IET Software

Research Article

IET Journals
The Institution of Engineering and Technology

Software structure evolution and relation to subgraph defectiveness

ISSN 1751-8806
Received on 2nd March 2018
Revised 3rd October 2018
Accepted on 16th November 2018
E-First on 27th February 2019
doi: 10.1049/iet-sen.2018.5060
www.ietdl.org

Ana Vranković[1] ✉, Tihana Galinac Grbac[1], Željka Car[2]

[1]*Faculty of Engineering, University of Rijeka, Vukovarska 58, HR-51000 Rijeka, Croatia*
[2]*Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, Zagreb, Croatia*
✉ *E-mail: avrankovic@riteh.hr*

Abstract: Network analysis has been successfully applied in software engineering to understand structural effects in the software. System software is represented as a network graph, and network metrics are used to analyse system quality. This study is motivated by a previous study, which represents the software structure as three-node subgraphs and empirically identifies that software structure continuously evolves over system releases. Here, the authors extend the previous study to analyse the relation of structural evolution and the defectiveness of subgraphs in the software network graph. This study investigates the behaviour of subgraph defects through software evolution and their impact on system defectiveness. Statistical methods were used to study subgraph defectiveness across versions of the systems and across subgraph types. The authors conclude that software versions have similar behaviours in terms of average subgraph type defectiveness and subgraph frequency distributions. However, different subgraph types have different defectiveness distributions. Based on these conclusions, the authors motivate the use of subgraph-based software representation in defect predictions and software modelling. These promising findings contribute to the further development of the software engineering discipline and help software developers and quality management in terms of better modelling and focusing their testing efforts within the code structure represented by subgraphs.

1 Introduction

One of the most challenging software engineering tasks is to produce and deliver fault-free software products. Evolving complex software systems (EVOSOFT) have become a central part of a rapidly growing range of applications, products, and services supporting daily human activities from all economic sectors. Still, people are the key element in software production, and human mistakes are the main source of software faults. Good and reliable techniques for software fault prevention and early detection are one of the major concerns of organisations developing such systems. Major project cost sharing involves verification activities aimed at delivering reliable software products.

One area of research in that direction is software defect prediction (SDP), which aims to predict fault-prone locations in the software code. Fault distributions in complex software systems are analysed in the context of structural abstractions of the software system. There are numerous possible structural abstractions in which we may observe large-scale complex software systems. Thus far, fault distributions in large-scale complex software systems have been analysed based on structural abstraction using main building elements such as modules, files, classes, packages, or software units. It has been identified that the majority of faults are located in a minority of system modules/classes and that this minority of system modules/classes constitutes a minority of system share [1–6]. In these studies, there are some indications that the most faulty system units are the most communication-intensive units and that there exists a principle for fault distribution over the system structure. That fact motivated us to analyse the software structure as a communicating network graph and to apply network analysis in order to observe the patterns inside the system units communication as the software structure evolves. The main building system units (files, classes, packages, and software units) may be represented as nodes, and communication dependencies among them, as links between these nodes.

Recent findings have shown that the network analysis approach to software evolution and defect prediction is a very promising area of research. A network graph is a structural system property that is defined independently of static code properties or the software development properties and can be defined for any software code. Owing to the fact that, it provides an important tool for cross-company, cross-product, and cross-software development phase comparisons of software structures no matter which programming language is used and which development process is followed. Therefore, this additional system property may be used to observe the effects of other system properties.

One part of network analysis is the exploration of network structures with the help of significant network substructures called motifs, aiming to uncover structural design principles in complex networks. Network motifs, proposed by Milo *et al*. [7], are patterns of interconnections in complex networks with an occurrence that is statistically higher than in random networks. It is identified that motifs could be useful for characterising universal classes of different complex network structures in different scientific fields such as medicine, sociology, and electrical engineering (e.g. *Escherichia coli*, World Wide Web, and feed-forward networks). In all these studies, only three-node structures were analysed because searching for higher-order node subgraphs, especially for motif identification, is a computationally intensive task. As part of our previous study [8], we performed the same analysis of identifying network motifs on network graphs from open-source software systems.

We obtained some interesting preliminary results. Motifs are shown to be consistent across system versions and across different software systems. Their significance seems to grow with the system growth and system maturity. Then, we observed that the same set of three-node subgraphs are present in all versions of Eclipse JDT and PDE system evolutions. This set of three-node subgraphs changes in different systems. With the help of three-node subgraphs, we proved that analysed systems evolve continuously and that the change in their structure is statistically significant. We could not confirm that the analysed systems tend to stabilise in terms of three-node subgraph frequencies. This is an important finding because it means that with the help of subgraph frequencies, we can better differentiate the software systems on the quantitative basis. We discovered that the system description with

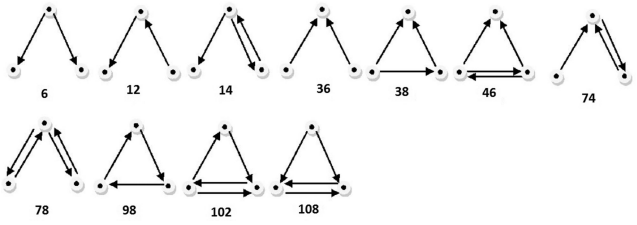


Fig. 1 *Types of three-node subgraphs*

the help of three-node subgraphs enables us to capture the system's behaviour across its evolution. Thus, a more sensitive system property can explain why SDP models cannot be easily reused in different versions of the same evolving system. Furthermore, we identified that the system defectiveness, measured as a number of defects reported and corrected in the system, is correlated with the frequency of occurrence of particular three-node subgraph types. This finding motivated us to further investigate if certain subgraph structures are more defective than others and if we can explain the stabilisation of system defectiveness through evolution with the help of subgraph defectiveness and its occurrence frequency in the system network graph.

Since the three-node subgraphs are the one well-recognised tool for network graph structure analysis and due to all of the above-mentioned findings from our previous work, we were motivated to further explore the tool's ability to capture the effects of system defectiveness. Three-node subgraph analysis can be used on any software system that can be translated as a collection of nodes with directed communication. We aim to observe subgraph defectiveness. There are 13 subgraph types that are presented in Fig. 1, where the number below the subgraph is the subgraph id used throughout this paper. Not all subgraph types appeared in all versions of the system. We want to explore if some subgraph types are more defective than others. Furthermore, we want to explore what happens with subgraph defectiveness across different versions of the system and if different subgraph types behave differently across the system during its evolution. To be more precise, we aim to investigate the following research questions:

RQ1 Which subgraphs with three nodes are the most defective in a software system?

By looking into the code structure, we can see if there is a trend where some particular types of subgraph have more defects than the others. The results could give us an insight into the defectiveness correlating with subgraph complexity and could be used to warn the programmer about problematic connections in the code.

RQ2 Is the average subgraph defectiveness similar across the different software systems and in the different software versions of the same evolving software system?

We want to see if there are the same subgraph types with similar average defectiveness present across all versions of the software system. We are also interested if there is some similarity between different subgraph types in the same version of the system.

RQ3 How does the subgraph defectiveness evolve over the software versions, and does it stabilise as the software system evolves?

As the software system evolves, the size of the code grows. Changes in the frequency of the subgraphs have been investigated in the previous study. In this study, we wish to examine the number of defects in the given subgraph types. Researching the evolution of the software system defectiveness will help us to see if all subgraph types have an increase in defectiveness as new code is added and a decrease as defects are corrected. There is a possibility of using information about evolution to point to problems where part of the code is not properly fixed.

RQ4 Is there a relation between the average subgraph defectiveness and the software system defectiveness?

By looking at average subgraph defectiveness, we could be able to get software system defectiveness. Each subgraph type has its own defectiveness, and when combined, they show software system defectiveness. It is possible that the subgraphs with lower frequencies have more influence on the software system defectiveness. The network subgraphs are overlapping within the network and one node is a part of a larger number of subgraphs. Its defects are indeed added to every subgraph and therefore, the sum does not equal the software system defectiveness. The information gathered could be useful in defect prediction.

In this paper, we present an analysis on the software systems by using Java classes as nodes in the three-node structure and communication between the classes as the edges. We cannot prove that the results would be similar if other software units were used and our results are based solely on the Eclipse software systems. The results of our study show that subgraph defectiveness is not the same across subgraph types and system versions. Furthermore, subgraph defectiveness does not stabilise over the system evolution. Finally, the most important finding is that we could not prove that there is no correlation between subgraph and software system defectiveness. Indeed, for some particular subgraph types, this correlation is statistically significant and very high. This finding is important because it may bring novel ideas and findings into SDP research. Furthermore, it may be used for better fault prevention, aiming to avoid certain subgraph types while designing software architecture and for more effective fault prevention while identifying potentially defective subgraph types in system structure.

The organisation of this paper is as follows. After this Introduction Section 1, we provide related work in Section 2. We provide details about the definitions and previous studies in Section 3 and basic information about the study and case description in Section 4. In Section 5, we provide a detailed analysis and the results of the study and discuss the findings. The work done in this paper is discussed in Section 6. Threats to validity are discussed in Section 7. We conclude the paper in Section 8.

2 Related work

There are numerous structural abstractions that we may define to analyse complex software systems such as software modules, classes, files, packages, and system units. Mostly, the empirical studies studying the behaviour of complex software systems chose abstractions that are equivalent to the basic building system element of the system they observe. Moreover, using different metrics on these basic analysis units further complicates the ability to generalise empirical observations. Empirical studies on SDP lack generalisability [9].

Fault distributions across system modules have been investigated by many authors. The first systematic study was conducted by Fenton and Ohlsson [3]. It was motivated by numerous earlier studies, in particular [10, 11], and a review of earlier studies by Hatton [12]. All these studies identified the uneven distribution of software faults over the system modules. Furthermore, these studies have been replicated many times and most recently by the authors in [1, 5]. All of these studied the empirically observed existence of the Pareto principle of uneven distribution of software faults over the system modules. Studies on the analytic model for fault distributions over the system modules have resulted in less consistent results in terms of the best-fitting particular analytic distribution to the empirical data of faults over the system modules [2, 4, 6].

One possible effect of fault distributions over the system modules is the communication patterns between them. In some previous studies, it was observed that the most frequently communicating software modules are the most faulty ones [1]. Furthermore, fewer serious failures are observed in components implementing commonality and fewer changes over time [13]. These observations motivated us to include studying communication structures as an element that influences system defectiveness. Software structure evolution and its relation to system defectiveness have been proven in [8]. The frequency of the occurrence of primitive communication substructures called subgraphs is investigated, following a similar approach investigated in other scientific communities for analysing some application domains (medicine, biology, the Internet, traffic, and telecommunications). Here, we extend this study to investigate substructure, as a primitive communication type, and its relation to

the substructure and system defectiveness. Furthermore, in [14], the authors used dependency graphs to identify central program units that are most likely to face defects. They compared network measures by using the Ucinet tool and complexity measures. A study that was performed on the binaries of a Windows Server 2003 computer identified that network measures are better predictors than classic complexity measures. However, replications of their study in small-scale projects by Turhan and co-workers [15] have shown that network measures are not suitable predictors. Network measures were also used in some other studies [16] that report interesting findings in relation to the system structure's ability, when represented as a network graph, to identify and predict structural changes that may be interesting for early fault prediction studies. In the paper by Concas *et al.* [17], the community structure of a complex software network is examined and they show that medium-size systems hold a community structure which appears related to the mean bug number for class.

One interesting approach using the complex network and representing the software system as a collection of nodes and edges has been used to identify the most critical nodes [18, 19].

System evolution principles have been stated in Lehman's statements about the evolution of software systems and their interrelations, as obtained by the analysis of large-program growth dynamics [20]. As the software system grows, declining quality is confirmed in other studies [21, 22]. Contradictory evidence is obtained for the statement of increasing complexity, and one probable reason for that is the existence of numerous approaches to structure definition. Each one brings its own perspective on complexity. By evaluating these empirical principles, numerous metrics have been used, and some simple ones are software source code size [23], types, global variables, cumulative number of additions and deletions, file changes ([24]) etc.

The persistence of SDP models over the system's evolution has been studied by Zimmermann *et al.* [14, 25]. They found that SDP models may not be so reusable from release to release in software evolution.

There is a group of studies aiming to identify design patterns and anti-patterns that are related to defects. These approaches aim to find already-known pattern templates in a particular programming language. However, here, we aim to find more generic structural communication patterns or anti-patterns that could guide software modelling and design on a more abstract level, independent of programming language. One similar approach to ours is [26]. It identifies connected subgraphs with the SGFinder tool. Their work is focused on the characterisation of the specific subgraphs and their relation to defectiveness. Work by Zhang *et al*. [27] is closely connected to the topic of this paper. In that work, they use motifs to present the connections inside the software structure and compare them to the number of defects, but they only focus on one specific subgraph type. The findings that they present are promising in terms of using subgraph types in defect prediction. Here, in this paper, we use all the connected subgraphs to represent the whole system and try to understand its evolutionary trends and defectiveness. Moreover, our study involves much larger systems with more releases.

3 Network analysis approach

In this paper, we will present a software system as a network graph. We define a network graph G as a directed graph represented by a set of vertices V called nodes (in our case, class), and the links that connect some pairs of nodes, which are called edges E (in our case, method calls), represent directed connections between these nodes. The graph is represented with an ordered pair $G = (V, E)$ comprising a set V of vertices together with a set E of edges (i.e. a method call is related with two classes ordered in the direction of that method call).

In the study, we analyse network subgraphs. Network subgraphs are abstract topological substructures of a network graph. A subgraph, H, of a graph, G, is a graph whose vertices V are a subset of the vertex set present in network graph G and whose edges are a subset of the edge set present in network graph G. In this paper, we restrict our analysis to three-node subgraphs. All possible topologically unique three-node connected and directed subgraphs are presented in Fig. 1. From the figure, we may observe that some subgraph types have simpler communication interactions than others, and that may also be one property that explains why they may behave differently. Note that some subgraphs involve bidirectional communication, and some involve only unidirectional communication between the nodes. Bidirectional communication may imply some more complicated data dependencies between the classes.

A network graph may be represented by a 'coordinate system' for such a collection of three-node subgraphs, and that can be used to compare software network subgraphs present in different application domains. Each particular three-node subgraph H_i has a frequency of occurrence f_i in network graph G. In our previous paper [8], we compared software network graphs G of different versions of the Eclipse system with respect to subgraph frequencies represented by such a coordinate system. Here, in this paper, we aim to compare the software network graphs of different versions of the Eclipse system with respect to subgraph defectiveness represented by such a coordinate system. We used code written in Java and in the Eclipse environment as we had developed the tools that work on software systems written in Java and were able to collect information about class defects for the Eclipse project. The approach of three-node subgraphs can theoretically be used on any programming language if the necessary tool is developed.

Subgraph frequency, $f_{S_{id}}$, represents the number of subgraph appearances of a particular subgraph type S_{id} in an analysed software network graph. The defectiveness of the ith subgraph, the random variable measuring defectiveness of subgraph type S_{id}, represents a sum of the numbers of faults detected in each node of the ith subgraph and is denoted as $d_{S_{id}}$. The sum of subgraph defectiveness for all subgraphs $i = 1, \ldots, S_{id}$ of a certain subgraph type S_{id} divided by the subgraph frequency of that subgraph type $f_{S_{id}}$ we call average subgraph defectiveness and denote as $\bar{d}_{S_{id}}$.

4 Study design

4.1 Hypotheses

The following hypotheses were analysed.

Hypotheses about average subgraph defectiveness:

RQ1

H1: The average subgraph defectiveness is the same for different subgraph types of the software program. If there are different subgraph types present in the software project, we want to test for the equality of their average defectiveness to see if there is any difference, and if there is, if there are any groups that have a similar average defectiveness.

RQ2

H2: The average subgraph defectiveness is the same across different versions of the software program. It is expected that during the process of software testing, the defects will be fixed. If there is no change in some subgraph type's defectiveness through the software evolution, it could point to problems where part of the code is not properly fixed.

H3: The average subgraph defectiveness is the same between different software programs.

As both tested projects were part of the Eclipse organisation, they have a similar working environment. Testing for the equality of the subgraph defectiveness between the two projects could show us if working in similar conditions has the same outcome in terms of the average software defectiveness.

Hypotheses about subgraph defectiveness evolution:

RQ3

H4: Subgraph defectiveness does not change over system evolution.

We test for the equality of the subgraph defectiveness distribution between different subgraph types and between one subgraph type in different versions of the system. If the hypotheses could not be rejected, it could be used in defect predictions.

H5: The distribution of subgraph defectiveness is the same for different subgraph types within one version of the software program.

We test for the equality of the subgraph defectiveness distribution between one subgraph type in different versions of the system. If the hypothesis could not be rejected, it could also be used in defect predictions.

H6: Subgraph defectiveness tends to stabilise during system evolution.

We wish to see if there is a stabilisation of different subgraph types' defectiveness through software evolution.

Hypotheses on effects of subgraph defectiveness evolution on system defect proneness:

RQ4

H7: Subgraph defectiveness is a good predictor of the number of defects in a system version.

The hypothesis is tested to see if there is a possibility that information about subgraph types, frequency, and defectiveness could be used as a defect prediction method.

4.2 Data collection

The data we analysed came from open-source Eclipse projects written in Java, Java development tools (JDT), and a plug-in development environment (PDE). In JDT, we choose 12 versions, and in PDE, we choose 13. The JDT plug-in was developed in a period of 12 years through 14 different versions, and the PDE plug-in, in a period of 11 years through 13 different versions.

The data collection process was very systematic and involved several months of investigation and tool development needed to produce reliable data sets. Here, we describe the steps in data collection.

Source code files: Firstly, we collected project files for each project and project versions from the Eclipse Git repository. We removed all the test files from each project file by removing all folders that contain the word test in its name. Only org.eclipse.ui[.*] components for JDT and PDE plug-ins were analysed. Other components were too small for the network analysis.

Network files: Network graphs of the software communicating structure in a.graph format were created through the rFind tool developed as part of research reported in [28]. The inputs were project files gathered in the previous step, from which the rFind tool reads all classes and graph nodes and related method calls between classes as directed links.

Collecting data about defects: The defects of the analysed projects were obtained through the Bugzilla repository. Previously, we developed a BuCo tool [29] that searches for links between class files and defects and, for each class file, reports details such as the number of lines of code, 52 other static code attributes, and the number of bugs. BuCo tool is a data collection tool that uses a bug-code linking technique based on regular expression. The result of the BuCo tool is a csv file where each row represents one class and measurements, stored in columns, detail static attributes followed by the number of bugs. More details can be found in [29].

Collecting defect data for subgraphs: For the purpose of this analysis, we developed a SuBuCo tool that searches for all subgraph types given in Fig. 1, identifies classes present in each three-node subgraph, and finally sums up the number of bugs that appears in those classes for a given subgraph. Finally, the obtained data files (one per project version) that we used for our analysis contain subgraph data organised in a table. Rows in Tables 1 and 2 represent data measured for every three-node subgraph found in the whole network graph and include information about subgraph id and the total number of faults present in three of its classes. The subgraph types 110 and 238 did not appear in any of the systems and therefore, those types are not presented in the tables nor in the rest of the analysis results.

4.3 Data analysis techniques

For the analysis, we use observational conclusions based on the figures and tables and several statistical tests. For the analysis of the subgraph defectiveness tendency, we used descriptive statistics: box and whisker plots for data distributions and medians. For analysing the distribution of defects over subgraph ids in one version and the distribution of defects of each subgraph id over all versions, the Mann–Whitney–Wilcoxon, Kruskal–Wallis, and median tests were used. Additionally, the Jonckheere–Terpstra test [30] was used to see if the subgraph defectiveness measurements of different ids come from the same population. For the analysis, Statistica software and IBM SPSS Statistic were used.

5 Analysis

When looking at the types of classes involved in each subgraph type, we came to the conclusion that in the most common subgraph types, 36 and 38, the classes that are involved are usually Java libraries for basic operations and libraries for data types such as String, Array, and others. Views and actions are usually involved in subgraph types 78 and 14, while API classes are present in types 12, 74, and 46. In all subgraph types, there are all types of classes, but these are the most common that appear in each type. That does not mean that those types of classes are the most faulty ones but rather only indicates the type of communication they are involved in. In other words, it means that they are the most used ones by other user classes. The faults could be contained within the other classes that are part of the three-node subgraph and are caused by the improper implementation of interface to those classes implementing basic functionalities. Our analysis involved 25 tables (one per software version, 12 JDT and 13 PDE) each with over a million rows representing all subgraphs found in each software release. Information about the size of each project version in terms of lines of code is given in Table 3. The summary of results involving subgraph frequency $f_{S_{id}}$, information about the number of subgraphs found per subgraph type S_{id} (all subgraph types are given in Fig. 1) and for each project version is in Tables 1 and 2. We provide in separate subsections the analysis results for the hypotheses provided in Section 5.1.

5.1 Hypotheses about average subgraph defectiveness

5.1.1 H1 and H2: The rows in Tables 1 and 2 present data for different versions of the Eclipse JDT or PDE software. The label of the software version is presented in the first column of the table. Other columns in the table represent subgraphs present in the software network of each software version. The additional information about the descriptive statistics on the subgraph defect appearance in the JDT and PDE software, such as standard deviation, can be found in the additional document at the project website [31].

The average subgraph defectiveness $\bar{d}_{S_{id}}$ row represents the ratio of the total sum of subgraph defects for a specific subgraph id and subgraph frequency. For example, in version JDT 2.0, subgraph id 38 has frequency 955, the sum of the defects present in all subgraphs of that type is 4998, and the average subgraph defectiveness of subgraph 38 is then 5.233. As the larger number of defects is to be expected with larger frequencies of a subgraph type, we used the average subgraph defectiveness for this analysis. We can see in Table 1 and Fig. 2 that the average subgraph defectiveness $\bar{d}_{S_{id}}$ for JDT is decreasing over the system versions. However, during the system evolution over the system versions, there is some fluctuation of average subgraph defectiveness for subgraph types 14, 48, 74, and 78. On the other hand, for the PDE product, we can observe a somewhat different behaviour of the average subgraph defectiveness from Table 2 and Fig. 3. We can see some increase in the average subgraph defectiveness until version 3.4 and then a decrease. Some larger fluctuations can be observed for subgraphs 36, 38, 46, and 74. From this analysis, we may say that the average subgraph defectiveness $\bar{d}_{S_{id}}$ for JDT and PDE products behaves differently during system evolution.

A Kolmogorov–Smirnov test was performed on distributions of the subgraph defectiveness random variable $d_{S_{id}}$ for each subgraph type and for each software version to see if samples are normally distributed. A normality test performed on the data set has shown that all samples of subgraph defectiveness are not normally

Table 1 Data set in the analysis presenting subgraph frequencies $f_{S_{id}}$, average subgraph defectiveness $\bar{d}_{S_{id}}$, and Standard Deviation for the JDT project

Version	Info	6	12	14	36	38	46	74	78	102	108	Defects
JDT 2_0	$f_{S_{id}}$	27,419	3825	346	175,621	955	18	268	28	0	0	965,114
	$\bar{d}_{S_{id}}$	6.599	7.718	16.442	4.207	5.233	16.278	15.933	21.107	0	0	
	$\sigma_{S_{id}}$	8.24	10.04	12.49	6.25	7.44	14.03	13.73	11.82	0	0	
JDT 2_1	$f_{S_{id}}$	42,739	7097	658	312,163	1527	20	397	43	0	0	927,892
	$\bar{d}_{S_{id}}$	2.963	3.573	1.801	2.476	2.696	2.85	3.479	0.954	0	0	
	$\sigma_{S_{id}}$	4.23	4.53	4.21	3.47	3.79	4.97	5.54	2.49	0	0	
JDT 3_0	$f_{S_{id}}$	66,509	11,025	912	575,120	2188	24	483	43	2	0	1,907,898
	$\bar{d}_{S_{id}}$	3.569	6.003	3.59	2.766	3.2556	7.125	6.073	1.558	2.5	0	
	$\sigma_{S_{id}}$	4.71	8.87	8.28	3.84	5.39	11.21	10.01	4.38	0.5	0	
JDT 3_1	$f_{S_{id}}$	79,670	11,038	875	714,762	2563	26	546	45	1	0	1,736,144
	$\bar{d}_{S_{id}}$	2.136	2.891	1.766	2.133	2.567	3.538	2.658	1.667	2	0	
	$\sigma_{S_{id}}$	3.02	3.53	2.62	2.96	3.72	2.94	3.19	2.65	0	0	
JDT 3_2	$f_{S_{id}}$	93,771	13,550	887	988,131	3121	31	610	40	3	0	2,074,675
	$\bar{d}_{S_{id}}$	1.746	1.983	2.249	1.896	2.233	4	2.626	2.475	5.333	0	
	$\sigma_{S_{id}}$	2.74	2.76	2.73	2.80	3.15	2.86	2.83	2.91	0.7	0	
JDT 3_3	$f_{S_{id}}$	100,921	15,454	923	1,052,913	3492	34	901	54	2	0	1,431,157
	$\bar{d}_{S_{id}}$	1.102	1.369	1.026	1.227	1.52	1.294	1.009	0.667	2.5	0	
	$\sigma_{S_{id}}$	2.10	2.54	1.66	1.92	2.74	1.78	1.26	1.18	0.5	0	
JDT 3_4	$f_{S_{id}}$	112,266	18,270	1027	1,193,467	4364	51	993	58	2	0	1,160,511
	$\bar{d}_{S_{id}}$	0.872	0.779	1.142	0.874	0.843	1.941	0.752	1.052	2.5	0	
	$\sigma_{S_{id}}$	1.97	1.42	1.54	1.70	1.55	1.88	1.25	1.44	0.5	0	
JDT 3_5	$f_{S_{id}}$	126,144	18,483	947	1,297,590	4379	45	931	45	2	0	739,885
	$\bar{d}_{S_{id}}$	0.477	0.376	0.227	0.517	0.59	0.644	0.245	0.4	0	0	
	$\sigma_{S_{id}}$	1.16	1.08	0.67	1.19	1.35	1.00	0.70	1.07	0	0	
JDT 3_6	$f_{S_{id}}$	126,144	18,862	945	1,333,151	4545	45	949	45	2	0	511,972
	$\bar{d}_{S_{id}}$	0.311	0.349	0.227	0.348	0.404	0.356	0.26	0	0	0	
	$\sigma_{S_{id}}$	0.75	0.75	0.80	0.79	0.85	1.15	0.55	0	0	0	
JDT 3_7	$f_{S_{id}}$	127,006	18,290	949	10,735,562	4384	44	949	45	2	0	400,263
	$\bar{d}_{S_{id}}$	0.338	0.302	0.178	0.033	0.387	0.25	0.341	0.133	1.5	0	
	$\sigma_{S_{id}}$	1.01	0.97	0.73	0.85	1.22	0.61	0.59	0.50	0.5	0	
JDT 3_8	$f_{S_{id}}$	132,736	18,535	951	1,349,285	4469	44	952	45	3	0	249,348
	$\bar{d}_{S_{id}}$	0.2716	0.066	0.013	0.156	0.21	0.25	0.123	0.044	0.333	0	
	$\sigma_{S_{id}}$	0.63	0.44	0.38	0.57	0.65	0.53	0.30	0.20	0.9	0	
JDT 4_2	$f_{S_{id}}$	132,736	18,535	951	1,349,284	4469	44	952	45	3	0	25,629
	$\bar{d}_{S_{id}}$	0.019	0.015	0	0.017	0.033	0	0.014	0	0	0	
	$\sigma_{S_{id}}$	0.13	0.11	0	0.12	0.17	0	0.11	0	0	0	

distributed. Therefore, we avoided applying statistics for normally distributed data in our further analysis.

We applied the Mann–Whitney–Wilcoxon test to analyse hypotheses H1 and H2. We compared distributions of average subgraph defectiveness $\bar{d}_{S_{id}}$ between subgraph types while testing hypothesis H1 and compared distributions between different versions while testing hypothesis H2. The null hypothesis for the test was that the different average subgraph defectiveness between subgraph types contains samples drawn from the same population. The p-value for the tests was 0.05. In Tables 4 and 5, we present the results of that test. All members of one group have samples drawn from the same population or from the population with similar mean value. In Table 4, members of the groups are the subgraph types, while in Table 5, those members are the software versions. As we can see in Table 4, some subgraph types have similar mean values. Despite this finding, there is no visible trend or similarity in those groups. Even though there is some similarity between some types, we cannot say that they all have the same average subgraph defectiveness, and therefore, hypothesis H1 is rejected. Testing H2, which says that the distributions of average subgraph defectiveness $\bar{d}_{S_{id}}$ between versions are the same, has given similar results as testing for H1, as shown in Table 5. We can see that subgraph types behave similarly in different versions of the project. In versions 4.2 and 2.0, there is no version for any subgraph type that has the same mean value, except for type 78, where version 4.2 is in a group with few other versions. The average subgraph defectiveness is not the same across all versions of the project, and hypothesis H2 is rejected. Owing to limited space, we did not report the tables for the PDE project, but they can be found at the project website [31].

Our observations while testing for hypotheses 1 and 2 are as follows:

- The PDE project is much smaller in terms of the number of classes and subgraphs than the JDT project.
- Subgraphs that contain more edges (78, 74, and 46) have a higher average defectiveness in the first considered version of JDT and PDE projects.
- In the JDT project, average defectiveness is decreasing with every new release, while in the PDE project, the average subgraph defectiveness seems to be decreasing only in the last few versions but increases again in the last version of the system.
- Despite some subgraph types having similar means and despite the fact that they are grouped together, there is no visible trend or similarity in those groups.
- There is no or very little similarity of mean values between subgraph types in different versions of a project.

With this analysis, hypotheses H1 and H2 were rejected.

5.2 Hypotheses about subgraph defectiveness evolution

5.2.1 H3: We used the Mann–Whitney–Wilcoxon test for testing H3 – testing that the average subgraph defectiveness $\bar{d}_{S_{id}}$ is the same between different software programs. We compared the average mean values for both the projects for every present subgraph for all the releases of the JDT project against all the releases of the PDE project. The p-value was 0.037, <0.05, at

Table 2 Data set in the analysis presenting subgraph frequencies $f_{S_{id}}$, average subgraph defectiveness $\bar{d}_{S_{id}}$, and standard deviation for the PDE project

Version	Info	6	12	14	36	38	46	74	78	102	108	Defects
PDE 2_0	$f_{S_{id}}$	3617	86	0	63,252	31	0	0	0	0	0	87,797
	$\bar{d}_{S_{id}}$	1.469	2.302	0	1.3	1.193	0	0	0	0	0	
	$\sigma_{S_{id}}$	2.61	2.86	0	2.11	1.51	0	0	0	0	0	
PDE 2_1	$f_{S_{id}}$	5178	179	0	108,516	48	0	0	0	0	0	104,125
	$\bar{d}_{S_{id}}$	0.952	0.972	0	0.912	0.708	0	0	0	0	0	
	$\sigma_{S_{id}}$	1.81	1.64	0	1.78	1.12	0	0	0	0	0	
PDE 3_0	$f_{S_{id}}$	5532	138	0	124,298	25	0	0	0	0	0	179,707
	$\bar{d}_{S_{id}}$	1.039	0.935	0	1.398	1.24	0	0	0	0	0	
	$\sigma_{S_{id}}$	1.47	1.49	0	1.66	1.58	0	0	0	0	0	
PDE 3_1	$f_{S_{id}}$	14,799	727	0	185,422	263	0	0	0	0	0	331,361
	$\bar{d}_{S_{id}}$	1.626	2.528	0	1.644	2.6	0	0	0	0	0	
	$\sigma_{S_{id}}$	1.99	2.43	0	1.95	2.39	0	0	0	0	0	
PDE 3_2	$f_{S_{id}}$	32,149	3024	56	354,217	852	13	48	0	0	0	777,763
	$\bar{d}_{S_{id}}$	1.772	2.075	1.25	2.011	2.587	1.615	0.771	0	0	0	
	$\sigma_{S_{id}}$	1.94	2.15	1.28	2.08	2.15	1.38	1.01	0	0	0	
PDE 3_3	$f_{S_{id}}$	39,654	4566	58	529,612	1177	10	51	0	0	2	1,743,146
	$\bar{d}_{S_{id}}$	2.819	3.677	1.483	3.039	4.304	1.9	0.824	0	0	1.5	
	$\sigma_{S_{id}}$	3.66	4.08	2.66	3.54	3.08	2.84	1.27	0	0	0.5	
PDE 3_4	$f_{S_{id}}$	31,026	3182	35	803,161	809	4	7	0	0	0	2,204,252
	$\bar{d}_{S_{id}}$	3.57	2.419	5.543	2.594	3.038	6.5	5.714	0	0	0	
	$\sigma_{S_{id}}$	8.25	6.62	0.95	6.27	8.90	1	0.95	0	0	0	
PDE 3_5	$f_{S_{id}}$	33,318	3564	46	920,810	942	2	0	1	0	0	2,294,674
	$\bar{d}_{S_{id}}$	2.669	2.337	10.348	2.383	3.014	2	0	14	0	0	
	$\sigma_{S_{id}}$	4.55	5.10	3.72	3.95	6.22	0	0	0	0	0	
PDE 3_6	$f_{S_{id}}$	34,997	3631	57	1,009,765	988	3	0	1	0	0	1,052,128
	$\bar{d}_{S_{id}}$	1.061	0.927	0.298	1	1.44	1	0	0	0	0	
	$\sigma_{S_{id}}$	2.53	1.35	0.42	1.30	1.55	0	0	0	0	0	
PDE 3_7	$f_{S_{id}}$	35,943	3706	57	1,038,078	1022	3	0	1	0	0	1,156,743
	$\bar{d}_{S_{id}}$	0.866	0.975	0.228	1.08	1.036	1	0	0	0	0	
	$\sigma_{S_{id}}$	1.28	1.35	0.42	1.30	1.55	0	0	0	0	0	
PDE 3_8	$f_{S_{id}}$	36,763	3679	57	1,063,298	1018	3	0	1	0	0	98,483
	$\bar{d}_{S_{id}}$	0.101	0.081	0	0.089	0.099	0	0	0	0	0	
	$\sigma_{S_{id}}$	0.36	0.35	0	0.35	0.36	0	0	0	0	0	
PDE 4_2	$f_{S_{id}}$	36,763	3679	57	1,063,298	1018	3	0	1	0	0	158,371
	$\bar{d}_{S_{id}}$	0.16	0.187	0	0.143	0.246	0	0	0	0	0	
	$\sigma_{S_{id}}$	0.44	0.51	0	0.43	0.55	0	0	0	0	0	
PDE 4_3	$f_{S_{id}}$	36,556	3671	81	1,064,393	1042	1	11	0	0	0	896,536
	$\bar{d}_{S_{id}}$	0.704	0.925	1.852	0.814	0.966	1	0	0	0	0	
	$\sigma_{S_{id}}$	1.04	1.05	2.2	1.03	1.18	0	0	0	0	0	

which value the test is statistically significant. Therefore, we can say that we cannot reject hypothesis H3.

Table 3 Lines of code for each project version in both the JDT and PDE projects

JDT		PDE	
Version	LOC	Version	LOC
R2_0	136,405	R2_0	52,716
R2_1	215,150	R2_1	74,855
R3_0	307,699	R3_0	83,865
R3_1	396,635	R3_1	107,700
R3_2	482,249	R3_2	143,221
R3_3	517,613	R3_3	177,781
R3_4	540,357	R3_4	237,670
R3_5	557,419	R3_5	293,180
R3_6	568,147	R3_6	316,383
R3_7	585,968	R3_7	334,962
R3_8	599,865	R3_8	330,582
R4_2	599,865	R4_2	330,592
—	—	R4_3	333,390

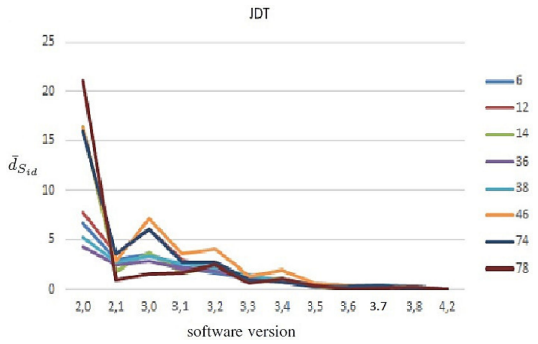


Fig. 2 *Average subgraph defectiveness $\bar{d}_{S_{id}}$ over versions in JDT*

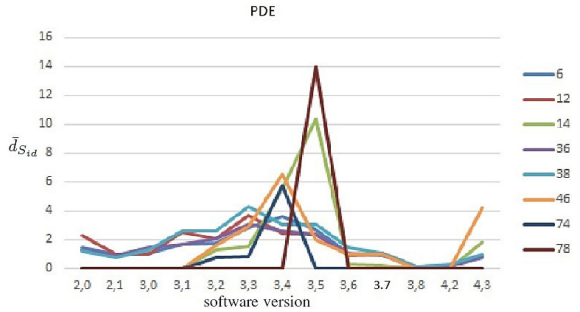


Fig. 3 *Average subgraph defectiveness $\bar{d}_{S_{id}}$ over versions in PDE*

5.2.2 H4 and H5: The box plots in Figs. 4 and 5 describe the central tendency of the subgraph defectiveness $d_{S_{id}}$ in the JDT project in terms of the median of the values (represented by the smallest box in the plot). The spread (variability) in the variable values is represented in this plot by the quartiles (the 25 and 75 percentiles, larger box in the plot) and the minimum and maximum values of the variable. As we can see from the figures, subgraphs' defectiveness medians are different for each subgraph type in software versions in earlier releases, although subgraph types 36, 38, 6, and 12, which are the most common since they are very simple, are close to each other in the beginning until version JDT R3_2, while other subgraph types' medians are much higher. In version JDT 3.6, all subgraph types are stabilised and have mostly no defects.

Figs. 6 and 7 describe subgraph defectiveness in the PDE project. In the PDE project, there are fewer subgraph types; all but 36, 38, 6, 14, and 12 are too rare to have a meaningful analysis. In the first version, subgraph type 12 seems to have the most subgraph defects and that continues in the first four versions, until version PDE 3.2. In version PDE 3.5, subgraph type 14 has a rise in defects compared to other types, and after that, the defect number decreases again. In the last version, it seems that all the defectiveness of subgraph types increased, and the most stable versions were PDE R3_8 and PDE R4_2.

The Kruskal–Wallis test is a non-parametric alternative to one-way (between-groups) ANOVA. It is used to compare three or more samples, and it tests the null hypothesis, which states that the different samples in the comparison were drawn from the same distribution or from distributions with the same median. The Kruskal–Wallis test is highly significant ($p = 0.000$) in the JDT project. Thus, we can conclude that the subgraph defectiveness values of different subgraph types in the same version are significantly different from each other. The test of all versions proves that the subgraph defectiveness of different subgraph types in the same version was significantly different from each other.

When comparing only subgraphs with a higher defectiveness to test if they are statistically from the same distribution, the results were also negative. Subgraphs with higher defectiveness do not come from the same distribution or from distributions with the same median.

The median test simply counts the number of cases in each sample that fall above or below the common median and computes the χ^2 value for the resulting $2 \times k$ samples contingency table. Under the null hypothesis (all samples come from populations with identical medians), we expect ∼50% of all cases in each sample to fall above (or below) the common median. The $p-$value, in the JDT project, of 0.000 indicates that the probability that the χ^2 value occurs if there are actually no differences between the subgraph types in the same version is 0%. Therefore, it can be concluded that there is at least one significant difference between the groups. The median is getting lower with each version as defects are being corrected, and in the last version, only 2% of subgraphs have some defects.

Just as in the JDT project, in PDE, the Kruskal–Wallis test is also highly significant ($p = 0.000$) in all versions except in version

Table 4 Mann–Whitney–Wilcoxon test between subgraph ids

Version	Group 1	Group 2	Group 3	Group 4	Group 5
JDT R2_0	36	38	6	12	74 46 14 78
JDT R2_1	78	14	36	38 46 6	74 12
JDT R3_0	78	36	38	6 14	12 74 46
JDT R3_1	78 14	36 6	38 74 12 46	—	—
JDT R3_2	6	36	12	38 14 78 74	46
JDT R3_3	78 74 14 6	36 46 12 38	—	—	—
JDT R3_4	74 12	38 6 36 78 14	46	—	—
JDT R3_5	14 74	12 78	6	36	38 46
JDT R3_6	78 14	74	6	12 46 46	38
JDT R3_7	78 74 46 36	12	6 14	38	—
JDT R3_8	78 74	12 14	36	38	46 6
JDT R4_2	78 46 14	74 12 36	6	38	—

Table 5 Mann–Whitney–Wilcoxon test for JDT project subgraph types

JDT	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7	Group 8	Group 9	Group 10	Group 11	Group 12
6	4.2	3.8	3.6	3.7	3.5	3.4	3.3	3.2	3.1	2.1	3.0	2.0
12	4.2	3.8	3.6	3.7	3.5	3.4	3.3	3.2	3.1	2.1	3.0	2.0
14	4.2	3.8	3.6 3.5	3.7	3.3 3.4	3.1 2.1	3.2	3.0	2.0	—	—	—
36	4.2	3.8	3.6	3.7	3.5	3.4	3.3	3.2	3.1	2.1	3.0	2.0
38	4.2	3.8	3.6 3.7	3.5	3.4	3.3	3.2	3.1 2.1	3.0	2.0	—	—
46	4.2	3.7 3.8 3.6 3.5	3.3 3.4	2.1 3.1 3.2 3.0	2.0	—	—	—	—	—	—	—
74	4.2	3.8	3.7	3.5 3.6	3.4	3.3	3.2 3.1	2.1	3.0	2.0	—	—
78	3.6 4.2 3.8 3.7	3.5	3.3 2.1 3.4	3.0 3.1 3.2	2.0	—	—	—	—	—	—	—

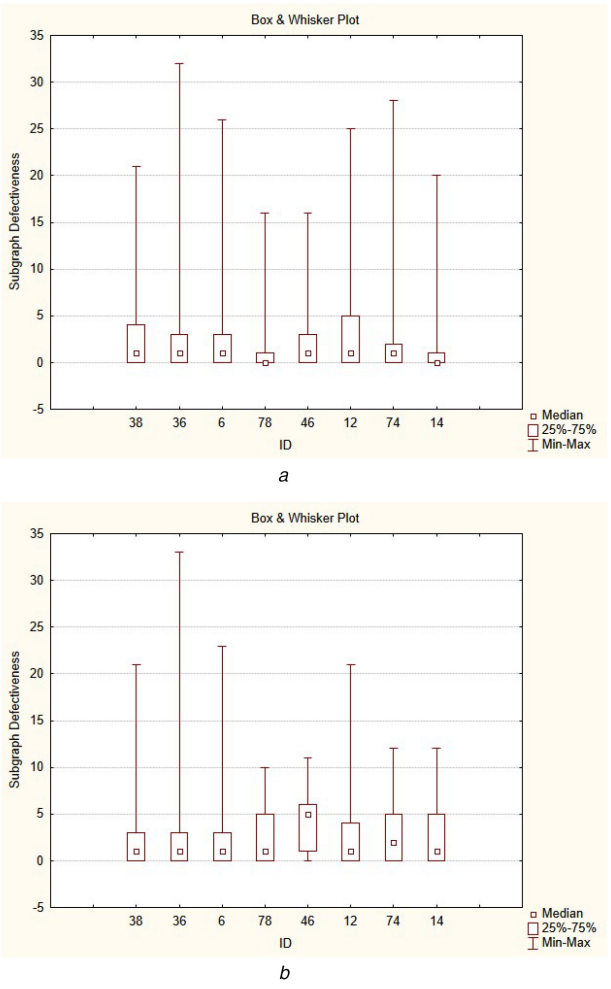


Fig. 4 *Box plots for subgraph defectiveness. Each figure represents one Eclipse JDT version and each subfigure presents box plots for all subgraphs*

(a) JDT version R2.1, ***(b)*** JDT version R3.2

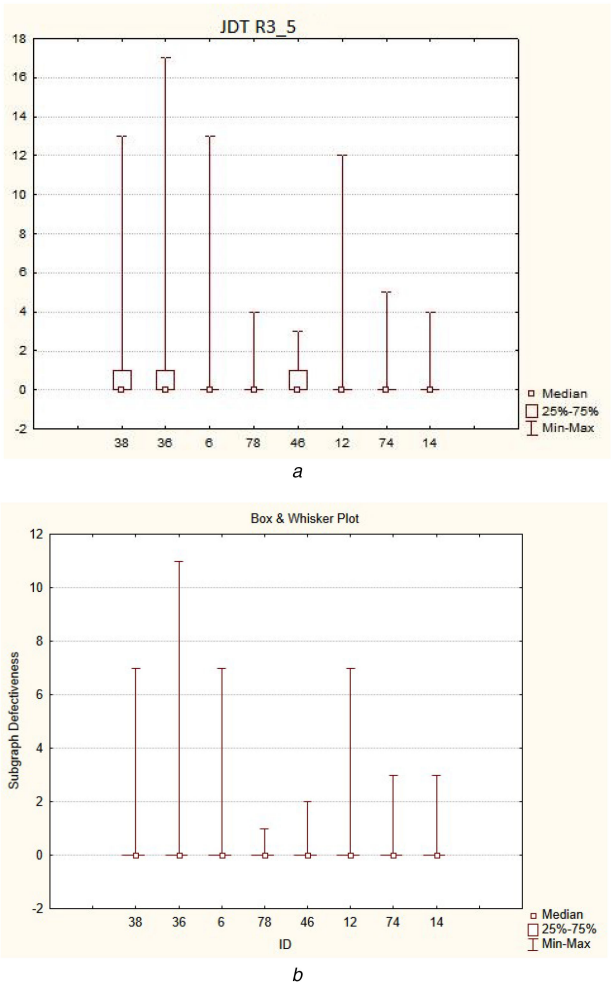


Fig. 5 *Box plots for subgraph defectiveness. Each figure represents one Eclipse JDT version and each subfigure presents box plots for all subgraphs*

(a) JDT version R3.5, ***(b)*** JDT version R3.8

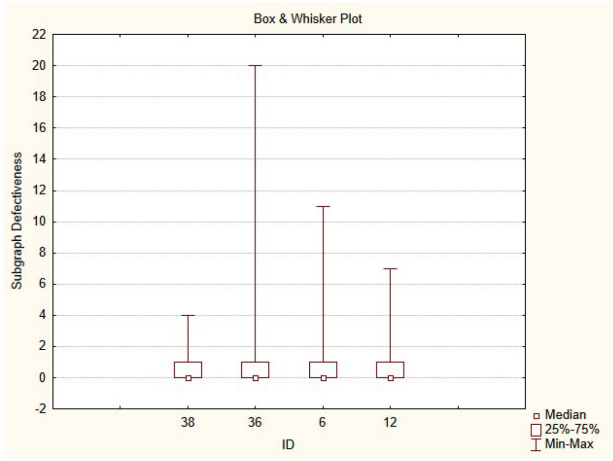
PDE 2.1. We can conclude that the subgraph defectiveness values of different subgraph types in the same version were significantly different from one another. In version PDE 2.1, we can see a $p-$value of 0.9924, which means that the subgraph defectiveness values of different subgraph types in the same version are not statistically significantly different. It is possible that the sample size is simply too small, which means that the Kruskal–Wallis test will always give a $p>0.05$, no matter how much the groups differ.

In the PDE project, the $p-$value is 0.000, which indicates that the probability that the χ^2 value occurs, if there are actually no differences between the subgraph types in the same version, is 0%. Therefore, it can be concluded that there is at least one significant difference between the groups.

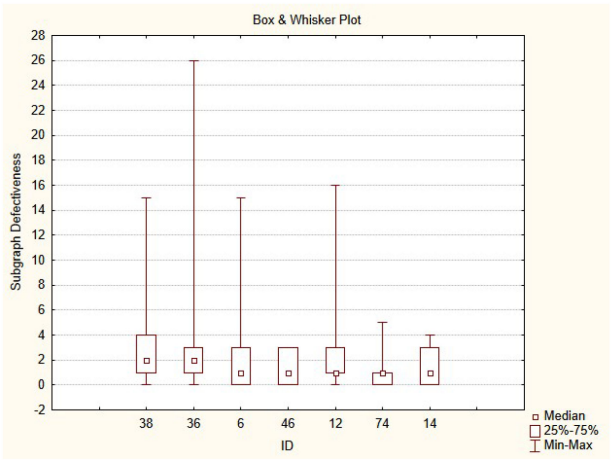
Just as in JDT, the median is getting lower with each version as defects are being corrected until the last version. In the last version, 49.5% of subgraphs have some defects, while in the previous version R4_2, only 12.3% had some defects. There were probably some changes in the code, or added code was poorly written. It is also possible that there was an increase in testing effort.

In version R2_1, as in the Kruskal–Wallis test, the $p-$value is >0.05, and χ^2 is low. It seems that the medians are equal in the PDE 2.1 version in subgraph types.

The box plots in Figs. 8–10 describe the central tendency of the subgraph defectiveness of each version in the JDT project. As we can see from the figures, the subgraph defectiveness median is getting lower with the software version except for subgraph types 46, 74, and 78, for which it varies over versions and stabilises in version JDT R3_5. Subgraph types 46 and 78 are rare and are irrelevant in comparison to other types. It is expected that subgraph defectiveness will decrease since with every version, defectiveness is corrected. In version JDT 3.0, new defects were added in all subgraph types, while in version JDT R3_2, new defects were added in subgraphs 14, 74, and 78. After version JDT R3_4, no more defects seem to be added to the subgraphs.

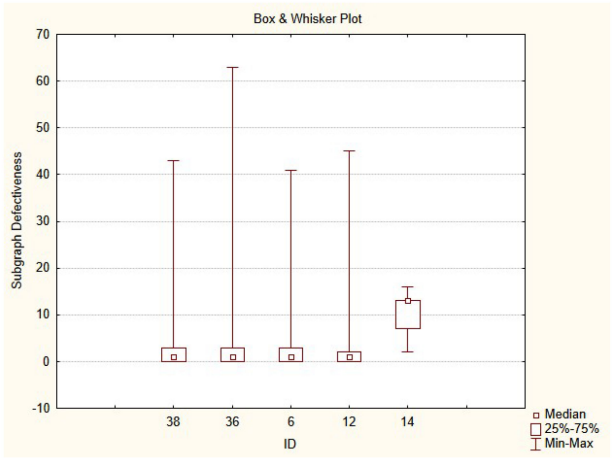


a

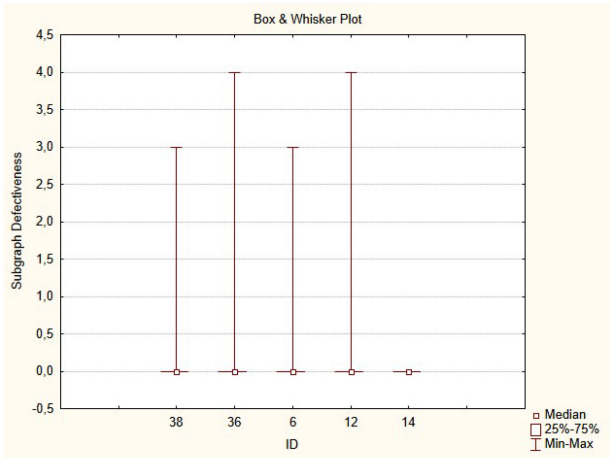


b

Fig. 6 *Box plots for subgraph defectiveness. Each figure represents one Eclipse PDE version and each subfigure presents box plots for all subgraphs*
(a) PDE version R2.1, ***(b)*** PDE version R3.2



a



b

Fig. 7 *Box plots for subgraph defectiveness. Each figure represents one Eclipse PDE version and each subfigure presents box plots for all subgraphs*
(a) PDE version R3.5, ***(b)*** PDE version R3.8

The box plots in Figs. 11 and 12 describe the central tendency of the subgraph defectiveness of each version in the PDE project. All subgraph types seem to be unstable through the development of the software. Versions PDE R3_8 and PDE R4_2 are the only versions in which all subgraph types seem to have low to no defectiveness. It is expected for subgraph defectiveness to decrease since with every version, defectiveness is corrected. Subgraph type 14 appears after version R3_2, since it has high defectiveness, but it is also the rarest subgraph type of the eight subgraph types that appear. Subgraph types 46, 78, and 74 also appear but in much smaller numbers, and therefore, it was not possible to analyse them so that the analysis would be significant.

The Kruskal–Wallis test is highly significant ($p = 0.000$) for both the PDE and JDT projects. We can conclude that the subgraph defectiveness values of subgraph types in different versions are significantly different from each other both in the JDT project and in the PDE project.

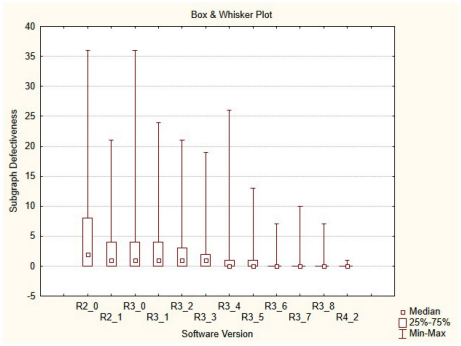
The $p-$value in the median tests is 0.0, in both the PDE and JDT projects, which indicates that the probability that the χ^2 value occurs if there are actually no differences in subgraph defectiveness between different versions is 0%.

Our observations while testing for hypotheses 4 and 5 are as follows:

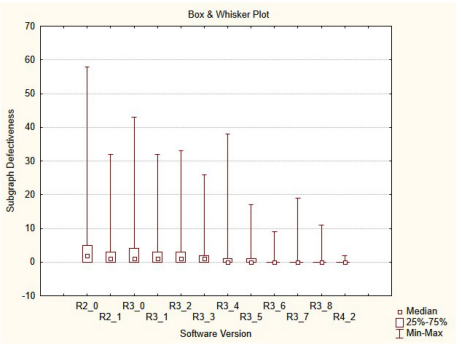
- The subgraph defectiveness medians are different between different subgraph types in the same versions for both the JDT and PDE projects.
- The subgraph defectiveness medians are very similar in the last few versions of the JDT project as they are all getting closer to 0.
- The subgraph defectiveness values of each type, in both the JDT and PDE projects, are not the same as the software evolves. With the testing process, subgraph defectiveness changes in each version.

With this analysis, we have shown that subgraph defectiveness changes over system evolution and that hypotheses H4 and H5 are rejected.

5.2.3 H6: The Jonckheere–Terpstra test is a test for an ordered alternative hypothesis within independent samples. It is similar to the Kruskal–Wallis test. The hypothesis is that independent samples are from the same population. However, with the Kruskal–Wallis test, there is no a priori ordering of the populations from which the samples are drawn. Therefore, the Jonckheere test has more statistical power than the Kruskal–Wallis test. The test has the null hypothesis that the distribution of subgraphs is the same across different software versions. The statistical significance value of the Jonckheere–Terpstra test was 0. As $p < 0.05$, the null hypothesis can be rejected, which means that there is a difference in subgraph defectiveness over versions. With that finding, an alternative hypothesis is accepted, meaning that subgraph defectiveness is decreasing with software evolution. Null hypothesis H6 was rejected, and therefore, the alternative hypothesis, which states that subgraph defectiveness is changing as software evolves, was accepted for all subgraph types.

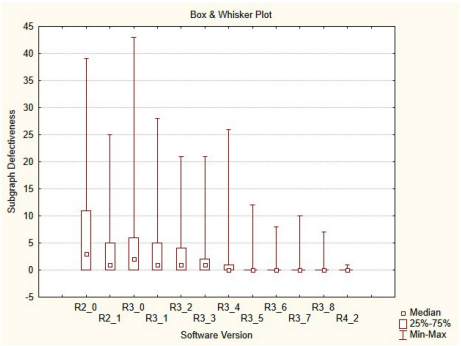


a

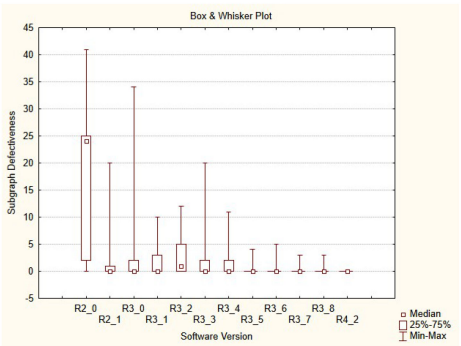


b

Fig. 8 *Box plots for JDT subgraph ids*
(a) Subgraph ID 38, ***(b)*** Subgraph ID 36

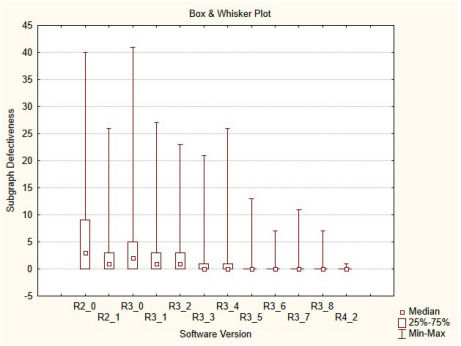


a

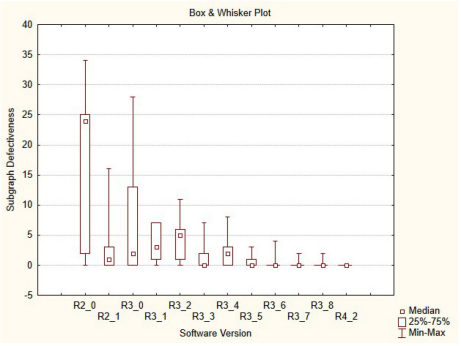


b

Fig. 10 *Box plots for JDT subgraph ids*
(a) Subgraph ID 12, ***(b)*** Subgraph ID 14

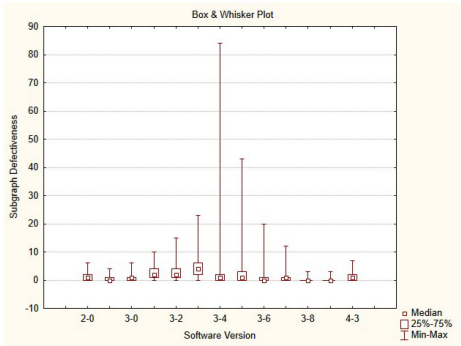


a

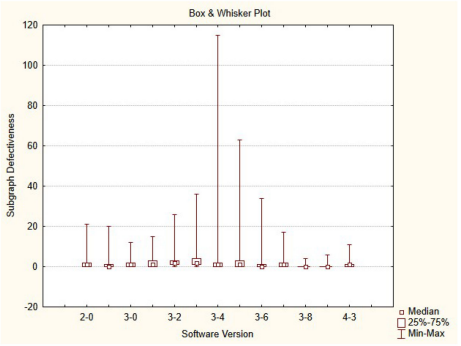


b

Fig. 9 *Box plots for JDT subgraph ids*
(a) Subgraph ID 6, ***(b)*** Subgraph ID 46



a



b

Fig. 11 *Box plots for PDE subgraph ids*
(a) Subgraph ID 38, ***(b)*** Subgraph ID 36

5.3 Hypotheses about effects of subgraph defectiveness evolution on system defect proneness

5.3.1 H7: A Spearman correlation coefficient between average defectiveness and total defectiveness by subgraph type is presented in Table 6 for the JDT project and in Table 7 for the PDE project. We can observe that not all types have a $p-$value < 0.05, which means that we can reject the idea that the correlation is due to random sampling for types 6, 12, 36, 38, and 78, but have no compelling evidence that the correlation is real and not due to chance for types 14, 46, and 74. Since there are much fewer cases where types 14, 46, and 74 are present, it is to be expected that they have much less influence on total defectiveness. For types 6, 12, 36, and 38, the correlation value (S value) is higher than for the rest, indicating that there is a very strong positive relation between those types of defectiveness and total defectiveness, for the type 74 relation is strong or moderate and weak for types 14 and 46. These tests have shown that the average number of defectiveness has a

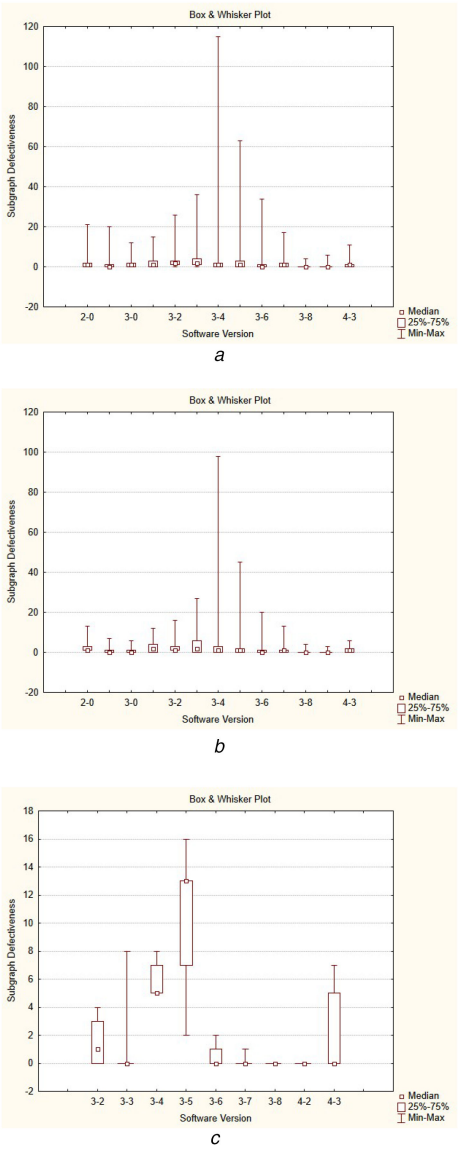


Fig. 12 *Box plots for PDE subgraph ids*
(a) Subgraph ID 6, ***(b)*** Subgraph ID 12, ***(c)*** Subgraph ID 14

correlation with the total number of defects in versions, which means that H7 cannot be rejected.

6 Discussion

As SDP is a growing field of research, our aim in this paper was to explore the usage of software structure in terms of network graphs as a possible direction in that research. In our research, we used network subgraphs and information about defects in every defined subgraph in the projects that we used for the analysis.

At the beginning of the research, we defined four research questions in Section 1 that we wished to answer in this paper.

6.1 RQ1 – Which subgraphs with three nodes are the most defective in a software system?

The first question we investigated was whether there is a subgraph type that is the most defective in the software system. We looked at the average defectiveness in all the subgraph types. Hypothesis H1 was used to help us answer this question. We tested the hypothesis by comparing the average defectiveness of the subgraph types. The subgraph types with fewer connections (edges) were more common than the types with multiple connections. Those simple types of subgraphs most often have connections as library calls, using interfaces and communications where one node calls two nodes that are not mutually connected. Despite the fact that those subgraph types were the most common, they did not have the highest average defectiveness. The types that are rarer have significantly higher average defectiveness as the software system starts evolving. At the beginning of the software, in the first few versions, average defectiveness is truly higher in the subgraph types with larger numbers of connections. As the system evolves, average defectiveness equalises through all subgraph types. Our answer to the question of whether there is a subgraph type that is more defective than the others is yes, types with more connections, such as types 78, 46, and 14, but only as the software system starts evolving. As the testing process becomes more intense and the system starts to grow, there does not seem to be a considerable difference between the types. With this finding, it is possible to alert the programmer as the complex communication patterns based on subgraph types are being formed inside his code.

Table 6 Spearman coefficient on the JDT project

Subgraph id	$s-$value	$p-$value
6	0.864	0.00
12	0.854	0.00
14	0.336	0.17
36	0.997	0.00
38	0.855	0.00
46	0.313	0.219
74	0.515	0.059
78	0.651	0.0415

Table 7 Spearman coefficient on the PDE project

Subgraph id	$s-$value	$p-$value
6	0.9174	0.00
12	0.874	0.00
14	0.886	0.00
36	1	0.00
38	0.874	0.00

6.2 RQ2 – Is the average subgraph defectiveness similar across the different software systems and in the different software versions of the same evolving software system?

The second question investigated concerned the differences between subgraph type average defectiveness across different versions; in other words, we investigated if the average subgraph defectiveness is similar across different systems and in different versions of the same evolving system. We defined hypotheses H2 and H3 in order to see how the given data sets behave in the context of this research question. We used the average number of defects present in all occurrences of each subgraph type in all versions. For comparison, we used the average number of defects of one type in all versions. Analysis was run using different sets of statistical test. It was run on all subgraph types appearing in the system versions. The results were that there is a difference in the average defectiveness of a subgraph type as the system evolves, which is to be expected. As the new versions of the system are launched, defects are being corrected as the testing process goes on. There are some versions in which the number of defects grows for every subgraph type, such as in the PDE versions from 3_3 to 3_6. The reason for this could be adding code that is very faulty or simply that the testing effort has been greatly enhanced. Our hypothesis H2 stating that the average defectiveness is the same across different versions of the software program was rejected and the hypothesis H3 that states that the average subgraph defectiveness is the same between different software programs could not be rejected for the given data set. We concluded that in this software system, the average subgraph defectiveness is not similar across the different software systems and in the different software versions of the same evolving software system, but we cannot say that it is not the same when looking at the subgraph defectiveness between two different systems that come from the same Eclipse environment.

6.3 RQ3 – How does the subgraph defectiveness evolve over the software versions, and does it stabilise as the software system evolves?

For every system, it is conventional that as the system evolves, its size also grows. In terms of software, that size is the size of the software code. Gathering information about the size of the code and the number of defects for every part of the code helped us to answer the question about subgraph defectiveness evolution and its stabilisation through the hypotheses H4, H5, and H6. When analysing the hypotheses H4 and H5, we tested to see if the defectiveness is the same across the system evolution, both between different versions and between subgraph types in the same version. The defectiveness was not the same in both cases, and we concluded that the defectiveness changes for each subgraph type as the system evolves. In both the PDE and JDT projects, it seems that in releasing version 4_2, the only differences from version 3_8 were in the number of defects. It seems that between those versions, there was no new code added, and only the testing process was performed. In the analysed system, we cannot conclude that there is a strict direction in which the subgraph defectiveness evolves. In the JDT project, subgraph defectiveness generally tends to decrease with the system evolution, but in the PDE project, there are versions in which it decreases and grows unexpectedly, but the direction of those changes is the same for all the subgraph types. Owing to those unpredictable changes, we cannot say that the subgraph defectiveness stabilises during the system evolution in this study. Hypothesis H6 stating that the subgraph defectiveness tends to stabilise during system evolution was rejected and the alternative hypothesis which states that subgraph defectiveness is changing as the software evolves was accepted.

6.4 RQ4 – Is there a relation between the average subgraph defectiveness and the software system defectiveness?

The last question we analysed through hypothesis H7 was if there is any relation between average subgraph defectiveness and system defectiveness. The results for this hypothesis and possible conclusions based on those results are unclear. There is a strong relation between the average subgraph defectiveness of some subgraph types, but those subgraph types are present in the system in quite larger numbers than the other types. There are some implications that analysing the system using subgraph types could be beneficial for defect prediction. Further research in that direction could be favourable for SDP but at the moment, we can hardly reach definitive conclusions.

7 Threats to validity

Threats to internal validity are present when causal relations are examined. The study was conducted by looking at the relationship of the defects and code structures. Factors, such as the developing team, testing team, and organisation structure, were not taken into consideration. In the process of building subgraphs, communication between classes, such as inheritance and composition, was not taken into account because of the tool's limitations. It is possible that when including those types of relationships, the results could vary.

Threats to external validity are conditions that limit the ability to generalise the results. The data we tested were from an open-source project, Eclipse, written in Java. Such data cannot be generalised to industry projects or to software written in other programming languages, as we do not have any proof that our conclusions could be applicable outside of the Eclipse environment. In this study, only two subsets of the Eclipse project were tested. Only complex parts of systems are taken into account, since there are parts that are simply too small for subgraph analysis. Selecting only the more complex parts could represent bias. Furthermore, in this paper, we used the classes as nodes and we cannot claim that the results will be the same if other units were used. The greatest difference between the two data sets used in this study is their size. The difference in size can be seen in Tables 1 and 2. Despite those differences, the results obtained were similar in both projects. However, this is still too small sample to be able to give generalised conclusions.

In this study, we used structural measures that are independent of static code properties and therefore are more general than the classical code metrics used in various studies. The three-node subgraphs, the structural measure that we used in this paper, have been investigated on 234 open-source Git repositories and the results presented in [28] show similar and independent behaviour. In papers [32, 33], the data sets from Java Qualitas Corpus [34] are used for investigating the network metrics and the directed graphs as structural metric. Both papers came to the same conclusion. Those measures are independent from the specific software system and are general properties of the Java software systems. The comparison between Java and Erlang software systems by using structural metrics has been conducted in [35]. The analysis in the paper shows that the behaviour between systems, in terms of the three-node subgraphs, written in Erlang and Java is similar. Thus, all this findings encourage the use of structural metrics and its ability to generalise the conclusions. However, in this paper, we further investigate its relation to defects and we cannot claim that the conclusions obtained on limited data sets used in this study can be generalised before replicating the study to other software projects and contexts.

In order to run an analysis similar to this, the data set has to contain information that is quite hard to obtain from software industry because data set combines information from source code and defect detection process. The whole source code has to be available as well as information about the defects and its relation to the source code. For each defect, we have to have a place in the code where the defect was introduced. In other words, we need the name of the class in which the defect was found and therefore, the Java Qualitas Corpus [34] could not be used. For that reason, only Eclipse projects were used, as we did not have other appropriate data sets that were written in Java, since in this study, we focused solely on Java software. For extracting the necessary information, the rFind for structural metrics and BuCo tool for class–defect relation was used. Since the class–defect relation is not collected during the development process and is not available from the repositories, this relation we obtained from data mining process implemented in BuCo tool. A detailed analysis of the tool performance can be found in [29]. The results of the study show that the tool is highly effective. The results also show that when compared with other more complex techniques, such as the approach used by Relink [36], the BuCo tool is more efficient. The recall of the tool, when tested on Eclipse JDT R2_0, is 99.58%. The precision is 99.89%, and the *F*-score is 99.74%, both of which are higher results than for Relink.

Making the data set used publicly available is a tremendous problem, since the data set contains 32 GB of information, including details in addition to those described within this study, but it is available upon request.

8 Conclusions and future work

The application of network analysis approaches to software and other areas of science is still a widely open area of research. Here, we applied structural analysis based on subgraph types that are representing primitive three-node communication patterns within the graph structure. Analysing the evolution of software by representing the software system using the frequencies of subgraph types is beneficial because it allows for structural analysis involving communication patterns. Fault distributions over the software structure have been widely analysed in many studies, but these studies only weakly study the effects of communication structures within fault distribution analysis.

In this paper, we have shown that Eclipse software programs have similar behaviours in terms of average subgraph type defectiveness and distributions of average subgraph frequencies coming from the same population. However, in each software program, the defectiveness of different subgraph types – representing different communication patterns – behave differently, and we cannot find evidence that they come from the same population. This leads us to conclude that communication

interactions formalised within subgraph types indeed have an influence on system defectiveness.

The group of hypotheses related to the distributions of subgraph defectiveness and their evolution have shown that subgraph defectiveness changes over system evolution and that subgraph defectiveness does not tend to stabilise during system evolution. Although subgraph frequencies grow during the system evolution and do not stabilise, we can observe a contrary effect that the subgraph defectiveness is decreasing as the system matures. This approach to software analysis with the help of subgraph structures offers a liveness property that other metrics, such as static code attributes and software process metrics, may not be able to offer.

We found that subgraph defectiveness has a strong correlation with the number of defects in a system version. New insights obtained with this approach may be useful in better fault prevention by providing software architects with new architecture design guidelines or by providing software quality personnel with a set of risky subgraphs so that they may better plan verification and thus be more effective in defect detection activities.

In our future work, we aim to investigate how different subgraphs influence other metrics of software. Additionally, analysing software written in other languages could be interesting. We can also see if software evolution, in terms of subgraphs, in other languages tends to evolve in the same way and if the same subgraphs are also present.

Even though it seems that the network measures are general for the software systems, we do not know if the defects behaviour in terms of subgraph can be generalised. This question has to be further investigated. Further research can be beneficial for the SDP.

9 Acknowledgments

This work has been supported in part by Croatian Science Foundation's funding of the project UIP-2014-09-7945 and by the University of Rijeka Research grant no. 13.09.2.2.16.

10 References

[1] Andersson, C., Runeson, P.: 'A replicated quantitative analysis of fault distributions in complex software systems', *IEEE Trans. Softw. Eng.*, 2007, **33**, (5), pp. 273–286

[2] Concas, G., Marchesi, M., Murgia, A., *et al.*: 'On the distribution of bugs in the eclipse system', *IEEE Trans. Softw. Eng.*, 2011, **37**, (6), pp. 872–877

[3] Fenton, N. E., Ohlsson, N.: 'Quantitative analysis of faults and failures in a complex software system', *IEEE Trans. Softw. Eng.*, 2000, **26**, (8), pp. 797–814

[4] Galinac Grbac, T., Huljenić, D.: 'On the probability distribution of faults in complex software systems', *Inf. Softw. Technol.*, 2015, **58**, (2015), pp. 250–258

[5] Galinac Grbac, T., Runeson, P., Huljenić, D.: 'A second replicated quantitative analysis of fault distributions in complex software systems', *IEEE Trans. Softw. Eng.*, 2013, **39**, (4), pp. 462–476

[6] Zhang, H.: 'On the distribution of software faults', *IEEE Trans. Softw. Eng.*, 2008, **34**, (2), pp. 301–302

[7] Milo, R., Shen-Orr, S., Itzkovitz, S., *et al.*: 'Network motifs: simple building blocks of complex networks', *Science*, 2002, **298**, (5594), pp. 824–827

[8] Petrić, J., Galinac Grbac, T.: 'Software structure evolution and relation to system defectiveness'. Proc. of the 18th Int. Conf. on Evaluation and Assessment in Software Engineering EASE2014, London, UK, 2014, Article 34, 10 pages

[9] Hall, T., Beecham, S., Bowes, D., *et al.*: 'A systematic literature review on fault prediction performance in software engineering', *IEEE Trans. Softw. Eng.*, 2012, **38**, (6), pp. 1276–1304

[10] Adams, E.: 'Optimizing preventive service of software products', *IBM Res. J.*, 1984, **28**, (1), pp. 2–14

[11] Basili, V. R., Perricone, B. T.: 'Software errors and complexity: an empirical investigation', *ACM Commun.*, 1984, **27**, (1), pp. 42–52

[12] Hatton, L.: 'Reexamining the fault density-component size connection', *IEEE Softw.*, 1997, **14**, (2), pp. 89–97

[13] Krishnan, S., Lutz, R. R., Goševa-Popstojanova, K.: 'Empirical evaluation of reliability improvement in an evolving software product line'. MSR 2011, New York, NY, USA, 2011, pp. 103–112

[14] Zimmermann, T., Nagappan, N.: 'Predicting defects using network analysis on dependency graphs'. ICSE 2008, Leipzig, Germany, 2008, pp. 531–540

[15] Tosun, A., Turhan, B., Bener, A.: 'Validation of network measures as indicators of defective modules in software systems'. PROMISE 2009, New York, NY, USA, 2009, Article 5, 9 pages

[16] Bhattacharya, P., Iliofotou, M., Neamtiu, I., *et al.*: 'Graph-based analysis and prediction for software evolution'. ICSE 2012, Zurich, Switzerland, 2012, pp. 419–429

[17] Concas, G., Monni, C., Orru', M., *et al.*: 'A study of the community structure of a complex software network'. Proc. of the 4th Int. Workshop on Emerging Trends in Software Metrics (WETSoM), San Francisco, CA, USA, 2013, pp. 14–20

[18] Li, Z., Tian, J., Zhao, P.: 'Software reliability estimate with duplicated components based on connection structure', *Cybern. Inf. Technol.*, 2014, **14**, (3), pp. 3–13

[19] Qian, Y., Minyan, L., Luyi, L.: 'Critical nodes evaluation in large-scale software based on static structure and runtime information'. IEEE Int. Conf. on Software Quality, Reliability and Security, Companion, Vancouver, BC, 2015, pp. 186–187

[20] Lehman, M. M., Belady, L. A.: '*Program evolution: processes of software change*' (Academic Press Prof., Inc., San Diego, CA, 1985)

[21] Fernandez-Ramil, J., Lozano, A., Wermelinger, M., *et al.*: 'Empirical studies of open source evolution', in Mens, T., Demeyer, S. (Eds.): '*Software evolution*' (Springer, Berlin, Heidelberg, 2008), pp. 263–288

[22] Mohagheghi, P., Conradi, R.: 'Quality, productivity and economic benefits of software reuse: a review of industrial studies', *Empir. Softw. Eng.*, 2007, **12**, (5), pp. 471–516

[23] Herraiz, I., González-Barahona, J. M., Robles, G., *et al.*: 'On the prediction of the evolution of libre software projects'. ICSM 2007, Paris, France, 2007, pp. 405–414

[24] Graves, T.L., Karr, A.F., Marron, J.S., *et al.*: 'Predicting fault incidence using software change history', *IEEE Trans. Softw. Eng.*, 2007, **26**, (7), pp. 653–661

[25] Zimmermann, T., Nagappan, N., Gall, H., *et al.*: 'Cross-project defect prediction: a large scale experiment on data vs. domain vs. process'. Proc. of the ESEC/FSE '09, New York, NY, USA, 2009, pp. 91–100

[26] Belderrar, A., Kpodjedo, S., Guéhéneuc, Y., *et al.*: 'Sub-graph mining: identifying micro-architectures in evolving object-oriented software'. CSMR 2011, Oldenburg, Germany, 2011, pp. 171–180

[27] Zhang, S., Ai, J., Li, X.: 'Correlation between the distribution of software bugs and network motifs'. IEEE Int. Conf. on Software Quality, Reliability and Security (QRS), Vienna, 2016, pp. 202–213

[28] Petrić, J., Galinac Grbac, T., Dubravac, M.: 'Processing and data collection of program structures in open source repositories'. Proc. of SQAMIA 2014, Lovran, Croatia, 2014, pp. 57–66

[29] Mauša, G., Galinac Grbac, T., Dalbelo Bašić, B.: 'A systematic data collection procedure for software defect prediction', *COMSIS J.*, 2016, **13**, (1), pp. 173–197

[30] Cardillo, G.: 'Jonckheere–Terpstra test: a non-parametric test for trend'. Available at http://www.mathworks.com/matlabcentral/fileexchange/22159, accessed July 2018

[31] http://www.seiplab.riteh.uniri.hr/wp-content/uploads/2018/07/Software+structure+evolution+and+relation+to+subgraph+defectiveness+graphs.pdf

[32] Tonelli, R., Concas, G., Marchesi, M., *et al.*: 'An analysis of SNA metrics on the Java Qualitas Corpus'. Proc. of the 4th Annual India Software Engineering Conf., ISEC 2011, India, February 2011, pp. 205–213

[33] Chong, C. Y., Lee, S. P.: 'Analyzing maintainability and reliability of object-oriented software using weighted complex network', *J. Syst. Softw.*, 2015, **110**, (2015), pp. 28–53

[34] Tempero, E., Anslow, C., Dietrich, J., *et al.*: 'The Qualitas Corpus: a curated collection of Java code for empirical studies'. 2010 17th Asia Pacific Software Engineering Conf. (APSEC), Sydney, NSW, Australia, 2010, pp. 336–345

[35] Vrankovic, A., Galinac Grbac, T., Tóth, M.: 'Comparison of software structures in Java and Erlang programming languages'. Proc. of SQAMIA 2017, Beograd, Serbia, September 2017, pp. 18–26

[36] Wu, R., Zhang, H., Kim, S., *et al.*: 'Relink: recovering links between bugs and changes'. Proc. of ESEC/FSE 11, New York, USA, 2011, pp. 15–25