A Design Structure Matrix Approach for Measuring Co-Change-Modularity of Software Products

Robert Benkoczi, Daya Gaur, Shahadat Hossain, Muhammad A. Khan*
University of Lethbridge, Department of Mathematics and Computer Science, Lethbridge AB, Canada
robert.benkoczi@uleth.ca,daya.gaur@uleth.ca,shahadat.hossain@uleth.ca,ma.khan@uleth.ca

ABSTRACT

Several authors have quantified the modularity of software systems in terms of coupling and cohesion metrics. Most of these approaches focus on functional and procedural dependencies in the system. Although highly relevant at the design phase, these static dependencies alone do not account for how a software product evolves over time. Instead, this is also dictated by logical and hidden dependencies between system files. To a large extent, the co-change (co-commit) relation captures these different types of dependencies. In this paper, we define two measures of co-change-modularity of a software product based on a weighted design structure matrix (DSM). The first metric, called the weighted propagation cost, uses matrix exponential to measure how changes to one system file potentially affect the whole product. The second metric, called the weighted clustering cost, uses the output of the first metric to measure the partitionability of the system based on the co-change relation. In addition, we provide a visual representation of how the co-change structure of a system evolves over time. We discuss the theoretical foundation of our work and highlight its advantages over existing methodologies. We apply our approach to GNU Octave and show the findings to be consistent with the available literature on the evolution of Octave. Our analysis is extensible and applicable to a range of scenarios including open source systems.

KEYWORDS

Co-change structure, software modularity, software evolution, coupling and cohesion, design structure matrix, clustering.

ACM Reference Format:
Robert Benkoczi, Daya Gaur, Shahadat Hossain, Muhammad A. Khan. 2018. A Design Structure Matrix Approach for Measuring Co-Change-Modularity of Software Products. In *MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3196398.3196409

1 INTRODUCTION

There is a consensus in software engineering community that modularity is a key feature of good software design [2, 23]. A 'modular' software product is one with a high degree of intra-modular cohesion and a low degree of inter-modular coupling. Most of the cohesion and coupling metrics in the literature use static dependencies between system elements to describe modularity [11]. These elements could be system files, classes or other appropriate units of analysis. Static analysis is a good approach during the development stage of the software life-cycle. However, during the evolution stage, logical and hidden dependencies between system elements play a more dominant role [7, 11]. The former type of dependency exists between two elements that logically serve the same purpose in the system, and the latter type of dependency is the result of unexpected side-effects of changes.

Given a series of commits during software revision, a co-change relation is said to exist between two system elements if they are committed together at some point. It has been argued that logical and hidden dependencies in a software product arise as co-change relations [11]. In addition, co-change data can be easily mined from software repositories without analyzing the code-base. This motivated Gall, Hajek and Jazayeri [11], and others [7, 22] to define coupling and cohesion in terms of co-change relations. Geipel and Schweitzer [12] and the subsequent work [1, 6, 19] shows that the nature of the interplay between static dependencies and co-changes is not completely understood although reasonably strong correlation exists between the two. Nevertheless, co-changes encompass a wide array of dependencies [6, 11, 12]. Last but not the least, from a software evolution perspective [6] co-change relations are more relevant than function call dependencies.

A design structure matrix (DSM) [9] provides a useful representation of any dependency structure. For instance, MacCormack, Rusnack and Baldwin [16] use a function call DSM to define the propagation and clustered costs of a software product. The lower these costs, the more cohesive and loosely coupled the files are in the system. In this paper, we significantly refine the notions of propagation and clustered costs and adapt them to examine the co-change-modularity of a software product. Moreover, we overcome certain weaknesses of MacCormack et al.'s measures [16] which we explain in Section 3. Our unit of analysis is system file and we consider changes committed in a given interval of time. The weighted propagation cost measures how changes made to one system file are likely to propagate through the system, whereas the weighted clustering cost determines the clusterability of the system into cohesive and loosely coupled groups based on the co-change relations. The lower the two costs, the more modular the system is with respect to the co-change relation. Our analysis produces numerical and visual data describing the temporal evolution of the co-change-modularity of a system.

We survey the relevant literature in Section 2. In Section 3, we discuss the motivation and the mathematics behind our measures, and how they improve on the earlier work. Section 4 presents an application of our approach to Octave [8] - an open source scientific computing software - and features our results. We choose Octave for the case study as it has evolved through three distinct phases of initial development, stabilization, and large-scale growth. Furthermore, we have a first-hand account [8] of its evolution from the primary developer and maintainer. This allows us to test our approach against real-life verifiable information. Our numbers successfully explain and distinguish between the three evolutionary phases of Octave. Finally, we outline some conclusions in Section 5.

2 RELATED WORK

In recent years, several researchers have used the co-change relation for clustering code artefacts [4], and detecting fault-proneness [15]

*Corresponding author

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05.
https://doi.org/10.1145/3196398.3196409

and design deficiencies [18] in software. Here we analyze the co-change-modularity of software.

Geipel and Schweitzer [12] empirically studied the nature of the relationship between static dependencies among Java classes and co-change. Based on their experiments they conclude "any metric which uses dependencies alone to pass judgment on the evolvability of a piece of Java software is thus unreliable". Their claim is further substantiated by the work of Ajienka and Capiluppi [1], and Oliva and Gerosa [19]. In particular, Cafeo et al. [6] observed a strong correlation between co-changes and dependencies. All of this motivates our exploratory study.

Sullivan et al. [23] identified the problem of evaluating the modularity of a software design and used the DSM methodology of Baldwin and Clark [2] to attack it. Since then, the DSM methodology has found uses in the design, modularization, refactoring and evolution management of software. It has been pointed out that static dependencies lead to the propagation of change. MacCormack et al. [16] proposed DSM based metrics to capture the propagation of changes. We significantly enhance the approach of [16] and use it to study the co-change-modularity of software products.

The use of clustering based on static dependencies to manage the evolution of software was initiated by Sangal et al. [21]. Gall et al. [11] were the first to study the evolutionary dependencies based on the commit history. They proposed a scheme to modularize the system based on such logical and hidden dependencies. Beyer and Noack [5] were the ones to coin the term 'co-change graph', and cluster the co-change graphs with a focus on visualization. Artifacts that have changed together appear closer to each other in the layout computed by them. Silva et al. [22] considered the co-change graph as a sparse graph and retrieved the co-change clusters of system classes using Chameleon clustering [14]. This direction was further pursued by de Oliviera et al. [7] who provided empirical evidence that static dependencies across the modules do not truly capture the future maintenance cost. They also argued that hidden dependencies have to be considered in any redesign on the system. They measured the stability of the system in terms of the propagation cost and clustered cost measures of MacCormack et al. [16]. As pointed out in [1, 6, 12], we still do not have a deep appreciation of the nature of the static dependencies that translate into co-change relations. However, since co-changes capture logical and hidden dependencies [11] as well as a portion of static dependencies [6, 12], we use them in our study of software modularity.

Zimmerman et al. [24–26] used association rule mining on co-change graphs to predict changes and to determine the evolutionary coupling between fine-grained modules. Their definition of a co-change is albeit different. A co-change is said to have occurred between two units if they are committed within Δ time units. A sliding window is used to compute the co-change relationships. Our definition of co-change agrees with the one used by Ball et al. [3] and Geipel and Schweitzer [12]. An important conclusion that we use from [12] is that co-change propagates through a path of static dependencies. This implies that co-change propagates through a path of co-changes. We believe that the chance of this happening diminishes as the path length increases.

3 THEORETICAL FRAMEWORK AND METHODOLOGY

In [16], two measures of modularity, namely, the propagation cost and the clustered cost, are defined. The authors consider a DSM with 1/0 entries representing whether or not a file makes a function call to another file. The propagation cost is calculated by summing all the entries of the transitive closure of this binary DSM. The underlying idea is that higher propagation cost implies a higher degree of coupling through transitive dependencies. The original DSM (and not its closure) is then clustered in a standard manner using different penalties for dependencies that (i) run across two clusters, (ii) fall within a cluster (iii) involve a 'bus'. Here a bus is a system file to which many other files make function calls - for example, a header file. A dependency of type (i) incurs a heavy cost, type (ii) has a lower cost that depends on the cluster size and (iii) is penalized the least. The intuition is that a low clustered cost indicates a more cohesive system, and that a function call-modular system should be making most of its function calls within modules of relatively small sizes.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \quad \text{(a)} \qquad \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \quad \text{(b)} \qquad \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \quad \text{(c)}$$

Figure 1: Three DSMs with propagation cost (a) 15, (b) 15 and (c) 25 computed as the sum of entries in the transitive closure of each DSM. Since the DSM (b) is the transitive closure of (a), the propagation cost is unable to distinguish between them. The weighted propagation costs are (a) 29.56, (b) 56.74 and (c) 36.94 computed as the sum of entries in the matrix exponential of the DSM.

Although widely cited by the DSM and software communities, the above approach has its drawbacks, which we address here.

(1) The measures do not distinguish whether a file calls another once or multiple times.
(2) A transitive dependency resulting from multiple static dependencies has the same weight as a direct static dependency. For instance, the same propagation cost of 15 is attributed to the DSMs in Fig. 1 (a) and (b) despite one of them corresponding to a directed path and the other to a complete directed acyclic graph.
(3) The existence of a symmetric pair of dependencies can drastically change the propagation cost. For example, adding the dependency (E, A) to the DSM in Fig. 1 (a) gives the DSM in Fig. 1 (c). This changes the propagation cost to 25.
(4) The two measures are independent of each other.

3.1 Weighted Propagation Cost

The matrix exponential of a square matrix $\mathbf{M}$, denoted by $e^{\mathbf{M}}$, is defined using the power series expansion $e^{\mathbf{M}} = \sum_{k=0}^{\infty} \frac{\mathbf{M}^k}{k!}$. Any square matrix $\mathbf{M}$ with non-negative integral entries can be thought of as the adjacency matrix of a directed graph $G_{\mathbf{M}}$ with the (i, j)-entry, $(\mathbf{M})_{ij}$, equal to the number of edges directed from node i to j. With this interpretation, for any non-negative power $\mathbf{M}^k$ of $\mathbf{M}$, $(\mathbf{M}^k)_{ij}$ counts the number of walks of length k in $G_{\mathbf{M}}$ directed from node i to j. Therefore, the (i, j)-entry of the matrix exponential $e^{\mathbf{M}}$ counts the total number of walks of all lengths directed from i to j while dampening the effect of longer walks. Estrada and Hatano [10] initiated the use of matrix exponential for measuring propagation in physical networks arguing that longer walks contribute less. Moreover, Geipel and Schwitzer [12] have empirically justified the diminishing effect of long walks in co-change and dependency graphs. This justifies the use of matrix exponential in our study. In practice, $e^{\mathbf{M}}$ is approximated using Padé approximation or other suitable methods [13, 17] and contains valuable information as to how changes might propagate in $G_{\mathbf{M}}$.

Given a software product, we form a symmetric DSM, **D**, with diagonal entries equal to 1 and any off-diagonal (i, j)-entry equal to the number of times files i and j are committed together in a given time interval. For the purpose of our study we chose one-year intervals. We call **D** the *co-change DSM*. If n is the total number of files committed over the given time interval, then **D** is an $n \times n$ matrix. We define the *weighted propagation cost* over that time interval as

$$\text{WtProp} = \sum_{i,j=1}^{n} (\mathcal{E})_{ij}, \tag{1}$$

where $\mathcal{E} = e^{\mathbf{D}}$. Intuitively, the quantity WtProp measures how co-changes propagate through a path of co-changes and affect the whole system. Thus we enrich the notion of propagation cost [16] at two levels, first, by using an integral DSM rather than a binary one, and second, by using matrix exponential instead of the transitive closure. This addresses items 1-3) raised above. For instance, the DSMs in Fig. 1 (a), (b) and (c) have distinct weighted propagation costs of 29.56, 56.74 and 36.94, respectively, clearly differentiating between their dependency structure.

3.2 Weighted Clustering Cost

A 'greedy' clustering algorithm is an iterative procedure that makes the best local move during each iteration to improve the quality of clustering. The quality of clustering is determined by some measure of clustering error or cost. We apply such a procedure to the DSM $\mathcal{E}$ of order $n \times n$. This provides a bridge between our measures and takes care of item 4) above.

We begin with each file in its own cluster. At each step, the algorithm moves a file between clusters that reduces the cost. The possible moves are considered in a random order and the first improving move is accepted. We adopt the cost function of MacCormack et al. [16] with some adjustment. At each iteration, a pair (i, j) of files is assigned a cost of $(\mathcal{E})_{ij}\, m^2$ if both the files belong to the same cluster of order $m \times m$, and a cost of $(\mathcal{E})_{ij}\, n^2$ otherwise. The algorithm terminates when no local move can decrease the cost beyond a change threshold δ. Let **C** denote the resulting clustered matrix and cost_{ij} denote the cost associated to a pair (i, j) at the terminal stage. Then the *weighted clustering cost* is

$$\text{WtClust} = \sum_{i,j=1}^{n} \text{cost}_{ij}. \tag{2}$$

Clearly, $\text{WtProp} \leq \text{WtClust} \leq n^2\, \text{WtProp}$. Overall, the objective is to determine a clustering of system files into a relatively large number of small clusters such that WtClust is minimized. The DSM, **C**, can be visualized to observe the co-change clusters of a software product at any given time.

The lower the weighted propagation and clustering costs, the more co-change-modular the system is. The following points are worth noting.

(i) *Lack of buses:* If we focus on system files, excluding text and log files, we do not observe any buses (in the sense of [16]) in the co-change DSM. The reason is that a single system file is not likely to be repeatedly co-committed with a large number of other files.

(ii) *Metrics as functions of time:* The measures WtProp and WtClust are functions of time. As a software product evolves, these function values indicate if the product is becoming more or less modular.

(iii) *Normalizing the co-change DSM:* Numerical blow-up is a likely prospect in handling real-life data. Normalizing or scaling a matrix is a standard way of getting around this problem [13]. In our case study, we divide each entry of the co-change DSM, **D**, by the maximum column sum of **D**. Our measures are computed afterward.

(iv) *Computational complexity:* Computing (approximately) the matrix exponential of an $n \times n$ matrix requires $O(n^3)$ operations [17] and the same is needed to calculate WtProp. The expensive local search algorithm to compute WtClust can be sped up considerably by increasing the change threshold δ or by using the fully polynomial time approximation scheme (FPTAS) introduced in [20]. However, we do neither.

4 CASE STUDY: THE EVOLUTION OF GNU OCTAVE

Octave is a popular open source numerical computing platform. It got its first alpha release in 1993 and has been under active development and maintenance since. Our choice of Octave for this case study is dictated by several considerations.

In a 2001 article [8], John Eaton - the primary developer of Octave - gave a detailed account of the history, development, maintenance and foreseeable future of Octave. Based on this, the evolutionary cycle of Octave can be divided into three phases: (i) *the early development* with Eaton as the sole/chief developer, (ii) *maintenance and stabilization* by Eaton around the turn of the century, and (iii) *open development* as Eaton stepped back and a large number of developers got involved. We wanted to see if our approach could distinguish between these phases. Secondly, Octave source code and historical commit data are easily accessible.The source is sizeable and mainly consists of three types of files: header files with .h extension, core C/C++ files with .cc or .cp extensions, and module files with a .m extension. At least in the later stages of the evolutionary cycle, one would expect a lot more commits for .m files than .c and .h files. Furthermore, .h files would potentially form the most stable part of the system. Here, we analyze the three file types independently and ignore the other less frequently occurring file types.

We downloaded the source code from https://www.gnu.org/software/octave/. We extracted all commit logs from 1993-2017 and formed normalized co-change DSMs over each year for each file type. In all, this resulted in 75 DSMs with the number of files (nodes) in the range 60-2,000 and number of commits (dependencies) ranging from 150-11,000. We coded our weighted propagation and clustering algorithms in julia (version 0.6.0). We ran our programs on a cluster (with theoretical CPU performance of 936 teraflops), in a massively parallel fashion, where each input instance ran on a separate CPU simultaneously. We use $\delta = 0$ as the change threshold (see Section 3.2). We set a 24-hour time limit for processing a single DSM and found that the process was completed within 1-6 hours in 73 out of 75 instances, with the clustering part unfinished for the remaining instances. We collect the values of the weighted propagation and clustering costs in Table 1 and 2, respectively. The resulting clustered DSMs for selected years are presented in Fig. 2. Note that for a co-change-modular system we should observe relatively small dark-coloured diagonal blocks and few off-diagonal dark patches (cross-relations).

The results provide a clear demarcation between the three phases of evolution. From 1993-97, the weighted propagation and clustering costs mostly increase - rather drastically for .m files. This is understandable as Octave was in its infancy and a lot of new files must have been added and bugs fixed. Although Eaton was the only developer with little help from others, he developed the system very actively during this period [8]. Both the metrics sharply decrease in 1998. This can probably be attributed to Octave moving to GNU in 1997 [8]. From 1998-2001 our metrics remain in the lower-bracket except for some surges in 2000. This is the period when Eaton was

Table 1: Octave: weighted propagation cost over the years.

Year	1993	1994	1995	1996	1997	**1998**	**1999**	**2000**	**2001**	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017
.c Files	268.23	218.88	310.55	384.19	674.62	158.29	164.57	338.42	118.91	452.47	323.26	375.74	754.02	393.60	966.64	566.81	847.96	611.48	895.66	2432.53	2040.65	1226.52	1096.28	1341.37	1276.12
.h Files	186.18	175.44	393.26	381.04	645.79	130.61	93.13	307.35	111.87	342.96	274.32	297.28	674.83	284.83	729.09	299.98	470.36	606.42	967.20	1842.56	1488.48	994.16	1034.78	1118.16	1059.11
.m Files	174.87	354.61	357.52	1922.92	2253.57	807.55	593.47	1157.25	120.79	1000.97	1429.29	580.59	1597.84	1267.16	2332.17	4123.28	1475.90	1128.36	1897.04	2344.32	2995.97	3507.82	2371.76	2712.68	2691.30

Table 2: Octave: weighted clustering cost (in millions) over the years. An 'x' indicates that clustering did not finish within 24 hours.

Year	1993	1994	1995	1996	1997	**1998**	**1999**	**2000**	**2001**	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017
.c Files	2.42	1.42	3.83	8.13	4.37	0.51	0.54	5.24	0.20	12.94	3.68	7.58	56.17	8.50	97.86	23.02	86.62	33.30	103.62	x	1279.56	244.30	190.89	308.72	259.92
.h Files	0.82	0.75	7.77	7.82	37.70	0.28	0.09	4.08	0.18	5.47	2.07	3.81	42.37	2.84	39.18	2.70	15.16	31.35	132.54	919.82	496.56	117.14	128.54	172.22	144.28
.m Files	0.53	5.02	5.68	714.22	1631.68	76.42	30.55	226.58	0.21	140.30	355.58	27.72	532.08	282.56	1419.05	8429.75	473.78	204.54	997.28	1802.77	3478.16	x	1763.66	2320.71	2229.39

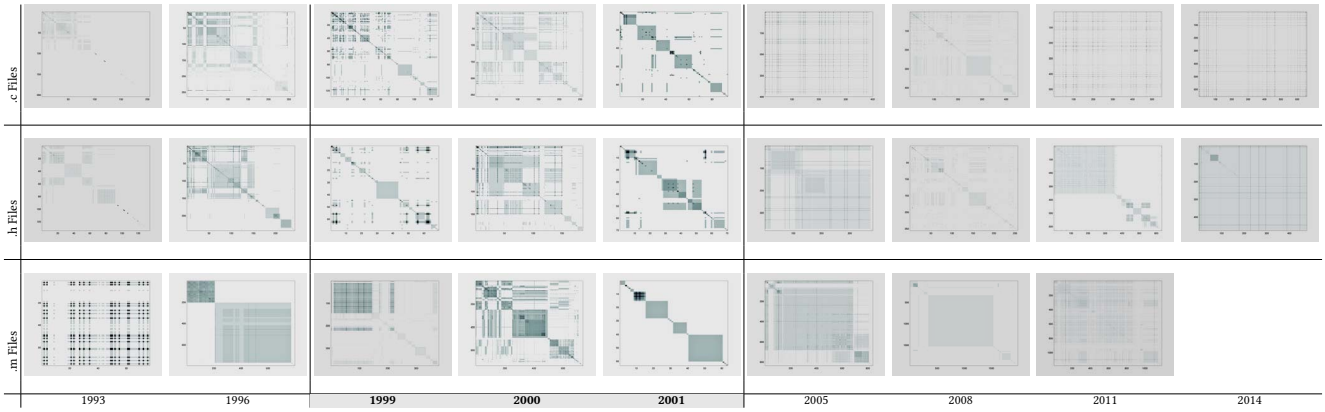

Figure 2: Evolution of co-change clusters of Octave. The blank cell corresponds to an instance of clustering algorithm requiring more than 24 hours of processing time.

contemplating taking a break from Octave and was focusing more on maintenance than development [8]. The surges in the year 2000 can be attributed to bug fixes related to Y2K. We also observe the sharpest clusters during these years. From 2002 onwards, the metrics progressively increase with weighted clustering cost blowing up for .m files. After 2001, Octave has undergone a lot of open source development and maintenance. The lack of a single primary developer and a loose organizational structure have led to changes and commits being made in an unstructured fashion. We observe that almost throughout, .m files demonstrate the least co-change-modularity, while .h files exhibit it the most. Overall, the results match our expectations based on Eaton's article [8] and the Octave file structure quite well. The authors will make the case study data and the code generating it available to anyone interested.

5 CONCLUDING REMARKS

For our exploratory study on measuring co-change-modularity we build on the work of MacCormack et al. [16]. This has motivations in the study by Geipel and Schweitzer [12], who showed that dependencies based metrics alone cannot be used to study the evolution of a Java software. Often, the co-change data for software products is more readily accessible than the source code. Our paper exploits this information to measure the modularity of any software product based on its co-change structure and without examining the source code. We represent the historical co-change data as a weighted DSM and propose two metrics, weighted propagation, and weighted clustering costs. The first determines how changes to one file impact the rest of the system, and the second measures the clusterability of the source files based on the co-change relation. Our approach provides a numerical as well as a visual representation of the evolution of the co-change-modularity of any software system over time. We apply our approach to GNU Octave [8], a fairly large open source project with a strong developer and user base. We analyze its evolution over 25 years and show that after a modular phase, Octave has progressively become less modular. As a test for validity, we demonstrate that our approach can correctly identify certain watershed moments in the evolution history of Octave [8]. The insight gained from our approach can help developers in optimizing their software development and maintenance processes.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable suggestions. The third author acknowledges support from his NSERC Discovery Grant. This research was enabled in part by support provided by WestGrid (www.westgrid.ca) and Compute Canada (www.computecanada.ca).

REFERENCES

[1] N. Ajienka and A. Capiluppi. 2017. Understanding the interplay between the logical and structural coupling of software classes. *J. Systems Software* 134 (2017), 120–137.

[2] C. Y. Baldwin and K. B. Clark. 1999. *Design Rules: The Power of Modularity*. Vol. 1. MIT Press, Cambridge, MA, USA.

[3] T. Ball, J. Kim, A. Porter, and H. Siy. 1997. If your version control system could talk. In *Proc. ICSE Workshop Process Modelling and Empirical Studies of Software Eng.*

[4] F. Beck and S. Diehl. 2013. On the Impact of Software Evolution on Software Clustering. *Empirical Software Eng.* 18 (2013), 970–1004.

[5] D. Beyer and A. Noack. 2005. Clustering software artifacts based on frequent common changes. In *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*. 259–268.

[6] B. P. Cafeo, E. Cirilo, A. Garcia, F. Dantas, and J. Lee. 2016. Feature dependencies as change propagators: an exploratory study of software product lines. *Inf. Software Technol.* 69 (2016), 37–49.

[7] M. C. de Oliveira, R. Bonifácio, G. N. Ramos, and M. Ribeiro. 2016. Unveiling and reasoning about co-change dependencies. In *Proc. of the 15th International Conference on Modularity (MODULARITY 2016)*. 25–36.

[8] J. W. Eaton. 2001. Octave: past, present and future. In *Proc. of the 2nd International Workshop on Distributed Statistical Computing*. Technische Universität Wien, Vienna, Austria.

[9] S. D. Eppinger and T. R. Browning. 2012. *Design Structure Matrix Methods and Applications* (1 ed.). Vol. 1. MIT Press, Cambridge, MA, USA.

[10] E. Estrada and N. Hatano. 2008. Communicability in complex networks. *Physical Rev. E* 77 (2008), 036111.

[11] H. Gall, K. Hajek, and M. Jazayeri. 1998. Detection of logical coupling based on product release history. In *Proc. of the International Conference on Software Maintenance (ICSM '98)*. 190–198.

[12] M. M. Geipel and F. Schweitzer. 2012. The link between dependency and cochange: empirical evidence. *IEEE Trans. Software Eng.* 38, 6 (Nov.-Dec. 2012), 1432–1444.

[13] N. J. Higham. 2006. The scaling and squaring method for the matrix exponential revisited. *SIAM J. Matrix Anal. Appl.* 26, 4 (2006), 1179–1193.

[14] G. Karypis, E. H. S. Han, and V. Kumar. 1999. Chameleon: hierarchical clustering using dynamic modeling. *Computer* 32, 8 (1999), 68–75.

[15] S. Kirbas, B. Caglayan, T. Hall, S. Counsell, D. Bowes, A. Sen, and A. Bener. 2017. The relationship between evolutionary coupling and defects in large industrial software. *J. Software Evol. Proc.* 29 (2017), e1842.

[16] A. MacCormack, J. Rusnak, and C. Y. Baldwin. 2006. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. *Management Science* 52, 7 (2006), 1015–1030.

[17] C. Moler and C. Van Loan. 2003. Nineteen dubious ways to compute the exponential of a matrix twenty-five years later. *SIAM Rev.* 45, 1 (2003), 3–49.

[18] M. Mondal, C. K. Roy, and K. A. Schneider. 2013. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In *Proc. of the 2013 21st International Conference on Program Comprehension (ICPC)*. San Francisco, CA, 103–112.

[19] G. A. Oliva and M. A. Gerosa. 2015. Experience report: How do structural dependencies influence change propagation? An empirical study. In *Proc. of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 250–260.

[20] J. B. Orlin, A. P. Punnen, and A. S. Schulz. 2004. Approximate local search in combinatorial optimization. *SIAM J. Comput.* 33, 5 (2004), 1201–1214.

[21] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. 2005. Using dependency models to manage complex software architecture. *ACM SIGPLAN Notices* 40, 10 (Oct. 2005), 167–176.

[22] L. Silva, M. T. Valente, and M. Maia. 2014. Assessing modularity using co-change clusters. In *Proc. of the 13th International Conference on Modularity (MODULARITY 2014)*. 49–60.

[23] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. 2001. The structure and value of modularity in software design. *ACM SIGSOFT Software Eng. Notes* 26, 5 (2001), 99–108.

[24] T. Zimmermann and P. Weissgerber. 2004. Preprocessing CVS data for fine-grained analysis. In *Proc. 2004 Mining Software Repositories (MSR)*. 2–6.

[25] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. 2004. Mining version histories to guide software changes. In *Proc. of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, 563–572.

[26] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. 2005. Mining version histories to guide software changes. *IEEE Trans. Software Eng.* 31, 6 (2005), 429–445.