]. In addition, these generic processes assessments are sometimes tedious and not specifically tailored. Other demotivators are higher management support, training, awareness, allocation of resources, staff involvement, experienced staff and defined software improvement process implementation methodology is critical for software process improvement [8].

There has been several related work into trying to make the models easier and better to use. Yoo [9] suggests a model that combines CMMI and ISO methods. Amburst [10] takes a different approach by treating software as manufacturing product lines, therefore making the processes systematic and generic.

An alternative measure of software processes can be done through the inspection of process execution histories. Performed at the developer's level, these measures are referred to as fine-grained processes 'micro processes'. Related research has been done by Morisaki el al [11]. This approach analyzes the quality of fine grained processes by studying the process execution histories such as change logs. This is called Micro Process Analysis (MPA). Quality of the process is measured by the execution sequence and occurrences of missing or invalid sequences.

Program Slicing is a concept first described by Weiser [12] as a method to study the behavior of source code through the flow dependency and control dependency relationship among statements. Program slicing metrics can be utilized various applications such as software inspection, which can be used to reduce defects at a fine-grained level [13]. Other related works have used program slicing metrics to classify bugs. Pan et. al. [14] showed that program slicing metrics work as well as conventional bug classification methodologies. This research proposes program slicing based metrics to describe code changes.

Combining both fields, our research aims better understand fine-grained software process by studying MPA and their code change characteristics using program slicing techniques. Our previous research with MPA and program slicing techniques [15] had focused particularly on the bug fixing process and analyses of the program slicing metrics were performed at file level. The research indicated that there is indeed a relationship between how the micro processes are executed and code-based characteristics such as McCabe's Cyclomatic Complexity (CC) [16] and lines of code (LoC).

1.2 Research Objective

Our motivation is to present a simple alternative model for software process improvement at the micro level. Current models of process assessment are at the software life cycle level and require complicated assessments with highly trained assessors. More specifically, this research is part of a proof of concept towards a prediction model based on source code properties.

Building from our previous works [16][18], this paper explores program slicing metrics specifically at *function level*, with the aim to improve from our work by only including functions edited during a code change. Also previous works focused on bug fixes and the bug fixing process, however, it was found that bug fixes, enhancements and patches are not clearly categorized. Therefore this research we broadened our scope to all code changes and the micro processes that they follow.

Based on our objective and considering code characteristics are the properties of a program slice, we formulated and tested the following research questions:

- *RQ1:* Do Code changes with similar micro process execution have similar code characteristics?
- *RQ2*: Are Program slicing metrics useful when finding a relationship between code change processes and code characteristics?

Section 2 presents the methodology as well as the proposed approach. Section 3 then explains the experiment using the approach and tools to carry out the experiments. Section 3 also presents the results of the experiments. Section 4 is a discussion and analysis of the results as well as future work. Finally, section 5 outlines the conclusion.

2. METHODOLOGY

2.1 Code Change Processes

This research focuses on the day to day processes performed in the development of software. This paper limits its data collection of code change information from the following two repositories:

1) Issue Tracking Repository

Typically a software development team uses a system to manage bugs and patches in a medium to large scale project. Trac is a wiki and issue tracking system Trac is used to track progress of any type of changes to the source code. This research utilized the Trac Management System[1] to help track all the code changes in a software project.

2)Source Code Repository

The source code repository refers the system manages changes to source code in a software project. The two main system used is the Subversion (SVN) and Concurrent Versioning System (CVS). For this research, both SVN and CVS repositories were used to gather data on bug characteristics.

[1] http://trac.edgewall.org/

To understand the micro processes behind code fixes we inspected the workflows of how fixes and changes are implemented in a software project. Inspecting the workflow of the bugs and patches in the tracking system we can be able to group and categorize code changes. Since each project has its own tailored workflow, we used workflows tailored to the project.

2.2 Characterization of Code Changes using Program Slicing Metrics.

To assess the bug characteristics, program slicing metrics are used in this approach to describe code change characteristics. This research looks to use program slicing as measure of the code changes.

In this research we refer to *editedfunction* as functions that are edited during a code change. This is the criterion of the program slice. *Back slice functions* of a program slice as functions that the edited function is dependent on. For *forward slice functions* we refer to functions that the edited function has an effect on. This is illustrated in Figure 1 below.

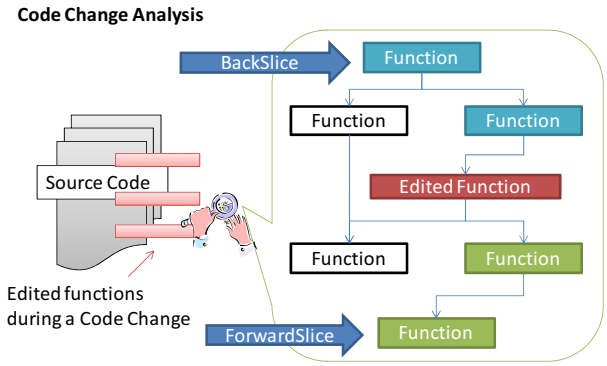


Figure 1. Example of BackSlice and ForwardSlice for a program slice

2.3 Code Change Extraction and Analysis

The analysis of the code change process includes 3 steps: 1) micro process extraction and analysis and 2) code based metrics extraction and analysis, and 3) grouping and comparing data. Details of each step are described as follows:

1) Micro process extraction and analysis:

This step involves data mining and extraction of code chance related attributes from both the source code repository and the issue tracking system. From the source code repository for each fix, the functions that were edited were extracted along with the date of edit. During the extraction, the exact affected functions are retrieved by comparing the fixed change set revision against the previous revision.

2) Code Based Metrics:

This step involves analysis of the code using the program slicing metrics. The following explains in detail how each metric was used to for bug classification. Our previous work [17] used

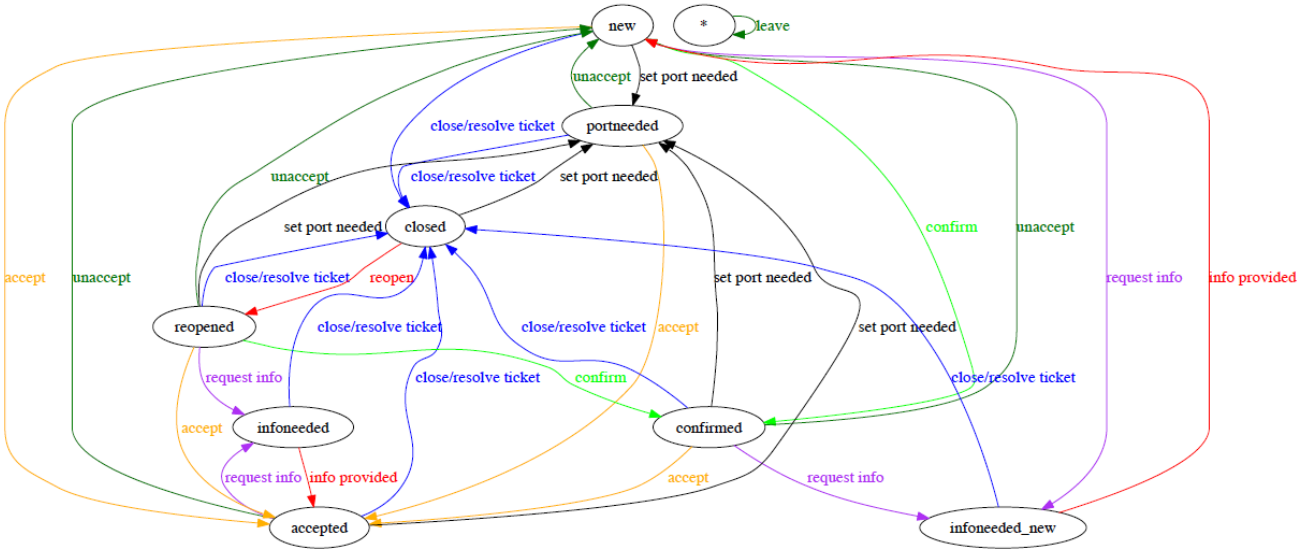


Figure 2. The micro process workflow of code change in wxWidgets (*http://trac.wxWidgets.org/wiki/wxTracWorkflow*).

Cyclomatic Complexity (CC) and Lines of code (LoC) as bug characteristic metrics. These two metrics were originally chosen as CC would be an attempt to measure the complexity of the function and LoC to measure the size of the change. In this research we use CC, however, we replaced LoC with counting the number of functions as a similar measure of size. Also we have some non-program slicing metrics to compare effectiveness.

The follow metrics were proposed and used:

- **Edited Function CC:**

This is a non-program slicing metric.If *editedfunction* is defined as a set of functions that were edited during a code change and *CC(f)* is the Cyclomatice Complexity (CC) of a function, EditedFuncCC metric is defined as:

$$EditedFuncCC = \bigcup_{f \in editedfunction} CC(f) \quad (1)$$

- **Back and Forward CC slice functions:**

Equations 2 and 3 show the calculation of how a backSliceCC and fwdSlice is calculated. As explained in Figure 1, *BackSlice* and *forwardSlice* results of when each *editedfunction* is sliced to find its depending and dependant functions. The CC for all associated functions for all *editedfunctions* are combined to give the resultant CC for the code change.

$$BackSliceCC = \bigcup_{f \in backSlice} CC(f) \quad (2)$$

$$FwdSliceCC = \bigcup_{f \in forwardSlice} CC(f) \quad (3)$$

- **Number of Edited Functions:**

$$NumEditFunc = |editedfunction| \quad (4)$$

Equation 4 states that *NumEditFunc* is the total number of edited functions. This metric is used to count how many functions are edited in a code change. This is non program slicing metric.

- **Number of Back and Forward Slice functions:**

$$BackSliceCount = \bigcup_{f \in backslice} |f| \quad (5)$$

$$FunctionSliceCount = \bigcup_{f \in forwardSlice} |f| \quad (6)$$

Equations 5 and 6 illustrate that by taking the cardinality of the *backSlice* and *forwardSlice* in relation to the *editedfunction*, we can count how many functions were associated with the code change. This metric is used to count the functions that are either depend and are dependent on a code change.

3) Groupings and comparing data:

After grouping the code changes based on their micro process execution we applied the metrics formulated in the previous step. We then analyze to find trends and relationships between micro process execution and the metrics.

3. CASE STUDY: WXWIDGETS

To test the methodology we conducted our case study using the Open Source Software wxWidgets[2] WxWidgets is a C++ library that lets developers create applications for Windows, OS X, Linux and UNIX on 32-bit and 64-bit architectures, as well as several mobile phones platforms including Windows Mobile, iPhone SDK and embedded GTK+.

The program slicing tool CodeSurfer [18] was used to generate the program slices. Our java based extraction tool acted as a web spider searching and parsing the online data repositories. Data

[2] http://www.wxWidgets.org/

was extracted from the wxWidgets Trac system and source code repository.[3]

3.1 Findings

3.1.1 Data Extraction and Selection

We analyzed wxWidgets version 2.8.11 (as of 04/16/2010) which had a size of 620389 lines of code containing 23203 functions.

Using our data extraction tool we analyzed 660662 change sets (Revision changesets 1 – 64005). The criteria for change set selection was based on the following factors 1)The code fix was linked to the wxWidgets Trac, therefore having a issue identification number and 2).The code fix contained functions that existed in the wxWidgets version that was sliced. 507 change sets passed this criteria and therefore used for analysis.

3.1.2 Micro Processes of Code Changes

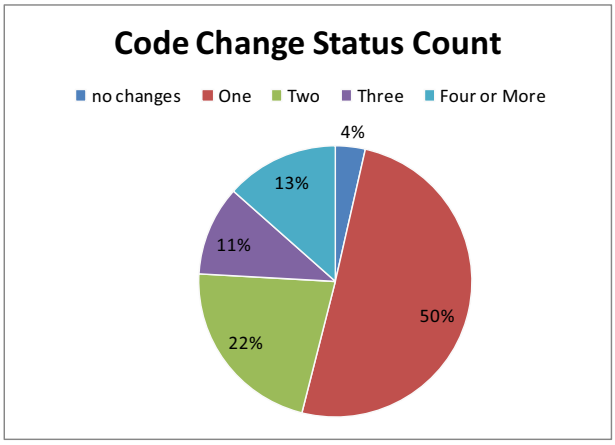


Figure 3. Status change for code change

Figure 2 illustrates the workflow used by wxWidget developers when dealing with a code change, this is regardless of being a bug or a patch or an enhancement model. Using this model we can see the states of the process of implementing a new code change. The change states are: *new, confirmed, closed, reopened, info_needed, port needed and new info_needed new*. As shown in Figure 3 using our extraction tool we were able to for each fix, find the number of status changes.

3.1.3 Grouping and comparing Code Changes

Using the grouping of the status counts as outlined in Figure 3, we then applied our proposed program slicing metrics. The results are illustrated in both Figures 4 and 5.

Figure 4 shows the proposed metrics used to compare and measure the sum of the CC of the functions affected. Starting from the left, the first graph shows the sum of the CC of the functions. However, when looking at the other two graphs, the program slice metrics for *BackSliceCC* in particular does not follow the trend as code changes with more than three changes have lower *BackSliceCC* edited. As seen, there is a trend that as the number of status changes increases with CC.

Figure 5 also suggests the correlation between the change status count and the number of functions edited. Similar to the CC metrics, it seems the number of functions increase as the number of times a status changes increase. Furthermore, with the program slicing based metrics *BackSliceCount* and *ForwardSliceCount* also support the trend.

4. DISCUSSION

4.1 Threats to Validity

Firstly the main threats to validity would be the accuracy of the data extracted as well as statistical analysis of the groupings. Still as a proof of concept, we are more interested in a correlation between program slicing metrics and micro processes. It is envisioned that once a reliable set of metrics is found, further methods of statistical tests can be applied to prove the validity of the data. We also hope to test across more different software projects.

Another threat is that the data analyzed are related to the software release version analyzed. In addition, CodeSurfer does not handle interface related code. When performing the data extraction, we only realized that out 660662 change sets, only 500 code sets were related to the version that we are analyzing. This is probably due to disappearance of functions or files or interface related code changes.

Finally code changing workflows are tailored specific therefore the status counts and complexity of micro processes will differ from organization to organization.

4.2 Evaluation

When looking at the workflow of wxWidgets in Figure 2, the workflow seems to be tedious it can be assumed that since 50% just use one state change, it does contradict the need for such a complex workflow.

Another interesting discussion point is the determination of the complexity of a code change based on its code changes. According to normal micro process analysis (MPA) [11] a normal change entails the following sequence: 1) reporting and confirmation of an issue, 2) assignment to developers, 3) Testing and applying code changes 4) closing of the issue. Findings however, suggest maybe the steps maybe be skipped. Causes could be single developer assignment and working on code change, being a simple fix or such events are not reported into the system. Further work will involve further investigation of the exact type of status change and to identify what are the 'normal' state change for a typical code change.

When applying the proposed metrics to the code changes based on their status counts we found an interesting correlation. Although not statistically proven, the number of functions increases as the change status increases. This was even evident in non-program slicing metrics of sum CC of edited functions and number of edited functions. Again further work is proposed to investigate

[3] http://trac.wxWidgets.org/ and
http://svn.wxWidgets.org/viewvc/wx/wxWidgets/

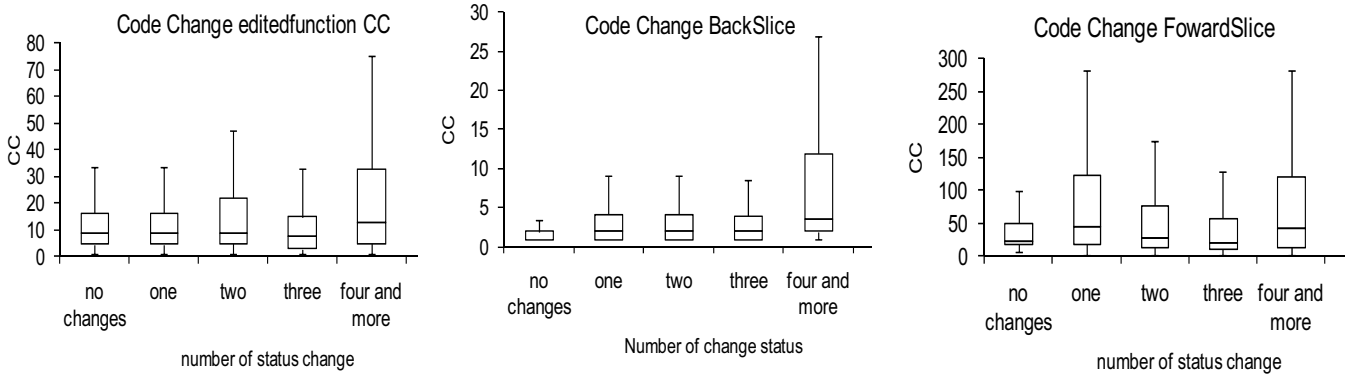


Figure 4. Graphs showing the CC of edited and program sliced functions grouped by status count of code change

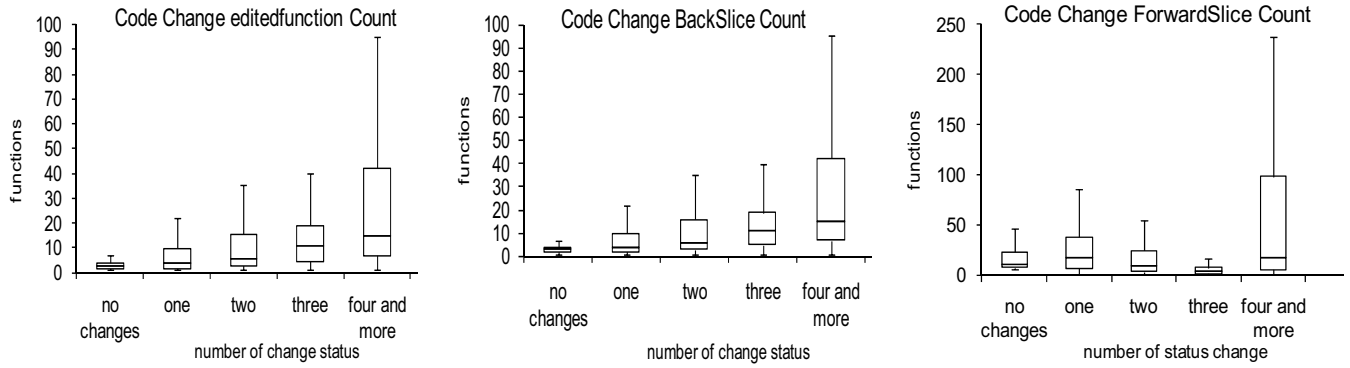


Figure 5. Graphs showing the number of edited and program sliced functions grouped by status count of code change

this further. However this correlation is not consistent with the CC metrics as seen with *ForwardSliceCC*. Again further case studies and other metrics may be proposed to better evaluate these trends.

4.3 Testing Hypothesis and Future Work

When referring to the research questions RQ1 was proven correct as we see a relationship between the complexity of the micro processes and the program slicing metrics of the affected code. Results showed that for RQ2, more current results have mixed results as *BackSliceCC* did not follow a relationship. However it may be due to being a bad metric as other program slicing metrics showed the trend. Therefore RQ2 still is not fully proven to be true.

In regards to the objective, this research contributes to the feasibility of the relationship of micro processes of code changes and their code based characteristics. This can be implemented by a framework that will help assist developers identify code changes that have a high likelyhood of having complicated micro processes.

This work is showing promise as it agrees with previous work that there is a correlation between code-based characteristics and MPA.

Further investigation will be performed to find out the exact state changes. Moreover, formulating and testing additional program slicing metrics. Once usable metrics are established, other software projects can be used as case studies.

5. CONCLUSION

The main objective of the research is to provide a proof of concept that there is indeed a relationship between a code change attributes and how its related processes are executed. The results suggest there is a correlation between how much of the complexity of the micro process and its program slicing metrics. Further work is proposed to refine the metrics as well as test on other similar software projects.

It is envisioned that the research will contribute to a better understanding and classification of code changes based on the nature of code, therefore using the nature of the code to suggest improvement of software processes at the micro level.

ACKNOWLEDGMENTS

Special thanks to Shinji Kawaguchi for his contribution to this research in MPA. This work 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. This research was also supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (No.22500027), and Grant-in-Aid for Research Activity start-up (No.22800040 and 22800043).

REFERENCES

[1]. J. Herbsleb, A. Carleton, J. Rozum, J. Siegel and D. Zubrow, "Benefits of CMM-Based Software Process Improvement: Initial Results", (CMU/SEI-94-TR-13). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1994.

[2]. K. K. Margaret and J. A. Kent, "Interpreting the CMMI: A Process Improvement Approach", Auerbach Publications, (CSUE Body of Knowledge area: Software Quality Management), 2003.

[3]. C. H. Schmauch, "ISO 9000 for Software Developers", 2nd. ASQ Quality Press, 1995.

[4]. S. Beecham, T. Hall and A. Rainer, "Software process problems in twelve software companies: an empirical analysis", Empirical Software Engineering, v.8, pp. 7-42, 2003.

[5]. N. Baddoo and T. Hall, "De-Motivators of software process improvement: an analysis of practitioner's views", Journal of Systems and Software, v.66, n.1, pp. 23-33, 2003.

[6]. M. Niazi, M. A. Babar, "De-motivators for software process improvement: an analysis of Vietnamese practitioners' views", in Product Focused Software Process Improvement PROFES 2007, LNCS, v.4589, pp. 118-131, 2007.

[7]. J.G. Brodman and D.L. Johnson, "What small businesses and small organizations say about the CMMI", In Proceedings of the 16th International Conference on Software Engineering (ICSE), IEEE Computer Society, pp. 331 – 340, 1994

[8]. A. Rainer and T. Hall, "Key success factors for implementing software process improvement: a maturity-based analysis", Journal of Systems and Software v.62, n.2, pp.71-84, 2002.

[9]. C. Yoo, J. Yoon, B. Lee, C. Lee, J. Lee, S. Hyun and C. Wu, "A unified model for the implementation of both ISO 9001:2000 and CMMI by ISO-certified organizations", The Journal of Systems and Software 79, n.7, pp. 954-961, 2006.

[10]. O. Armbrust, M. Katahira, Y. Miyamoto, J. Münch, H. Nakao, and A. O. Campo, "Scoping software process lines", Softw. Process , v.14, n.3, pp.181-197, May, 2009.

[11]. S. Morisaki and H. Iida, "Fine-Grained Software Process Analysis to Ongoing Distributed Software Development", 1st Workshop on Measurement-based Cockpits for Distributed Software and Systems Engineering Projects (SOFTPIT 2007), pp.26-30, Munich, Germany, Aug., 2007.

[12] M. Weiser, "Program slicing", In Proceedings of the 5th International Conference on Software Engineering. (ICSE). IEEE Press, Piscataway, NJ, pp.439-449, Mar. 09 - 12, 1981

[13]. P. Anderson, T. Reps, T. Teitelbaum, M. Zarins, "Tool Support for Fine-Grained Software Inspection," IEEE Software, pp. 42-50, July/August, 2003

[14]. K. Pan, S. Kim, E. J. Whitehead, Jr., "Bug Classification Using Program Slicing Metrics", Source Code Analysis and Manipulation, IEEE International Workshop, pp.31-42, 2006

[15]. R. G. Kula, K. Fushida, S. Kawaguchi, and H. Iida, "Analysis of Bug Fixing Processes Using Program Slicing Metrics," in Product-Focused Software Process Improvement PROFES 2010, vol. LNCS 6156, pp. 032-046, June 2010.

[16]. A. Watson and T. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric", National Institute of Standards and Technology, Gaithersburg, MD, (NIST) Special Publication, pp.500-235, 1996.

[17]. R. G. Kula, ``Using Program Slicing Metrics for the Analysis of Bug Fixing Processes," Master thesis, Institute of Science and Technology, Nara, Japan, 2010.

[18]. P. Anderson , M. Zarins, "The CodeSurfer Software Understanding Platform", Proceedings of the 13th International Workshop on Program Comprehension, pp.147-148, May 15-16, 2005

Flexibly Highlighting in Replaying Operation History

Takayuki Omori
Dept. of Computer Science
Ritsumeikan University
Kusatsu, Shiga, Japan
takayuki@fse.cs.ritsumei.ac.jp

Katsuhisa Maruyama
Dept. of Computer Science
Ritsumeikan University
Kusatsu, Shiga, Japan
maru@cs.ritsumei.ac.jp

ABSTRACT
In empirical software engineering, historical information of software development is one of significant facets since knowledge derived from such information can improve quality of software product and development process. In these years, several tools recording actual operations on integrated development environments have been proposed. They are promising for improving correctness of examining operation history. However, it is necessary to narrow a range of targets to be examined because of variety and massiveness of recorded operations. We proposed OperationReplayer which supports examining operation history by enabling to replay past operations as if they are been performing right now. This paper proposes a highlight plug-in, which extends OperationReplayer. Users can rapidly identify target operations of their examinations by using their own highlight plug-ins. This paper also describes a case study to show usefulness of the proposed flexible highlighting.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments.*

General Terms
Experimentation, Management, Human Factors.

Keywords
Software evolution, Integrated development environment, Fine-grained changes, Software process analysis.

1. INTRODUCTION
In empirical software engineering field, analyzing history of software development is an essential approach to improve software quality and productivity. Several methods for recording actual operations on integrated development environments (IDEs) have been proposed[1-3]. Since the methods enable replaying recorded operations, we can examine them and attain precise historical information. However, such examination is fault-prone and time-consuming because of variety and massiveness of recorded operations. To address this problem, this paper proposes flexible visual abstraction in addition to simply replaying past operations. This work promotes our previous work of OperationReplayer[1] and introduces plug-in mechanism to it for constructing a flexible time-series visual summary called a highlight. A user (who replays past edits) can easily identify noteworthy operations by seeing highlights on time-line bars, if the prepared highlights are appropriate for his/her purpose. The original OperationReplayer provides static drawing of highlights. On the other hand, its extended version allows the user to determine how highlights are drawn by switching existing highlight plug-ins or creating new ones. Accordingly, the extended version of OperationReplayer facilitates his/her examination of operation history.

2. OperationReplayer
This section describes an examination task of operation history by using OperationReplayer. The highlight plug-in mechanism is also described.

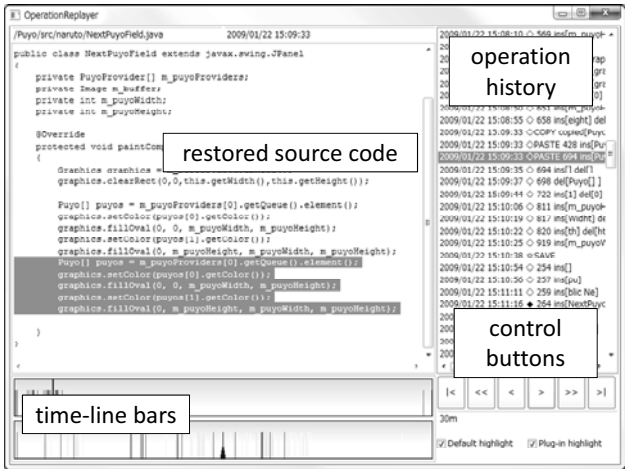


Figure 1. Screenshot of extended OperationReplayer

2.1 Examination of Operation History
Figure 1 shows a screenshot of the extended version of OperationReplayer. Appearance of the window is similar between the original and extended one.

A user can replay past operations in a single source file on a single replayer window. He/she can change the focal time by pushing control buttons, selecting an operation from the operation history, or specifying the time on the time-line bars. The content of source code is restored in conjunction with the focal time. For the two time-line bars, the upper one denotes a project life-time, and the lower one shows detailed information within a specified time-period of the project life-time. In the extended OperationReplayer, these time-line bars are drawn based on a highlight plug-in the user constructs.

We assume that an operation-history examination is iteratively performed with different views. This is because operation history is versatile. Determining a suitable highlight is difficult and needs a number of trials-and-errors. Therefore, the user repeats installing (switching and creating) his/her highlight plug-in and replaying operations to examine them. After doing iterative processes, he/she attains visual abstraction (highlights) more

suitable for his/her purpose. Thus, flexible highlighting is essential for the examination of the operation history.

2.2 Highlight Plug-in

A highlight plug-in is implemented as an extension for `highlightPoint` extension point of OperationReplayer. Using APIs provided by OperationReplayer, the plug-in can access various kinds of information on recorded operations, such as the time when the operation was performed, the name of the developer who performed the operation, the deleted/inserted string, and the offset where the operation was performed. Moreover, OperationReplayer provides APIs for accessing any snapshot of source code in the past and its syntactical information. To display highlights on the time-line bars, the plug-in instantiates `SpotHighlight` and/or `RangeHighlight`. These instances are responsible for drawing lines and rectangles corresponding to times and time-periods specified on the time-line bars.

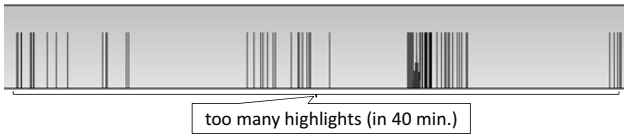
3. Case Study

To show usefulness of flexible highlighting proposed here, we have created a highlight plug-in and used it to examine actual operation history. In this case study, the purpose of this examination is to distinguish periods for intensive refactoring from ones for other modifications. We checked if the created highlight plug-in has been improved in three trials.

1st trial: In the operation history, all refactoring operations invoked by Eclipse's menu items are held as instances of `TriggeredOperations`. The initial plug-in simply creates `SpotHighlight` instances corresponding to the operations, which are colored with blue (blue highlights). In addition, all operations that delete any character (the length of the deleted text is not zero) are colored with cyan (cyan highlights). The highlight plug-in has 49 lines of source code. We applied this highlight plug-in to all operations recorded in real software development. As a result, the plug-in created a large number of highlights including noises shown on the window. The noises obscured true refactorings. Figure 2(a) shows a part of the time-line bar resulting from this trial.

2nd trial: To reduce the noise, we removed the cyan highlights (a single line of source code was deleted). Instead, to detect move operations of fields and methods which are frequently done in refactorings, cut and paste operations are colored with red and orange, respectively (six lines of source code were inserted). We applied this new version of the plug-in to the same recorded operations. The result was better than one in the 1st trial; however it still includes many noises. By replaying operations around several blue highlights, we found some of the highlights involve "`organize/add imports`" operations that do not seem to be refactorings.

3rd trial: To further reduce noisy highlights, we improved the algorithm related to blue highlights. Finally, the plug-in detects operations of which labels are same as Eclipse's "`Refactor`" submenu items and colors them with blue (ten lines of source code were inserted). Applying this plug-in, we detected a possible refactoring period with intensive blue (deeply colored) lines shown in Figure 2(b). Each of these lines indicates a "`rename`" refactoring operation.



(a) Highlights including many noises (1st trial)

(b) Detecting intensive refactoring operations (3rd trial)

Figure 2. Case study

To assess that the highlight plug-in is truly improved through this experiment, we manually examined operations around intensive highlights reported by the highlight plug-ins in 2nd and 3rd trials. Actually, we intuitively extracted several time-periods based on our understanding for the reported highlights, and then checked if each period involves true refactorings. We did not examine highlights reported in the 1st trial since it created too many worthless highlights.

As a result, in the 2nd and 3rd trials, the highlight plug-in totally detected 66 and 47 refactoring periods, respectively. In both the trials, 16 true refactoring periods were detected among them. Besides, 50 and 31 faulty periods were respectively reported. The precision rates are 24% (16/66) and 34% (16/47), respectively. From these experimental results, the precision was improved with little effort (inserting and deleting source code) although the experiment is quite limited. Here, we were supposed to count the total number of periods for real refactorings within the development and show recall rates in this experiment. However, we could not precisely count it due to vast numbers of recorded operations.

4. Conclusion

This paper proposed an operation replayer with a mechanism supporting flexible highlighting. It allows users to flexibly change visual abstraction of operation history on a window of OperationReplayer. We are planning to extend it to effectively treat histories of multiple source files and developers. Enhancing flexibility and scalability of visualization is also future work.

5. REFERENCES

[1] Omori, T. and Maruyama, K. 2009. Identifying Stagnation Periods in Software Evolution by Replaying Editing Operations. In *Proceedings of 16th Asia-Pacific Software Engineering Conference*, 389-396.

[2] Robbes, R. and Lanza, M. 2007. A Change-based Approach to Software Evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93-109.

[3] Hattori, L., Lungu, M., and Lanza, M. 2010. Replaying Past Changes in Multi-developer Projects. In *Proceedings of IWPSE-EVOL 2010*, 13-22.