Co-evolution of Logical Couplings and Commits for Defect Estimation

Maximilian Steff, Barbara Russo
Faculty of Computer Science
Free University of Bozen-Bolzano
Bozen, Italy
{*maximilian.steff,barbara.russo*}*@unibz.it*

***Abstract*—Logical couplings between files in the commit history of a software repository are instances of files being changed together. The evolution of couplings over commits' history has been used for the localization and prediction of software defects in software reliability. Couplings have been represented in class graphs and change histories on the class-level have been used to identify defective modules. Our new approach inverts this perspective and constructs graphs of ordered commits coupled by common changed classes. These graphs, thus, represent the co-evolution of commits, structured by the change patterns among classes. We believe that co-evolutionary graphs are a promising new instrument for detecting defective software structures. As a first result, we have been able to correlate the history of logical couplings to the history of defects for every commit in the graph and to identify sub-structures of bug-fixing commits over sub-structures of normal commits.**

***Keywords*-change coupling; defects; commit graphs; commit history**

I. Introduction

A logical coupling between files exists if those files change together over the course of development [1]. Logical couplings are typically derived from the commit history for individual files and are sometimes aggregated to modules of the system. A logical coupling between two files is an instance of those two files being changed together in the same commit. It can be complemented by an actual coupling, i.e. a syntactic relationship between the corresponding source code entities. If there is no syntactic relationship, a logical coupling might indicate possible or even necessary refactorings [2]. Logical couplings can also be used to provide additional information to developers pointing to non-obvious changes necessitated by what they have already or are intending to do [3], [4]. The presence of logical couplings arguably adds extra complexity to a system. In this vain, logical couplings can also be harnessed to estimate and predict defects in a system [5].

Typically, there is no logical coupling between all pairs of files in a system. Modularization and object-oriented programming decouple large parts of every system both logically and syntactically. That does not preclude the set of logical couplings for a file to change over time. Logical couplings among files confer a structure on the commits in which a set of files was changed. Hence, we gain a comprehensive, evolutionary perspective of the development of a system by examining the co-evolution of commits and files.

We create a graph from the commit history that represents the structure among commits as conveyed through logical couplings among the files in those commits. Figure 1 illustrates an example for a commit graph. Each node is a commit from the repository, the lower case letters in the nodes indicate the files changed in that commit. Edges are labeled with file names as well, denoting that two nodes are connected by a file that has been changed in both commits. Assuming that commit 17 is the last commit in this repository, the example is complete from the bottom. We use bug-fixing commits in the commit history as an approximation of the number of defects related to the logical couplings of the history. In this setting, we investigate two hypotheses:

1) The history of a commit as conveyed by its files is correlated to defects.
2) Topological properties of the commit graph are correlated to fault-proneness of files.

In the remainder of this paper, we will first introduce our dataset and how we create a commit graph from it. Then, we will evaluate our hypotheses, present our findings and their limitations, and conclude with a discussion.

II. Data

We have used a dataset from the Subversion repository of the Spring project[1]. The Spring Framework is the premiere Java application framework. We consider the entire history of the repository used for the development of version 3 of the system, spanning the time from July 2008 up to commit 5116 in late-October 2011. We only consider commits to the *trunk*, comprising 2,960 commits for that timespan. Within those commits, we only consider Java source files and exclude all tests. Within this setting, the system grows from initially about 2,000 to 2,800 classes and from 7,000 to 13,000 syntactic dependencies between classes. In the following, we will use classes and files interchangeably since the system is written in Java.

The Spring development team maintains a JIRA issue tracker[2]. We retrieved all defects labeled as affecting any

[1]http://www.springsource.org
[2]http://jira.springsource.org

978-1-4673-1761-0/12/$31.00 © 2012 IEEE

MSR 2012, Zurich, Switzerland

commit of Spring version 3 and a status of 'fixed' for a total of 1,060 defects. We mapped these defects to commits in our dataset using regular expressions matching the occurrence of unique defect identifiers from the tracker to commit messages. In this manner, we identified 432 commits as bug-fixing commits, out of which 147 change more than one file. We call the latter 'non-isolated' commit since the fault might not be contained within a single file. In this setting, the number of bug-fixing commits is a good approximation for the number of defects included in the commit history of a given commit. In what follows, any correlation with defects we perform will always assume this fact.

III. Commit Graphs

To create the commit graph, we collect all files in each commit and for each file the commit it was changed in last. Each commit now represents a node in the commit graph. An edge exists between two nodes if both commits changed one or more identical files and no other commit changed any of these files in between. In other words, a commit is a tuple (t,$\mathcal{F}$) where t is a timestamp and $\mathcal{F}$ a set of files. A commit graph CG is a directed graph that consists of

- a set of commits $\{$(t,$\mathcal{F}$)$\}$,
- a set of links between commits, $\{\rightarrow_{i,j}\}$, such that $(t_1, \mathcal{F}_1) \rightarrow_{1,2} (t_2, \mathcal{F}_2)$ iff $t_1 < t_2, \mathcal{F}_1 \cap \mathcal{F}_2 \neq \emptyset$, and $\forall (t, \mathcal{F})$ with $t_1 < t < t_2 : \mathcal{F}_1 \cap \mathcal{F}_2 \cap \mathcal{F} = \emptyset$.

In our case, we have a directed graph with 2,960 nodes. The graph has one giant connected component comprising 2,948 nodes and 12 isolates having neither in- nor outgoing edges. They occur, for example, if a single file is added in a commit and never modified afterwards. The 2,948 nodes are connected by 5,828 edges, meaning the graph is rather sparse. Also, the average node degree - the total of all in- and outward links of a node - is slightly below 2, indicating that the average node has one in- and one out-going edge. The sparsity of the graph may not be accidental. Typical design principles, like modularity and decomposition, might influence how files change together and the commit graph density might not go beyond a certain threshold.

In the commit graph there are two special kinds of nodes, root and end nodes. A root node has no in-degree meaning none of its files has been ever changed before, i.e. they have been added in a root commit. Similarly, an end node has no out-degree meaning none of its files has been changed later on. In our case, we have 86 root and 393 end nodes. Isolated commits are, of course, both root and end nodes. These commits are not relevant to our analysis as in the case of our dataset they are not bug-fixing commits.

IV. Results

A. Commit Histories

As the history of a commit, H(t,$\mathcal{F}$), comprises information about typical indicators of defective files such as frequently changed files [6] or logical couplings [5], we believe that

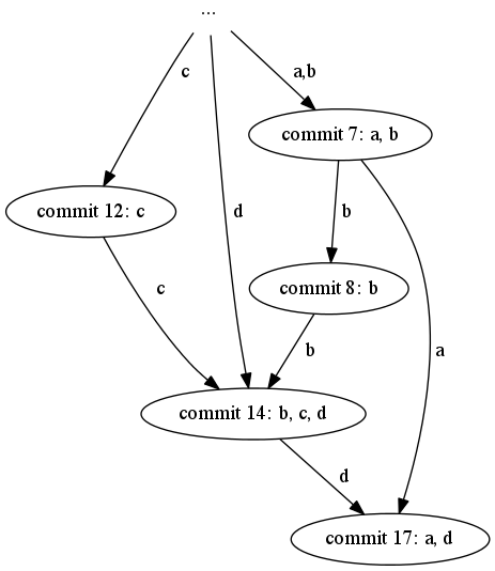

Figure 1. Commit history example.

measures based on H(t,$\mathcal{F}$) will be at least a good indicator of the number of defects. As such, our first hypothesis is that H(t,$\mathcal{F}$) is correlated to the number of defects in this history. To prove this we first need to define the history of a commit (t,$\mathcal{F}$).

To determine H(t,$\mathcal{F}$) we first invert all the edges of CG and calculate the shortest paths from (t,$\mathcal{F}$) to all its reachable roots. For a commit (t,$\mathcal{F}$), we define its history as

$$H = H(t, \mathcal{F}) = \{(t_i, \mathcal{F}_i) | t_i < t, \mathcal{F}_i \cap \mathcal{F} \neq \emptyset\}.$$

Then we define G_H as the set of files changed in all the commits of H. We set

$$H^{Bugs} = \{(t, \mathcal{F}) \in H | (t, \mathcal{F}) \text{ is a bug-fixing commit}\}$$

and G_H^{Bugs} the set of files changed in H^{Bugs}. For example, the history of commit 8 in Figure 1 comprises of commit 7 and all predecessors of commit 7. The files we collect are a and b, and all files in predecessors of commit 7. In total, we get 89,610 shortest paths with an average length of 8.2 original and root node.

For every commit (t,$\mathcal{F}$), we compute the cardinality of the four sets $H, H^{Bugs}, G_H, G_H^{Bugs}$ to define the values of the variables h, h_b, g, and g_b respectively. Then we correlate h with h_g and g with g_b using Spearman correlation as the data is not normally distributed. Finally, we run the same correlations excluding isolated bug-fixing commits, i.e. those bug-fixing commits where changes affect only a single file. Table I reports all these correlations and shows that they are rather strong supporting our initial assumption. Table I indicates two interesting facts. First, correlations with non-isolated bug-fixing commits are high. Considering also

TABLE I
CORRELATIONS BETWEEN THE HISTORY OF EACH COMMIT AND DEFECTS IN THIS HISTORY, SIGNIFICANT AT $\alpha = .001$.

History as ...	**Spearman's ρ**
number of files with all defects	.865
number of files with non-isolated defects	.837
number of commits with all defects	.889
number of commits with non-isolated defects	.851

that they are about 34% of the total number of bug-fixing commits, the original graph filtered on non-isolated bug-fixing commits can be a good starting point for identifying topological structures that help understand code defectiveness (see next section). Second, correlations on the number of files suggest that the greater the number of changed files the greater is the number of defects in a commit history. This result might indicate that files not directly coupled can contribute to defect estimation. According to this result, we extend the definition of logical coupling between two classes saying that that two classes are in higher-order logical coupling when they belong to the same history. If they, in addition, belong to the same commit they are in fact in a standard (first-order) logical coupling. For example, in Figure 1, there is a logical coupling between files *a* and *d* via commit 17. Since file *d* is logically coupled with files *b* and *c* in commit 14, we have a connection between *a* and *c* mediated by *d*. *a* and *c* are then in second-order coupling. Under these premises, commit histories and higher-order coupling can reveal potential code dependencies that are not evident with other approaches. In particular, considering the whole history of files changed has further advantages over traditional code churn and history [6] or logical coupling [5]. In the former case, the focus is on the single file and eventually its history. Considering, as we did, all the files in the history of a commit extends the information on changes to file dependencies that typically reveal potential bad software design [2]. In the latter case, the focus is on pairs of files that change together. In this case, we extend the concept of logical coupling to higher order coupling using the commit history. This supports the idea that not simply a pair of classes changed together but rather sets of classes changing together - but not necessarily with each other - are responsible for software issues.

B. Topological Properties

Our second hypothesis is that topological properties of nodes in the graph are correlated to fault-proneness in files. As we mentioned, in this exploratory analysis, we focus on non-isolated bug-fixing commits. A further reason for this is that the more files a commit contains the longer history it has and, in turn, the more files the history contains. It is also reasonable to assume that bug-fixing commits affecting multiple files are generally indicative of a higher difficulty for fixing the corresponding defect or higher-risk

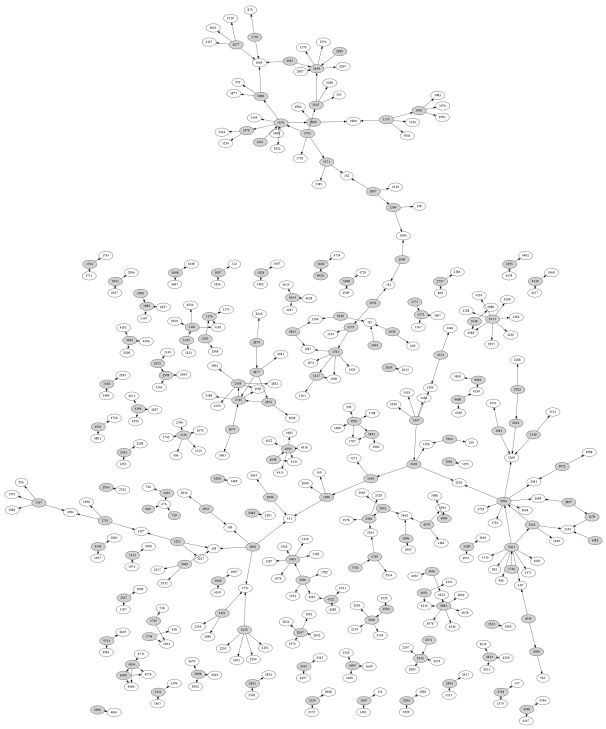

Figure 2. Non-isolated bug-fixing commits and their preceding commits.

files. Therefore, investigating topological properties of non-isolated bug-fixing commits can shed some light on file defectiveness. To explore the topological structure of the graph we use two types of measures: bug-fixing commit proximity and density, and fan-in and -out values.

Figure 2 shows the subgraph for all non-isolated bug-fixing commits and their respective preceding commits. The graph has a total of 414 nodes (with 147 non-isolated bug-fixing commits) and 362 edges spread across 62 connected components. Only a small minority of non-isolated bug-fixing commits has a single predecessor. Roughly half of the components contains at least two non-isolated bug-fixing commits. This means that roughly half of all non-isolated bug-fixing commits occur in proximity of another non-isolated bug-fixing commit, i.e. at most two steps removed in the graph. To understand further how non-isolated bug-fixing commits are embedded in the overall graph, we calculated all shortest paths between the graph's root and end nodes. This amounts to 10,965 shortest paths. Then we exclude all the non-isolated bug-fixing commits and we find only 8,887 shortest paths, 19% fewer paths for about 5% fewer nodes. Thus, non-isolated bug-fixing commits are apparently well-connected and relatively dense in the whole graph.

Table II illustrates the number of commits with different degree values (fan-in, fan-out, and both) together with the number (and percentage) of non-isolated bug-fixing commits. Commits are classified against the degree average we

TABLE II
TOTAL NUMBER OF COMMITS AND NON-ISOLATED BUG-FIXING COMMITS FOR DIFFERENT NODE DEGREES.

	#nodes	#defects	%defects
out-degree > 1	1,134	124	11%
out-degree ≤ 1	1,814	23	1%
in-degree > 1	1,044	91	9%
in-degree ≤ 1	1,904	56	3%
degree > 2	1,294	129	10%
degree ≤ 2	1,654	18	1%

have found in the graph (i.e. greater or lower than 2). We can see that the percentage of non-isolated bug-fixing commits with degree (in, out, or both) greater than the average is significantly higher than in the complementary sets with lower degrees. This indicates that bug-fixing commits that change multiple files tend to be more dependent on preceding commits. If we further explore the predecessors of non isolated bug-fixing commits with degree > 2, we get 267 additional nodes out of which 179 have a degree > 2 as well. This implies that a total of 308 nodes with degree > 2 are immediately related to bug-fixing commits that change multiple files, i.e. 24% of all the commits with degree > 2. The complementary set for non-isolated bug-fixing commits and their predecessors with lower degree is below 9%. This may indicate that higher commit dependency and higher logical coupling can be a promising instrument to identify high-risk files or difficult fixes. It is also worth mentioning that, by definition, the isolated bug-fixing commits have degrees less or equal to 2. As such, their incidence is limited (Table II) and their topological characteristics are not detected with our present method.

V. FINDINGS AND LIMITATIONS

We first found that both number of commits and number of files occurring in a commit's history are highly correlated to defects. This was not completely obvious as typically frequently changed or first-order logical couplings are considered indicators of fault-proneness [5], [6]. Secondly, we found that bug-fixing commits are well-connected among themselves and distributed across the whole graph. In this sense, we found that defects are much more likely to occur in nodes with an above-average degree. Finally, we characterize the bug-fixing commits that change multiple files (e.g. defects with high intrinsic complexity) as commits that exhibit a higher dependency with two-step preceding commits. This implies that higher order logical coupling can be a promising instrument to identify risky files.

In terms of limitations, our method of reconstructing commits' histories might not cover the whole graph and thus miss some information. For example, our method does not characterize the topology of isolated bug-fixing commits or does not consider weights in the graph, e.g. the age and distance of a node to the original node or the order of the logical coupling between nodes in histories. Furthermore, we have to evaluate our method on different datasets. Our findings are preliminary and derived from a single data set. This is a typical bias of empirical analysis that can be controlled by replicating the study for different software products. Similarly, we are only considering commits into the *trunk*, neglecting commits in the *branches*. This might create a partial view of the actual defects in the product but it is justified by the exploratory nature of our analysis.

VI. DISCUSSION AND FUTURE WORK

We present a novel method to investigate logical couplings and their relation with defect occurrences. In our method, we map commits to nodes in a directed graph and link them if they occur subsequently and share at least one file. We define size of a node's history and its degree as measures on the commit graph. We use these measures and the topological structure of the graph for defect estimation and localization. We show that the co-evolution of commits and files provides valuable information on the fault-proneness of a system.

The co-evolution of commits and files warrants further attention. Higher-order couplings and topological properties of histories in the graph have the potential to reveal defective code structures in those files. In particular, they can be used to focus on recent commits in ongoing development and identify possible defects.

There are ways to further improve our current method. These include considering nodes' content, e.g. overall churn in that commit, and age when collecting a node's history. The search itself could be improved by employing a different search algorithm, e.g. breadth-first search. Also, we should distinguish different kinds of nodes, e.g. refactorings or initial commits in our case.

REFERENCES

[1] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *ICSM'98*, Nov 1998, pp. 190 –198.

[2] J. Ratzinger, M. Fischer, and H. Gall, "Improving evolvability through refactoring," in *MSR'05*, May 2005, pp. 1–5.

[3] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Soft. Eng.*, vol. 31, no. 6, pp. 429 – 445, Jun 2005.

[4] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Soft. Eng.*, vol. 30, pp. 574–586, Sep 2004.

[5] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *WCRE '09*, Oct 2009, pp. 135 –144.

[6] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Soft. Eng.*, vol. 26, no. 7, pp. 653 –661, Jul 2000.