Micro Interaction Metrics for Defect Prediction

Taek Lee[†,*], Jaechang Nam[§], DongGyun Han[§], Sunghun Kim[§], Hoh Peter In[†]

[†]Department of Computer Science and Engineering
Korea University, Seoul, Korea
{comtaek, hoh_in}@korea.ac.kr

[§]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong
{jcnam, handk, hunkim}@cse.ust.hk

ABSTRACT

There is a common belief that developers' behavioral interaction patterns may affect software quality. However, widely used defect prediction metrics such as source code metrics, change churns, and the number of previous defects do not capture developers' direct interactions. We propose 56 novel micro interaction metrics (MIMs) that leverage developers' interaction information stored in the Mylyn data. Mylyn is an Eclipse plug-in, which captures developers' interactions such as file editing and selection events with time spent. To evaluate the performance of MIMs in defect prediction, we build defect prediction (classification and regression) models using MIMs, traditional metrics, and their combinations. Our experimental results show that MIMs significantly improve defect classification and regression accuracy.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software maintenance*

General Terms

Algorithms, Measurement, Experimentation

1. INTRODUCTION

Defect prediction has been a very active research area in software engineering [10, 14, 15, 21, 26, 28, 36, 37]. Many effective new defect prediction models including new metrics have been proposed. Among them, source code metrics (CMs) and change history metrics (HMs) are widely used and yield reasonable defect prediction accuracy. For example, Basili et al. [3] used Chidamber and Kemerer metrics, and Ohlsson et al. [27] used McCabe's cyclomatic complexity for defect prediction. Moser et al. [23] used the number of revisions, authors, and past fixes, and age of a file as defect predictors.

In addition to software CMs and HMs, developers' behavioral patterns are also believed to be an important factor affecting software quality. Ko et al. [16] identified possible causes for programming errors using a breakdown model of cognitive chains. DeLine et al. [19] surveyed developers' work habits and found that work interruptions or task switching may affect programmer productivity. Their studies imply a correlation between developers' behaviors and software quality. Specifically, unexpected or abnormal behaviors may introduce bugs and cause software defects. Therefore, it is desirable to use developers' interaction information when building defect predictors. However, current defect metrics such as CMs and HMs do not directly capture developers' interactions, since current version control systems or bug report systems do not record developers' interactions.

We use Mylyn, a context storing and recovering Eclipse plug-in, to capture developers' interaction events. Examples of such events include selecting and editing software source files. Additionally, the order and time span of each event are also provided by Mylyn. Since developers' interaction patterns may affect software quality and developer productivity, metrics based on developers' interactions could be important indicators to predict defects.

In this paper, we propose 56 micro interaction metrics (MIMs) based on developers' interaction information stored in Mylyn. For example, *NumMultiTasks* measures task complexity by observing the number of ongoing tasks. *RepeatedlySelectedFileNum* and *RepeatedlyEditedFileNum* measure repeated work activities by counting selecting and editing events for the same file. (The full list of MIMs and their descriptions are shown in Appendix.)

To evaluate the performance of MIMs in defect prediction, we build defect prediction (classification and regression) models using MIMs, CMs, HMs, and their combinations. We evaluated various defect prediction models on Eclipse subprojects, which include the Mylyn data. Our evaluation results showed that MIMs significantly improve

[*]Taek Lee was a visiting PHD student at the Hong Kong University of Science and Technology when this work was carried out.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

prediction accuracy. The defect classification F-measure is 0.49 with MIMs, while it is only 0.26 with the combination of CM and HM. The mean square error of the defect regression model with MIMs is 0.76, while it is 0.81 with the combination of CM and HM.

These results concur with previous findings [16, 19]. Since developers' interactions affect software quality and programmer productivity, MIMs that capture a certain degree of developers' interactions play an important role in defect prediction. Our paper makes the following contributions:

- **56 MIMs**, which capture developers' interaction information from the Mylyn data.
- **Empirical evaluation** of the role of MIMs in defect prediction.

In the remainder of the paper, we start by presenting MIMs in Section 2. Section 3 describes our experimental setup including various defect prediction models, metrics used, and evaluation measures. Section 4 presents results of various defect models with/without MIMs. Section 5 discusses threats to validity of our study. We discuss related work in Section 6 and conclude the paper with future directions of our research in Section 7.

2. MICRO INTERACTION METRICS

This section proposes MIMs based on data collected by Mylyn, an Eclipse plug-in. We briefly introduce Mylyn and its data in Section 2.1, and describe the proposed MIMs in Section 2.2.

2.1 Mylyn

Mylyn is an Eclipse plug-in that records the context of developers' task such as editing or selecting files [8]. The recorded context is restored when developers want to resume the task. In this way, even after a task switching, developers can focus on the files they have previously worked on. The stored context can be shared among developers and help other developers understand what files were browsed and edited for the task.

The Mylyn data are stored as an attachment to the corresponding bug reports in the XML format. The data include events and their attributes performed by developers. Currently, Mylyn records six types of events: selection, edit, command, propagation, prediction, and manipulation as shown in Table 1. When a developer selects a file, a selection event occurs. Edit events are recorded when developers edit a file. Propagation events occur when a developer uses automatic refactoring features in Eclipse. For example, files can be modified automatically via Eclipse refactoring feature, and this is recorded as a propagation event.

Table 1: Event Type in the Mylyn Data [25]

Type	Description
Selection	Select a file in the explorer
Edit	Edit a file in the editor
Command	Invoke command by developer
Propagation	Propagated interaction
Prediction	Predict future interaction
Manipulation	Manipulate DOI value.

Each event is recorded with attributes including start-date, end-date, structure-handle, and degree-of-interest (DOI). The start-date and end-date attributes represent the starting and ending time of the event. The structure-handle attribute denotes corresponding files of the event. For example, for edit events, the structure-handle attribute indicates which file is edited. The DOI value indicates developer's interest in the corresponding file. The DOI value of a file increases when developers select or edit the file. DOI values help developers identify more/less important files for the task. Currently, DOI values are automatically computed based on the frequency of developers' interactions [11, 12].

When developers double click a file and open it in an editor, an edit event with no time spent, zero-time edit occurs. The time spent can be easily computed from end-date and start-date attributes. The zero-time edit event means the start-date and end-date of the event are the same. When developers change the content of the file, such events are recorded as non-zero-time events. We distinguish between these two types of events because it is important to separate real edits from simple double-clicks.

More information about the Mylyn data is available at the Mylyn Project home page[25].

2.2 Design of MIMs

The design principle of MIMs is quantifying the complexity and intensity of developers' interaction activities such as browsing or editing of files. Based on this principle, we designed two levels of MIMs, file-level and task-level.

The file-level MIMs capture specific interactions for a file in a task. For example, *NumEditEvent* represents the specific file edit events in one task. The task-level MIMs represent properties per task. For example, *TimeSpent* shows the time spent on a given task.

We designed three categories of file-level MIMs:

- **Effort**: Since Mylyn stores developers' interactions, this category measures developers' effort for a given file such as the number of events on a file.
- **Interest**: By using DOI value, we infer developers' interest in a specific file.
- **Intervals**: This category measures time intervals between events.

Task-level MIMs include six categories:

- **Effort**: Similar to the effort category in the file-level, we measure the effort made for a given task.
- **Distraction**: This measures developers' distraction for corresponding tasks such as low DOI events and non-java file edit events.
- **Work Portion**: All events in one task are divided into three event periods: Before beginning the first edit, edit, and after finishing the last edit sections. This category measures how much time was spent for each section.
- **Repetition**: There are some repeated events such as repeated selection or editing of the same file. This category counts this kind of repeated events.
- **Task Load**: This category measures the task load by observing the number of simultaneously on-going tasks (NumMultiTasks).
- **Event Pattern**: Since Mylyn contains interactions, we identify common patterns of sequential events. This cat-

egory captures the number of identified sequential patterns.

The complete list of MIMs with descriptions is shown in Appendix.

3. EXPERIMENTAL SETUP

This section describes our experimental setup including data collection, prediction models, and evaluation measures.

3.1 Bug Prediction Process

The commonly used file-level bug prediction process is used for our experiments as shown in Figure 1 [23, 28, 36].

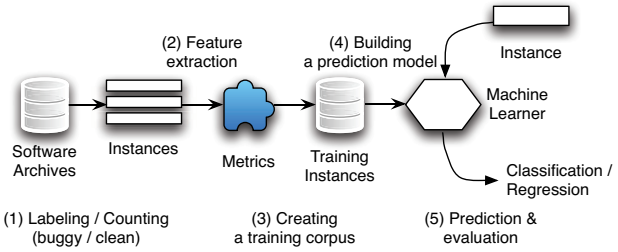

Figure 1: Overall steps of bug prediction process

First, we collect files as instances (in the machine learning sense) and count post-defects for each file. For the regression model, we predict the defect numbers. For classification, we label a file as *buggy* if it has any post-defect (post-defect number >= 1), or *clean* otherwise. The detailed post-defect counting process is described in Section 3.2.

Then, we extract MIMs, CM, and HMs for each instance as explained in Sections 3.3 and 3.4. Finally, we train prediction models using machine learning algorithms implemented in Weka [9]. The trained prediction models classify instances as *buggy* or *clean* (classification), or predict the post-defect numbers (regression). Detailed cross validation models and evaluation measures are explained in Section 3.5.

3.2 Data Collection

We extracted a total of 7,785 Mylyn tasks from Eclipse Bugzilla attachments between Dec 2005 [1] and Sep 2010.

To explicitly separate *metrics extraction* and *post-defect counting* periods, we set arbitrary time split points, P as shown in Figure 2. It is important not to collect any metrics from the post-defect counting period. Time P represents the present, and our model predicts future defects (after P) using metrics from the past to P. Thus, we computed all metrics (MIMs, CMs, and HMs) of instances before P, and counted post-defects after P. In our experiments, we used various time split points: 5:5, 7:3, and 8:2 to compare prediction results with different time split points. For example, Table 2 shows the number of instances and defect ratios for Eclipse subprojects from the 8:2 time split point. The goal of various time splits is evaluating MIM models by following the random split convention widely used in the literature [37, 38].

To collect metrics and count defects, instances must exist in both time periods (i.e. after and before Time P). If a file does not exist in the metrics collection period, there is no metrics to use for prediction. On the other hand, if a file does not exist in the post-defect counting period, the defect number for the file is always zero, which is misleading. To use only files which existed in both periods, we checked their existence using Eclipse CVS. For this reason, when we use different time split points, 5:5, 7:3, and 8:2, the number of instances changes.

[1]Mylyn was released and widely used from Dec 2005.

Table 2: Collected file instances and post defects for the 8:2 split.

Subjects	# of instances (files)	% of defects
Mylyn	1061	14.3%
Team	239	35.5%
Etc.	1041	5.4%
All	2341	12.5%

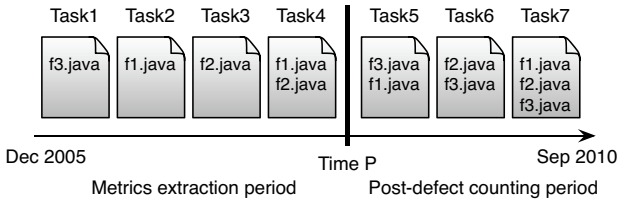

Figure 2: Time Split for metrics extraction and post-defect counting periods.

To count post-defects, we used edited file information in Mylyn tasks. Since each task is directly attached to a bug report, we checked if the corresponding bug report was a fixed bug. If it was not fixed or not a bug (such as feature-enhancement or trivial), we assumed the edited files for the particular bug report are not fixes. We marked the edited files only for fixed bug reports as fixed and increased the defect numbers for the files. For example, suppose Tasks 5 and 6 in Figure 2 were attached to fixed bug reports, and Task 7 is attached to a feature-enhancement bug report. In this case, the post defect number of 'f3.java' is two, since Task 7 is for feature enhancement. In this way, we could avoid false-positives in marking defects.

The following sections describe metrics extraction techniques and evaluation measures for prediction models in detail.

3.3 Extraction of MIMs

All MIMs listed in Appendix were extracted from the Mylyn data. Since our models are file-level defect predictors, we need to compute MIMs for each file.

Computation of file-level MIMs is straightforward. We first compute file-level MIMs for a given file. If the file is edited multiple times in the *metrics extraction period*, then it will have multiple metric values. We just average and total the multiple values and use the totaled values as features (in the machine learning sense) of the file.

For task-level MIMs such as *TimeSpent* of a task, first we propagate the metric values to all edited files in the task. Then, the propagated metric values are regarded as file level metrics. If a file is edited multiple times, and it has multiple metric values (propagated from tasks), we average the values and use them as features.

3.4 Extraction of CMs and HMs

We collected CMs at Time P as shown in Figure 2 since CMs can be extracted from a snapshot. We used the Understand tool [33] to extract CMs. The Understand tool extracts 24 file-level and 18 class-level metrics such as Chidamber and Kemerer [5] and Object-Oriented metrics. If a file has more than one class, we derived file-level metrics from multiple class-level metrics. The Understand tool mostly provides two kinds of metrics: Avg* and Count*. To generate file-level metrics from multiple classes in a file, we averaged Avg* class-level metrics. However, when we get file-level metrics from Count* classs-level metrics, we added the values together. We used all 42 CMs for our experiments. Selected CMs are listed in Table 3.

Table 3: List of selected source code metrics (CMs)

Metrics	Description
CountLineCode	Lines of code
CountSemicolon	# of semicolons
CountStmtDecl	# of declarative statements
SumEssential	Sum of essential complexity of methods
CntClassCoupled	Coupling between object classes (CBO)
CntClassDerived	# of Child classes (NOC)
CntDeclMethod	# of local methods (NOM)
CntDeclMethodPublic	# of local public methods (NOPM)
MxInheritanceTree	Depth of Inheritance Tree (DIT)
PcntLackOfCohesion	Lack of cohesion (LCOM)

In addition, we collected 15 HMs following Moser et al.'s approach [23]. All HMs were collected from the change history stored in Eclipse CVS repository[2] during the metrics extraction period as shown in Figure 2.

Table 4 lists 15 HMs used in our experiments. The *Refactorings* metrics indicates if a file change is refactoring [23]. This is determined by mining CVS commit logs: if they contain the keyword 'refactor', we assume it is a refactoring. We counted the number of all refactored revisions of a file in the metrics extraction period. The *Age* metric indicates the period of file existence [23]. The *BugFixes* metric represents the defect numbers in the metrics extraction period. To compute this, we mined commit logs to search explicit bug IDs in the logs. Then, we checked the bug reports, and if they were fixed bugs (not feature enhancement), we marked the change as bug-fix. In addition, we searched for specific keywords, *bug* or *fix*[3], which indicate bug fix changes [23]. If change logs had such keywords, we marked the changes as bug-fixes [23].

3.5 Performance Measures

In this section, we explain evaluation methods used in our prediction models. We compare the prediction performance of MIMs with other metrics. Thus, we build a model by each metrics (i.e. MIM, CM, HM) and combination of them (i.e. CM+HM). To evaluate our prediction models, we used 10-fold cross validation, which is widely used to evaluate prediction models [14, 23, 20]. We repeated 10-fold cross validation 100 times for each prediction model on each different time split to validate the prediction performance of MIMs by t-test.

[2]http://archive.eclipse.org/arch/

[3]The keywords, *postfix* and *prefix* are excluded [23].

Table 4: List of history metrics (HMs)

Metrics	Description
Revisions	# of revisions of a file
Refactorings	# of times a file has been refactored
BugFixes	# of times a file was involved in fixing bugs
Authors	# of distinct authors committing a file
LOC_Added	Sum of the lines of code added to a file
Max_LOC_Added	Maximum number of lines of code added
Ave_LOC_Added	Average lines of code added
LOC_Deleted	Sum of the lines of code deleted in a file
Max_LOC_Deleted	Maximum number of lines of code deleted
Avg_LOC_Deleted	Average lines of code deleted
CodeChurn	Sum of (added LOC – deleted LOC)
Max_CodeChurn	Maximum CodeChurn for all revisions
Ave_CodeChurn	Average CodeChurn per revision
Age	Age of a file in weeks
Weighted_Age	Age considering LOC_Added

3.5.1 Classification

To evaluate performance differences between different prediction models, we used F-measure. Usually, F-measure represents harmonic mean of precision and recall. We first computed precision and recall values of buggy instances, and then we obtained F-measures. The following outcomes were used to define precision, recall, and F-measure: (1) predicting a buggy instance as buggy (b→b); (2) predicting a buggy instance as clean (b→c); (3) predicting a clean instance as buggy (c→b). We use the above outcomes to evaluate the prediction accuracy of our prediction models with the following measures [1, 30]:

- **Precision:** the number of instances correctly classified as buggy ($N_{b\to b}$) over the number of all instances classified as buggy.

$$Precision\ P(b) = \frac{N_{b\to b}}{N_{b\to b} + N_{c\to b}} \quad (1)$$

- **Recall:** the number of instances correctly classified as buggy ($N_{b\to b}$) over the total number of buggy instances.

$$Recall\ R(b) = \frac{N_{b\to b}}{N_{b\to b} + N_{b\to c}} \quad (2)$$

- **F-measure:** a composite measure of precision *P(b)* and recall *R(b)* for buggy instances.

$$F\text{-}measure\ F(b) = \frac{2 * P(b) * R(b)}{P(b) + R(b)} \quad (3)$$

3.5.2 Regression

By using linear regression models, we predicted the number of post-defects. To compare the prediction performance of models of MIMs and other metrics, we calculated correlation coefficient, mean absolute error, and root mean squared error [34]. Correlation coefficient measures the correlation between predicted and real defect numbers. If the correlation coefficient is closer to 1, the metrics are more correlated to post-defects. Both mean absolute error and root mean squared error represent the difference between predicted and actual post-defects numbers. If both error values of a prediction model are less than others, it means the model has higher prediction accuracy. To compare the pre-

diction model of MIMs with those of CMs, HMs and their combination, we measured these three values. We also repeated 10-fold cross validation 100 times to validate regression models for each metrics by t-test.

3.5.3 T-test

To check statistical significance of prediction performance of MIMs and CMs, HMs and CM+HM, the simple t-test was used [6]. We checked if mean of F-measure values of MIMs was not equal to the mean of F-measures of CM and HM. Specifically, the null and alternative hypotheses for t-test are:

- **H0** F-measure mean of CM+HM is *equal to* the F-measure mean of MIM.
- **H1** F-measure mean of CM+HM is *not equal to* the F-measure mean of MIM. (i.e. MIMs have better performance if the mean value is higher)

We rejected the null hypothesis *H0* and accepted the alternative hypothesis *H1* if the p-value was smaller than 0.05 (at the 95% confidence level) [6].

3.6 Dummy Classifier

To evaluate the performance of classifiers using MIMs, CM, and HMs, we introduce a baseline: *Dummy classifier* – guessing a change/file as buggy or clean in a purely random manner. Since there are only two labels, buggy and clean changes, the dummy predictor could also achieve certain prediction accuracy. For example, if there are 12.5% of changes in a project are buggy, by predicting all changes as buggy, the buggy recall would be 1, and the precision would be 0.125. It is also possible that the dummy predictor randomly predicts a change as buggy or clean with 0.5 probability. In this case, the buggy recall would be 0.5, but the precision will still be 0.125.

We used the F-measure of the dummy predictor as a baseline when showing the classification results. We computed the dummy F-measure assuming the dummy predictor randomly predicts 50% as buggy and 50% as clean. For example, for a project with 12.5% buggy changes as shown in Table 2, the dummy buggy F-measure is 0.2 ($2 \times \frac{0.5 \times 0.125}{0.5+0.125}$).

4. RESULTS

This section presents performance of bug prediction models using MIM, CM, HM, and their combinations.

4.1 Result Summary

This section provides a quick summary of results from different experiments. Details of settings and results are explained in subsections.

- MIM outperforms existing metrics (CM and HM) and their combination (CM+HM) for different subjects (Section 4.2.1).
- MIM outperforms CM, HM and CM+HM for different classification algorithms (Section 4.2.2).
- MIM outperforms CM+HM for different split points (Section 4.2.3).
- Among the top 56 important metrics for classification models, 44 (79%) are from MIMs. All top 42 important metrics are from MIMs (Section 4.2.4).
- MIM outperforms CM+HM for regression models (Section 4.3).

4.2 Predicting Buggy Files

4.2.1 Different Subjects

To evaluate the performance of MIMs for various subjects, we built defect prediction models using J48 Decision Tree for three Eclipse subprojects, *Mylyn*, *Team* and *Etc.* from the 8:2 split point as shown in Table 2. To build the models, MIM, CM, HM, CM+HM, and MIM+CM+HM were used.

We use F-measure to evaluate prediction performance as described in Section 3.5.3. The ten-fold cross validation is used to train and test models. Since ten-fold cross validation randomly samples instances and puts them in ten folds [1], we run the ten-fold cross validation 100 times to avoid sampling bias.

Figure 3 shows F-measure values for various Eclipse subjects and various metrics. F-measure values vary, but the trend is clear: MIM outperforms CM, HM, and their combination (CM+HM) for *All*, *Team*, and *Etc.* For *Mylyn*, the F-measure of MIM is slightly lower than that of CM+HM, but MIM+CM+HM outperforms CM+HM. In addition, the Dummy F-measure values are shown as a baseline, the solid line in Figure 3. Generally, CM and HM outperform the baseline, and MIM significantly outperforms the baseline.

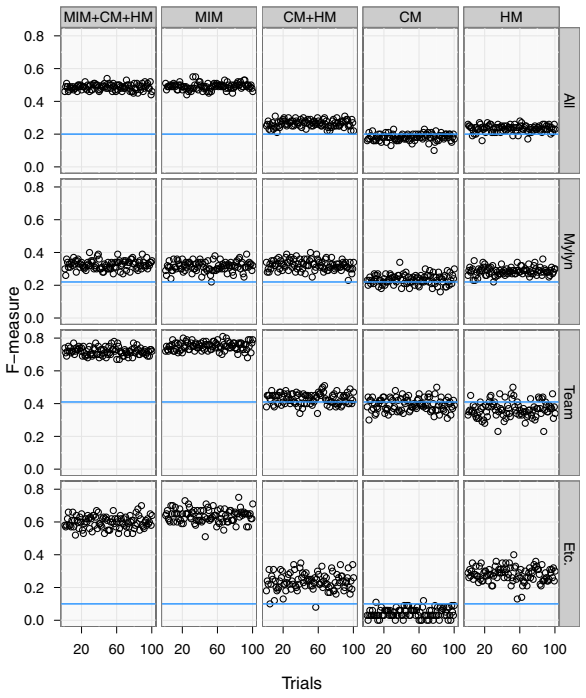

Figure 3: Performance comparison by different subjects (Classier: J48 decision tree). The Dummy F-measure is shown as a solid line.

Table 5 shows the mean of F-measure values from 100 ten-fold cross validations and their statistical significance. If F-measure values of MIM+CM+HM or MIM are significant (p-value < 0.05) in comparison to CM+HM, the corresponding values are in bold. For example, the F-measure (0.75)

of MIMs in *Team* is better than that of CM+HM (0.43), and it is statistically significant. For *Mylyn*, the F-measure (0.31) of MIM is not better than that of CM+HM (0.32). However, MIM+CM+HM outperforms CM+HM, which indicates that MIM complements CM+HM to yield better prediction accuracy. Generally, the results in Table 5 indicate that MIM and/or MIM+CM+HM outperform traditional metrics (CM+HM). In addition, Dummy F-measures are shown in Table 5.

Table 5: F-measure mean values of each metrics in different subjects. (The F-measures in bold indicate the value difference in comparison to CM+HM is statistically significant. (p-value < 0.05))

Subjects	MIM+CM+HM	MIM	CM+HM	CM	HM	DUMMY
All	**0.49**	**0.49**	0.26	0.18	0.23	0.20
Mylyn	0.33	0.31	0.32	0.24	0.29	0.22
Team	**0.72**	**0.75**	0.43	0.39	0.36	0.41
Etc.	**0.60**	**0.64**	0.24	0.04	0.28	0.10

4.2.2 *Different Machine Learner*

This section compares the results of prediction models using three widely used classification algorithms, Bayesian Network, J48 decision tree, and logistics in Weka [9]. All instances with the 8:2 time split (in Table 2) are used for this experiment.

Figure 4 shows F-measures from 100 ten-fold cross validations. The F-measures vary, but they show a trend that MIM outperforms CM, HM, and their combination (CM+HM) in all algorithms.

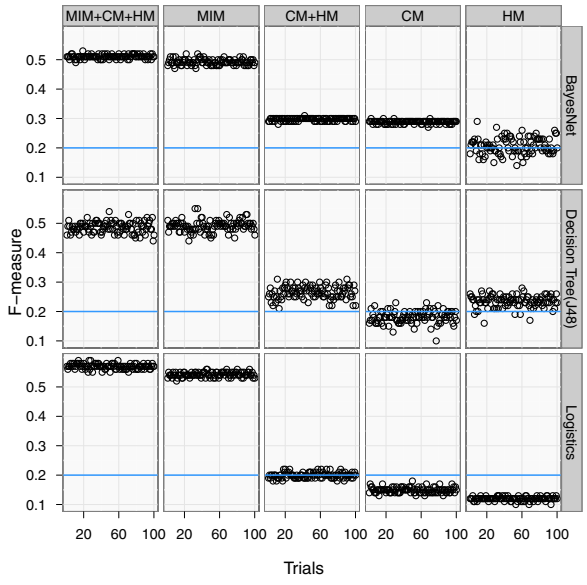

Figure 4: Performance comparison by different algorithms (All instances). The Dummy F-measure is shown as a solid line.

Table 6 shows mean F-measure values from 100 ten-fold cross validations. The F-measure mean (0.49) of MIMs in the BayesNet is better than that of CM+HM (0.30). If F-measure mean values of MIM+CM+HM or MIM are statistically significant in comparison to CM+HM, we mark the values in bold. For all algorithms, MIM outperforms CM+HM with statistical significance.

Table 6: F-measure mean values of each metrics in different algorithms. (The F-measures in bold indicate the value difference in comparison to CM+HM is statistically significant. (p-value < 0.05))

Measures	MIM+CM+HM	MIM	CM+HM	CM	HM	DUMMY
BayesNet	**0.51**	**0.49**	0.30	0.29	0.21	0.20
Decision Tree	**0.49**	**0.49**	0.26	0.18	0.23	0.20
Logistics	**0.57**	**0.54**	0.19	0.15	0.12	0.20

4.2.3 *Different Split Points*

To evaluate prediction performance of MIMs on different time splits, we built prediction models using three different time split points, 5:5, 7:3, and 8:2 as explained in Section 3.2. Table 7 shows corpus information. Since we collected instances before and after Time point P, the number of instances varies for different time split points (Section 3.2). The defect ratio varies, since we have different periods for post-defect counting as shown in Figure 2.

Table 7: Sample Conditions per Split Period

Time split point, P	# of instances	% of defects
Apr. 2008 (50:50)	1155	21.2%
Mar. 2009 (70:30)	2022	14.6%
Sep. 2009 (80:20)	2341	12.5%

Figure 5 shows F-measure values from 100 ten-fold cross validations. It indicates that MIM outperforms CM+HM for three different time split points.

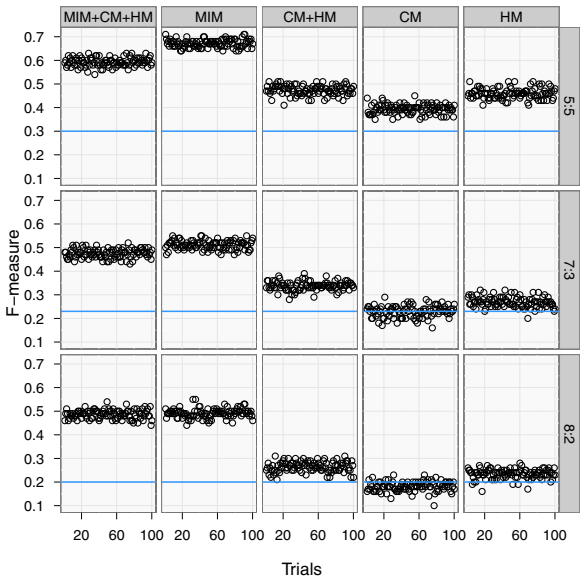

Figure 5: Performance comparison by time split (All instances, Classifier: J48 decision tree). The Dummy F-measure is shown as a solid line.

Table 8 shows F-measure mean values of MIMs, and values are in bold if they are statistically significant in comparison to CM+HM. For example, the F-measure mean (0.67) of MIM in the 5:5 time split point is better than that (0.47) of CM+HM, and it is statistically significant. As we observe in Table 8, MIM outperforms CM+HM for all time split points.

Table 8: F-measure mean values of each metrics at different split points. (The F-measures in bold indicate the value difference in comparison to CM+HM is statistically significant. (p-value < 0.05))

Measures	MIM+CM+HM	MIM	CM+HM	CM	HM	DUMMY
5:5	**0.59**	**0.67**	0.47	0.39	0.46	0.30
7:3	**0.48**	**0.51**	0.34	0.23	0.27	0.23
8:2	**0.49**	**0.49**	0.26	0.18	0.23	0.20

4.2.4 Metrics Effectiveness Analysis

To evaluate the effectiveness of each metric for classification, we measured the information gain ratio [18, 17] of MIMs, CMs, and HMs, and ranked them accordingly. All collected instances and post defects from the 8:2 split point as shown in Table 2 are used for this analysis. The information gain ratio indicates how well a metric distinguishes labels (i.e., buggy or clean) of instances. Even though the metrics effectiveness may differ based on machine learning algorithms, generally metrics with a high information gain ratio is regarded as important [13, 31].

Top 56 ranked metrics (among 113 metrics) based on the gain ratio are shown in Figure 6. Among the top 56 important metrics for classification models, 44 (79%) metrics are from MIMs. Especially, the top 42 metrics are MIMs. The best metric is *NumLowDOIEdit* followed by *NumPatternEXSX* and *TimeSpentOnEdit*. *NumLowDOIEdit* represents the number of low DOI file editing events, and editing low DOI files might affect software quality. *NumPatternEXSX* captures the event pattern of editing and selecting error-prone files consecutively. This pattern could be a defect prone interaction. The average time spent on editing events is also an important metric to predict defects.

4.3 Predicting Defect Numbers

To evaluate the regression performance using MIMs, we built prediction models using linear regression [9] with the 8:2 time split point. MIM, CM, HM, and their combinations were used to build regression models. We repeated 10-fold cross validation 100 times and computed mean values of *correlation coefficient*, *mean absolute error*, and *root mean squared error*.

Figure 7 shows correlation coefficients of various metrics. The correlation coefficient of MIM is better than CM+HM, which indicates that prediction results using MIMs are more correlated to real defect numbers.

In terms of error, MIMs yield lower errors than CM+HM as shown in Figure 8.

Table 9 shows mean values of measures. Statistically significant values are in bold. For example, correlation coefficient (0.41) of MIMs is better than that (0.30) of CM+HM, and it is statistically significant. In terms of error, the mean absolute error (0.34) and root mean squared error (0.76) of MIM are lower than the values of CM+HM (0.37 and 0.81 respectively). These results show that MIM outperforms CM+HM in regression models.

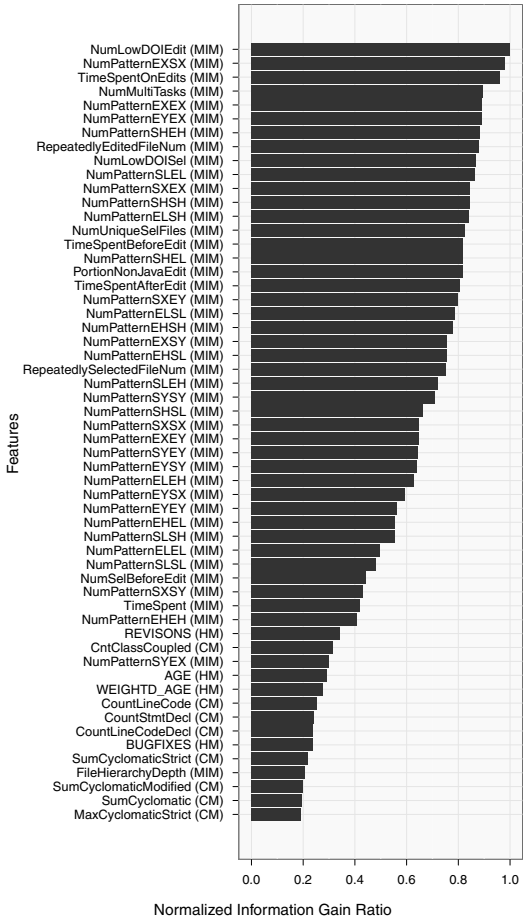

Figure 6: Top 56 ranked metrics (among 113 metrics) based on the gain ratio.

4.4 Predicting CVS-log-based Defects

This section introduces prediction results using CVS-based defect counting. As explained in Section 3.2, we counted the number of post-defects based on edited files recorded in Mylyn tasks to avoid false positives in marking fixed files. However, it is possible that some developers may not have used Mylyn to fix bugs, i.e. these fixes are not recorded in the Mylyn data. This may lead to biased post-defect numbers.

To address this issue, we repeated experiments using the same instances but with a different defect counting method. We used traditional heuristics to count post-defects [22, 32] by searching for 'fix' or 'bug' keywords and bug report IDs in change logs. The change logs containing the keywords or bug IDs were marked as fix changes. After counting post-defects in this approach, we repeated ten-fold cross validation 100 times using three different classification algorithms used in Section 4.2.2. We used the 8:2 time split point for this experiment. The number of instances was 2341, the same number as in Table 7. However, the defect rate increased to 32.4%, since we counted defects by mining CVS change logs.

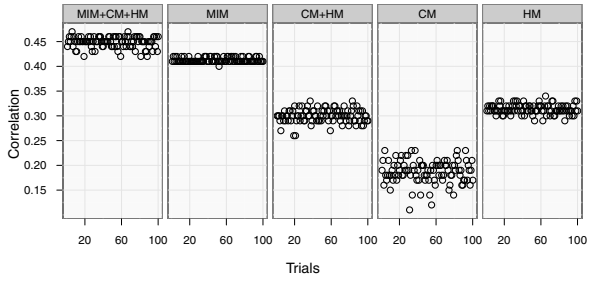

Figure 7: Correlation coefficient comparison (All instances, 8:2 split point, Linear Regression)

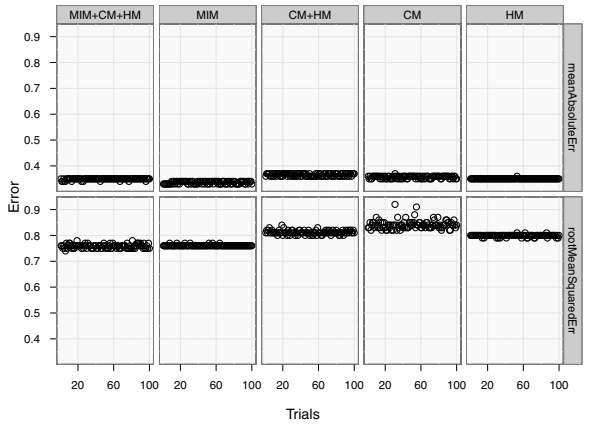

Figure 8: Error comparison (All instances, 80:20 split point)

Figure 9 and Table 10 show F-measure values for each metrics and classification algorithms. For *BayesNet* and *Logistics*, MIM+CM+HM turns out to have the best overall performance. For *Decision Tree*, HM is slightly better than MIM+CM+HM. We would like to point out that in this experiment, using MIMs alone does not yield better performance than CM or HM. One possible explanation is that even though Mylyn is widely used, some of the tasks are still performed without it. In this case, we lost developers' interactions for these tasks, and the information loss may affect the defect prediction performance. However, this issue will be automatically addressed when all files are edited using the Mylyn plug-in. Another reason could be the false positives in CVS-based defect counting [4], and they may affect the defect prediction performance.

5. THREATS TO VALIDITY

We have identified the following threats to validity.

- **Systems examined might not be representative.** Since MIMs rely on the Mylyn data, we intentionally chose subprojects which include the Mylyn data. We might have a project selection bias. In this sense, our approach using MIMs is not generally applicable for projects, which do not use Mylyn.

Table 9: Correlation Coefficient of MIM was validated against CM+HM metrics. (The values in bold indicate the value difference in comparison to CM+HM is statistically significant. (p-value < 0.05))

Measures	MIM+CM+HM	MIM	CM+HM	CM	HM
Correlation Coefficient	**0.45**	**0.41**	0.30	0.19	0.31
Mean absolute error	**0.35**	**0.34**	0.37	0.36	0.35
Root mean squared error	**0.76**	**0.76**	0.81	0.84	0.80

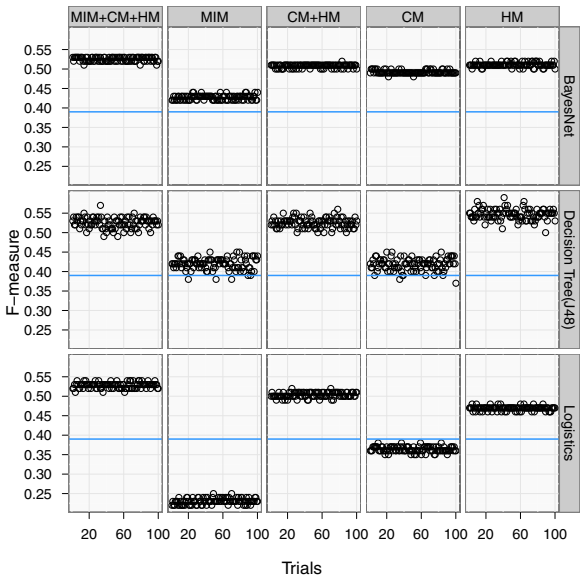

Figure 9: Performance comparison of prediction models by CVS-based labeling (All instances, 8:2 time split point). The Dummy F-measure is shown as a solid line.

- **Systems are all open source projects.** All subprojects examined in this paper are developed as open source projects. Hence they might not be representative of closed-source projects. Commercial software developers may have different micro-level interaction patterns.
- **Defect information might be biased.** We collected defect information from changed files in CVS and edited files recorded in the Mylyn data. In addition, we verified whether the corresponding bug reports were really fixed. However, our defect information might be biased, since some files are edited without using Mylyn and developers may not leave explicit bug IDs in CVS change logs.

Table 10: F-measure mean values of each metrics in CVS-based labeling. (The F-measures in bold indicate the value difference in comparison to CM+HM is statistically significant. (p-value < 0.05))

Measures	MIM+CM+HM	MIM	CM+HM	CM	HM	DUMMY
BayesNet	**0.525**	0.43	0.50	0.49	0.51	0.39
Decision Tree	0.525	0.42	0.525	0.42	0.55	0.39
Logistics	**0.528**	0.23	0.50	0.36	0.47	0.39

6. RELATED WORK

6.1 Defect Prediction

Software defect prediction is a very active research area [3, 14, 15, 27, 28, 37] in software engineering. Researchers have proposed new defect prediction algorithms and/or new metrics to effectively predict defects. Source code metrics such as complexity metrics are widely used for defect prediction, since there is a common understanding that complicated software may yield more defects. For example, Basili et al. [3] used Chidamber and Kemerer metrics, and Ohlsson et al. [27] used McCabe's cyclomatic complexity for defect prediction.

Recently, change history based metrics have been proposed and widely used for defect prediction. Nagappan et al. proposed the code churn metric, which is the amount of changed code, and showed that code churn is very effective for defect prediction. Moser et al. [23] used the number of revisions, authors, past fixes, and age of a file as defect predictors. Kim et al. used previous defect information to predict future defects. Hassan adopted the concept of entropy for change metrics, and found their approach is often better than the code churn approach and the approach based on previous bugs [10]. D'Ambros et al. conducted an extensive comparison of existing bug prediction approaches using source code metrics, change history metrics, past defects and entropy of change metrics [7]. They also proposed two noble metrics: churn and entropy of source code metrics.

Defect metrics other than CMs and HMs have also been proposed. Zimmermann and Nagappan predicted defects in Windows server 2003 using network analysis among binaries [37]. Bacchelli et al. proposed popularity metrics based on e-mail archives [2]. They assumed the most discussed files are more defect-prone. Meneely et al. proposed developer social network based metrics to predict defects [20].

These proposed metrics play an important role in defect prediction, and yield reasonable prediction accuracy. However, they do not capture developers' direct interactions. Proposed MIMs are the first metrics using developer interaction data to predict defects. We have also showed that developers' interaction based metrics outperform traditional metrics such as CMs and HMs.

6.2 Developer Interaction History

In recent years, researchers used developer's interaction history for facilitating software development and maintenance. Zou et al. proposed how to detect interaction coupling from task interaction histories [39]. Their case study showed the information of interaction coupling is helpful to comprehend software maintenance activities. Robbes and Lanza proposed a code completion tool based on programmer's code editing interaction history [29]. Ying and Robillard analyzed the influence of program change tasks based on developers' editing behavior and found editing patterns that are helpful for software tool designers [35].

Kersten et al. suggested task context model and implemented Mylyn to store/restore task context when developers switch their task context [11, 12]. As Mylyn is getting popular, there are many available developers' interaction history data captured by Mylyn. Murphy et al. analyzed statistics about IDE usage using the Mylyn data and showed the most used UI components and commands [24].

These approaches are similar to our work in that they are leveraging developer interactions to improve software quality. However, they do not address the software defect prediction issue using developers' interaction history, while we extract MIMs for defect prediction.

7. CONCLUSIONS

We proposed 56 micro interaction metrics, and showed that they significantly improve defect classification and regression accuracy. Our findings concur with previous studies [16, 19], which indicates developers' interaction patterns affect software quality.

In our experimental evaluation, MIM based defect prediction models are applied to Eclipse subprojects. We plan to extend our experiments by adding more subjects including industrial projects. Current MIMs depend on the Mylyn data, which may not be available for some projects. Therefore, we plan to extend MIMs leveraging other sources of developers' interaction data.

Overall, we expect that future defect prediction models will use more information from developers' direct and micro level interactions for effective defect prediction. MIMs are a first step in this direction.

All data we used in our experiments are publicly available at `http://www.cse.ust.hk/~jcnam/mim`.

8. ACKNOWLEDGEMENTS

Our thanks to Ananya Kanjilal, Seonah Lee, Yida Tao, Annie T.T. Ying, and the anonymous reviewers of ESEC/FSE for valuable and helpful suggestions on earlier versions of this paper.

This work was supported in part by Mid-career Researcher Program through NRF grant funded by the MEST (No. 2010-0000142) and in part by the R&BD Support Center of Seoul Development Institute and the South Korean government (Project title: WR080951, Establishment of Bell Labs in Seoul / Research of Broadband Convergent Networks and their Enabling Technologies).

9. REFERENCES

[1] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[2] A. Bacchelli, M. D'Ambros, and M. Lanza. Are popular classes more defect prone? In *Fundamental Approaches to Software Engineering*, volume 6013, pages 59–73. 2010.

[3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22:751–761, October 1996.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *ESEC/FSE'09*, 2009.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.

[6] P. Dalgaard. *Introductory statistics with R*. Springer, 2nd edition, 2010.

[7] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31 –41, May 2010.

[8] Eclipse Mylyn. http://www.eclipse.org/mylyn.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[10] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, 2009.

[11] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 159–168, 2005.

[12] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 1–11, 2006.

[13] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung, and S. Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Trans. Softw. Eng.*, 99, 2011.

[14] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34:181–196, March 2008.

[15] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 489–498, 2007.

[16] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *J. Vis. Lang. Comput.*, 16:41–84, February 2005.

[17] S. Kullback. *Information Theory and Statistics*. John Wiley and Sons, 1959.

[18] S. Kullback and R. A. Leibler. On information and su ciency. In *The annals of Mathematical Statistics*, volume 22, pages 79–86, 1951.

[19] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 492–501, 2006.

[20] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, 2008.

[21] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33:2–13, January 2007.

[22] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, 2000.

[23] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190, 2008.

[24] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23:76–83, July 2006.

[25] Mylyn/Integrator Reference. http://wiki.eclipse.org/Mylyn_Integrator_Reference.

[26] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, 2006.

[27] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw.*, 22(12):886 –894, Dec. 1996.

[28] 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.*, 31:340–355, April 2005.

[29] R. Robbes and M. Lanza. How program history can improve code completion. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 317 –326, sept. 2008.

[30] S. Scott and S. Matwin. Feature engineering for text classification. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 379–388, 1999.

[31] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim. Reducing features to improve bug prediction. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 600–604, 2009.

[32] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, 2005.

[33] Understand 2.0. http://www.scitools.com/products/understand/.

[34] I. H. Witten and E. Frank. *Data mining : practical machine learning tools and techniques*. Morgan Kaufmann, Amsterdam, 2 edition, 2005.

[35] A. T. Ying and M. P. Robillard. The influence of the task on programmer behaviour. In *Program Comprehension, 2011. ICPC '11. 19th IEEE International Conference on*, June 2011.

[36] H. Zhang. An investigation of the relationships between lines of code and defects. In *ICSM 2009*, pages 274 –283, 2009.

[37] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, 2008.

[38] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, 2007.

[39] L. Zou, M. Godfrey, and A. Hassan. Detecting interaction coupling from task interaction histories. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 135 –144, June 2007.

APPENDIX: List of Micro Interaction Metrics

Category		Name	Description
File Level	Effort	NumSelectionEvent NumEditEvent NumManipulationEvent NumPropagationEvent	The number of each kind of events of the file.
	Interest	AvgDOI VarDOI	Average and variance of Degree of Interest (DOI) values of the file.
	Intervals	AvgTimeIntervalEditEdit	The average time interval between two edit events of the file.
		AvgTimeIntervalSelSel	The average time interval between two selection events of the file.
		AvgTimeGapSelEdit	The average time interval between selection and edit events of the file
Task Level	Effort	NumUniqueSelFiles	The unique number of selected files in a task
		NumUniqueEdit	The unique number of edited files in a task.
		TimeSpent	The total time spent on a task.
	Distraction	NumLowDOISel	The number of selection events with low DOI (DOI < median of all DOIs).
		NumLowDOIEdit	The number of edit events with low DOI (DOI < median of all DOIs).
		PortionNonJavaEdit	The percentage of non-java file edit events in a task.
	Work Portion	TimeSpentBeforeEdit	The ratio of the time spent (out of total time spent) before the first edit.
		TimeSpendAfterEdit	The ratio of the time spent (out of total time spent) after the last edit.
		TimeSpentOnEdits	The ratio of the time spent (out of total time spent) between the first and last edits.
		NumSelBeforeEdit	The number of unique selections before the first edit event.
		NumSelAfterEdit	The number of unique selections after the last edit event.
	Repetition	RepeatedlySelectedFileNum	The number of files selected more than once in one task.
		RepeatedlyEditedFileNum	The number of files edited more than once in one task.
	Task Load	NumMultiTasks	The number of ongoing tasks at the same time.
		FileHierarchyDepth	The average depth of file hierarchy.
	Event Pattern	NumPatternSXSX NumPatternSXSY NumPatternSYSX NumPatternSYSY NumPatternSXEX NumPatternSXEY NumPatternSYEX NumPatternSYEY NumPatternEXSX NumPatternEXSY NumPatternEYSX NumPatternEYSY NumPatternEXEX NumPatternEXEY NumPatternEYEX NumPatternEYEY	The number of adjacent sequential event patterns in one task. S and E represent selection and edit events respectively. For example, S?E? describes adjacent sequential interactions of selecting a file and continuously editing a file. We classify files into two groups: X and Y. If a file satisfies one of the following three conditions, we mark them as Group X. 1) If the file has been ever fixed before due to a defect. 2) If the file is one of frequently edited files in a task. (frequency threshold: top 50% of file editing frequencies). 3) If the file has been recently selected or edited within $\pm n$ time spent of a given task, where $n = \frac{totaltime}{4}$ Otherwise, we put the file in the Y group. The group X implies high locality of file accessing with error-prone interaction. The hint of locality concept is from [15].
		NumPatternSLSL NumPatternSLSH NumPatternSHSL NumPatternSHSH NumPatternSLEL NumPatternSLEH NumPatternSHEL NumPatternSHEH NumPatternELSL NumPatternELSH NumPatternEHSL NumPatternEHSH NumPatternELEL NumPatternELEH NumPatternEHEL NumPatternEHEH	As explained above, S and E share the same meaning. But instead of X or Y encoding, H or L encoding is used here. If the DOI value of the edited or selected file is higher than the median of all the DOI values in a task, it is denoted by H. Otherwise, it is L.