On the Relationship Between Change Coupling and Software Defects

Marco D'Ambros, Michele Lanza, Romain Robbes
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—Change coupling is the implicit relationship between two or more software artifacts that have been observed to frequently change together during the evolution of a software system. Researchers have studied this dependency and have observed that it points to design issues such as architectural decay. It is still unknown whether change coupling correlates with a tangible effect of design issues, i.e., software defects.**

In this paper we analyze the relationship between change coupling and software defects on three large software systems. We investigate whether change coupling correlates with defects, and if the performance of bug prediction models based on software metrics can be improved with change coupling information.

***Keywords*-Change coupling; Software defects**

I. Introduction

The analysis of the evolution of software [16], has two main goals, namely to infer causes of its current problems, and to predict its future development. Many approaches based on evolutionary information demonstrated that not only can this information be used to predict a system's future evolution [17], [24], but it can also provide good starting points for reengineering activities [12].

The history of a software system also holds information about *change coupling* [2], [10]. These are implicit and evolutionary dependencies between the artifacts of a system which, although potentially not structurally related, evolve together and are therefore linked to each other from an evolutionary point of view. In short, these coupled entities have changed together in the past and are thus likely to change in the future. Change coupling information reveals potentially misplaced artifacts in a software system, because entities that evolve together should be placed close to each other for cognitive reasons: A developer who modifies a file in a system could forget to modify related files because they are placed in other subsystems or packages.

Change coupling has been considered a bad symptom in a software system [10], [11]: At a fine grained level because a developer who changes an entity might forget to change related entities or, at the system level, because high change coupling among modules points to design issues such as architecture decay. Researchers have studied change coupling in order to address these two issues using change recommendation systems and software evolution analysis approaches.

To address the issue of co-changing entities, change recommendation systems use fine grained change coupling information to predict entities that are likely to be modified when another is being modified [5], [28]. To pinpoint architectural design issues, software evolution analysis approaches abstract change coupling information to analyze the architecture of the system and/or to detect candidates for reengineering [4], [10], [11], [22]. In this scenario, change coupling information is used to find good starting points for the reengineering process, because coupled artifacts lead to maintenance problems, while decreasing the change coupling in a system leads to an improved system structure.

All these approaches are based on the assumption that change coupling indeed is a cause of issues in a software system. However, the relationship between change coupling and a tangible effect of software issues has not been studied yet. To perform such a study, one needs an objective quantification of the issues that affect a software system and its components. A software defect repository, which records all the known issues about a software system, provides such a quantification. Eaddy et al. performed a similar study linking cross-cutting concerns with defects [8], but the specific case of change coupling remains unaddressed.

In this paper we define various measures of change coupling and analyze their correlations with software defects. The contributions of this paper are:

- We provide empirical evidence, through three case studies, that indeed change coupling correlates with defects extracted from a bug repository, and investigate the relationships of change coupling with defects based on the severity of the reported bugs.
- We compare the correlation of change coupling with a catalog of complexity metrics (including the Chidamber & Kemerer metrics suite [6]), and find that change coupling correlates with defects better than complexity metrics.
- We show that the performance of defect prediction models, based on complexity metrics, can be improved with change coupling information.

Structure of the paper. We describe our dataset in Section II, introduce change coupling measures in Section III and analyze their correlation with software defects in Section IV. In Section V we explain how to enrich defect prediction models with change coupling information. We address the threats to validity in Section VI, look at related work in Section VII and conclude in Section VIII.

II. Data Collection

To analyze the relationship between change coupling and software defects, we first need to create and populate a model with the necessary data.

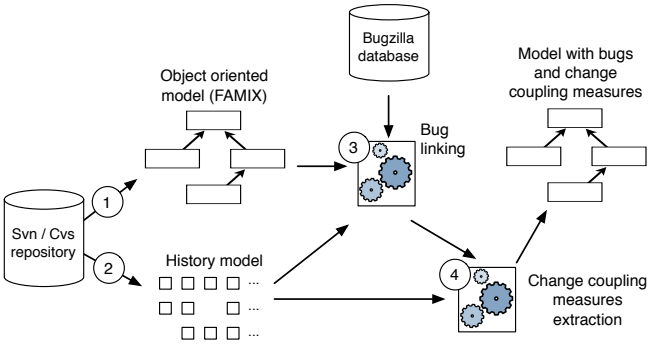


Figure 1. Creating a model with bug and change coupling information.

Figure 1 shows how we create a model of a software system, containing source code data, bug information and change coupling measures.

Creating a Source Code Model (Figure 1.1): To create the source code model, we retrieve and parse the source code, by performing a check out from the SVN (or CVS) repository and by parsing it using the iPlasma tool (available at: http://loose.upt.ro/iplasma). The result is a FAMIX [7] model of the source code.

Creating the History Model (Figure 1.2): To detect co-change occurences, we model how the system changed during its lifetime by parsing the versioning system log files and by creating a model of the history of the system. We model the system's history with the transactions extracted from the SCM system's repository. A transaction corresponds to a commit in the SCM repository, i.e., it is a set of files which were modified and committed to the repository, together with the commit timestamp, the author and the comment written by the author at commit time. SVN marks co-changing files at commit time as belonging to the same transaction while in CVS the transactions must be inferred from the modification time (plus commit comment and author) of each file. In the case of CVS, we reconstruct the transactions using a sliding time window approach.

Linking Classes with Bugs (Figure 1.3): To reason about the presence of bugs affecting parts of the software system, we first map each problem report with the components of the system that it affects. We link FAMIX classes with versioning system files and these files with bugs retrieved from a Bugzilla repository, as shown in Figure 2.

A file version in the versioning system contains a developer comment written at commit time, which often includes a reference to a problem report (e.g., "fixed bug 123"). Such references allow us to link problem reports with files in the versioning system, and therefore with source code artifacts,

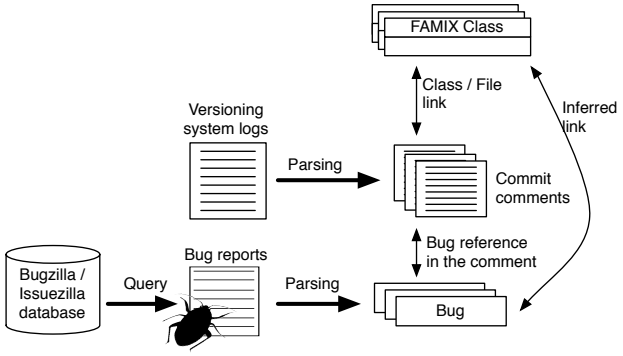


Figure 2. Linking bugs, SCM files and FAMIX classes.

i.e., classes. However, the link between a CVS/SVN file and a Bugzilla problem report has not yet been formally defined. To find a reference to the problem report id, we use pattern matching techniques on the developer comments, an approach widely used in practice [9], [27].

Due to the file-based nature of SVN and CVS and to the fact that Java inner classes are defined in the same file as their containing class, in our approach several classes might point to the same CVS/SVN file, i.e., a bug linking to a file version, might actually be linking to more than one class. We are not aware of a workaround for this problem, which in fact is a shortcoming of the bug tracking system. For this reason, we do *not* consider inner classes.

Change Coupling Computation (Figure 1.4): At this point we have a model including source code information and defects data. The last thing we do, before proceeding with the analysis, is to enrich the model with the four change coupling measures that we define in the next section, and compute their values for each FAMIX class.

Case Studies: To study the relationship between change coupling and software defects we analyze three large Java software systems: ArgoUML, Eclipse JDT Core, and Mylyn.

	ArgoUML	**JDT Core**	**Mylyn**
System version	0.28	3.3	3.1.0
Versioning system	SVN	CVS	CVS
# Classes	2197	1193	3050
# Transactions	15257	13186	9373
Avg. # transactions per class	14.3	68	11.7
Avg. # shared transactions per class	0.37	5.3	0.39

Table I
Measures of the studied software systems.

Table I shows the size of the systems in terms of classes and transactions. In computing the change coupling, we filtered out the transactions involving more than 100 classes, which were 86 for ArgoUML, 59 for Eclipse JDT Core, and 102 for Mylyn. We manually inspected the commit comments of these transactions, the vast majority of which concerned license changes, Javadoc and documentation updates.

III. Change Coupling Measures

Change coupling is the implicit and evolutionary dependency of two software artifacts that have been observed to frequently change together during the evolution of a software system. The more they changed together, the stronger the change coupling dependency is. However, there is no consensus on the formal definition of change coupling, and several alternative measures exist. We formally define 4 measures of change coupling emphasizing different aspects.

To measure the correlation of change coupling with software defects we need change coupling measures which are defined for each entity in the system. The entity in our case is a class, as classes are a cornerstone of the object-oriented paradigm, and we want to be able to compare change coupling with object oriented metrics.

The measures we define concern the coupling of a class with the entire system. An alternative is a measure of change coupling for each pair of entities in the system. However since bugs are often mapped to one entity only, a coupling measure involving only one entity is prefereable. We can define a measure of coupling of a class with the entire system simply by aggregating the pairwise coupling measures.

In the following definitions we use the concept of *n-coupled classes*. We consider two classes n-coupled when there are at least n transactions which include both the classes. Thus, all our change coupling measures are functions of n. Given two classes c_1 and c_2, they are n-coupled if the following condition holds:

$$|\{t \in T | c_1 \in t \wedge c_2 \in t\}| \geq n \tag{1}$$

where T is the set of all the transactions. Given a class c we define the *set of coupled classes* (SCC) as:

$$\text{SCC}(c, n) = \{c_i | c_i \neq c \wedge c \text{ is } n\text{-coupled with } c_i\} \tag{2}$$

Figure 3 shows an example scenario with 5 classes and 6 transactions. In this case $\text{SCC}(c1, 3) = \{c2, c5\}$, $\text{SCC}(c1, 4) = \{c2, c5\}$ and $\text{SCC}(c1, 5) = \{c5\}$. In computing n-coupled classes we filter out large transactions, as previously mentioned.

A. Number of Coupled Classes (NOCC)

The first per-class measure of change coupling is the number of classes n-coupled with a given class c. This measure emphasizes the raw number of classes with which a given class is coupled with. NOCC is defined as:

$$\text{NOCC}(c, n) = |\text{SCC}(c, n)| \tag{3}$$

The NOCC measure is the cardinality of the set of coupled classes. In the example in Figure 3 $\text{NOCC}(c1, 3) = 2$.

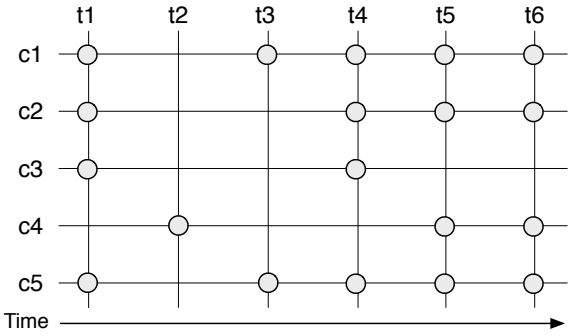


Figure 3. Sample scenario of classes and transactions.

B. Sum of Coupling (SOC)

The sum of coupling is the sum of the shared transactions between a given class c and all the classes n-coupled with c. We define SOC as:

$$\text{SOC}(c, n) = \sum_{c_i \in \text{SCC}(c,n)} |\{t \in T | c_i \in t \wedge c \in t\}| \tag{4}$$

The SOC measure is the sum of the cardinalities of the sets of transactions which include the class c and the classes n-coupled with c. Compared to NOCC, SOC also takes into account the strength of the couplings. In Figure 3 $\text{SOC}(c1, 3) = 4 + 5 = 9$.

C. Exponentially Weighted Sum of Coupling (EWSOC)

EWSOC is a variation of SOC, where the shared transactions are exponentially weighted according to their distance in time, emphasizing recent changes over past changes. We define EWSOC as:

$$\text{EWSOC}(c, n) = \sum_{c_i \in \text{SCC}(c,n)} \text{EWC}(c_i, c), \text{ where} \tag{5}$$

$$\text{EWC}(c_i, c) = \sum_{t_k \in T(c)} \begin{cases} 0 & \text{if } c_i \notin t_k \\ \frac{1}{2^{|T(c)|-k}} & \text{if } c_i \in t_k \end{cases} \tag{6}$$

$T(c)$ are all the transactions, sorted by time, which include the class c. Figure 4 shows an example of computation of EWSOC for the class $c1$ for $n = 3$. In this case $T(c1) = \{t1, t3, t4, t5, t6\}$, $|T(c1)| = 5$ ($t2$ is not included in the computation since c is absent in it) and therefore

$$\begin{aligned} \text{EWSOC}(c1, 3) &= \text{EWC}(c1, c2) \\ &= \frac{1}{2^{5-5}} + \frac{1}{2^{5-4}} + \frac{1}{2^{5-3}} + 0 + \frac{1}{2^{5-1}} \end{aligned}$$

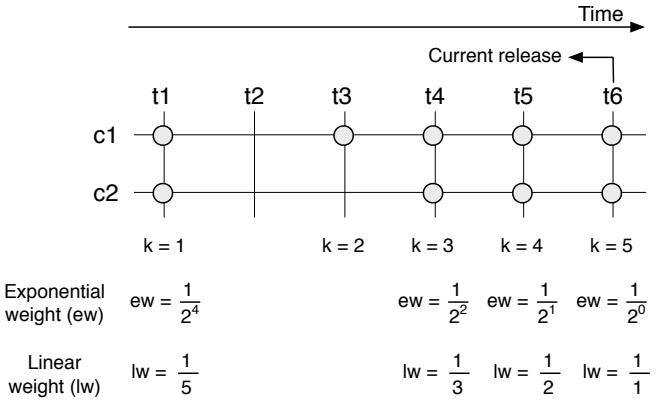


Figure 4. Example EWSOC and LWSOC computations.

D. Linearly Weighted Sum of Coupling (LWSOC)

The last per-class measure of change coupling is another variation of the sum of coupling, in which the shared transactions are linearly weighted according to their distance in time. Like EWSOC, LWSOC emphasizes recent changes, penalizing past changes less. We define LWSOC as:

$$\text{LWSOC}(c,n) = \sum_{c_i \in \text{SCC}(c,n)} \text{LWC}(c_i, c), \text{ where} \quad (7)$$

$$\text{LWC}(c_i,c) = \sum_{t_k \in T(c)} \begin{cases} 0 & \text{if } c_i \notin t_k \\ \frac{1}{|T(c)|+1-k} & \text{if } c_i \in t_k \end{cases} \quad (8)$$

In Figure 4, LWSOC($c1$,3) is equal to LWC($c1$,$c2$)

$$\text{LWC}(c1,c2) = \frac{1}{6-5} + \frac{1}{6-4} + \frac{1}{6-3} + 0 + \frac{1}{6-1}$$

E. Common Behaviors and Differences

All the measures are defined on a class-by-class level, and aggregated to recover a measure of the coupling of one class with the entire system. All the defined measures decrease if n increases, as the set of coupled classes at the value n shrinks if n increases. Beyond that, these 4 measures emphasize different aspects of change coupling: NOCC measures only the number of co-change occurences of a class with all the other classes that exceed the threshold n. On the other hand, SOC takes into account the magnitude of each coupling relationship beyond the threshold, so that a pair of classes changing extremely often together is taken into account differently. EWSOC and LWSOC function similarly, but take into account the recency of the co-change relationships. A reason for this is that two classes may have been co-changed heavily in the past, but have since been refactored to not depend on each other. Their past behavior should not affect their current coupling value. EWSOC discounts the past more quickly than LWSOC does.

IV. Correlation Analysis

The goal of our study is to answer the following questions:

1) Does change coupling correlate with software defects? If so, which change coupling measure correlates best?
2) Does change coupling correlate more with severe defects than with minor ones?

To answer these questions we use the Spearman correlation coefficient, which measures the correlation between two rankings. High correlations are indicated by values close to 1 and -1, in which 1 denotes an identical ranking and -1 an opposite ranking, while values close to 0 indicate no correlation. All the Spearman correlations we report are significant at the 0.01 level. We compute the values of the correlation between the number of defects per class (or number of defects with a given severity) and the various measures of change coupling. For comparison purposes, we also compute the Spearman coefficient for other metrics: The Chidamber & Kemerer object oriented metrics suite –CK metrics– [6] (WMC: Weighted Method Count, DIT: Depth of Inheritance Tree, RFC: Response For Class, NOC: Number Of Children, CBO: Coupling Between Objects, LCOM: Lack of Cohesion in Methods), a selection of other object-oriented metrics (NOA: Number Of Attributes, NOM: Number Of Methods, FANIN, FANOUT, LOC: Lines Of Code) and the number of changes to a class (Changes).

A. Results

Figure 5 and Figure 6 show the Spearman correlation of the number of bugs with the metrics we tested accross the 3 case studies. Figure 5 displays the correlation for all levels, while Figure 6 shows it for selected categories of bugs, according to their labels in the bug tracking system (major bugs and high priority bugs). All the graphs follow the same format: The Spearman correlation is indicated on the y axis, while the x axis indicates the threshold used for the computation of change couplings metrics (i.e., the value of n used as a basis to compute n-coupled classes). For example, all the change coupling measures at the x position of 3 are computed using the set of 3-coupled classes. Metrics which do not depend on this threshold (such as Changes, FANOUT, or CBO) are hence flat lines. The metrics on each graph are the 4 coupling metrics (NOCC, SOC, EWSOC, LWSOC), the number of changes metric, and the best-performing among the object-oriented metrics for each project and each bug category.

Correlation with all types of bugs: Figure 5 shows the correlation of metrics with all types of bugs, for all systems. The best performing object-oriented metrics are: Fan out for Eclipse and ArgoUML, and CBO for Mylyn. For all the software systems change coupling indeed correlates with the number of bugs, since the Spearman correlation reaches values above 0.5, especially for Eclipse where the maximum Spearman is above 0.8. The SOC measure is the best for

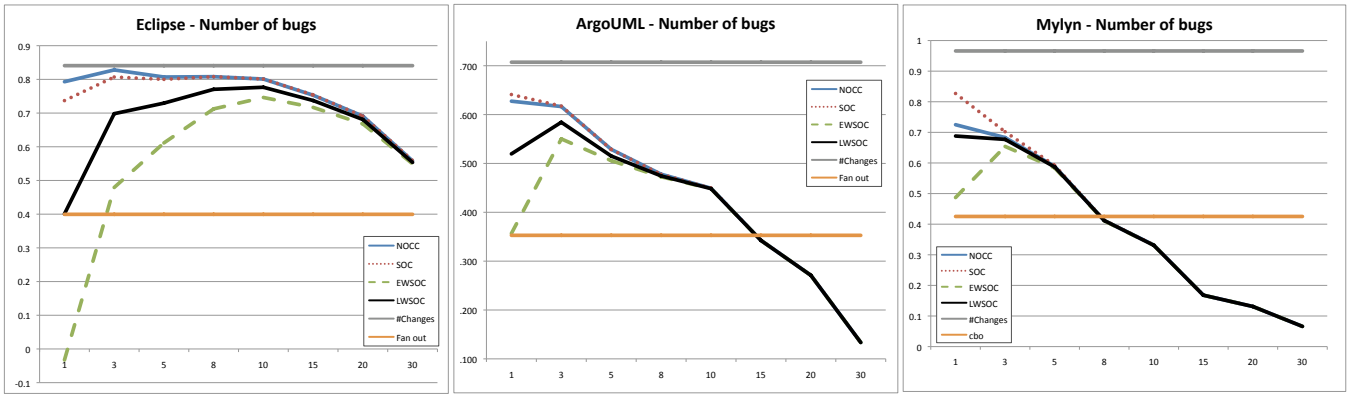


Figure 5. Correlations between number of bugs and change coupling measures, number of changes and the best object-oriented metric. The correlations are measured with the Sperman correlation coefficient.

ArgoUML and Mylyn, and the second best for Eclipse. All the coupling measures decrease after a certain value of n: 3 for ArgoUML and Mylyn, 10 for Eclipse. EWSOC and LWSOC do not correlate for low values of n, while they are comparable with NOCC and SOC for $n \geq 3$ in ArgoUML and Mylyn, $n \geq 10$ for Eclipse.

Correlation with major bugs: Figure 6(a) shows the Spearman correlation between the number of major bugs and change coupling measures. We consider a bug as major if its severity is major, critical or blocker. We also show the correlations with number of changes and the best object-oriented metric: Fan out for Eclipse and LOC for Mylyn. For Eclipse, with $3 \leq n \leq 20$ NOCC and SOC are very close to number of changes (about 0.7). EWSOC and LWSOC have bad performances with $n < 10$, while starting from 10 they are above 0.6. In the case of Mylyn the correlations are lower, with a maximum of circa 0.4. For $n = 5$ all the change coupling measures are at the maximum and above number of changes, and for $n > 8$ they rapidly decrease. We do not show the result for ArgoUML because the number of major bugs is not large enough to get significant correlations.

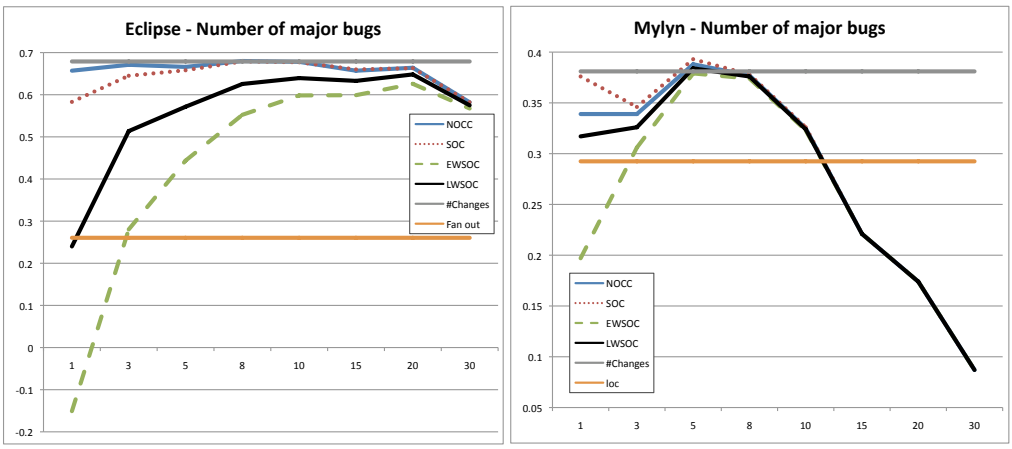
Correlation with high priority bugs: Figure 6(b) shows the Spearman correlations for the number of high priority bugs. This time, the best object-oriented metrics are Fan out for Eclipse and CBO for ArgoUML. For this particular type of bugs, the correlations are weaker, with a maximum around 0.55 for Eclipse and 0.45 for ArgoUML. The change coupling measures are often better than the number of changes. In the case of Eclipse, NOCC is always better, SOC is better for $n \geq 5$ and LWSOC for $n \geq 15$, while EWSOC is always worse. For ArgoUML all the change coupling measures have a maximum for $n = 8$, which is greater than the correlation of the number of changes. After that, for $n > 8$ the correlations rapidly decrease. We do not show the result for Mylyn because the number of high priority bugs is not large enough to get significant correlations.

B. Discussion

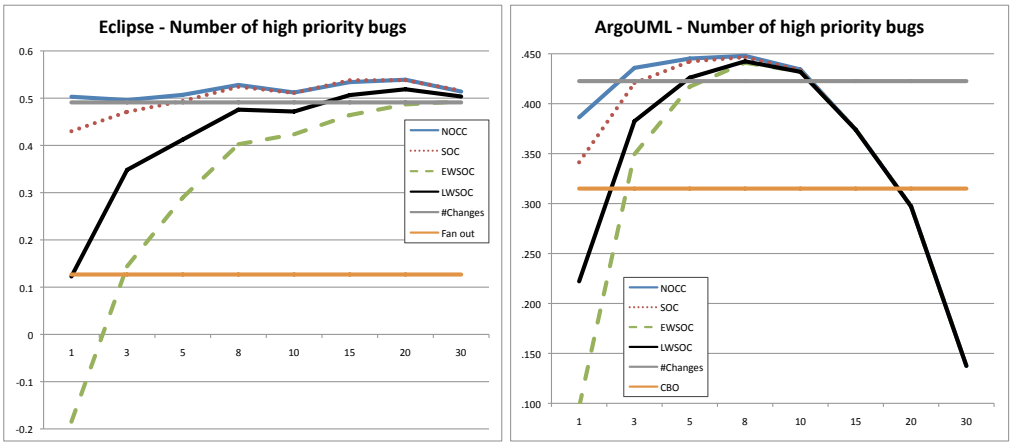
Based on the data presented in Figure 5 and Figure 6 we derive the following insights.

Change coupling works better than metrics: From Figure 5 we see that, for every system, there is a range of values of n in which change coupling measures indeed correlate with number of defects. They correlate more than the CK and other object-oriented metrics, but less than the number of changes. The fact that number of changes correlates with number of defects was already assessed by Nagappan and Ball [19]. One possible reason why the number of changes correlates more is that this information is defined for every class in the system, while only some classes have change coupling measures greater than 0. This also explains why change coupling measures peak at a given index and then decrease in accuracy, as very few classes have a change history large enough to exceed moderately high thresholds of co-change. Further, since not all the bugs are related to a change coupling relationship, all in all the number of changes have an higher correlation with defects. Similar to this situation, Gyimóthy et al. found that LOC is among the best metrics to predict defects [14], since it is defined for all entities in the system. In conclusion, we can answer question 1: *Change coupling correlates with defects, more than metrics but less than number of changes*.

Change proneness plays a role: Another observable fact in Figure 5 is that for ArgoUML and Mylyn the correlation of the change coupling measures rapidly decreases with $n \geq 5$, while for Eclipse this happens with $n \geq 20$. The reason behind this is that Eclipse classes have, on average, many more changes and more shared transactions than classes in the other two systems. In Eclipse the average number of changes per class is 68, while in ArgoUML it is 14.3 and in Mylyn 11.7. The average number of shared transactions per class is 5.3 for Eclipse, 0.37 for ArgoUML and 0.39 for Mylyn. Since we consider three systems, we cannot derive a general formula, but limit ourselves to note that the



(a) Major bugs



(b) Bugs with high priority

Figure 6. Spearman correlations between number of major/high priority bugs and change coupling measures.

correlation depends on the change proneness of the system. In short the insight is the following: *The correlation between change coupling measures and defects varies with n. The trend and the maximum correlation values depend on the software system and in particular on its change proneness.*

Change coupling is harmful: The situation in Figure 6 is different from the one in Figure 5. The average value of the Spearman correlation is lower when considering only major or high priority bugs than with all the bugs. This is not surprising, since there is a smaller amount of data and therefore the correlation is less precise. The interesting fact here is the delta beween the number of changes and the change coupling measures: It is lower for major and high priority bugs, with respect to all the bugs, and it is often negative, i.e., change coupling measures correlate more than number of changes with number of major/high priority bugs. One possible explanation is that change coupling can be detected only in the evolution of a system. As such, this type of dependency is often hidden and might be related to bugs with a high priority or a high severity. The answer to question 2 is then: *On average the correlation between change coupling measures and number of major/high priority bugs is lower than with all the bugs. For these particular bugs change coupling measures are always better than software metrics and, in many cases, than number of changes.*

Sometimes it is better *not* to forget the past: One last observation from both Figure 5 and Figure 6 is that the correlation for EWSOC is always below the one for LWSOC, and the latter one is always below NOCC and SOC. From this we infer that "penalizing" couplings in the past does not work in correlating with number of defects, i.e., couplings in the past also correlate with defects. EWSOC, which penalizes the past more than LWSOC, correlates less with defects. *"Penalizing" change coupling in the past decreases the correlation with number of defects. The best change coupling metrics are then NOCC and SOC.* This is the second part of the answer to question 1.

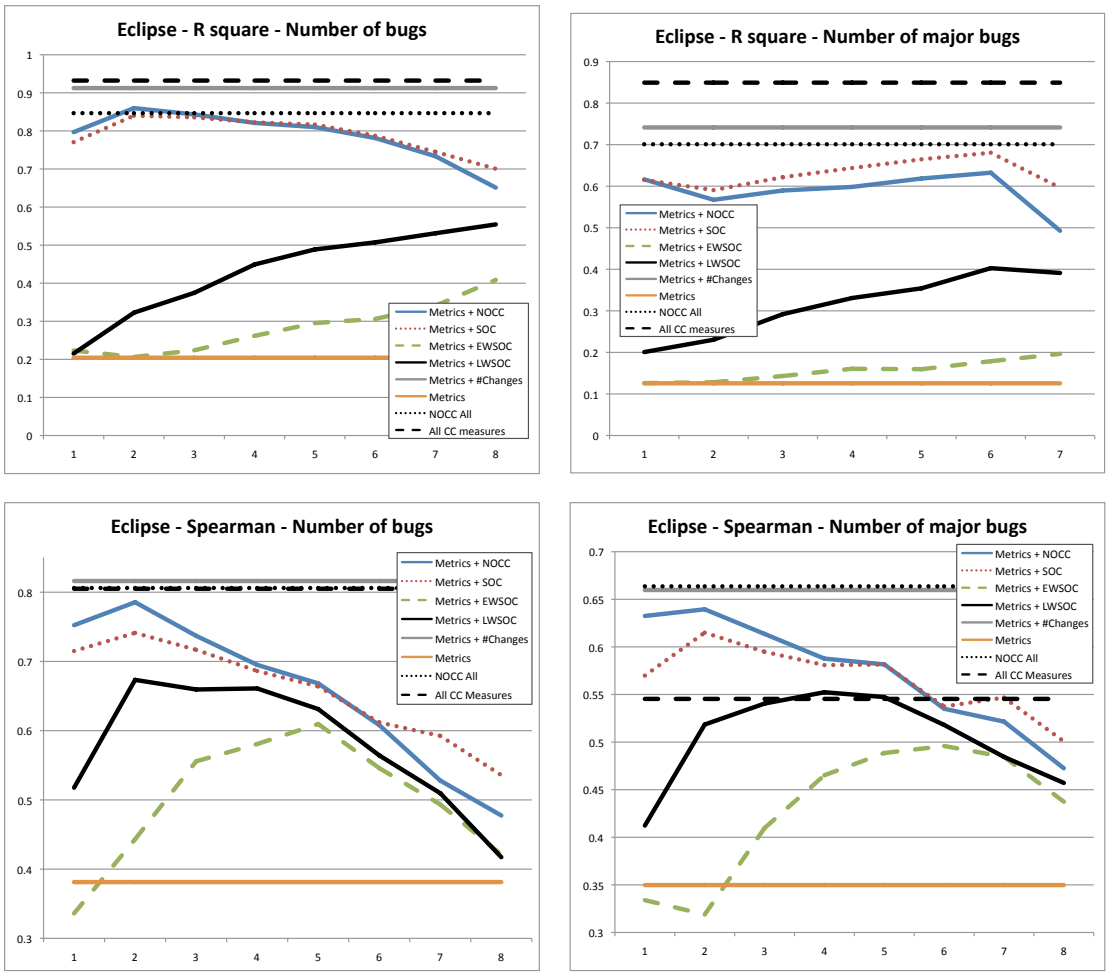


Figure 7. Results of the regression analysis for Eclipse.

V. Regression Analysis

Our goal is to answer the following questions:

1) Does the use of change coupling improve explanative and predictive powers of bug prediction models based on software metrics?
2) Is the improvement greater for severe bugs?

To do so, we create and evaluate different regression models in which the independent variables (for predicting) are respectively metrics, change coupling measures, number of changes and their combinations, while the dependent variable (the predicted one) is the number of bugs, the number of major bugs and the number of high priority bugs. Our experiments follow the methodology proposed by Nagappan et al. in [20] and used also in [26], which consists in the following steps: Principal component analysis, building regression models, evaluating explanative power and evaluating prediction power.

Principal Component Analysis: Principal component analysis (PCA) is a standard statistical technique to avoid the problem of multicollinearity among the independent variables. This problem comes from intercorrelations amongst these variables and can lead to an inflated variance in the estimation of the dependent variable.

We do not build the regression models using the actual variables (e.g., metrics, change coupling measures) as independent variables, but instead we use sets of principal components (PC). PC are independent and therefore do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible. In our experiments we select sets of PC that account for a cumulative sample variance of at least 95%.

Building Regression Models: To evaluate the predictive power of the regression models we do cross validation, i.e., we use 90% of the dataset (90% of the classes) to build the prediction model, and the remaining 10% of the dataset to evaluate the efficacy of the built model. For each model we perform 50 "folds", i.e., we create 50 random 90%-10% splits of the data.

Evaluating Explanative Power: To evaluate the explanative power of the regression models we use the R^2 coefficient. It is the ratio of the regression sum of squares to the total sum of squares. R^2 ranges from 0 to 1, and the higher the value is, the more variability is explained by the model, i.e., the better the explanative power of the model is. Another indicator of the explanative power is the adjusted R^2, which takes into account the degrees of freedom of the independent variables and the sample population.

We also test the statistical significance of the regression models using the F-test. All the regression models that we build are significant at the 99% level ($p < 0.01$).

Evaluating Prediction Power: To evaluate the prediction power of the regression models, we compute the Spearman correlation between the predicted number of defects and the actual number. We compute the Spearman on the validation set, which is 10% of the original dataset. Since we perform 50 folds cross validation, the final value of the Spearman is the average over the 50 folds.

Results: Figure 7 shows the results of our experiments for Eclipse in terms of explanative power (R^2) and predictive power (Spearman correlation). We show the results for the regression models built using the following sets of variables: (1) object-oriented metrics, (2) metrics and number of changes, (3) metrics and NOCC, (4) metrics and SOC, (5) metrics and EWSOC, (6) metrics and LWSOC, (7) all NOCC, i.e., the NOCC metrics for each value of n and (8) all CC measures, i.e., all the measures of change coupling for each value of n. For space reasons we do not show all the results, but only a subset of them including all bugs and major bugs for Eclipse. The other results, for the high priority bugs and for the other two systems (ArgoUML and Mylyn) are on the same line with the ones presented in Figure 7. We do not show the values of adjusted R^2, since it tends to remain comparable to R^2.

Discussion: Regression models based on object-oriented metrics and change coupling information have a greater explanative and predictive power than models based only on object-oriented metrics. However, the model based on metrics and number of changes have a slightly better prediction power, than "all CC measures" and "NOCC all", and a slightly worse explanative power, than "all CC measures". This answers the first question mentioned in this section.

When considering only major bugs the overall performance is lower, but the models based on change coupling are better than the one based on number of changes. This answers our second question. The model based on "all CC measures" is the best in terms of explanative power, but it also suffers for overfitting, since its prediction performance is much lower. On the other hand, the model based on "NOCC all" is the best in terms of prediction (slightly better than number of changes), but not in terms of R^2.

The conclusions drawn for the correlation analysis are still valid for the regression. First it is better not to forget the past, i.e., change coupling measures which penalize past coupling relationships (EWSOC and LWSOC) have bad explanative and prediction power. Second, change proneness play a role, i.e., the change coupling measures have different trends for different software systems. This is because different systems have different average numbers of transactions and shared transactions per class. For lack of space we do not show the regression results for ArgoUML and Mylyn, but the trends of NOCC, SOC, EWSOC and LWSOC are similar to the ones presented in Figure 5 for the correlation analysis.

VI. Threats to Validity

Threats to construct validity: These threats regard the relationship between theory and observation, i.e., the measured variables may not actually measure the conceptual variable. A first construct validity threat concerns the way we link bugs with versioning system files and subsequently with classes. In fact, the pattern matching technique we use to detect bug references in commit comments does not guarantee that all the links are found. In addition to this, we made the assumption that commit comments do contain bug fixing information, which limits the application of our approach only to software projects where this convention is used. However, this technique currently represents the state of the art in linking bugs to versioning system files and is widely used in the literature [9], [27]. A second threat concerns inner classes. Linking a Java class with a versioning system file implies that we cannot consider inner classes, because in Java they are defined in the same file in which the container class is defined. A last construct validity threat is due to the noise affecting Bugzilla repositories. Antoniol et al. showed that a considerable fraction of problem reports marked as bugs in Bugzilla (according to their severity) are indeed "non bugs", i.e., problems not related to corrective maintenance [1]. As part of our future work, we plan to apply the approach proposed by Antoniol et al. to filter "non bugs" out.

Threats to statistical conclusion validity: These concern the relationship between the treatment and the outcome. In our experiments all the Spearman correlation coefficients and all the regression models were significant at the 99% level.

Threats to external validity: These concern the generalization of the findings. In our approach there are three threats belonging to this category: First we have analyzed only three software systems and, second, they are all open-source. This is a threat because of the differences between open-source and industrial development. The last threat concerns the language: The considered software systems are all developed in Java. To generalize more the results, as part of our future work we plan to apply our bug prediction approach to industrial systems as well as systems written in other object-oriented languages such as C++ and Smalltalk.

VII. Related Work

To our knowledge, ours is the first study on the relationship between change coupling and software defects. However, change coupling has been intensively studied in the literature and a number of approaches for bug prediction were proposed.

A. Change Coupling

The concept was first introduced by Ball and Eick [2]. They used this information to visualize a graph of co-changed classes and detect clusters of classes that often changed together during the evolution of the system. The authors discovered that classes belonging to the same cluster were semantically related.

A number of approches exploited fine grained change coupling information. Gall et al. detected change couplings at the class level [11] and validated it on 28 releases of an industrial software system. The authors showed that architectural weaknesses, such as poorly designed interfaces and inheritance hierarchies, could be detected based on change coupling information. Ying et al. proposed an approach that applies data mining techniques to recommend potentially relevant source code to a developer performing a modification task [25]. The authors showed that the approach can reveal valuable dependencies by applying it to the Eclipse and Mozilla open source projects. Zimmermann et al. proposed a technique which predicts entities (classes, methods, fields etc.) that are likely to be modified when another is being modified [28]. Breu and Zimmermann [5] applied data mining techniques on co-changed entities to identify and rank crosscutting concerns in software systems.

Several approaches abstract the change couplings to the level of modules or (sub)system. Gall et al. analyzed the dependencies between modules of a large telecommunications system and showed that the change coupling information helps to derive useful insights on the system architecture [10]. Pinzger et al. proposed a visualization in which they represent modules as Kiviat diagrams and change coupling between modules as edges connecting the Kiviat diagrams [22], showing that the visualization facilitates the detection of potential refactoring candidates.

Other visualization approaches, as the seminal work of Ball and Eick [2], use an energy-based layout to cluster groups of files which have been frequently changed together. The Evolution Storyboards [4], by Beyer and Hassan, is a sequence of animated panels that shows the files composing a CVS repository, where the distance of two files is computed according to their change coupling. The visualization allows the user to easily spot clusters of related files.

B. Bug Prediction

In [27] Zimmermann et al. used object-oriented metrics and past defects to predict future defects. The technique performed well, producing a Spearman correlation of 0.907 at the file level. Nagappan et al. proposed a technique which uses historical data to select appropriate metrics and build regression models to predict post-release defects [20]. They applied the approach on 5 Microsoft software systems and concluded that (1) complexity metrics should not be used for prediction without previously validating them on the project (exploiting the historical data) and (2) metrics which were validated from history should be used to identify low-quality components. Nagappan and Ball used particular types of historical data, the code churn metrics, to predict defect density [19]. They proved that source files with high activity rate are more likely to generate bugs than files with low activity rate. Moreover, they found out that relative measures are better predictors than absolute measures of code churn. In [15] Khoshgoftaar et al. classified modules as defect-prone based on the number of past modifications to the source files composing the module. They proved that the number of lines added or removed in the past is a good predictor for future defects at the module level. Ostrand et al. [21] proposed a regression model to predict defect density and location in large industrial software systems. They used historical data, such as bug and modification history from up to 17 releases, together with the code length of the current release to predict the files with the highest defect density in the next release. Graves et al. developed an approach based on statistical models to find the best predictors for modules' future faults [13]. They found out that the best predictor is the sum of contributions to a module in its history.

Several techniques use information extracted from the source code for defect prediction. One of the first approaches to prove that object-oriented metrics correlate with defects was proposed by Basili et al. [3]. In [23] Subramanyam et al. provided empirical evidence, through eight industrial case studies, that object-oriented metrics are significantly associated with defects. Nagappan et al. computed the static analysis defect density and used it as a predictor for pre-release defect density [18], obtaining Spearman correlations above 0.5. In [26] Zimmermann et al. provided empirical evidence that network measures of the software dependency graph correlate with number of defects, and can be used to enrich regression models based on standard software metrics.

VIII. Conclusion

Change coupling has long been considered a significant issue. However, no empirical study of its correlation with actual software defects had been done until now. We performed such a study on three large software systems and found that there was indeed a correlation between change coupling and defects which is higher than the one observed with complexity metrics. Further, defects with a high severity seem to exhibit a correlation with change coupling which, in some instances, is higher than the change rate of the components. We also enriched bug prediction models based on complexity metrics with change coupling information,

and the results –in terms of explanative and predictive power– corroborate our previous findings.

In the future, we plan to replicate our experiments on a larger number of systems and improve the quality of the dataset by filtering non-bugs out as proposed by Antoniol et al. [1].

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063).

REFERENCES

[1] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of CASCON 2008*, pages 304–318. ACM, 2008.

[2] T. Ball, J.-M. K. Adam, A. P. Harvey, and P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

[3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[4] D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *Proc. 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 199–210. IEEE CS Press, 2006.

[5] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230. IEEE Computer Society, 2006.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[8] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Software Eng.*, 34(4):497–515, 2008.

[9] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 23–32. IEEE CS Press, 2003.

[10] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198. IEEE Computer Society Press, 1998.

[11] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Los Alamitos CA, 2003. IEEE Computer Society Press.

[12] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49. IEEE Computer Society, Sept. 2004.

[13] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

[14] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[15] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of ISSRE 1996*, page 364. IEEE CS Press, 1996.

[16] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[17] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution)*, pages 83–86, 2001.

[18] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of ICSE 2005 (27th International Conference on Software Engineering)*, pages 580–586. ACM, 2005.

[19] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of ICSE 2005 (27th International Conference on Software Engineering)*, pages 284–292. ACM, 2005.

[20] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of ICSE 2006 (28th International Conference on Software Engineering*, pages 452–461. ACM, May 2006.

[21] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[22] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[23] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[24] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[25] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):573–586, 2004.

[26] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, May 2008.

[27] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of ICSEW 2007 (29th International Conference on Software Engineering Workshops)*, page 76. IEEE Computer Society, 2007.

[28] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.