Using Network Properties to Study Congruence of Software Dependencies and Maintenance Activities in Eclipse

Markus Strohmaier[1] Michel Wermelinger[2] Yijun Yu[2]
[1] Knowledge Management Institute, Graz University of Technology, and Know-Center Graz, Austria
[2] Department of Computing and Centre for Research in Computing, The Open University, UK

Abstract

In this paper we present a preliminary investigation on whether a dependency between two software components leads to more people having to be involved with both *components during maintenance. We compute a technically-inferred and a socially-inferred network of components and rank their edges according to their weight. A good edge ranking correlation would indicate that the strongly dependent components also require more people looking at bug reports and feature requests of both components.*

1 Introduction

Software engineering is inherently a socio-technical endeavour and Conway's law [5] is one of the early acknowledgements of the implications of that fact. Conway not only argued that there is a congruence between the social structure of an organisation and the technical structure of the designs it produces, but also that the former causes the latter. These observations have caused researchers to study socio-technical congruence and its implications both from a scientific and a practical perspective.

For example, Nagappan et al. [7] took Conway's law as a motivation to investigate whether organizational metrics can predict the most faulty components, and Cataldo et al. [4] found a higher correlation between Conway's congruence and the productivity of software projects than just considering the modularity of the code. There has also been work on mining social networks of contributors from source code [6, 3], e-mail archives [2] and bug reports [1].

However, the field is not yet (nor could it be) mature, and various approaches to analysing and applying congruence have been put forward. In particular, it is not yet clear which kinds of congruence are useful and how to measure and use them for practical purposes. In this paper, we use a measure that is less strict than structural graph similarity [8], and thereby wish to contribute to the discussion about the nature of congruence and its measurement.

In particular, we are interested in investigating whether the technical structure of component dependencies correlates with the social structure when maintaining those components. The research question is whether the more a component depends on another one, the more people will be involved in maintaining both. This would enable studies related to causality between social and technical systems.

The notion of congruence we are proposing is whether the ranking of the dependency strength between components correlates with the ranking of the number of common contributors. Our approach is as follows. We first take two independently acquired data sources, one related to maintenance (mainly bug reports), the other to software (architectural metadata in the source code). Next we generate two comparable networks with the same type of node (components) but with different edge types: the *socially-inferred* network connects two components if they share contributors to their maintenance, while the *technically-inferred* network connects two components if they have code dependencies. The weight of an edge represents the strength of the relationship. Finally, we rank the edges according to weight and correlate the rankings. The following sections will explain the process in detail.

The idea of taking two similar, yet differently constructed, component networks was used before by Bowman and Holt [3]. However, their basic assumption is different from ours. They assumed that the technical component dependency structure does impact and constrain the social structure of component maintenance and hence used the latter to guide the reverse engineering of the former. Instead, we assume the architecture is correct, because it is explicitly given in metadata, and check whether it correlates with the social maintenance structure.

2 Data collection

We selected the Eclipse SDK for our case study because: it is a non-trivial project, both in social and technological terms; the lead of IBM ensures some social continuity; we had mined it before [10, 11] and hence could reuse some

of our mining infrastructure; and a Bugzilla database is available[1] with a sufficiently long history of social interactions. We restricted our study to the SDK, i.e. the core Eclipse platform that includes e.g. the Java and plugin development environments. Note that the information contained in the Bugzilla database is dominated by bug reports (maintenance), but also contains a small set of feature requests (development). Due to the dominance by bug reports we characterize the information in the Bugzilla dataset as *maintenance-related* and henceforth refer to it as *requests* (whether of bug fixes or new features).

The socially-inferred network Using only the Bugzilla dataset, we extracted for each of the change requests its unique id, the current component related to that request, and the people associated to the request: the request reporter, the current assignee (i.e. the person implementing the change), and the (zero or more) past discussants.

With this information we constructed a network consisting of three types of nodes: people P (22,532 nodes), requests R (101,966 nodes) and software components C (16 nodes). There is an arc from person p to request r, if p reported, worked on, or discussed r. There is an arc from r to c if r was reported for component c. Next we created a two-mode network PC, i.e. with only two types of nodes where an arc from person p to component c will be weighted with the number of requests about c that mention p, i.e. the number of paths from p to c in the original PRC network.

To make a meaningful analysis, it is necessary to avoid 'noise' due to people that had only a very small intervention in the project. We therefore introduced a threshold k: arcs with a weight less than k are removed from the PC network, and so are any nodes that become detached. After applying k, we removed all weights from the remaining edges, as those weights are no longer needed.

Finally, we folded the two-mode network $PC(k)$ into a one-mode Bugzilla-inferred network $C_B(k)$ that contains only components, by calculating $PC(k)^T \times PC(k)$ and removing the diagonal (i.e. self-loops) [9]. Thereby, all pairs of edges $c-p_i-c'$ are folded into a single edge $c-c'$ having as weight the number of pairs of edges that were folded, i.e. the number of people p_i that were attached to *both* components in $PC(k)$. The left of Figure 1 shows $C_B(128)$ but with weights omitted to avoid cluttering the visualization. In other words, two components are connected if they share at least one person who has reported, fixed and/or discussed at least k=128 requests for each of the two components.

The technically-inferred network The Eclipse SDK is implemented as a set of groups of Java classes called *plugins*. In the remaining of the paper, we say that plugin X *statically depends* on plugin Y if the compilation of X requires Y , and we say that X *dynamically depends* on Y if X uses at run-time an extension point that Y provides. Note that the dynamic dependencies are at the architectural level; they do not capture run-time calls between objects.

We extracted both kinds of dependencies from Eclipse's metadata, as reported in [10]. For this study, we only used the dependencies for Eclipse 3.3.1.1, the release at which the Bugzilla database was built.

However, requests are not at the level of plugins but at the level of coarser-grained components. We manually went through the official web pages for the Eclipse SDK projects, collected the lists of plugins belonging to each component, and lifted the static and dynamic dependency relations among plugins to those among components: the weight of an edge between components A and B states how many dependencies exist between plugins of A and those of B. We computed three code-inferred component networks C_C, one taking only static dependencies into account, another one only dynamic dependencies, and the third one with the union of both. The latter is shown on the right of Figure 1, again omitting the weights.

3 Results

It is immediately apparent that the two networks are similar, in spite of the different data sources. However, we are not interested in the direct structural similarity of the networks, but whether a stronger dependency among two components correlates with more people having to deal with both components in maintenance activities. To study congruence between the Bugzilla-inferred component network C_B and the code-inferred component network C_C we calculate a ranking correlation between the most important edges in C_B, C_C and a set of random networks C_R. C_R have the same number of nodes (16) and a number of edges comparable to the C_C networks. We measured edge importance by using edge weight, where the edge with the highest weight ranks the highest. Pairs of nodes that are not connected (i.e. the edge weight is zero) rank lowest. Because the choice of parameter k can be assumed to influence the results, we experimented with a sample of 10 exponentially growing thresholds, from $k = 2^1$ to $k = 2^{10}$.

Having obtained the edge rankings for several networks, we compared them using Spearman ranking correlation ρ, a value in the interval [-1, +1]. A positive correlation between C_B and C_C would mean that the relative importance of edges is similar. Ideal correlation would allow predicting the rank of an edge in C_B when knowing the ranking of the corresponding edge in C_C and vice versa.

Figure 2 shows the results of the correlation analysis between C_B and C_C and between C_B and C_R by presenting the corresponding correlation values ρ. We can observe that

[1] http://msr.uwaterloo.ca/msr2008/challenge

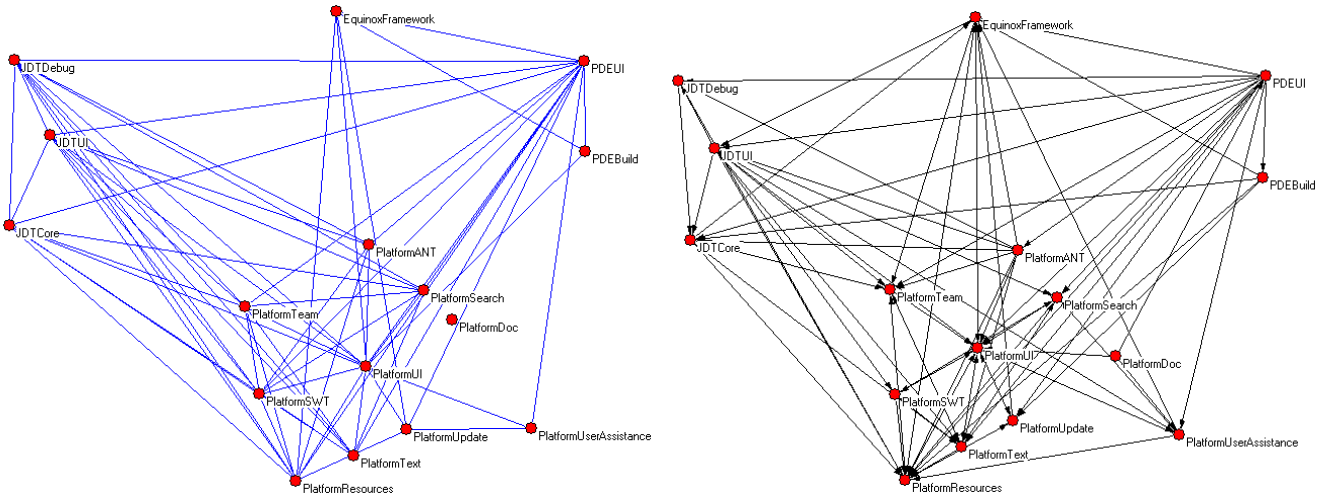

Figure 1. Component networks based on common contributors (C_B, $k = 128$, left) and on code dependencies (C_C, static and dynamic, right)

correlation between C_B and C_C is always higher for static than for dynamic and for combined dependencies.

We also calculated the significance level p, an estimate of the probability that our results have occurred by chance. A p of ≤ 0.01 would mean that there is a $\leq 1\%$ chance that the result is accidental. The lower p is, the higher the significance. All p in Figure 2 have significance $p \leq 0.01$ except for 1) all C_R values, which means that those results are not significant and 2) correlation values for all C_B(k=1024) and correlation between C_B(k=512) and C_C (Static).

Correlation values ρ between code-inferred networks C_C and the random networks C_R are substantially lower than between C_C and Bugzilla-inferred networks C_B across different values of k. This shows that the correlations between the socially- and technically-inferred networks are not due to chance. Correlations decline for larger k, because there are fewer people satisfying the threshold, thereby reducing the number of edges in C_B, whereas those in C_C are fixed.

4 Discussion

The high correlation between edge rank in the code-inferred networks and edge rank in the Bugzilla-inferred networks suggests that strongly dependent software components indeed require *more* people to look at bug reports and feature requests of both components. While this appears intuitive, our case study provides interesting *empirical* evidence that software dependencies correlate with social interactions during maintenance activities. Whether those two systems reinforce themselves (leading to higher correlation over time) or seek some stable equilibrium (leading to an 'optimal' correlation) remains an open question.

Our data reveals that certain social structures seem to correlate better with software structures than others. Static dependencies show a stronger correlation with social structures than dynamic dependencies, which is the case for all thresholds. One interpretation of this is that the extension points used in dynamic dependencies insulate a component from changes better than static dependencies. Requests involving statically dependent components are hence more likely to require developers to look at both components.

A high edge correlation between technical-inferred and social-inferred networks also means that by analyzing only one of these networks, one can approximate certain characteristics of the other. This might be useful as an indication of necessary development and communication effort related to specific software dependencies, or might help in allocating work tasks among software developers.

Because the two networks in our study were produced from independently acquired datasets (Eclipse architecture vs. Bugzilla data), our findings might also open up new ways of studying software dependencies in software architectures without direct access to code, based on datasets such as mailing lists or maintenance systems.

Threats to Validity On a technical level, we measured component dependencies by capturing static, dynamic and combined dependencies mined from the Eclipse SDK code. In this sense, our notion of dependencies is a strict and technical one, not capturing more implicit dependencies. On a social level, we measure dependencies by the number of developers involved in discussing requests for two components based on varying degrees of involvement k. While certain thresholds k produce higher correlations, our re-

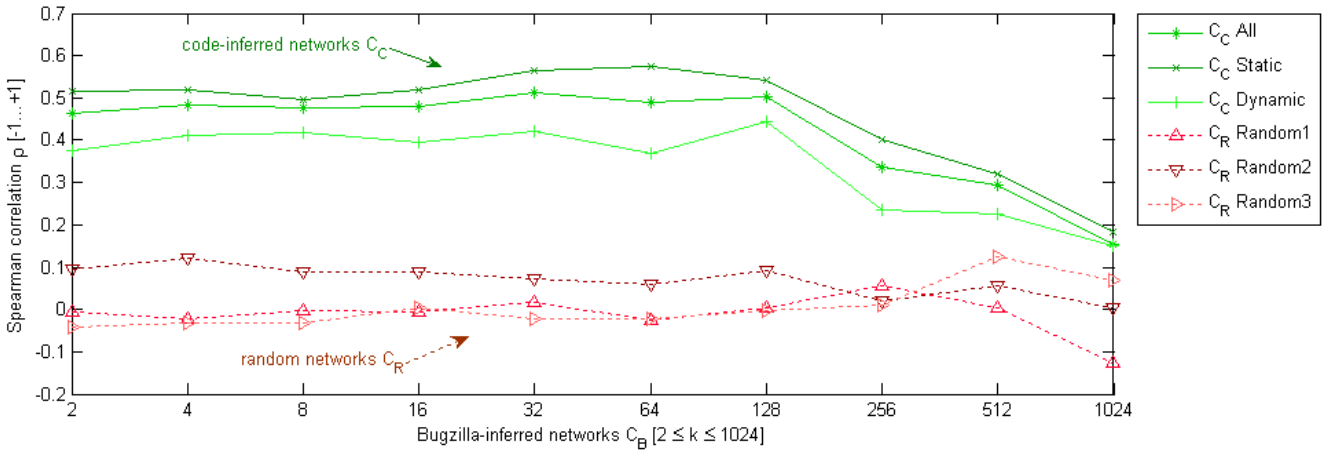

Figure 2. Correlation ρ of edge weight rankings between networks $C_B(2 \leq k \leq 1024)$, C_C and C_R

sults show good correlations between Bugzilla- and code-inferred networks across different values of k. We chose edge rank correlation over other network congruence metrics (such as arc mirroring [8]) due to its symmetric nature, and its applicability for our research question. C_B is build from the accumulated bug reports and feature requests over all SDK releases from 1.0 to 3.3.1.1, while C_C is built only from the latter. Thus, components not longer existing at version 3.3.1.1 are not taken into account even if they were renamed and hence the person counts should be merged for both components.

5 Concluding remarks

While Conway postulates that the social structure of an organisation impacts the technical structure of a design, we are interested in expanding our view further down the development process, including maintenance. Our findings suggest that Conway's Law on design of software systems might extend to software and maintenance processes.

Towards that goal, this paper presented a preliminary study in which we have constructed both socially and technologically inferred networks of Eclipse components, using different datasets (code vs. bug repository) and a set of different parameters (k threshold). We found that the strongest congruence, as measured by edge weight ranking, is obtained when comparing the socially constructed network obtained with a threshold of $k = 64$ bug reports per person with the static component dependencies. This result encourages further studies, as static code dependencies are easier to obtain than dynamic ones, and the lower the threshold, the less data is necessary.

Our work contributes a different notion and measurement of congruence, i.e. edge weight ranking, which we hope may be useful to other researchers, when other measurements of congruence (like [8]) are not appropriate due to the nature of the data or the research question addressed.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of the 28th Int'l Conf. on Software Engineering*, pages 361–370, 2006.

[2] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *SIGSOFT '08/FSE-16: Proc. of the 16th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, pages 24–35, New York, NY, USA, 2008. ACM.

[3] I. T. Bowman and R. C. Holt. Software architecture recovery using Conway's law. In *Proc. of the 1998 Conf. of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1998.

[4] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, 2008, Kaiserslautern, Germany*, pages 2–11, 2008.

[5] M. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[6] C. de Souza, J. Froehlich, and P. Dourish. Seeking the source: software source code as a social and technical artifact. In *Proc. of the Int'l ACM SIGGROUP Conf. on Supporting group work*, pages 197–206. ACM, 2005.

[7] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proc. of the 30th Int'l Conf. on Software Engineering*, pages 521–530. ACM, 2008.

[8] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams. Using software repositories to investigate socio-technical congruence in development projects. In *Proc. 4th Int'l Workshop on Mining Software Repositories*. IEEE, 2007.

[9] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[10] M. Wermelinger, Y. Yu, and A. Lozano. Design principles in architectural evolution: a case study. In *Proc. 24th IEEE Int'l Conf. on Software Maintenance*, pages 396–405, 2008.

[11] M. Wermelinger, Y. Yu, and M. Strohmaier. Using formal concept analysis to construct and visualise hierarchies of socio-technical relations. In *Proc. 31st Int'l Conf. on Software Eng., Companion volume*. IEEE, 2009. To appear.