Received: 8 April 2015 | Revised: 21 November 2016 | Accepted: 2 December 2016

DOI: 10.1002/smr.1842

WILEY Journal of Software: Evolution and Process

RESEARCH ARTICLE

The relationship between evolutionary coupling and defects in large industrial software

Serkan Kirbas[1,2] | Bora Caglayan[3] | Tracy Hall[2] | Steve Counsell[2] | David Bowes[4] | Alper Sen[1] | Ayse Bener[3]

[1]Department of Computer Engineering, Bogazici University, Istanbul, Turkey
[2]Department of Computer Science, Brunel University London, London, United Kingdom
[3]Data Science Laboratory, Ryerson University, Toronto, Ontario, Canada
[4]School of Computer Science, University of Hertfordshire, Hatfield, United Kingdom

Correspondence
Serkan Kirbas, Department of Computer Engineering, Bogazici University, Istanbul, Turkey.
Email: serkan.kirbas@boun.edu.tr

Funding Information
Scientific and Technological Research Council of Turkey (TUBITAK), Grant/Award Number: B.14.2.TBT.0.06.01-214-115535; Bogazici University Research Fund (7223) Turkish Academy of Sciences and by Engineering and Physical Sciences Research Council (EPSRC), Grant/Award Number: EP/L011751/1; NSERC Discovery, Grant/Award Number: 402003-2012

Abstract

Evolutionary coupling (EC) is defined as the implicit relationship between 2 or more software artifacts that are frequently changed together. Changing software is widely reported to be defect-prone. In this study, we investigate the effect of EC on the defect proneness of large industrial software systems and explain why the effects vary. We analysed 2 large industrial systems: a legacy financial system and a modern telecommunications system. We collected historical data for 7 years from 5 different software repositories containing 176 thousand files. We applied correlation and regression analysis to explore the relationship between EC and software defects, and we analysed defect types, size, and process metrics to explain different effects of EC on defects through correlation. Our results indicate that there is generally a positive correlation between EC and defects, but the correlation strength varies. Evolutionary coupling is less likely to have a relationship to software defects for parts of the software with fewer files and where fewer developers contributed. Evolutionary coupling measures showed higher correlation with some types of defects (based on root causes) such as code implementation and acceptance criteria. Although EC measures may be useful to explain defects, the explanatory power of such measures depends on defect types, size, and process metrics.

KEYWORDS

evolutionary coupling, industrial software, legacy software, mining software repositories, measurement, software defects

1 | INTRODUCTION

Software constantly changes for many reasons.[1–4] Studies have shown that changing software may be a defect-prone activity.[5–8] Code that is changed most frequently is likely to be most defect-prone.[7,9–11] Evolutionary coupling (EC) could explain some of this defect proneness because when code with high EC is changed, a high number of changes must be made to related parts of the system. The locations of these related changes may be scattered within the application or even across applications in a software ecosystem. Correctly making related changes across these locations is likely to be challenging. Developers may miss some locations, which should have been cochanged, and this may cause unforeseen ripple effects and problems.

Evolutionary coupling information is generally extracted from the commit history of version control systems (VCSs). It is based on the assumption that artifacts committed together are logically coupled. This makes EC relatively simple to calculate compared with other types of coupling. For example, structural[12] and semantic[13] coupling are both measured on the basis of the static and text analysis of source code. Often, this source code is difficult to obtain from closed source developers. Dynamic coupling[14] analyses execution traces and so requires the software to be executed. Evolutionary coupling requires access to only the VCS and is thus a relatively easy way to measure coupling, particularly for industrial closed source systems.

Evolutionary coupling has previously been shown to indicate architectural and design problems. Gall et al[15,16] showed that EC can discover design flaws such as God classes or Spaghetti code, without analysing the source code. Gall et al's results also identified architectural weaknesses such as poorly designed inheritance hierarchies and blurred interfaces between modules and submodules. Breu and Zimmermann[17] showed that EC information and data mining techniques could detect crosscutting concerns in software systems. Such crosscutting concerns emerging overtime may contain functionality, which does not align with its architecture. Furthermore, Eaddy et al[18]

argued that crosscutting concerns were harder to implement and change consistently because multiple (possibly unrelated) locations in the code have to be found and updated simultaneously. Their study suggested that increased crosscutting concerns may actually cause or contribute to defects. Our own previous study of EC in a banking system[19] suggested that EC does impact defects.

Conversely, Graves et al[8] showed that module-level EC measures were a poor predictor of defect proneness. Knab et al[20] also found that EC did not predict defects in the Mozilla project; no studies exist to explain these contradictory findings. Furthermore, EC in large commercial systems has rarely been empirically investigated. Most previous studies are based on the analysis of open source systems.

In this study, we analysed the correlation between EC measures and the number of defects and defect density in 2 large software systems in industrial software development environments. Correlation analysis is performed separately for each module.* We also built logistic regression models. In this study, multivariate regression analysis is used to explore the relationship between EC (independent variable) and defects (dependent variable) to understand how helpful EC measures are in defect analysis compared with other process metrics (we build correlation models rather than prediction models). We also analysed the relationship between EC and defect types. Our research questions are as follows:

- **(RQ1) What is the relationship between EC and software defects?**

The results of our study showed that there was, in general, a relationship between EC and software defects in the software maintenance/evolution phase of the industrial software systems under study. We detected a positive correlation between EC measures and defects. Compared with other process measures such as the number of commits and the number of developers, EC measures seem to contain additional, sometimes important, information about defects: for every additional EC, the module is 8% more likely to be defective. However, correlation strength varied across modules and in some modules EC and defects were not correlated. On the basis of these findings, we added the following research question to our study:

- **(RQ2) What factors explain why the relationship between EC and software defects is different for different modules?**

Modules, which were small in Lines of Code (LOC) and developer numbers, tended to be less correlated with EC. Fewer defects due to EC seem to occur in small modules. Evolutionary coupling also appeared to be more highly correlated with some types of defects such as code implementation, acceptance criteria, and analysis problems. Overall, regression analysis showed that EC may be useful for explaining defects in industrial systems.

We make the following contributions in this paper. Firstly, we analyse large commercial systems, which have rarely been empirically studied to understand the relation between EC and defects. Secondly, we show that the effect of EC on defects varies depending on the module. Thirdly, the explanatory power of EC measures varies depending on defect types and module features such as size and developer activity.

This paper is organised as follows: In the next section, we summarise related work. In Section 3, we present our methodology including measures, data extraction, and analysis methods. Section 4 shows the results of applying our methodology to 2 industrial systems. The discussions and threats to validity of this study are then addressed in Sections 5 and 6, respectively. Finally, in Section 7, we summarise and present our conclusions.

2 | RELATED WORK

2.1 | Evolutionary coupling

Evolutionary coupling was first identified in 1997 by Ball et al.[21] Early studies on EC focused on the relationship between EC and architectural problems with EC used as an indicator of architectural weaknesses and modularity problems. Classes that were frequently changed together during the evolution of a system were presented visually using EC information by Ball et al.[21] Clusters of classes were identified according to EC measures. Ball et al showed that classes belonging to the same cluster were semantically related. Evolutionary coupling among different clusters was used as an indicator of ineffective class partitioning. Gall et al analysed EC at a module level and reported that EC provides useful insights into system architecture.[15] They identified potential structural shortcomings and detected modules and programs that should undergo restructuring or even reengineering. Another study by Gall et al analysed EC at a class level on an industrial software system.[16] This study was important because it demonstrated that EC could be used to identify architectural weaknesses such as poorly designed interfaces and inheritance hierarchies. Pinzger et al showed that candidate modules for refactoring could be detected by showing ECs between modules on Kiviat diagrams.[22] Beyer and Hassan explored EC data in the calculation of the distance between 2 files in a VCS and displayed results as a series of animated panels.[23] They showed how the structure of a software system decayed or remained stable overtime.

Besides detecting architectural problems, EC has also been used to predict possible cochanges and to recommend such cochanges to developers. In a study by Ying et al, an approach using data mining techniques was developed to recommend related source code parts to software developers with assigned modification requests (MRs).[24] Applying the approach to open source projects revealed important dependencies. A study by Zimmermann et al presented a technique, which predicted the parts of source code likely to change, given the already changed parts of source code (at file, class, property, and method levels).[25] Association rule mining was used to detect ECs.

Several previous studies have used EC in the detection of crosscutting concerns scattered across software systems. Breu et al[17] leverage EC information to mine aspect candidates (identifying crosscutting concerns). Eaddy et al[18] argued that crosscutting concerns were harder to implement and change consistently because multiple (possibly unrelated) locations in the code have to be found and updated

*A module is part of a software system. A software system is composed of one or more independently developed modules. Similar functionality is contained within the same module, and a module is generally composed of many source files. A module is generally owned by a specific team, and the team members are responsible for its development and maintenance. In the systems analysed in this study, modules are also part of subsystems. There is a one-to-many relationship between subsystems and modules. A module can be part of only one subsystem, and a subsystem may have many modules. But subsystems are not covered in the scope of this work.

simultaneously. Their study suggested that increased crosscutting concerns may cause or even contribute to defects. Adams et al[26] developed an aspect-mining technique based on coaddition or coremoval of dependencies on program entities overtime. They suggest that detailed knowledge about crosscutting concerns in the source code is crucial for the cost-effective maintenance and successful evolution of large systems.

In our recent study of EC measurement,[27] we evaluated the measurement of EC in software artefacts from a measurement theory perspective. We defined 19 evaluation criteria based on the principles of measurement theory and metrology. We evaluated previously published EC measures by applying these criteria. Our evaluation results revealed that current EC measurement has the particular weaknesses around establishing sound empirical relation systems, defining detailed and standardised measurement procedures as well as scale type and mathematical validation.

2.2 | Relationship between EC and defects

Evolutionary coupling measures have also been used in defect prediction studies. These studies are related to our first research question (RQ1). First, we focus on the studies, which reported a relation between EC and defects. Steff and Russo created sequential commit graphs of evolutionary coupled classes.[28] They showed that the graphs could be used for defect prediction. A study by Tantithamthavorn et al proposed improvements to existing defect localisation methods by using EC information.[29] The proposed method was applied and verified on 2 open source projects (Eclipse SWT and Android ZXing).

D'Ambros et al[30] analysed 3 open source software systems and detected correlation between EC and software defects. This was the first study focusing explicitly on the relationship between EC and software defects, which corresponds to our RQ1. They found a positive correlation between EC and defects. Furthermore, they reported that defects with a high severity exhibited a correlation with EC. This study considered only EC between classes within a project. Another study reported by Kouroshfar concluded that cross-subsystem EC measures are more related to defects than within-subsystem EC.[31] Kouroshfar's findings are related to our second research question (RQ2), as Kouroshfar proposed different kinds of EC (between subsystems vs within subsystems) as a factor affecting the relationship between defects and EC.

Other studies using EC metrics suggest that EC does not contribute to defects and is not useful for identifying defects. These studies could not find any relationship between EC and defects, which is related to our RQ1. In a study conducted by Graves et al, various statistical models were developed to assess which features of the revision history of a module could be used for defect prediction.[8] Results from study showed that prediction performance of the models using EC measures were lower compared with other models. Another study by Knab et al found that EC measures did not give good results for predicting defects.[20] In that study, the ability of EC to predict defect density was tested. In our previous studies of EC,[19,32] we examined the effect of EC on software defects for an industrial legacy banking system. For some modules, we observed significant correlation between EC and defect measures, whereas for others, no relation was detected. This study is different from our previous studies in companies involved and the analysis applied. In this study, we analyse also a large modern telecommunication software and used different analyses such as multivariate regression analysis, defect type, and module characteristic analysis.

The previous studies on EC focused on open source projects. Also, most of these studies did not investigate large projects. Our study is different in the sense that we investigate industrial projects, which have very different software development processes and culture than the open source projects. Moreover, the sizes of the projects we analysed are different to existing studies. For example, the sizes of the projects studied by D'Ambros et al.[30] were between 1 and 3 K (number of classes). The sizes of the projects that we studied were 20 and 150 K (as number of files). Large industrial systems have rarely been empirically studied to understand the relationship between EC and defects. This is an important contribution of our work on existing knowledge of EC.

In contrast to previous studies, we also show that the relationship between EC and defects varies for different modules even in the same system. We provide the distribution of numerical values for EC-defect relationship as histograms. This introduces a more realistic and probabilistic model for the EC-defect relationship and can be also used to explain the contradictory results reported by different studies. Furthermore, we attempt to explain factors affecting the relationship between EC measures and defects, which has not been explicitly addressed by previous studies.

3 | METHODOLOGY

This section explains the setting and data sources that we used as well as how we extracted and analysed the data.

3.1 | Study context

We performed our study on 2 large industrial systems. One of the systems was a large financial legacy system that had evolved for over 25 years to support the back-end business processes of a large financial institution (henceforward known as "Company 1"). Much of the code was written in PL/I and COBOL, but there were also files written in job control language, a scripting language used on mainframes to develop a batch job. The system consisted of 20 subsystems and 274 modules. The company started using a VCS in 2009. We analysed all subsystems and modules between 2009 and 2013, and the total size of the system was 87 million LOC, consisting of 150 K individual files. The applications analysed in this study are back-end banking applications. Company 1 uses a "modified" waterfall model for software development.

The other system studied was a large telecommunications system written in Java (henceforward known as "Company 2"). The company had used a VCS (SVN) since 2006; we analysed 4 subsystems and 11 modules between 2006 and 2013. Company 2 uses an agile methodology as well as test-driven development (TDD). The applications analysed in this study are web applications. Detailed information about the systems under study is given in Table 1.

TABLE 1 Summary about industrial systems under study

	Company 1	Company 2
Programming languages	COBOL,PL/I, & JCL	Java
Domain	Finance	Telecommunications
Versioning system	CA SCM	SVN
Defect-tracking system	Developed in-house	JIRA
Number of software subsystems	20	4
Number of software modules	274	11
Total number of developers	460	25
Total software size (LOC)	87 M	310 K
Total number of files	150 K	26 K
Total number of file versions	192 K	180 K
Total number of commits	50 K	24 K
Analysis period	2009-2013	2006-2013
Percentage of files changed	%11	%15

Abbreviations: CA SCM indicates CA Software Change Manager; LOC, Lines of Code; JCL, job control language.

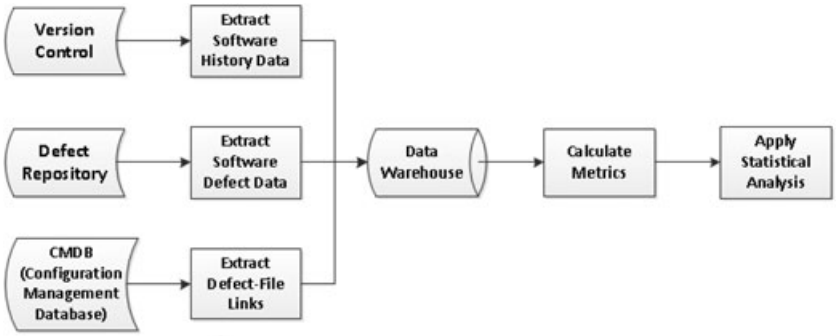


FIGURE 1 Data collection overview

3.2 | Data collection

We collected source code data from SVN and CA SCM VCSs, defect data from JIRA and in-house developed defect repositories, and the link between source code and defects from configuration management database (CMDB) at Company 1.

Figure 1 presents an overview of our approach to mine the data. We developed adapters for the 5 different data sources. The output of adapters containing the data retrieved from the data source for the specified period is stored in a database. For source code data, we fetch all versions created during the specified period in the VCS and store them in the database. We applied static code analysis on each file revision providing method-level and program-level static metrics (discussed in the next subsection). Commit information such as the developer ID that created the version, date of creation, and the related problem/request/project ID were available from the source code repository. We applied filtering to remove large commits that may have contained logically irrelevant changes. Commits containing more than 30 files were ignored and were not considered while calculating EC measures.

In CMDB, each software product is defined as a separate configuration item (CI) and each change is recorded and linked to the corresponding CI. In our study, we collected all source code–related changes performed in the scope of defect fixing or enhancement on the software product analysed over the defined period. In CMDB and JIRA, 2 different sources for a change were defined: Problem or Request. We could therefore distinguish between bug fixing and enhancement.

3.3 | Data sources

3.3.1 | Code repositories

Source code repositories are primarily used for storing and managing changes to source code artifacts. The full history of changes, the owner of the change, date of the change, and even the corresponding requirement or project task can be extracted. The tool used at Company 1 for managing the source code repository was a product of Computer Associates (CA), called CA Software Change Manager (CA SCM).[33] CA SCM provides change management in addition to its version control functionality. A developer must make changes in CA SCM in a change package, similar to the notion of "change set" in other SCM tools. A package groups and keeps all related changes together, corresponding to the same defect fix or enhancement. The versioning system used at Company 2 was Apache Subversion (SVN).[34] SVN is a popular versioning tool and is used by nearly half of all open source projects according to openhub.net.[†]

[†]The Open Hub tracks more than 650 000 free and open source software repositories and generates several statistics on the hosted source.

3.3.2 | Defect repositories: Company 1 (Finance)

We mined the defect repository to collect defect data reported. The defect repository at Company 1 was developed in-house by the company.

Mapping between defects and source code: We followed different approaches for the 2 companies for finding a mapping between defects and source code. For Company 1, we used the CMDB for this purpose. For both companies, we assumed that files involved in a defect fix contained the defect.

Configuration Management Database: Many companies store information related to components of their information system in a CMDB, which contains data describing the following entities[35]:

- managed resources such as computer systems and application software,
- process artifacts such as incident, problem and change records,
- relationships among managed resources and process artifacts.

In our study, we used the CMDB system at Company 1 to extract data about the relationship between MR and source code. The CMDB system was developed in-house by the company.

3.3.3 | Defect repository: Company 2 (Telecommunications)

Company 2 used JIRA,[36] a proprietary defect tracking product, developed by Atlassian.

Mapping between defects and source code: For Company 2, we used the defect IDs provided in the SVN commit comments by developers and the revision numbers provided in JIRA issues. For both companies, we assumed that files involved in a defect fix contained the defect. The percentage of fixed bugs linked to version control is changed between 73% and 79% yearly. The bugs that are not linked to version control include the defect fixes, which do not require source code change and version control commit such as database-related fixes. The mappings from SVN commit to defect and from JIRA issue to SVN commit were generally consistent.

3.4 | Descriptions of measures

Table 2 lists all the measures used in this study. The following sections will provide the details of these measures.

3.4.1 | EC measures

In the companies under study, any changes made to the source code were made based on MRs. An MR represents a conceptual software change, which includes modification of one or more source code files by one or more software developers. These changes can defect fixes or enhancements. We used an MR-based approach to calculate EC and formalise our approach as follows.

Let *MR* denote the set of MRs, *mr* denote a specific MR in *MR*, and *f* denote a source code file changed in the scope of *mr*. On the basis of these definitions, we calculate evolutionary coupled files and EC measures as follows:

The set of evolutionary coupled files of a file *f*:

$$SECF(f) = \{f_i | mr \in MR \wedge f_i \in mr \wedge f \in mr \wedge f_i \neq f\}$$

The total number of evolutionary coupled files of a file *f*:

$$NoECF(f) = |SECF(f)|$$

Set of evolutionary coupled files of a file *f* in the scope of a MR *mr*:

$$SECFMR(f, mr) = \{f_i | f_i \in mr \wedge f \in mr \wedge f_i \neq f\}$$

Sum of the number of evolutionary coupled files of a file *f* for all *mr*'s in *MR*:

$$NoECFMR(f) = \sum_{i=0}^{n} |SECFMR(f, mr_i)|$$

NoECF counts a coupling between 2 files as one even if they are coupled in the scope of multiple MRs. *NoECFMR* is different from *NoECF* in this respect. If 2 files are cochanged in the scope of 5 MRs, *NoECFMR* is calculated as 5, whereas *NoECF* is calculated as 1. *NoECFMR* considers the number of MRs, in which 2 files are coupled. We aim to use *NoECFMR* alongside *NoECF* to consider multiple MR cochanges, which may lead to stronger EC.

The following 3 issues were considered in the calculation of EC measures: (1) the level at which measures are taken, (2) the approach for grouping files, and (3) the boundary for finding coupled files. We calculated EC measures at file level; we chose file level, since defects are mapped to files in the companies under study. One approach for grouping file changes is using commit transactions of versioning systems that are the unique commit operations of a developer. In this approach, it is assumed that developers commit logically coupled files within a transaction. The system at Company 1 was a legacy system, and developers rarely committed more than 1 file in one transaction. Therefore, we found that a transaction-based approach was not appropriate to detect EC. We followed an MR-based approach and grouped the file changes according to the associated MR numbers.[8] In our approach, file changes spanning multiple transactions that were grouped together if they were associated with the same MR. The third issue considered for EC calculation was the boundary for finding coupled files. We chose module level to find coupled files that resided in the same module. We consider EC only within module boundaries. Alternative module boundaries could be subsystem or system level, which considers cross module couplings. In this study, we ignore any cross-module ECs.

3.4.2 | Size measures

Lines of Code was chosen for size measurement, and this is also used for normalising derived measures. We also used LOC to detect outliers in the data. To this end, we identified files whose size was greater than 10 K (0.4% of all files). These files were removed from the analysis as they were interpreted as outliers. Lines of Code is also used to investigate file size as a possible confounding factor. We check for correlation between LOC and other measures. Using defect density as normalised measure in our study mitigates the risk of size as a possible confounding factor.

TABLE 2 Summary of measures used in the study

		Description
NoECF	File level	The number of unique evolutionary coupled files of a file
NoECFMR	File level	Sum of the number of cochanged files (in the scope of an MR) of a file
LOC	File level	LOC, size measure
NoD	File level	Number of defects reported for a file
DD	File level	Defect density
NoCommits	File level	Number of commits—process metric for comparison purposes
NoDevs	File level	Number of developers—process metric for comparison purposes
TNF	Module level	Total number of files in a module
TNEFC	Module level	Total number of evolutionary file couplings in a module
TNFR	Module level	Total number of file revisions in a module
TNDVLP	Module level	Total number of developers contributing to a module
TND	Module level	Total number of defects of a module
TFSC	Module level	Total file size change in LOC for a module

Abbreviations: LOC indicates Lines of Code; MR, modification request.

We use the following measures for defects: number of defects reported for a file (NoD) and defect density (DD). We use the following formula for calculating defect density:

$$DD = NoD/LOC$$

3.4.3 | Defect types

We used the defect types listed in the Appendix (Table A1) and provide their descriptions. This defect type classification was used by Company 2 and each defect reported was tagged by one or more defect types (the defect repository stored the defect type data for each defect). The defect types in Table 13 are ordered on the basis of the defect type codes used by the company.

3.5 | Analysis method

3.5.1 | Analysis method for answering RQ1

Spearman correlation analysis was used to find the relationship between EC and defect measures. Since the data is not normally distributed, we apply Spearman rank correlation analysis. Spearman rank correlation analysis is a nonparametric test of correlation and assesses how well a monotonic function describes the association between variables. This is done by ranking the sample data separately for each variable. We used the Shapiro-Wilk test[37] to check for normality of the data. The null hypothesis of this test is that the population is normally distributed; if the p value is less than the chosen alpha level (.05), then the null hypothesis is rejected, and there is evidence that the data tested is not from a normally distributed population. Razali et al[38] report that Shapiro-Wilk is the most powerful normality test.

We set the p value (significance level) for Spearman correlation analysis to .05. If the data from the study results in a p value of less than .05, we conclude that the correlation is significant. The correlation coefficient or correlation strength is represented by ρ. It expresses the relationship between EC and software defects by a value between -1 and 1. ρ values of 1 or -1 indicate perfect positive or negative correlation, respectively. Values close to 0 indicate absence of correlation between measures. We considered ρ values less than 0.1 to be trivial, between 0.1 and 0.3 as low, between 0.3 and 0.5 as moderate, between 0.5 and 0.7 as high, between 0.7 and 0.9 as very high, and above 0.9 as almost perfect.[39,40]

Correlation analysis was applied on each module separately to obtain ρ, P and $StdErr$ values for each. We used histograms to summarise the correlation results and the SPSS[41] tool was used for the statistical analysis.

After correlation analysis was performed, we applied multivariate logistic regression and multicollinearity analysis with basic process metrics such as number of commits, number of developers, and prior number of defects as well as EC metrics. With this analysis, we are aiming to identify the relationship between metrics and metrics that do not add any new knowledge about defects.

The following describes the steps taken to build a logistic regression model for the EC metrics, process metrics, and the presence or absence of defects. The first step is to binarise the defect count such that a data point is labelled defective if the defect count is greater than 0. Then we build a logistic regression model using all terms and no interactions. Having built the model, we test for multicollinearity to find any independent variables, which are correlated. Then we build a model, which includes interaction terms and identify terms, which are correlated. Finally, we build an interaction model without correlated terms and apply stepwise reduction to remove terms, which are not significant.

By using regression models, we aim to determine whether a particular independent variable really affects the dependent variable and to estimate the magnitude of that effect, if any.

We diagnose collinearity through variance inflation factor (VIF) analysis.[42] We used 2.5 as the cutoff value for the simple model and 10 for the interaction model where collinearity naturally occurs by default. If a VIF value is greater than the cutoff value, the metric with the largest VIF is removed and the model rebuilt until all VIF values are less than the cutoff value.

3.5.2 | Analysis method for answering RQ2

We used box plots to determine differences between the modules where significant correlation was or was not observed. We drew box plots for the following measures:

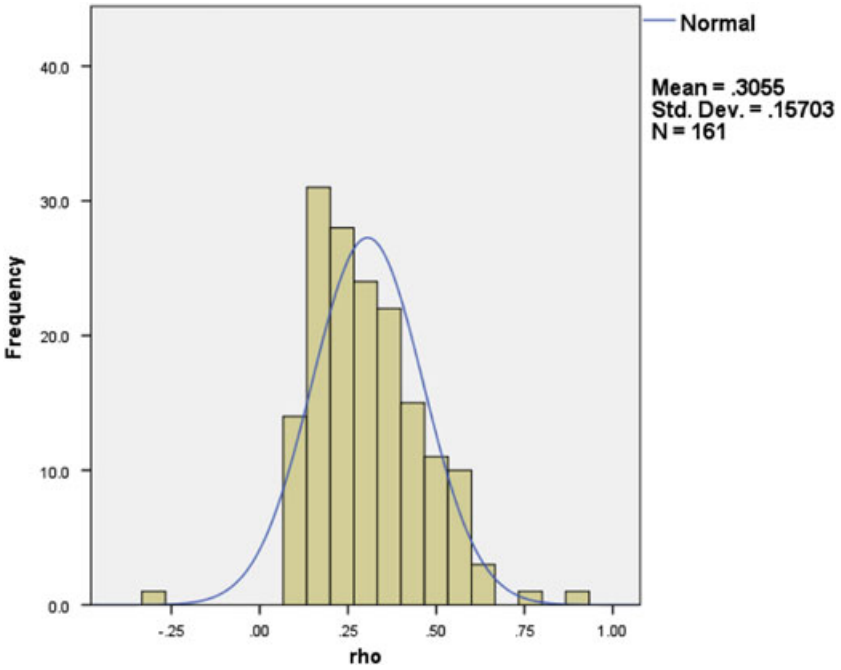


FIGURE 2 Company 1: Histogram of Spearman ρ values for correlation between evolutionary coupling (NoECF measure) and number of defects

- TNF: Total Number of Files in a Module
- TNEFC: Total Number of Evolutionary File Couplings in a Module
- TNFR: Total Number of File Revisions in a Module
- TNDVLP: Total Number of Developers contributing to a Module
- TND: Total Number of Defects of a Module
- TFSC: Total File Size Change in LOC for a Module

These measures were chosen based on availability and their power to reflect different attributes of modules characterising size, developer activity, and defects. Many studies in the literature suggest that size is generally an important factor. Since EC is dependent on developer activity, we have also added it as a factor. To check whether the difference is statistically significant, we apply a *t* test if data is parametric and a Mann-Whitney test if nonparametric. We again take a significance level of .05.

To check the role of defect types, we repeated the correlation analysis between EC and defect measures, but this time for each defect type. We aimed to find defect types that were likely to be related to EC, and we checked the distribution of defect types for each module.

4 | RESULTS

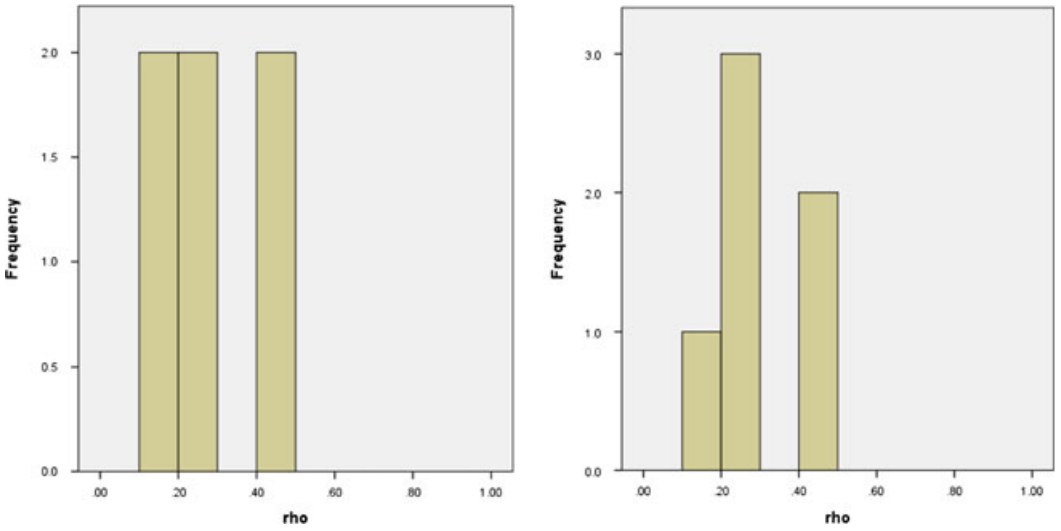
4.1 | RQ1: What is the relationship between EC and software defects? Correlation analysis results

For 161 of 274 (59%) software modules analysed at Company 1 and 6 of 11 at Company 2, we observed significant correlation ($p < .05$) between the NoD and EC measures using Spearman analysis. A Shapiro-Wilk test indicated that data distribution was not normal ($p = 0.0 < .05$) and so consequently, we used Spearman analysis. For 32 of 113 modules at Company 1 for which no significant correlation was observed, the number of commit values was either 0 or low values ($< = 10$). Evolutionary coupling measures need a lead period (Zimmerman et al[25]) and sufficient version control activity (prerequisite for EC measurement). Otherwise, they may not be useful. For Company 2, the modules for which no significant correlation is observed were all small and the number of defects was also low for these small modules.

The distribution of ρ values of these 161 modules at Company 1 can be seen in the histogram in Figure 2.[‡] The correlation observed was generally low and moderate. For 21 modules, high correlation was observed. Figure 3A,B shows the distribution of ρ values on the histogram for Company 2. The correlation values do not seem to be high but while interpreting these results, we need to consider that we are only analysing one factor among many, which can have a relationship with defects. From this perspective, having 59% of modules with significant correlation and low to moderate correlation strength is an important result.

If we compare the analysis results of the 2 companies, we observe that Company 2 has relatively fewer modules with high correlation values. The practices such as Agile and TDD used by Company 2 may have affected this result. Such practices may lead to lower coupling in systems. This result may also be due to the different architectures used by these 2 systems. Company 2 used the Model-View-Controller architectural pattern in its projects, which divides a software application into 3 interconnected parts, so as to separate internal representations of information from the ways that information is presented to, or accepted from, the user. Whereas the architecture in the Company 1 systems is more ad hoc since these legacy systems have been evolved over a long period. Organizational structure of the companies may also have impact on the design and coupling of the systems analysed as suggested by Conway law.[43] However, this should be investigated further.

[‡]This figure only shows the histogram of Spearman ρ values for correlation between NoECF and NoD. The histogram for correlation between NoECFMR and NoD is not shown in the main text, as it is very similar to the former one. However, it can be seen in Figure A4 in the Appendix.



(A) Correlation between NoECF and NoD (B) Correlation between NoECFMR and NoD

FIGURE 3 Company 2: Histogram of Spearman ρ values for correlation between evolutionary coupling measures and NoD

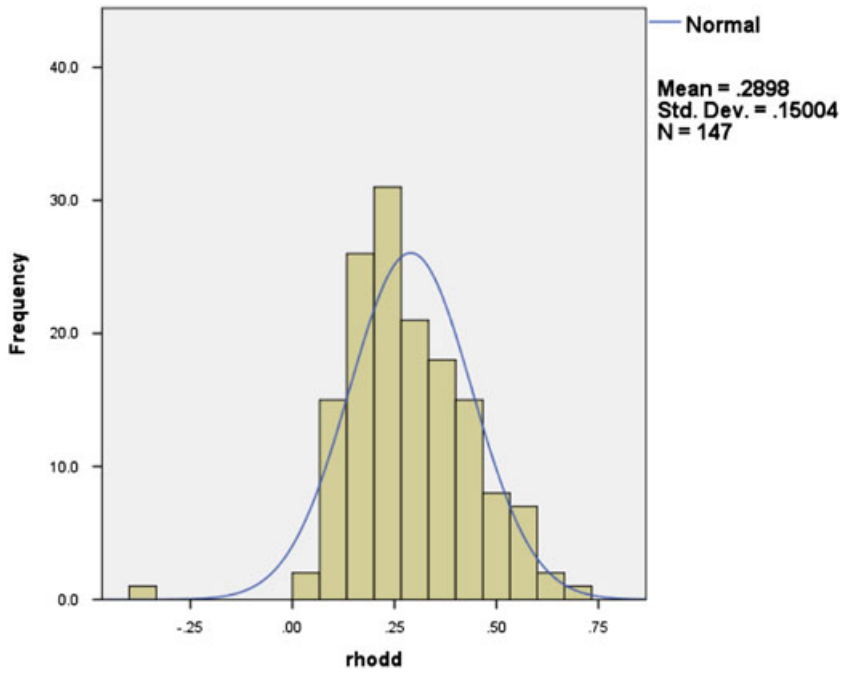


FIGURE 4 Company 1: Histogram of Spearman ρ values for correlation between evolutionary coupling (NoECF measure) and defect density

We also applied Spearman analysis for EC measures and DD. For 147 of 274 software modules analysed at Company 1, we observed significant correlation ($p < .05$) between DD and EC measures by using Spearman analysis. The distribution of ρ values can be seen in the histogram in Figure 4. Although there are slightly fewer modules identified as significant compared with the previous analysis, the distribution of ρ values shown in the 2 histograms shows great similarity. In keeping with the previous analysis results, the correlation observed was generally low and moderate; for a small number of modules, high correlation was observed. The results for Company 2 were similar to the previous analysis results.

We have also applied Spearman correlation analysis for basic process metrics such as number of commits, number of developers, and prior number of defects for comparison purposes. Table A3 summarises the results.

4.2 | RQ1: What is the relationship between EC and software defects? Regression analysis results

After correlation analysis, we applied multivariate logistic regression to build models, which indicate files which are likely to be defective. First, we built a logistic regression model using all terms and no interactions (Table 3).

Having built the model, we test for multicollinearity to find any independent variables, which are correlated (Table 4). We assess the VIF. A VIF > 2.5 is considered problematic requiring one or more variables

TABLE 3 First model with all terms and no interaction

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	− 2.9369	0.0122	− 240.71	0.0000
NoECFMR	0.0287	0.0048	5.95	0.0000
NoECF	0.0274	0.0056	4.88	0.0000
NoCommits	0.0213	0.0059	3.63	0.0003
NoDevs	0.8340	0.0250	33.41	0.0000

TABLE 4 Test for multicolinearity

	VIF
NoECFMR	21.29
NoECF	21.12
NoCommits	1.87
NoDevs	2.10

Abbreviation: VIF indicates variance inflation factor.

TABLE 5 Model for all Iindependent variables (IVs) without NoECFMR

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	− 2.9432	0.0122	− 241.67	0.0000
NoECF	0.0601	0.0014	44.02	0.0000
NoCommits	0.0247	0.0057	4.33	0.0000
NoDevs	0.8362	0.0247	33.88	0.0000

to be removed. "NoECFMR" and "NoECF" are identified as being correlated, and therefore, we remove "NoECFMR" from the model (Table 5). Multicollinearity analysis results and odds ratio (OR)[§] effect sizes after removing 'NoECFMR' are also provided in Table 6 respectively. The OR results suggest a rather low relation between EC and defects, although slightly higher than that of the number of commits.

Having identified individual variables, which make a significant contribution to the logistic regression model, we built a model that includes interaction terms (Table 7) and identify terms that are correlated (Table 8). Again, VIF values are highly likely to be correlated because we are using interaction terms; therefore, $VIF > 10$ is considered problematic (Table 8). Odds ratio effect sizes for this model are provided in Table 9.

Next, we built an interaction model without correlated terms and applied stepwise reduction to remove terms, which were not significant (Table 10). The multicollinearity analysis results and OR effect sizes for this model are also provided in Table 11. This analysis shows how unique the knowledge embedded in EC measures is compared to the other process metrics.

The final model includes the following significant terms: NoECF, NoCommits, NoDevs, and the interaction of NoECF with NoDevs. All terms apart from the interaction term are greater than 1.0 showing that when the independent variable increases, the propensity of a file to be defective increases. The interaction term (NoECF:NoDevs 0.98) is slightly less than 1.0 indicating that as both increase together, the linear model is adjusted to marginally decrease the increasing propensity of the model to predict a file as being defective.

[§]An odds ratio greater than 1.0 indicates that an increase in the variable will increase the propensity for the file to be defective.

TABLE 6 Multicolinearity and odds ratio (OR) effect size (without NoECFMR)

	VIF		OR
NoECF	1.29	(Intercept)	0.05
NoCommits	1.85	NoECF	1.06
NoDevs	2.09	NoCommits	1.03
		NoDevs	2.31

Abbreviation: VIF indicates variance inflation factor.

To check the relationship between EC measures and defect proneness of files from a different perspective, we drew box plots for EC measures of files with and without defects. A separate box plot for each module was created, and for some of the modules, these can be seen in Figure A2 in the Appendix (1: represents files with defects and 2: represents files without any defects).

We also performed manual analysis for some highly evolutionary coupled files and their defects to show how software defects were influenced by EC. In some defect instances, a highly evolutionary coupled file was changed, but this change was not accumulated to all coupled files correctly. This was the root cause of the fault. There was no structural or dynamic coupling between these files. We also observed similar instances but across different modules managed by different teams. A change made in a module was not accumulated to the evolutionary coupled modules. For some defect instances, a previous modification to a highly evolutionary coupled file caused some unanticipated behaviour in the coupled files.

4.3 | RQ2: What factors explain why the relationship between EC and software defects is different for different modules? Box plot analysis results

Figures 5, 6, and 7 show the box plots of module-level measures for modules where correlation is and is not detected. The y-axis of box plots is represented on a logarithmic scale, and the range of measurement values in Figure 6 (for Company 2) is perfectly separated. Although there is an overlap in the box plots for Company 1, the difference is statistically significant (< 0.05) for both companies according to the Mann-Whitney test. All modules for which correlation is observed have high values for total number of files, total number of evolutionary file couplings, and total number of file revisions. On the other hand, we did not observe a perfect separation of ranges for measures such as total number of developers contributed, total number of defects and total file size change in LOC for both companies. However, the difference is still statistically significant (< 0.05) according to the Mann-Whitney test. We also checked how balanced the set of modules were in defects with and without correlation, and they were mostly unbalanced. There are generally more files without defects than those with defects. We also analysed the relationship between module size and Spearman ρ values for the correlation between EC (NoECF measure) and number of defects. The results can be seen in Figure A5 and Table A5 in the Appendix. The correlation analysis showed a significant negative correlation ($p = .005 < 0.05$ and $\rho = -0.218$) between module size and ρ value.

TABLE 7 Model with interaction terms

	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	– 3.0191	0.0130	– 232.56	0.0000
NoECF	0.0817	0.0018	46.07	0.0000
NoCommits	0.0888	0.0116	7.63	0.0000
NoDevs	1.2002	0.0310	38.74	0.0000
NoECF:NoCommits	– 0.0021	0.0007	– 3.13	0.0017
NoECF:NoDevs	– 0.0356	0.0019	– 18.35	0.0000
NoCommits:NoDevs	– 0.0576	0.0064	– 8.94	0.0000
NoECF:NoCommits:NoDevs	0.0024	0.0003	7.40	0.0000

TABLE 8 Multicolinearity (with interaction terms)

	VIF
NoECF	2.52
NoCommits	8.20
NoDevs	3.79
NoECF:NoCommits	10.81
NoECF:NoDevs	7.34
NoCommits:NoDevs	10.21
NoECF:NoCommits:NoDevs	14.59

Abbreviation: VIF indicates variance inflation factor.

TABLE 9 Odds ratio (OR) effect size (with interaction terms)

	OR	2.5%	97.5%
(Intercept)	0.05	0.05	0.05
NoECF	1.09	1.08	1.09
NoCommits	1.09	1.07	1.12
NoDevs	3.32	3.12	3.53
NoECF:NoCommits	1.00	1.00	1.00
NoECF:NoDevs	0.96	0.96	0.97
NoCommits:NoDevs	0.94	0.93	0.96
NoECF:NoCommits:NoDevs	1.00	1.00	1.00

TABLE 10 Reduced model with interaction terms with no collinearity

	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	– 2.9878	0.0126	– 236.95	0.0000
NoECF	0.0747	0.0015	48.92	0.0000
NoCommits	0.0324	0.0054	5.95	0.0000
NoDevs	0.9823	0.0248	39.63	0.0000
NoECF:NoDevs	– 0.0184	0.0009	– 20.27	0.0000

TABLE 11 Multicolinearity and odds ratio (OR) effect size (final model) with confidence limits

	VIF		OR	2.5 %	97.5 %
NoECF	1.89	(Intercept)	0.05	0.05	0.05
NoCommits	1.94	NoECF	1.08	1.07	1.08
NoDevs	2.39	NoCommits	1.03	1.02	1.04
NoECF:NoDevs	2.40	NoDevs	2.67	2.54	2.80
		NoECF:NoDevs	0.98	0.98	0.98

Abbreviation: VIF indicates variance inflation factor.

4.4 | RQ2: What factors explain why the relationship between EC and software defects is different for different modules? Defect type analysis results

The results of correlation analysis for each defect type are summarised in Table 12. The columns of the table show the Spearman correlation strength (ρ values) between 2 EC measures and defect measures. Rows of the table represent different defect types. In the table, we include only the defect types that have at least 1 significant correlation result. Code Implementation has the highest correlation with EC, and moderate correlation was observed here. One interpretation is that developers tend to make coding errors while they work on source files, which are highly evolutionary coupled, and they should take into account more relations with more files when coding these files. For the defect types in the table, we observed low correlation, although they are significant. Defect types such as Acceptance Criteria and Analysis can be associated with external EC to other modules and applications. Involvement of more modules and applications may make analysis and defining acceptance test criteria more difficult. We can interpret the correlation with Test Implementation type in a similar way to Code Implementation. We have checked the defects of Not An Issue type with the project members. They explained that this defect type was generally used for deployment problems. Correlation between defects of this defect type and EC may be explained as that deploying highly evolutionary coupled files and modules may be more error-prone due to more dependencies to be considered and deployed together.

For the following defect types, correlation values observed were trivial and are therefore ignored: Wrong Properties, De defect Value, Process Failure, Database Upgrade Failure, Data Fix Errata, Unexpected Functionality, Incorrect Config, Infrastructure Issues, Missing or Incomplete Data Migration, and Acceptance Criteria Impl.

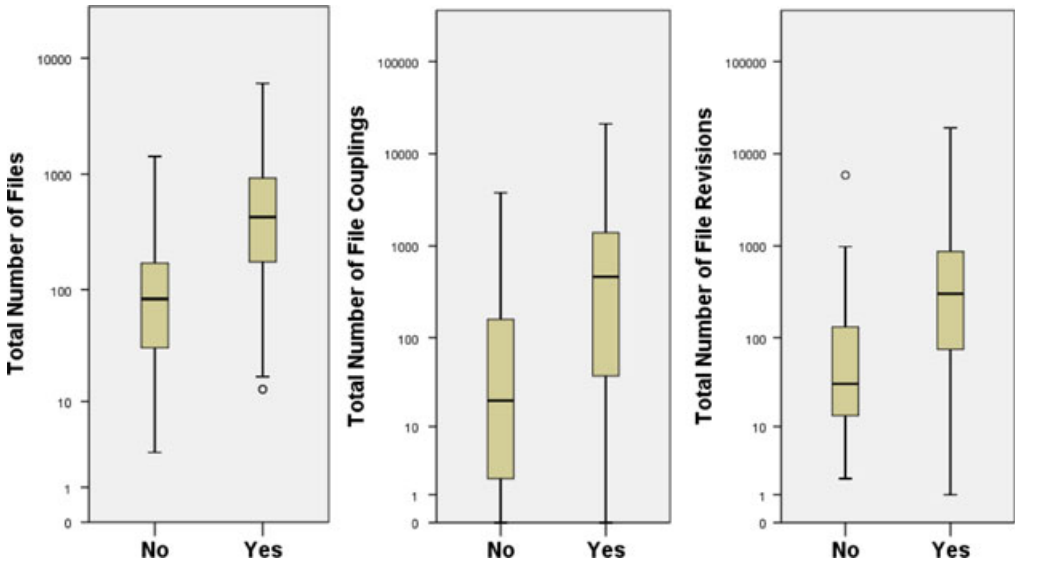
For the following defect types, no significant correlation was detected: Incorrect Environment, CRM Bug, User Error, and Database Disconnect-Reconnect Error.

5 | DISCUSSION

Our findings give insights to future researchers and practitioners on the effect of EC on defects.

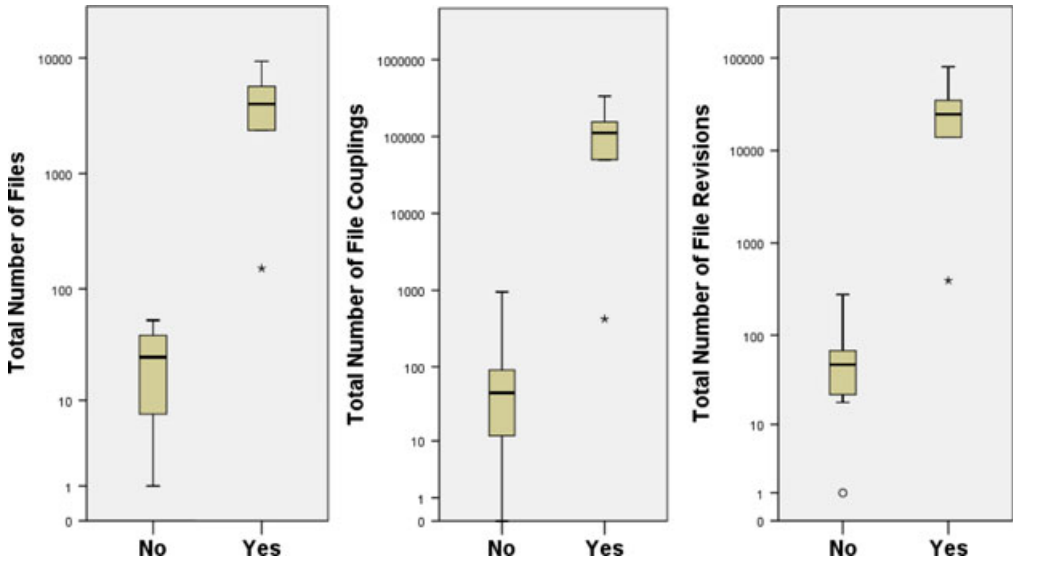
(RQ1) What is the relationship between EC and software defects?

Our results suggest that there is, in general, a significant positive correlation between EC measures and defects. This finding is consistent



(A) Total Number of Files (B) Total Number of File Couplings (C) Total Number of File Revisions

FIGURE 5 Company 1: box plots of different measures for modules in which correlation between EC and defects detected (Yes) and not detected (No)

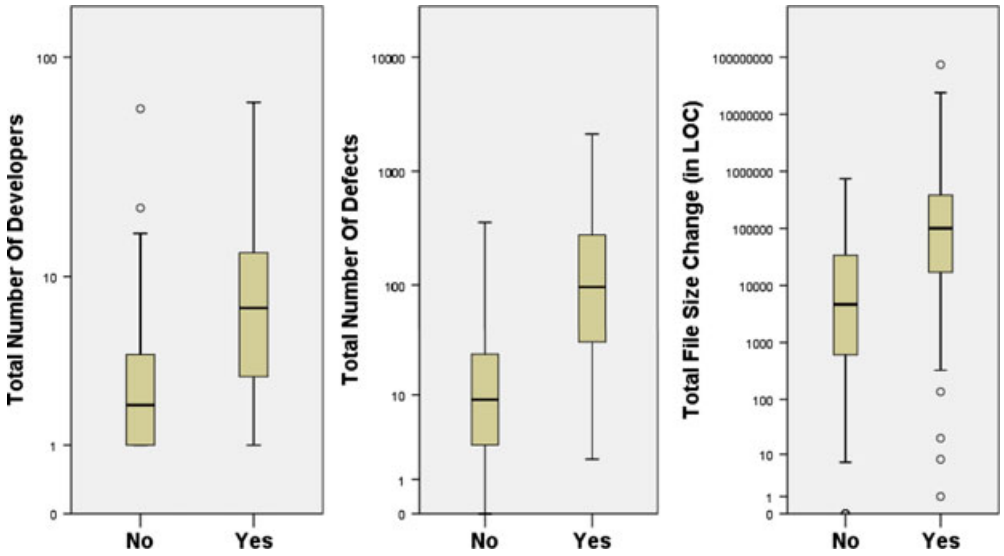


(A) Total Number of Files (B) Total Number of File Couplings (C) Total Number of File Revisions

FIGURE 6 Company 2: box plots of different measures for modules in which correlation between EC and defects detected (Yes) and not detected (No)

with the general opinion that low coupling is an important principle to follow for a high-quality software design and that high coupling can be related to defects.[12,44,45] Fewer interconnections between elements reduce the chance that changes in one element cause problems in other elements. Fewer interconnections between elements are also reported to reduce programmer time.[46] It is essential to keep the effect of a change in one element on another element low. However, our study shows that correlation strength between EC and defects varies across modules. Correlation strength had a wide range of values from 0 to 0.8. Furthermore, there are also modules in which EC and software defects are not correlated. This is an important finding, since it highlights that the effect of EC is likely to vary depending on the module analysed. It is likely that the context of each module affects the risk that making change will create unanticipated changes within other elements. A change made in source code may have different manifestations on defects based on the module context (eg, development process characteristics). Some modules carry higher risk than others. Therefore, it is important to consider the EC-defect relationship in the context of related modules.

The contradictory findings reported by previous EC studies such as Graves et al[8] and Knab et al[20] may be partially explained in the different systems and modules used in these studies. As shown by



(A) Total Number of Developers (B) Total Number of Defects (C) Total File Size Change (in LOC)

FIGURE 7 Company 1: box plots of different measures for modules in which correlation between EC and defects detected (Yes) and not detected (No)

existing studies,[47] the context has an important impact on studies. We have shown that EC has different effects on defects across different modules, because EC seems to manifest differently in different modules. The characteristics of the modules in individual systems must be accounted for in future studies, as suggested by Hall et al.[47] This finding also provides practitioners with valuable information on detecting defects and problematic code hot spots. However, as for all other code measures in relation to defects, EC does not contribute to defects equally for every module in a system, so EC use is not consistently helpful. We recommend that practitioners use EC for assessing the quality of their software design but also in conjunction with other module characteristics. That way, practitioners will get the best of both worlds.

TABLE 12 Spearman correlation analysis results between evolutionary coupling measures and different defect types (Appendix 9.1 provides more details of these defect types)

	NoECFMR vs NoD (Spearman Corr.)	NoECF vs NoD (Spearman Corr.)
Code implementation	$\rho = .176^{*}$	$\rho = .182^{*}$
	p = .000	p = .000
Acceptance criteria	$\rho = .111^{*}$	$\rho = .113^{*}$
	p = .000	p = .000
Functionality not implemented yet	$\rho = .088^{*}$	$\rho = .091^{*}$
	p = .000	p = .000
Analysis	$\rho = .083^{*}$	$\rho = .085^{*}$
	p = .000	p = .000
Not an issue	$\rho = .052^{*}$	$\rho = .045^{*}$
	p = .000	p = .000
Test implementation	$\rho = .060^{*}$	$\rho = .061^{*}$
	p = .000	p = .000
3rd party system defect	$\rho = .047^{*}$	$\rho = .045^{*}$
	p = .000	p = .000
Bad data	$\rho = .091^{*}$	$\rho = .093^{*}$
	p = .000	p = .000

*Correlation is significant at the 0.01 level (2-tailed).

(RQ2) What factors explain why the relationship between EC and software defects is different for different modules?

We also tried to explain possible reasons for the different effects of EC on software defects. We considered this issue from 2 perspectives: module characteristics and defect types. We found that EC was less likely to have an effect on software defects for modules with fewer files and where fewer developers contributed. This may be explained by fewer defects being caused by EC in relatively small modules. Potentially, there are fewer interconnections between elements in a small module. Let n denote the number of files in a module. The potential number of interconnections in a module is calculated as $\frac{n*(n-1)}{2} = \frac{n^2-n}{2}$. Interconnections between files in a module can grow quadratically with the number of files. The more interrelated the files are, the more difficult these modules are to understand, change, and correct and thus the more complex the resulting software system. This may eventually lead to defects. An alternative explanation at least for the nondensity models would be that such files typically have fewer defects.

We also recommend that practitioners add EC measures to their metric suite for software design evaluation. We recommend that researchers report process and size metrics of modules in their EC studies to account for the possible effect of context in their results. Furthermore, we found that EC may be more related to some defect types such as Code Implementation, Acceptance Criteria, and Test Implementation and less related to others such as Unexpected Functionality, Infrastructure Issues, Missing or Incomplete Data Migration, Incorrect Environment, and User Error.

We believe that defect types may be used to explain the contradictory findings reported by previous EC studies in the literature. The different systems and modules used in these studies have different defect types ,and EC has different relationships with different defect types. It is more likely that high EC will cause Code Implementation and Test Implementation defects, because a high number of changes must be

made to related parts of the system when code with high EC is changed. The locations of these related changes may be scattered within the application or even across applications in a software ecosystem; making related changes across these locations is likely to be challenging, and this can increase the cognitive load of developers.[48] Moreover, developers may miss some locations, which should be cochanged, and this may cause unforeseen code and test implementation problems. On the other hand, EC is unlikely to contribute to defects whose root cause is user error or infrastructure issues. If a module has defects caused mostly by user error or infrastructure issues, EC measures will not be useful for detecting defects and hot spots.

6 | THREATS TO VALIDITY

6.1 | Construct validity

Threats to construct validity relate to whether we measure what we intend to measure. When calculating EC measures, there are 2 ways in which to group file revisions in the source code repository: MR-based and transaction-based. Evolutionary coupling measures calculated on a transaction basis for the system under study that do not reflect the coupling relations between files; therefore, we preferred an MR-based approach to their calculation. The reason is that changes for a single MR were frequently split across multiple commit transactions for the systems under study. In contrast to open source systems previously analysed, in this study, we had a good defect linking that enabled us to use an MR-based approach.

Another threat is the potential overlap in knowledge of EC with existing process metrics. To mitigate this threat, we applied multivariate regression and multicollinearity analysis to understand the overlap and the unique knowledge embedded in EC measures.

The EC metrics used in our study do not consider the age and temporal aspects of EC. Evolutionary couplings that are temporary or no longer valid due to refactorings or restructurings could not be detected in our study. This is a limitation of the study.

We assume that any change made to source code is committed to code repositories. The software processes in both companies place check points (at compilation, moving to test/production) to guarantee this assumption. Practitioners should be careful in using EC measures, because they may not be reliable for some modules if VCSs are not used by their developers (or there is a low utilisation of VCS). Evolutionary coupling measures make sense if the VCS is used long enough and consistently. In this study, some modules had low utilisation for VCS, and this may be an indication of problems with VCS adaptation for some projects in the company. Variance inflation factor was introduced to some projects at Company 1 a few years ago. As a consequence, we recommend excluding these types of modules or systems when calculating EC measures and using EC measures in defect models.

All the files committed in a particular commit operation might not be logically coupled. This threat is mitigated by ignoring commit transactions having more than 30 files. Therefore large transactions, which may possibly include files from more than 1 MR (eg, the merge of a branch), are not used to calculate EC. Furthermore, we also performed a manual analysis of randomly chosen commits and MRs and checked the validity of this assumption. When huge commit transactions are removed, there are very few exceptions to this assumption. Another point is that there are differences between industrial and open source software development regarding this assumption. Companies generally place controls on MRs in the application life cycle (eg, mandatory MR numbers during check-in, allowing only files associated with an MR to move to the production, etc), and companies usually rigorously follow such conventions, unlike many open source projects.[49]

6.2 | Internal validity

In this study, we used CIs from the CMDB (for Company 1 only) attached to problem records and related requests (move to production, code review, move to test, etc) with which to match defects to source code files. Two assumptions were made at this stage:

1. Configuration items defined at CMDB correspond to source files changed in the scope of the resolution of a defect.
2. Configuration items of the source files changed in the scope of the resolution of a defect are linked to the problem record of the defect.

The validity of these 2 assumptions can be guaranteed for certain record types (move to production and code review), but in general, these cannot be guaranteed, that for each defect, all related source files are detected.

Another assumption is that developers commit source files changed in the scope of the same MR to the same package in the code repository. This assumption is used in the calculation of EC measures. We rely on the data collected from versioning systems, and any project, which is not managed in the versioning system (or any file which is not committed to versioning systems), is not considered in our study.

The measures and defect types chosen for answering RQ2 are not exhaustive and do not cover all characteristics of a module and all defect types, which can exist. An exhaustive examination may have revealed other factors that have a greater effect on defects and which may be confounding our results. In our study, we investigated file size (LOC) as a possible confounding factor. We observed that code size correlated with number of defects in some modules. Defect density however had either no significant correlation or only minor negative correlation. Using defect density in our study mitigates the risk of size as a possible confounding factor. We are planning future investigations to explore the effect size of a large number of factors related to defects.

6.3 | External validity

External validity relates to the generalisation of our study results. We only studied 2 industrial software systems. These systems may not be representative of the way developers develop systems more generally. We mitigate this risk by choosing 2 systems from different domains and with different technologies. In future work, we would like to extend this study by including more commercial systems and projects.

7 | CONCLUSIONS

In this paper, we presented a study on the relationship between EC and software defects in 2 large industrial software systems. We reported a positive correlation between EC and defect measures in the software maintenance/evolution phase of systems from 2 different companies. Our results indicated low-level, moderate-level, and high-level correlation, with varied correlation strength across modules. Our regression analysis results indicated that EC measures could be useful for explaining defects.

The box plots drawn for each module separately showed the potential of EC measures to distinguish defective and nondefective files. We also observed that the company using practices such as Agile and TDD had relatively fewer modules with high EC-defect correlation values. However, this finding needs to be further investigated on more companies for generalisable conclusions.

We also tried to understand the reasons for variation of the observed effect of EC on software defects for different modules. We found that modules, which were small in file and developer numbers, tended to be less correlated with EC. Interconnections between files in a module can grow quadratically with the number of files. The more interrelated the files are, the more difficult these modules are to understand, change, and correct and thus the more complex the resulting software system. This complexity may eventually lead to defects, and this may be one of the reasons for variation across modules. Furthermore, we observed that EC measures showed higher correlation with some types of defects (based on root causes) such as code implementation, acceptance criteria, and analysis problems. The dispersion of these defect types could be another reason for these varying effects. Different modules have different defect types, and EC has different relationships with different defect types.

Module characteristics and defect types may also explain why different results are reported by different studies in the literature. Different applications or modules analysed may have different characteristics and defect types. We recommend that researchers report characteristics and defect types of modules in their EC studies to account for the possible effect of the context in their results. We also recommend that practitioners add EC measures to their metric suite for software design evaluation and consider the characteristics and defect types of their modules in their evaluation.

ACKNOWLEDGMENTS

We would like to thank the Scientific and Technological Research Council of Turkey (TUBITAK) for its financial support (B.14.2.TBT.0.06.01-214-115535). This research was supported in part by Bogazici University Research Fund (7223) and the Turkish Academy of Sciences and by Engineering and Physical Sciences Research Council (EPSRC) of the UK (EP/L011751/1). Dr. Bener and Dr. Caglayan are supported by NSERC Discovery grant 402003-2012. We would also like to thank Thomas Shippey for his contribution on data cleaning and analysis.

REFERENCES

1. Lehman MM. Programs, life cycles, and laws of software evolution. *Proc IEEE*. 1980;68(9):1060–1076.
2. Bennett KH, Rajlich VT. Software maintenance and evolution: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*. ACM, Limerick, Ireland; 2000:73–87.
3. Mens T. *Introduction and Roadmap: History and Challenges of Software Evolution*. Springer Berlin Heidelberg; 2008.
4. Visser J. Change is the constant. *ERCIM News*. 2012;2012(88)3–3.
5. Śliwerski J, Zimmermann T, Zeller A. When do changes induce fixes?*ACM sigsoft Software Engineering Notes*. 2005;30(4):1–5.
6. Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *ACM/IEEE 30th Internatinal Conference on Software Engineering, 2008. ICSE' 08*. IEEE, Leipzig, Germany; 2008:181–190.
7. Nagappan N, Zeller A, Zimmermann T, Herzig K, Murphy B. Change bursts as defect predictors. *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, San Jose, CA, USA; November 2010:309–318.
8. Graves TL, Karr AF, Marron JS, Siy H. Predicting fault incidence using software change history. *IEEE Trans Softw Eng*. 2000;26(7):653–661.
9. Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*. IEEE, St. Louis, MO, USA; 2005:284–292.
10. Shin Y, Bell R, Ostrand T, Weyuker E. Does calling structure information improve the accuracy of fault prediction? *6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09*, Vancouver, BC, Canada; May 2009:61–70.
11. Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08*, Leipzig, Germany; May 2008:181–190.
12. Briand LC, Daly JW, Wust JK. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans Softw Eng*. 1999;25(1):91–121.
13. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. Using information retrieval based coupling measures for impact analysis. *Empirical Softw Eng*. 2009;14(1):5–32.
14. Arisholm E, Briand LC, Foyen A. Dynamic coupling measurement for object-oriented software. *IEEE Trans Softw Eng*. 2004;30(8):491–506.
15. Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. *International Conference on Software Maintenance, 1998. Proceedings*. IEEE, Bethesda, MD, USA; 1998:190–198.
16. Gall H, Jazayeri M, Krajewski J. CVS release history data for detecting logical couplings. *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings*. IEEE, Helsinki, Finland; 2003:13–23.
17. Breu S, Zimmermann T. Mining aspects from version history. *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE'06*. IEEE, Tokyo, Japan; 2006:221–230.
18. Eaddy M, Zimmermann T, Sherwood KD, et al. Do crosscutting concerns cause defects? *IEEE Trans Softw Eng*. 2008;34(4):497–515.
19. Kirbas S, Sen A, Caglayan B, Bener A, Mahmutogullari R. The effect of evolutionary coupling on software defects: an industrial case study on a legacy system. *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14. ACM, Torino, Italy; 2014:6:1–6:7.
20. Knab P, Pinzger M, Bernstein A. Predicting defect densities in source code files with decision tree learners. *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, Shanghai, China; 2006:119–125.
21. Ball T, Kim JM, Porter AA, Siy HP. If your version control system could talk. *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, Boston, Massachusetts, USA; 1997.
22. Pinzger M, Gall H, Fischer M, Lanza M. Visualizing multiple evolution metrics. *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, New York, NY, USA; 2005:67–75.

23. Beyer D, Hassan AE. Animated visualization of software history using evolution storyboards. *13th Working Conference on Reverse Engineering, 2006. WCRE'06*. IEEE, Benevento, Italy; 2006:199–210.
24. Ying AT, Murphy GC, Ng R, Chu-Carroll MC. Predicting source code changes by mining change history. *IEEE Trans Softw Eng*. 2004;30(9):574–586.
25. Zimmermann T, Zeller A, Weissgerber P, Diehl S. Mining version histories to guide software changes. *IEEE Trans Softw Eng*. 2005;31(6):429–445.
26. Adams B, Jiang ZM, Hassan AE. Identifying crosscutting concerns using historical code changes. *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, vol. 1. ACM, Cape Town, South Africa; 2010:305–314.
27. Kirbas S, Hall T, Sen A. Evolutionary coupling measurement: making sense of the current chaos. *Sci Comput Program*. 2016;135:4-19 Special Issue on Advances in Software Measurement.
28. Steff M, Russo B. Co-evolution of logical couplings and commits for defect estimation. *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, Zurich, Switzerland; 2012:213–216.
29. Tantithamthavorn C, Ihara A, Matsumoto KI. Using co-change histories to improve bug localization performance. *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, Honolulu, HI, USA; 2013:543–548.
30. D'Ambros M, Lanza M, Robbes R. On the relationship between change coupling and software defects. *16th Working Conference on Reverse Engineering, 2009. WCRE'09*. IEEE, Lille, France; 2009:135–144.
31. Kouroshfar E. Studying the effect of co-change dispersion on software quality. *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA; 2013:1450–1452.
32. Kirbas S, Sen A, Caglayan B, Bener A. Değişiklik bağlaşımı ve yazılım hataları ilişkisinin incelenmesi. *Proceedings of the 8th Turkish National Software Engineering Symposium*, Güzelyurt, KKTC, Turkey; September 8-10, 2014:419-430.
33. CASCM. Web page of ca software change manager. 2013.
34. SVN. Web page of apache subversion. 2015.
35. DMTF Configuration management database (cmdb) federation specification - dsp0252 1.0.1. April 2010.
36. JIRA. Web page of jira; 2015.
37. Shapiro SS, Wilk MB. An analysis of variance test for normality (complete samples). *Biometrika*. 1965:52(3-4):591–611.
38. Razali NM, Wah YB. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *J Stat Model Analytics*. 2011;2(1):21–33.
39. WG H. A new view of statistics SportScience; 2003.
40. Lu H, Zhou Y, Xu B, Leung H, Chen L. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Softw Eng*. 2012;17:200–242.
41. SPSS. Web page of spss. 2s013.
42. Montgomery DC, Peck EA, Vining GG. *Introduction to Linear Regression Analysis*, vol. 821: John Wiley & Sons, Hoboken, New Jersey; 2012 .
43. Conway ME. How do committees invent. *Datamation*. 1968;14(4): 28–31.
44. Briand LC, Wüst J, Daly JW, Porter DV. Exploring the relationships between design measures and software quality in object-oriented systems. *J Syst Softw*. 2000;51(3):245–273.
45. Baldwin CY, Clark KB. *Design Rules: The Power of Modularity*. Cambridge, Massachusetts, US, vol. 1: Mit Press; 2000.
46. Harrold MJ, Kolte P. A software metric system for module coupling. *J Syst Softw*. 2003;20(3):295–308.
47. Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng*. 2012;38(6):1276–1304.
48. Sweller J. Cognitive load during problem solving: effects on learning. *Cognitive Sci*. 1988;12(2):257–285.
49. Herzig K, Zeller A. The impact of tangled code changes. *Proceedings of the 10th Working Conference on Mining Software Repositories*; San Francisco, CA, USA, 2013:121–130.

SUPPORTING INFORMATION

Additional Supporting Information may be found online in the supporting information tab for this article.

How to cite this article: Kirbas S, Caglayan B, Hall T, et al. The relationship between evolutionary coupling and defects in large industrial software. *J Softw Evol Proc*. 2017;29:e1842. https://doi.org/10.1002/smr.1842

APPENDIX A

A.1 Defect root causes list and details

TABLE A1 Defect type list

	Defect Root Causes	Descriptions
1	Bad data	defects caused by invalid / unexpected data at persistence storage (databases)
2	Wrong properties	defects caused by properties set incorrectly for the application like timeout, thread pool size, etc.
3	Default value	defects caused by the de defect values of the deployed application
4	Process failure	defects caused by problems in software processes such as insufficient communication between teams
5	3rd party system defect	defects caused by problems occurred at other systems running in the same environment
6	Database upgrade failure	defects caused by incomplete/unsuccessful database schema updates / migrations
7	Data fix errata	defects caused by incorrect scripts that are added to fix the data at persistence storage as part of another defect
8	CRM defect	defects caused by problems at Customer Relationship Management (CRM) system
9	Functionality not implemented yet	defects caused by API methods or functionalities which are not implemented yet or not deployed with the existing software version
10	Unexpected functionality	defects experienced by users as an unexpected functionality such as response failures and performance problems
11	Incorrect environment	defects caused by non-satisfied prerequisites at the running environment of the application
12	Acceptance criteria impl	defects caused by missing / incomplete / incorrect automated acceptance tests
13	Database disconnect/reconnect error	defects caused by database (persistence storage) connection problems
14	Analysis	defects caused by missing / incomplete/ incorrect requirements / user stories
15	Incorrect config	defects caused by incorrect application configuration such as versions of feeds / services pointed
16	Infrastructure issues	defects caused by application infrastructure problems such as exhausted server swap memory or incorrect load balancing
17	Acceptance criteria	defects caused by missing / incomplete / incorrect acceptance criteria
18	Missing or incomplete data migration	defect s caused by incomplete / missing data migrations affecting a specific environment (test, qa, etc.)
19	User error	defects or unexpected behaviour due to a invalid usage of the application
20	Not an issue	defects which are not interpreted as defects (not supported scenario, no longer required behaviour, already fixed, etc.)
21	Test implementation	defects caused by missing / incomplete / incorrect automated (unit / integration) tests
22	Code implementation	defects caused by the defects inserted during implementation of new features or defect fixing

A.1.1 Defect root causes-frequencies:

Code Implementation and Not An Issue were the 2 most popular defect types, ≈%28 and ≈%20, respectively. The following defect types followed these 2 defect types: Incorrect Config (%5.2), Functionality Not Implemented Yet (%4.1), Bad Data (%3.7), 3rd Party System Defect (%3.4), Analysis (%3.2), Default Value (%2.9), Unexpected Functionality (%2.3), Wrong Properties (%2.1), and Acceptance Criteria (%2.0).

A.2 Box plots

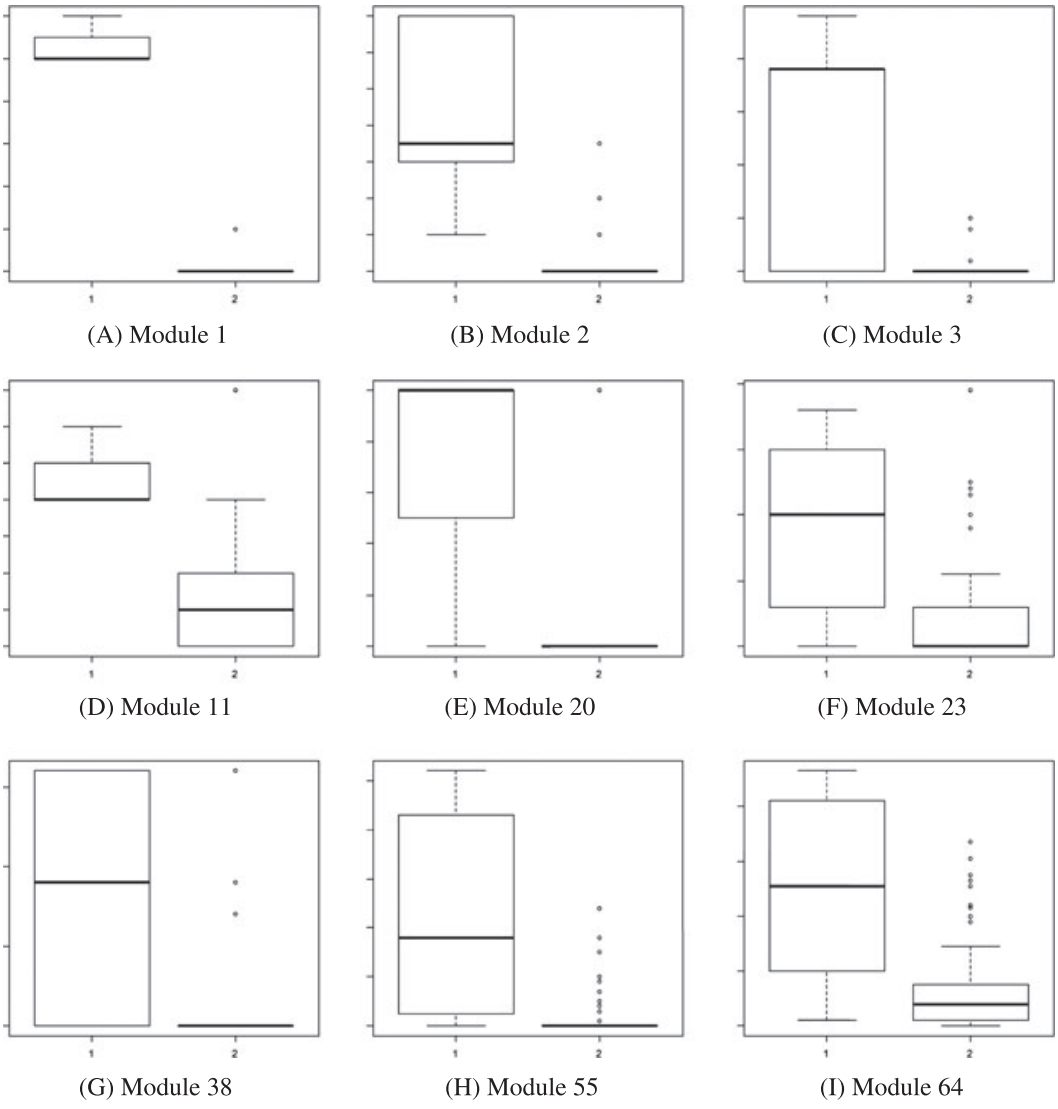


FIGURE A2 Box plots of evolutionary coupling measures for selected modules—files with defects vs files without defects: (1) represents files with defects and (2) represents files without any defects. Y-axes of box plots are removed to prevent revealing sensitive company data

A.3 Correlations analysis of the main variables

TABLE A3 Spearman correlation results of the process metrics

	NoECF	NoECFMR	NoD	NoCommits
NoECFMR	$\rho = .99$ p = .000			
NoD	$\rho = .28$ p = .000	$\rho = .28$ p = .000		
NoCommits	$\rho = 0.57$ p = .00	$\rho = 0.57$ p = .00	$\rho = .24$ p = .00	
NoDevs	$\rho = 0.57$ p = .00	$\rho = 0.57$ p = .00	$\rho = .24$ p = .000	$\rho = 0.99$ p = .00

A.4 Histogram for correlation between NoECFMR measure and number of defects

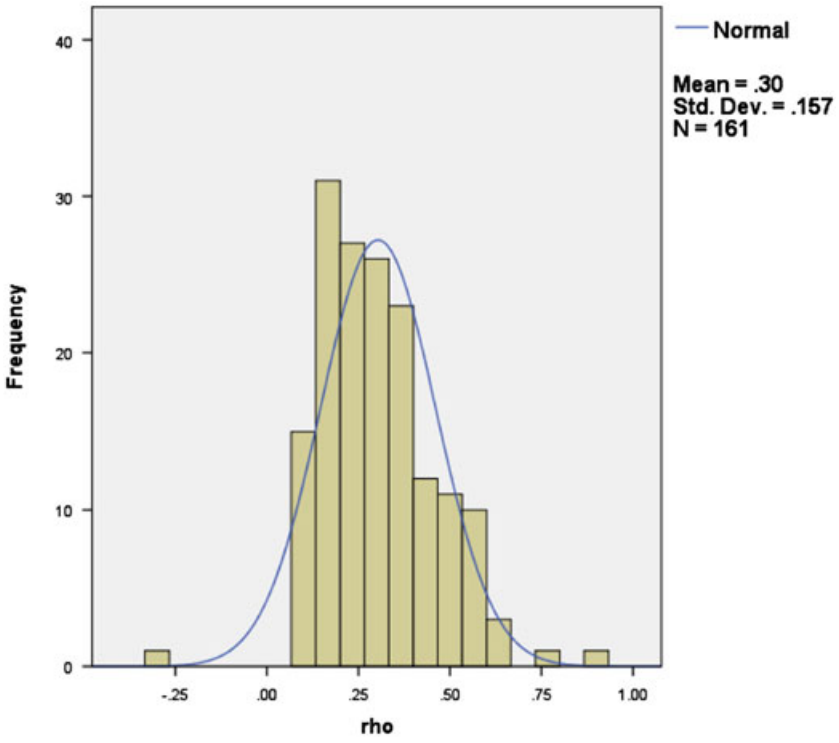


FIGURE A4 Company 1: histogram of Spearman ρ values for correlation between EC (NoECFMR measure) and number of defects

A.5 Module size vs ρ values of EC-defect correlation

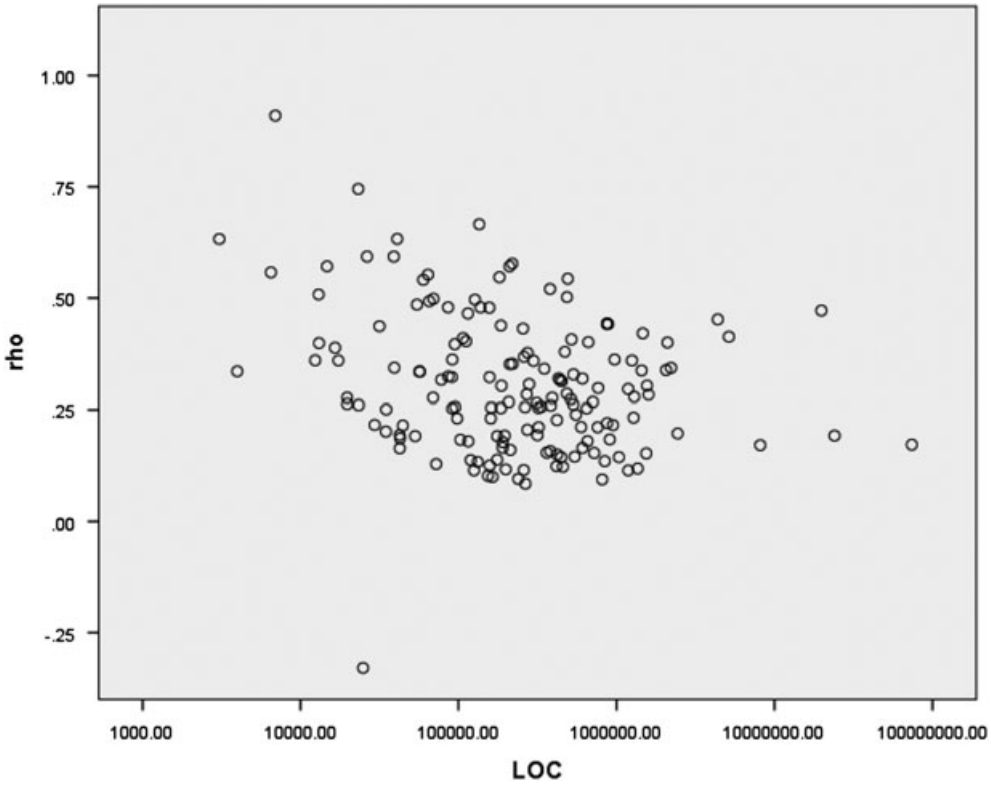


FIGURE A5 Module Size vs ρ values of EC-defect correlation. LOC indicates Lines of Code

TABLE A5 Spearman correlation results

	LOC
rho	$\rho = -0.218^*$
	p = .005

Abbreviation: LOC indicates Lines of Code.