Improving the Detection Accuracy of Evolutionary Coupling by Measuring Change Correspondence

Manishankar Mondal Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{mshankar.mondal, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—If two or more program entities change together (i.e., co-change) frequently (i.e., in many commits) during software evolution, it is likely that the entities are related and we say that the entities are showing evolutionary coupling. Association rules have been used to express evolutionary coupling and two related measures, support and confidence, have been used to measure the strength of coupling among the co-changed entities. However, an association rule relies only on the number of times the entities have co-changed. It does not analyze whether the changes are corresponding and whether the entities are really related. As a result, association rule often reports false positives and also, ignores important coupling among the infrequently co-changed entities. Focusing on this issue we propose to calculate a new measure, *change correspondence*, blending the idea of concept location in a code-base to determine whether the changes to the co-changed entities are corresponding and thus, whether they are really related. Our preliminary investigation result on four subject systems written in two programming languages shows that *change correspondence* has the potential to accurately determine whether two entities are related even if they co-changed infrequently. Thus, we believe that our new measure will help us improve the detection accuracy of evolutionary coupling.**

***Keywords*-Association Rule; Evolutionary Coupling; Method Co-change; Concept Location;**

I. Introduction

Detection and analysis of evolutionary coupling is a promising area in the realm of software engineering research. The central concept is that if a group (two or more) of program entities change together (i.e., co-change) frequently (i.e., in many commits) during software evolution, then it is likely that the entities (in the group) are related to one another. We say that the entities exhibit evolutionary coupling. Detection of evolutionary coupling is important from the perspective of software maintenance. While changing a particular entity, E, during maintenance if we already know that E is likely to be related with some other entities because E has shown evolutionary coupling with them, then we can analyze whether the changes in E will have negative effects to the other entities and whether we need to propagate changes to these entities.

Evolutionary coupling [5] has been investigated a lot by the existing studies resulting in a number of tools and techniques that can help us identify the frequently co-changed entity sets. Association rules [1] have been used to express evolutionary coupling. If two or more entities have ever co-changed, we can assume association rule(s) for them. Two related measures - *support* and *confidence* have been widely used to determine the strength of relationship of the entities in an association rule. Higher values of these measures indicate higher likeliness of an underlying relationship among the associated entities. However, an association rule only emphasizes the number of times the entities have co-changed. It does not analyze whether the changes to the two or more co-changed entities are really corresponding (i.e., changes in one entity required corresponding changes to the other entities to ensure consistency) and thus, whether entities are really related. As a result, association rules sometimes report false positives and often ignore important coupling among the infrequently co-changed entities because of unpromising support.

Focusing on the above issue we introduce a new measure, *change correspondence*, to quantify the identifier correspondence and constant correspondence between the changed portions (i.e., statements) of the co-changed entities. The basic idea of measuring change correspondence originates from another promising research area centered on concept location in (i.e., information retrieval from) a code-base for a given query on the basis of semantic similarity between the query words and the identifiers and/or comments of different program entities (such as methods, classes) in the code-base. We plan to quantify the semantic similarity (including exact similarity) among the identifiers involved in the changed portions of the co-changed entities. If it is observed that two or more entities have co-changed and also, the identifiers involved in the changed portions of these entities have semantic similarity, then the changes to the entities are likely to be corresponding and we can assume a better likeliness of an existing relationship among these co-changed entities.

According to our early investigation on four subject systems written in two programming languages (C and Java), *change correspondence* has the potential to improve the detection accuracy of evolutionary coupling. We observe that many rules have the lowest support (support = 1) but promising *change correspondence* and their constituent methods are related. We believe that our proposed measure will complement the existing measures to better detect evolutionary coupling.

The rest of the paper is organized as follows. Section II describes association rule with the existing measures, Section III motivates the necessity of *change correspondence*, Section IV defines and discusses *change correspondence*, Section V demonstrates the experimental steps, Section VI presents our preliminary experimental results, Section VII discusses related work, and Section VIII concludes the paper.

II. Association Rule

Association rules have been widely used to represent and investigate evolutionary coupling among program entities. We define association rule and two existing related measures - support and confidence in the following way.

Association Rule. An association rule [1] is an expression of the form X => Y where X is the antecedent and Y is the consequent. Each of X and Y is a set of one or more program entities. The meaning of such a rule in our context is that if X gets changed in a particular commit operation, Y also has the tendency of getting changed in that commit operation. We can determine the *confidence* of a particular association rule by determining the *support* of its constituent parts.

Support and Confidence. Support is the number of commit operations in which an entity or a group (two or more) of entities changed together. We consider an example of two entities E1 and E2. If E1 and E2 have ever changed together, we can assume two association rules, $E1$ => $E2$ and $E2$ => $E1$, from them. Suppose, E1 changed in four commits: 2, 5, 6, and 10. E2 changed in six commits: 4, 6, 7, 8, 10, and 13. Thus, *support(E1) = 4* and *support(E2) = 6*. However, *support(E1, E2) = 2*, because E1 and E2 co-changed in two commits: 6, and 10. Also, *support(E1 => E2) = support(E2 => E1) = support(E1, E2) = 2*.

Confidence of an association rule, X => Y, determines the probability that Y will change in a commit operation provided that X changed in that commit operation. We determine the confidence of X => Y in the following way.

$$confidence(X \Rightarrow Y) = support(X,Y)/support(X) \quad (1)$$

Implications of Support and Confidence. While a higher support value indicates a higher likeliness of an existing relationship among the co-changed entities, confidence determines which entity has the higher probability of triggering changes in the other entity. Let us consider two rules m_1 => m_2 and m_2 => m_1 consisting of the methods m_1 and m_2. If m_1 => m_2 has a higher confidence than m_2 => m_1, then there is a higher probability that a change in m_1 will require a corresponding change in m_2, and also, there is a comparatively lower probability that a change in m_2 will require a corresponding change in m_1. From this we understand that confidence is important only when the constituent entities are related. It might be the case that m_1 and m_2 co-changed several times but they are not related. In this case, we do not need to form any rule using m_1 and m_2.

In this research our goal is to determine whether the co-changed entities are related. Thus, we do not focus on confidence. We consider the support measure to be an indicator of an existing relationship among the co-changed entities. In the rest of this paper, we present and investigate our proposed measure *change correspondence* in comparison with *support* to determine whether *change correspondence* has the potential to better identify if the co-changed entities are related.

III. Motivation behind Change Correspondence

The existing measures regarding association rules only rely on the number of times a group of entities co-change. It is widely accepted that if a group of entities has a higher support (i.e., the entities co-changed in many commits), there is a higher likelihood that the entities are related. However, if we only rely on the support and confidence measures, we might often be confronted with the following two issues.

Possibility of false negatives. In general, if a group of entities has a lower support value, it might seem that the entities have a lower probability of being related and thus, the entity set is not a promising one. However, a lower support value might have the following implications too.

(1) The entities were recently created (i.e., the entities are newer compared to the other sets of entities with higher support). However, they are related and might co-change frequently during future evolution.

(2) The entities were created a while ago and also, they are related. However, the requirements corresponding to these two entities are stable and thus, they have not changed or co-changed frequently.

Thus, the infrequently co-changed (for example, co-changed only once and thus, support = 1) entities can also be related. Support measure does not consider the above two possibilities. By manually analyzing the association rules of our subject systems we found many rules with support = 1 but the constituent entities are related. So, if we rely only on higher support values, we might ignore important coupling relationships among the infrequently co-changed entity sets.

Possibility of false positives. It is also possible that a group of entities co-changed in a number of commits but they are not related. Such a situation is likely to occur if there are atypical commits during system evolution. According to the literature [7], in atypical commits unrelated entities change together. Atypical commits can occur because of major structural changes to the software systems.

The possibilities of false positives and false negatives mentioned above could be minimized if we could have a mechanism of automatically inspecting the changes to the co-changed entities and determining whether the changes are related (i.e., corresponding). From this perspective, we introduce a new measure *Change Correspondence* with an expectation of automatically determining whether the changes to the co-changed entities are related. To the best of our knowledge, our approach is the first one considering automatic code inspection to improve the detection accuracy of evolutionary coupling.

IV. Change Correspondence

For one change in a particular method, there can be corresponding changes to several other methods in the same commit operation. However, which type of change in one method will require which other types of changes in other methods has not yet been studied properly. According to our observation, a very small change in one method can sometimes require a comparatively larger change in another method. Changes (in a codebase) in most cases are dependent on the objective (derived from a client request) and context. Automatic identification of change correspondence still remains a challenge. However,

we plan to blend the idea of concept location with association rule mining technique in order to address this challenge.

A. Identifier correspondence

Identifiers are the names given to different program entities such as variables, methods, classes, packages, interfaces, structures, unions, etc. Our idea of incorporating identifier correspondence measure is described below.

The underlying idea. Concept location in a code-base [2], [9] using semantically similar words is a promising research area which has been explored much resulting in a great many studies, techniques and tools. The basic idea is that for a given query (consisting of one or more words) we identify program entities (e.g., methods) containing the same or semantically similar words (in terms of identifiers and comments). The program entities obtained as the query result are likely to implement the underlying concept of the query and thus, are expected to be related. Our plan of measuring identifier correspondence is based on this central concept which we can incorporate in our context in the following way.

We consider two methods m_1 and m_2 which have co-changed in a particular commit. The list of identifiers involved in the added, deleted or modified lines of m_1 is $list_1$. Similarly, the list regarding m_2 is $list_2$. We determine whether some of the identifiers in $list_1$ are the same or semantically similar to some of the identifiers in $list_2$. If such similarity is observed, we can assume a better likeliness of an existing relationship among the co-changed methods m_1 and m_2. Thus, we believe that concept location techniques can be used to improve the detection accuracy of evolutionary coupling.

Calculation of Identifier Correspondence. Let us consider an association rule consisting of two methods m_1 and m_2 that co-changed in a number of commits during the whole period of evolution of a software system. Suppose, C is the set of these commits. For each commit, we determine the identifier correspondence between the modified (i.e., added, deleted, or changed) lines (i.e., statements) of these two methods. Let us consider a particular commit, $c \epsilon C$. We determine the list of identifiers involved in the modified lines of m_1 in this commit (c). We denote this list by $L_{m_1,c}$. The corresponding list considering m_2 is $L_{m_2,c}$. We calculate the identifier correspondence of m_1 and m_2 for commit c using Eq. 2.

$$IDCorr_{m_1,m_2,c} = \left(\frac{|LS_{m_1,m_2,c}|}{|L_{m_1,c}|} + \frac{|LS_{m_2,m_1,c}|}{|L_{m_2,c}|}\right) \times \frac{1}{2} \quad (2)$$

Here, $IDCorr_{m_1 m_2 c}$ is the identifier correspondence between the modified lines of m_1 and m_2 considering commit c. $LS_{m_1,m_2,c}$ is the list of identifiers from $L_{m_1,c}$ that are semantically similar to some identifiers in $L_{m2,c}$. In the same way, $LS_{m_2,m_1,c}$ is the corresponding list of identifiers from $L_{m_2,c}$. Finally, we determine the overall identifier correspondence considering each of the commits in the set C (i.e., considering the whole period of evolution) in the following way.

$$OverallIDCorr_{m_1,m_2} = \frac{\sum_{c \epsilon C} IDCorr_{m_1,m_2,c}}{|C|} \quad (3)$$

Justification of the Equations. Observing Eq. 2 and Eq. 3, we can easily determine that the highest value of overall identifier correspondence regarding two co-changed methods is 1 and the lowest possible value is 0. Eq. 2 demonstrates that for a particular commit c, the identifier correspondence of two co-changed methods m_1 and m_2 can be 1 if for each of the identifiers in one list ($L_{m_1,c}$ or $L_{m_2,c}$) we get a semantically similar (also includes exact similarity) identifier in the other list. In such a situation, we can have a higher expectation that the changes to the two methods are corresponding and also, the methods are related. However, if we cannot infer any semantic similarity at all, then it is very unlikely that the changes are corresponding and thus, the methods are not likely to be related. Identifier correspondence is zero in this situation. Finally, we believe that *our equations for calculating identifier correspondence are reasonable and these equations essentially reflect our intention.*

In this preliminary investigation, we consider only the common identifiers involved in the modified lines of both methods. In this case, $LS_{m_1,m_2,c} = LS_{m_2,m_1,c} = L_{m_1,c} \cap L_{m_2,c}$.

B. Constant Correspondence

We also plan to determine *Constant Correspondence* by measuring the extent the modified lines of the co-changed methods involve the same constant(s). However, we did not consider constant correspondence in this preliminary study.

V. Experimental Steps

In this section we describe the experimental steps involved in extraction of method association rules from software evolution history and calculation of identifier correspondence for each rule. Before extracting association rules we perform some preliminary steps discussed briefly in the following paragraph.

For a particular subject system, we at first extract all of its revisions (as mentioned in Table I) from an open-source SVN repository. Then we detect and store methods from each of the revisions applying CTAGS. We detect method genealogies considering all the revisions following a procedure proposed by Lozano and Wermelinger [7]. Genealogy of a particular method helps us to inspect the evolution of the method. After genealogy detection, we detect changes between every two consecutive revisions and map these changes to the already detected methods. As we have already detected method genealogies, after mapping the changes to the methods we can easily determine how a particular method changed during software evolution. For the details of these preliminary steps we refer the interested readers to our earlier work [8].

Extraction of Method Association Rules. After completing the preliminary steps, we extract method association rules considering every possible method pair.

Method pair. *If two methods m_1 and m_2 co-changed (i.e., changed together) in at least one commit during evolution, we form a pair (m_1, m_2) consisting of these two methods.*

According to the above definition, if n methods co-changed in a particular commit operation, we construct $\binom{n}{2}$ method pairs (i.e., every possible pair) from these. If $n = 4$, count of method pairs = 6. From a method pair (m_1, m_2) we determine two association rules, $m_1 \Rightarrow m_2$ and $m_2 \Rightarrow m_1$. The supports

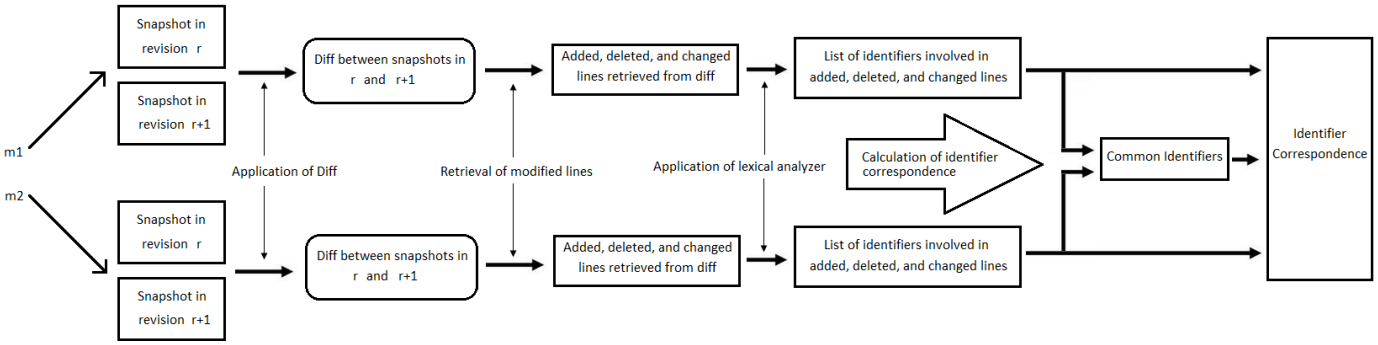

Fig. 1. Calculation of identifier correspondence for method m_1 and m_2 for the commit operation applied on revision r

of these two association rules are the same, however, their confidences can be different. We determine all method pairs as well as method association rules from each of our candidate subject systems. We determine identifier correspondence for a particular method pair in the following way.

Calculation of Identifier Correspondence. We consider a method pair (m_1, m_2). We at first determine the list of commits where they co-changed. We consider a particular commit c in this list. Suppose, the commit c was applied on revision r. Fig. 1 shows the sequential steps for calculating identifier correspondence of method pair (m_1, m_2) for the commit applied on revision r.

As shown in the figure (cf., Fig. 1), for each of the methods (m_1 and m_2), we collect two snapshots - (1) the snapshot in revision r and (2) the snapshot in revision $r+1$. Revision $r+1$ was created because of the commit c on revision r. Suppose, the snapshots of m_1 and m_2 in revision r are $S_{m_1,r}$ and $S_{m_2,r}$ respectively. In the same way, the snapshots of these methods in revision $r+1$ are $S_{m_1,r+1}$ and $S_{m_2,r+1}$. For each of the methods, we determine the *diff* of the corresponding snapshots as indicated in Fig. 1. *diff* gives us output in terms of addition (addition of lines in the new snapshot), deletion (deletion of lines from the previous snapshot), and change (change of one or more lines in the previous snapshot and the corresponding changed lines in the new snapshot). Analyzing the *diff*-output regarding a particular method (m_1 or m_2) we extract the followings.

- The lines deleted from the snapshot in revision r
- The lines added to the snapshot in revision $r+1$
- The lines in the snapshot in revision r that were changed
- The changed lines in the snapshot in revision $r+1$

We refer these four types of lines as *modified lines* as a whole. For each method we determine the $modified\ lines$. We apply a lexical analyzer (CCFinderX [1]) on these *modified lines* to determine the identifiers involved in these lines. Suppose the list of unique identifiers (i.e., each identifier is considered only once in the list) regarding the modified lines in m_1 is $list_1$ and the list of unique identifiers regarding m_2 is $list_2$. From these two lists we determine the common identifiers. Then, according to the equation Eq. 2, we determine the identifier correspondence for method pair (m_1, m_2) regarding commit operation c. By determining the identifier correspondence of the methods m_1 and m_2 for each of the commits where they co-changed, we determine the overall identifier correspondence using Eq. 3.

[1]CCFinderX: http://www.ccfinder.net/ccfinderx.html

TABLE I
SUBJECT SYSTEMS

Systems	Lang.	Domains	LOC	Revisions
Ctags	C	Code Def. Generator	33,270	774
QMailAdmin	C	Mail Management	4,054	317
jEdit	Java	Text Editor	1,91,804	4000
Plandora	Java	Project Management	94,076	73

TABLE II
STATISTICS REGARDING THE RULES WITH THE LOWEST SUPPORT

	Ctags	QMail Admin	jEdit	Plandora
No. of Rules (all rules in the system)	3660	1331	229140	12587
% of Rules that are LSR	89%	49.6%	97%	97.2%
% of LSR that have IDCorr > 0	28%	41.3%	13.9%	48%
% of LSR that are Target Rules	21%	9.3%	13%	35%
% of Target Rules that are True Positives	98%	94%	90%	92%

IDCorr = Identifier Correspondence
LSR = Lowest-Support Rules (i.e., The rules with support = 1)

VI. EXPERIMENTAL RESULTS AND DISCUSSION

We conducted our preliminary investigation on four subject systems listed in Table I. We investigated what proportion of the unpromising (because of lower support) rules have promising *identifier correspondence* and their constituent methods are related. In this preliminary study, we consider the rules with the lowest support value (i.e., support = 1) as the unpromising rules because, the constituent methods in such a rule has the lowest likeliness of being related according to the implication of support measure (Section II). From these rules (with support = 1), we automatically identify our *target rules*.

Taget Rule. A *target rule* is a rule with the lowest support (support = 1) but a higher *identifier correspondence*. We consider an *identifier correspondence* of at least 0.5 as a promising one in this preliminary experiment.

We manually analyze 50 *target rules* from each of the candidate systems to determine whether they are *true positives*.

True Positive. We consider a *target rule* as a *true positive*, if according to our manual analysis, the changes to the constituent methods are corresponding and also, the methods are related.

In Table II, we present statistics regarding the lowest-support rules (support = 1), *target rules*, and *true positives*. We

see that for most of the subject systems (except QMailAdmin), almost 90% of the total rules have the lowest support. If we ignore all of these rules because of their unpromising support value, we have the possibility of missing important coupling relationship among the constituent methods of these rules. However, in case of each of the subject systems we see that for a considerable amount of these lowest-support rules, *identifier correspondence* > 0. We also show the percentages of the lowest support rules that are our *target rules*. Promisingly, most of the *target rules* (at least 90%) that we manually analyzed are *true positives* in case of each of the candidate systems. Thus, *according to our investigation, these rules (i.e., true positives) with promising identifier correspondence should not be ignored although they have unpromising support and rather, should be considered as promising ones. We can extract important coupling relationship from the constituent methods of each of these rules.* As an example of a *true positive*, we can mention a rule (with *identifier correspondence = 0.6*) consisting of two methods - (1) *adduser* (file: *user.c*) and (2) *onevalidonly* (file: *alias.c*) from our candidate system QMailAdmin. These methods co-changed in only one commit applied on revision 143. However, the changes to these methods are corresponding and, the methods are related.

During manual analysis we found that in case of a number of rules (with lowest support) only parts of the identifiers are similar. For some rules (in Ctags), some constants involved in the modified lines of the two methods were common. According to our analysis these rules seemed to be promising and the constituent methods seemed to be related. However, as we considered only exact similarity of identifiers and did not consider constant correspondence, these rules had an identifier correspondence of zero. However, according to our analysis in this preliminary investigation we plan to do the followings in order to extend our research work.

(1) Identifier splitting and extracting semantic similarity between identifier-parts involving state of the art tools and techniques being used in concept location in code-base.

(2) Incorporation of constant correspondence.

(3) Rigorous analysis regarding the capability of our proposed measure *change correspondence* in detecting corresponding changes as well as in improving the precision and recall in detecting evolutionary coupling incorporating large subject systems of different programming languages.

(4) Investigation on whether *change correspondence* can also be used in predicting future co-change candidates considering method level granularity.

We believe that if properly implemented, *change correspondence* can help us improve the detection accuracy of evolutionary coupling.

VII. Related Work

Association rules were introduced by Agarwal et al. [1] and have been frequently used to find associated or co-changing program artifacts (also known as frequent itemsets). Gall et al. [5] introduced an approach for discovering logical dependencies analyzing the evolutionary coupling (or co-changing) of different program modules. Zimmermann et al. [11] used *association rules* with *support*, and *confidence* to represent the co-change relationships among different program artifacts. Jafar et al. [6] performed a comprehensive study on macro co-changes considering file level granularity and introduced the patterns, *macro co-changes* and *diphase macro co-changes*, that can help in retrieving file level coupling.

The existing studies related to association rules mainly depended on the support measure to assume the likeliness of an existing relationship among the entities in a rule. None of these studies proposed an automatic code inspection approach to improve the accuracy in determining whether the co-changed entities are in fact related. We propose a new measure *Change Correspondence* that involves automatic inspection of the modified lines of the co-changed entities with an expectation to improve the detection accuracy of evolutionary coupling. Moreover, we conduct our study considering a finer granularity, at the method level, while most of the existing studies were conducted at the file level.

VIII. Conclusion

In this paper, we propose a new measure *Change Correspondence* with an aim to improve the detection accuracy of evolutionary coupling. Calculation of this measure involves automatic inspection of the modified source code lines of the co-changed methods to infer whether the changes are corresponding on the basis of the semantic similarity of the identifiers involved in the modified lines. Our idea of determining identifier similarity originates from the promising research area centered around concept location in a code-base. From our preliminary investigation result on four subject systems written in two programming languages (C, and Java) we believe that our proposed measure *change correspondence*, in association with the existing ones, has the potential to improve the detection accuracy of evolutionary coupling.

References

[1] R. Agrawal, T. Imieliski, A. Swami, "Mining association rules between sets of items in large databases", Proc. *ACM SIGMOD*, 1993, Vol. 22, Issue 2, pp. 207–216.

[2] B. Cleary , C. Exton, "Assisting Concept Location in Software Comprehension", *PPIG*, 2007, pp. 42 – 55.

[3] M. D'Ambros, M. Lanza, M. Lungu, "Visualizing co-change information with the evolution radar", *TSE*, 2009, vol. 35, no. 5, pp. 720–735.

[4] D. Beyer, A. E. Hassan, "Animated visualization of software history using evolution storyboards", Proc. *WCRE*, 2006, pp. 199–210.

[5] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history", Proc. *ICSM*, 1998, pp. 190–199.

[6] F. Jaafar, Y. Gueheneuc, S. Hamel, G. Antoniol, "An Exploratory Study of Macro Co-changes", Proc. *WCRE*, 2011, pp. 32–334.

[7] A. Lozano, M. Wermelinger, "Assessing the effect of clones on changeability", Proc. *ICSM*, 2008, pp. 227–236.

[8] M. Mondal, C. K. Roy, K. A. Schneider, "Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems", Proc. *CASCON*, 2012, pp. 205 – 219.

[9] G. Sridhara, E. Hill, L. Pollock, K. Vijay-Shanker, "Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools", Proc. *ICPC*, 2008, pp. 123 – 132.

[10] A. Vanya, R. Premraj, and H. v. Vliet, "Interactive Exploration of Co-evolving Software Entities", Proc. *CSMR*, 2010, pp. 260 – 263.

[11] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, "Mining version histories to guide software changes", Proc. *ICSE*, 2004, pp. 563–572.