Preliminary study of higher dimensional software structures

Emili Puh[*], Tihana Galinac Grbac[*] and Neven Grbac

Juraj Dobrila University of Pula, Zagrebačka 30, Pula, HR-52100, Croatia

Abstract

Quality of large-scale mission-critical software systems depends on software system architecture. Although we design and create software system architecture we are still unable to evaluate how architectural decisions influence software behavior. This problem is becoming even more important in the context of future networks which assume autonomic software creation and interconnection guided by special needs and purposes of its creation where software architectures are created autonomously.

One of the approaches to this problem is in development of software structure metrics that can be used as characterization metrics for architecture comparison and evaluation of its properties. Our previous work proposed to use of motifs for such purposes. However, its application is limited because of computational efficiency and we were able to investigate only 3–node motifs. Here in this paper, we provide a preliminary study investigating how higher dimensional substructures within software architecture may be used as software metrics and if these higher dimensional structures bring benefit to software characterization. Here, we search for higher dimensional substructures and aim to investigate their growth and change trends in the context of evolving software systems. We find out that structure behavior we may only understand by having a multidimensional structure view. Furthermore, we observed that the initial project size may correlate to structure stabilization across the product evolution. More precisely, projects with larger initial sizes may be slower in stabilizing the structure of internal dependencies while projects with smaller initial sizes may stabilize the structure of internal dependencies in just a few project releases.

Keywords

Software structure, software call graphs, higher dimensional subgraphs, software topology

1. Introduction

One of the key research goals within the software engineering community was to understand the mechanisms of software dynamics and its relations to the software static attributes which we can measure during the software creation [8]. Although the software engineering discipline has matured over the years of evolution we still lack instruments to model software behavior or more precisely, to model software operational characteristics during the design time. In most cases, the software modeling approaches are focused on implementing logic that would deliver functional software characteristics not on modeling its operational behavior. One active

SQAMIA 2023: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10–13, 2023, Bratislava, Slovakia

[*]Corresponding authors.

[†]These authors contributed equally.

epuh@student.unipu.hr (E. Puh); tihana.galinac@unipu.hr (T. Galinac Grbac); neven.grbac@unipu.hr (N. Grbac)

0000-0002-4351-4082 (T. Galinac Grbac); 0000-0001-6657-6297 (N. Grbac)

direction of software engineering research is aiming to find appropriate instruments that would allow modeling software behavior already during the design time and thus more precisely design software products that would better fit its purpose and its operational goals.

The software architecture is one of the main artifacts we design and create during the software design phase. Usually, the modeling of software structure and architecture is based on static software attributes or based on human subjective decisions [7]. However, its impact on software dynamics has been weakly explored.

One large group of research is devoted to developing various models for *Software Defect Prediction, SDP* experimenting with various statistical approaches, machine learning, and artificial intelligence that aims to build models based on aforementioned static software metrics measured on software components such as software size, development effort, detected during the design phase, and various software complexity metrics. However, these approaches have almost neglected the existence of the software communication structure and focus solely on static software metrics on files, modules, and classes (we will call them modules further on), and correlate these static metrics with a number of failures or faults identified during the software operation. These approaches rely on the module clustering principle based solely on data similarity measured on those modules. However, many researchers have recognized the influence of communication patterns within the software system on the fault and failure behavior and operational software characteristics [22, 23, 24, 25]. Furthermore, communication patterns persist during the system evolution and may uncover fault and failure prediction abilities across system versions. However, a recent literature review of cross-project predictions showed that papers are mainly based on these static metrics aiming to improve data preprocessing and feature selection methods [12] and there the system architecture view is weakly represented within these metrics.

On the other hand, many approaches have been investigated for software structure analysis. Previous research efforts have mainly focused on applying complex network science based on the software dependency or call graph, [1, 2, 3, 4, 5]. Investigating the dependency network metrics [20, 21, 14] and network science approaches to software defect prediction is still an active area of research. Recent studies have identified that its direct application alongside all software files may not be effective. However, some great improvements may be achieved if these models are used to target some particular modules [13]. More precisely, these models may be beneficial for module clustering within the software structure.

Our previous study has been motivated by the aim to find appropriate software structure characterization metrics that would enable us to further improve software defect prediction modes. We found that information obtained from software structure expressed as the frequency of the three-node subgraphs may be in correlation with software defects, [5, 17, 15]. This finding is aligned with numerous previous observations that the most faulty modules are exactly the ones that implement the most communication links with other modules. Moreover, we identified that the same 3-node subgraphs are present across all system releases but their frequency changes as the system evolves and we statistically proved that such measured structure evolution is continuous and there are no signs of structure stabilization as the system matures [5]. Here in this paper, our research is further exploring higher dimension subgraphs and their representation within the software structure and aims to identify what subgraph dimensions are represented within the software structures and what is the highest subgraph dimension present within the software structures. Furthermore, we want to explore how subgraphs evolve and change across from the various structural dimensions within the system evolution.

The paper is structured as follows.

In Sect. 2 we provide an overview of the software structure analysis on the evolving software systems by introducing the concept of software structure, providing details of Eclipse dataset which is frequently used for SDP research, providing an overview of related works on software structures. Furthermore, in Sect. 3 we explain how we extracted the higher dimensions and the results we obtained. In Sect. 4 we discuss the issues that may represent threats to the results and conclusions we derive based on the selected study case. Finally, in the Sect. 5 we conclude the paper.

2. Software structure evolution analysis

2.1. Research background

In this section, we aim to provide an overview of the research in finding appropriate models for delivering best-effort software quality by doing appropriate actions during the design of the software. One of the research directions is to predict and detect as many as possible defects early, already during the design time, and to additionally invest in quality assurance activities on selected-prediction supported faulty parts. In this line of research belong numerous software defect prediction studies aiming to predict faulty software parts needed additional attention with the main benefit to save on focused quality activities that are invested into high-risk defective modules. The majority of such modules are employing machine learning and artificial intelligence tools to find high-risk modules based on historical data and fining relations between faulty modules' incidence and their static attributes. There are numerous metrics investigated to measure static code attributes and therefore a lot of studies reported challenges of dimensionality reduction and feature selection procedures, [9, 10]. In spite of numerous static metrics, it is identified that SDP models could be improved with the additional characterization of the problem it studies and that software structure is not well represented among existing static code attributes. The system structure is an important design time artifact however we still do not have adequate models to asses its goodness for the purpose it is developed. Our previous studies have already demonstrated that some representation of the software structure may contain useful information to model software defect prediction [15, 5]. Therefore, here in this paper, we focus our efforts on the further explanation of how some hidden structural characterization obtained from topological information and higher dimensional structure can support SDP.

Software architecture is an abstract term that we usually use in the software design phase. Mostly it is used to represent a software structure as a set of software components i.e. modules, files, or classes and their interactions [7]. Software structure may be represented by a call graph as indicated in [1]. We reused the concept of presenting software structure as a call graph from [19] and develop a rFind tool for automatic call graph extraction from Java source code that we presented in [5]. Actually, the call graphs are useful instruments to identify the dependencies that exist between parts of the software system. The working of the tool may be simply explained with the help of the figure 1 where on the left-hand side is represented

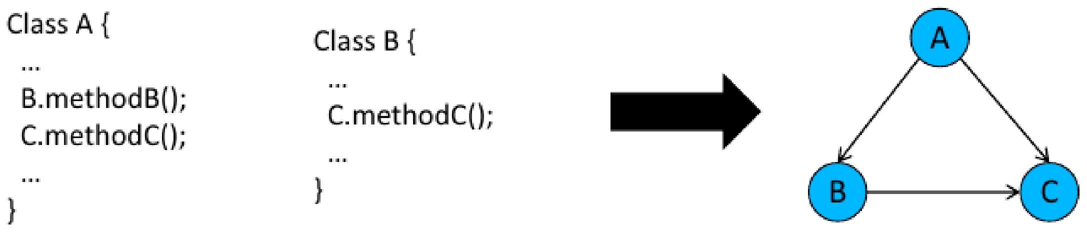

Figure 1: Call graph as a representation of software structure, as presented in our previous work [17].

software code and on the right-hand side is represented by a call graph. The example of software consists of three classes A, B, and C. Each class is represented by a node on the call graph of the right side. Furthermore, the method calls among the classes, methodB(), and methodC() are represented by vertices or edges of the software structure call graph presented on the right side of the 1.

2.2. Datasets in the study

In our previous works, we analyzed the software structure from open-source software repositories from the Eclipse community. Eclipse is an open-source community that stores its software versions on the open GIT repository (https://git.eclipse.org/c/). For the purposes of our study, we selected Java Development Tools (JDT) and Plug-in development environment (PDE) and Business Intelligence and Reporting Tools (BIRT) which is an Eclipse Platform-based reporting system used to create data visualizations and reports that can be embedded into rich client and web applications. These projects were selected because they are often analyzed in various research papers so our selection was motivated with the aim to understand the obtained results and their meaning when integrated into the existing knowledge base. Our study involves 14 JDT, 13 PDE, and 9 BIRT sequential releases. Most of the other studies analyzes only the first 3 releases of each of these projects. Here we wanted to analyze changes over the software evolution and therefore we undertake data collection for all available releases in the respective projects. The projects are developed during a longer period of time, BIRT in a period of 7 years, JDT 12 years of development and PDE is developed in 11 years.

The data collection process consists of the following steps. Firstly we downloaded from the GIT repository all class files related to each project release. Then we extracted call graphs with rFind tool that we developed within our research group and used in our previous studies for software structure analytics [5, 17, 15]. Furthermore, we collected a number of defect data per release in Bugzilla open-source project bug report repository. For precise mapping of classes with defects we have developed a tool BuCo [26, 27]. Furthermore, we calculated static metrics for each project release. The data set contains information for all classes involved within each analyzed software release (36 software releases in total). For each software release, we collected more than 50 static metrics per class and per release.

Here, in this paper, we will base our analysis solely on the call graphs collected with rFind tool. But our future plans are to extend the analysis also for the context of SDP.

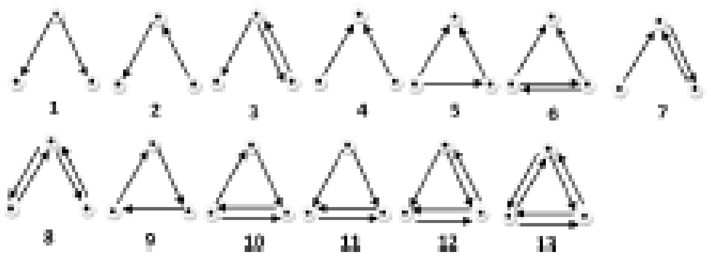

Figure 2: Subgraphs analysed within the graph software structure in our previous work [17]

2.3. Review on Eclipse studies

Firstly, in [5] we analyzed the characteristics of software structure when it is represented as a graph through the lens of subgraph occurrences and motifs. Motif is a graph property and is represented by the frequency of subgraph occurrence that is statistically significantly more represented in the observed graph than within a large number of random graphs. As has been already shown in many scientific fields such as medicine, sociology, and electrical engineering motifs are a good characterization of graph sources and may be used to differentiate graphs originating from different sources, e.g. motifs found in Escherichia coli, World Wide Web, and feed-forward networks differ, [16]. However, previous studies have focused their analysis only on three node subgraph motifs, which are presented in Fig. 2, since motif identification requires significant computational resources and finding higher order subgraph structures and their motifs is algorithmic complex and is costly in the sense of computational resources and time. In our previous study where we analyzed motifs within Eclipse software structures, we confirmed the finding from [16] that the object-oriented software has its unique characterization in terms of motifs 5 and 6 see Fig. 2. Furthermore, we proved that as software grows the significance of the specific motifs grows in the analyzed context of Eclipse software.

In our previous study [5] performed on Eclipse software, we also show that the same subgraph types are present in all versions of the system releases throughout the system evolution but their frequencies change as the system evolves. Also, not all subgraph types are present in all system versions. With the help of the subgraph occurrence instrument for representing the system structure, we proved that the system structure is significantly changing during the system evolution and that the system structure is continuously evolving. Moreover, we could not confirm that the software structure is tending to stabilize as the system matures. On the other hand, we found that the frequency of particular subgraph occurrences is correlated with system defects and is stabilizing as the system evolves. This finding has opened new research challenges that we further investigated and results are presented in [17]. In our further investigation, we were analyzing the impact of each three-node subgraph on the defectiveness of the code represented by this subgraph. We statistically proved on observed Eclipse software programs that different subgraph types behave differently in the sense of defectiveness of code represented by that subgraph. This finding leads us to the conclusion that communication interactions that are represented by different subgraph types behave differently and that the communication interactions among the classes of code have the influence of defects. It is worth noting that this conclusion is aligned with some previous research and empirical studies. However, this is the first study that has succeeded to measure and statistically prove the effect of communication patterns on software defectiveness. Furthermore, it is the same work we found that code defectiveness of different subgraphs does not stabilize as the system matures during the system evolution. Note that this finding is opposite to the code defectiveness of the system that tends to stabilize during the system evolution. Our findings are limited only to three-node subgraphs. Although we were able to generalize this finding on all three-node subgraphs represented within the analyzed system, we were not able to generalize this finding on higher-order subgraphs that may also be present within the system graph structure. It is interesting to further analyze at which higher-order subgraph this code defectiveness stabilization occurs.

Based on this analysis we may conclude that with the help of subgraph defectiveness, we may capture the code liveness property that other static code metrics may not be able to offer. Furthermore, we found that subgraph defectiveness is correlated to system release defectiveness and this finding opens new opportunities to further investigations within the Software Defect Prediction community.

Furthermore, we were analyzing the software structure with the help of network science models [20, 21]. Previous studies that imply graph theory for mathematical modeling of software architecture have analyzed the cyclomatic complexity of various structures, resilience to failures, and failure propagation through architecture, [7].

Here in this paper, we aim to further investigate higher-order structures in graphs that may bring some positive results in developing new models for smart software architecting. We believe that particular software graph structures may have bad influences on software behavior and should be avoided by proper software design decisions. Some earlier works have analyzed software modularity but we are not aware of studies analyzing this impact directly in relation to code defectiveness. Also, previous studies were analyzing the level of modularity between software classes as a number of communicating links but here we aim to investigate the behavior of higher-order software structures within the software evolution.

3. Results

The main goal of the study was to understand how software structures evolve with respect to the higher dimensional subgraph structures that are represented within the software structure. In our study, as explained in the previous sections, we observe software structure from the code dependency viewpoint and the main source of our analysis is the call graphs. As we have already explained, the call graphs are software structures formed from the nodes that represent Java classes from the Eclipse software and edges between these nodes that are represented by method calls between these classes.

Within the call graphs, the goal is to find how higher dimensional substructures are represented as follows:

- Zero-dimensional subgraphs (which we refer to as 0D) are the substructures present within the call graphs represented by nodes, i.e., these are just the classes present in the software.

- One-dimensional subgraphs (which we refer to as 1D)) are the substructures present within the call graphs represented by two nodes and an edge connecting these two nodes, i.e., these are just the method calls present in the software.
- Two-dimensional subgraphs (which we refer to as 2D) are the substructures present within the call graphs represented by three nodes fully connected by edges among them, i.e., these are the complete subgraphs with three nodes. These were studied as part of our research on motifs in the software [17, 15].
- Three-dimensional subgraphs (which we refer to as 3D) are the substructures present within the call graphs represented by four nodes fully connected by edges among them, i.e., these are the complete subgraphs with four nodes.
- And so on, k-dimensional subgraphs (which we refer to as kD) are the substructures present within the call graphs represented by $k + 1$ nodes fully connected by edges among them, i.e., these are the complete subgraphs with $k + 1$ nodes.

The dimension of these substructures refers to their geometric realization. The OD substructures are nodes, represented by points, the 1D substructures are edges, represented by segments, the 2D substructures can be viewed geometrically as triangles, the 3D substructures as tetrahedra, and so on. The higher dimensional substructure of dimension k can be viewed geometrically as the polyhedron with $k + 1$ vertices.

Figure 3 represents the trend of the software growth over the project releases a) in the zero dimension 0D represented by a number of nodes, and b) in the first dimension 1D represented by a number of edges. The figure provides also the results of linear regression analysis performed on the three Eclipse projects. The obtained regression parameters (a, b) are provided in Table 1 for each analyzed dimension and Eclipse product. From the figure, we can observe that there is bigger growth in 0D and 1D (the number of classes and method calls) at the beginning of the evolution, and then, in the middle of the evolution, this growth is gradually decreasing or if we imagine infinite evolution we can say that size measured in 0D and 1D dimension has an asymptote towards some finite software size. This is also evident from the table1 where we can observe a reductions of a shape parameter (a) in higher dimensions.

From the figures and tables provided for 0D and 1D we may observe that size of the slope parameter may be in correlation with the initial software size, software size at the first release. The initial size of the BIRT project and all its subsequent releases is more then doubled with respect to the JDT and PDE projects and the same may be observed with the slope parameter in the respective projects.

Moreover, variations in growth size between releases are unstable in the first several releases, and then after some point, these variations stabilize as software evolves for JDT and PDE projects and tend to converge to some asymptote. Projects with smaller initial sizes have smaller growth variations compared to projects with larger initial sizes.

In Figure 4 we present the trend of software growth with respect to the 2D subgraphs structures along with the regression lines for each analyzed project. From the figure we may observe that in the case of smaller Eclipse projects (JDT and PDE) the software evolution has similar behavior as in lower dimensional substructures. However, the behavior of the largest project BIRT is somewhat different. In 2D we may observe that bigger projects have weaker linear regression fit, and we may observe larger deviation of the measurements from the linear

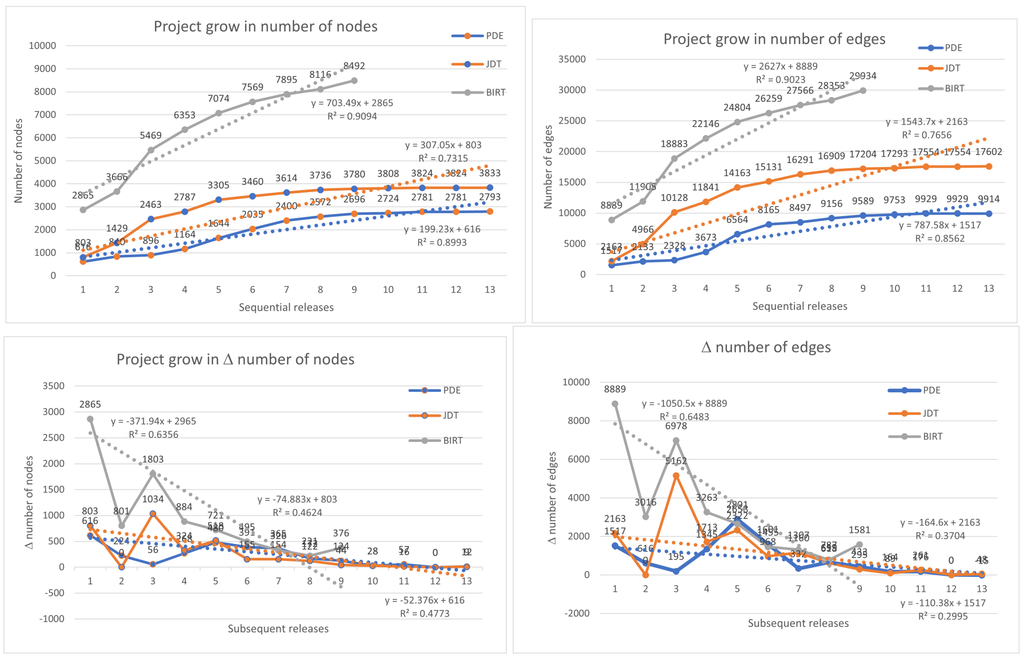

Figure 3: Project growth expressed by the growth of a) 0D - number of nodes, b) 1D number of edges, c) Δ 0D difference in number of nodes between the releases, and d) Δ 1D difference in number of edges between the releases

regression line. Software growth, expressed as a number of triangles in the 2D dimension, at the beginning of the evolution is much higher than in the second part of the analyzed evolution cycle. Like in 0D and 1D dimensions, we can observe that the slope parameter in the linear regression curve may be correlated with software size in the initial project release.

Variations in software change between the project releases, as a measure of structural change during the evolution, are more evident in the first half of the observed evolution cycle while in the second part of the evolution cycle, these structural changes tend to stabilize. It is interesting to observe the difference in software structure change trends for the three observed projects during the evolution. We can observe that the slope parameter in the linear regression curve of structure change is very low and resulting in an almost horizontal regression line while the slope parameter of the 2D structure change between releases is very high for BIRT in comparison with other two projects, JDT and PDE. For the BIRT project, we may also observe a larger deviation of the measured curve from the linear regression line than for the other two projects. It seems that larger projects have larger variations in structure change between the project releases than smaller projects.

Figure 5 presents the growth in the number of tetrahedral found in software structure and the change in the number of tetrahedral in software evolution. We can observe from the figure there is a significantly different trend in the number of tetrahedral over software evolution. The smallest project in the 0D dimensions, PDE, has a constant and very low number of tetrahedral and no growth during the evolution can be observed. Then, a project with a bit larger initial size, JDT, in the 0D dimension has a trend of almost linear growth in this 3D dimension. While,

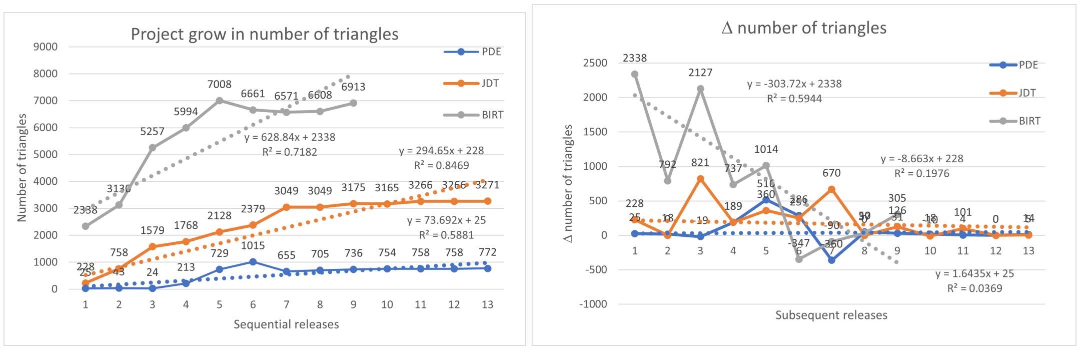

Figure 4: Project growth expressed by the growth of a) number of triangles, b) delta number of triangles

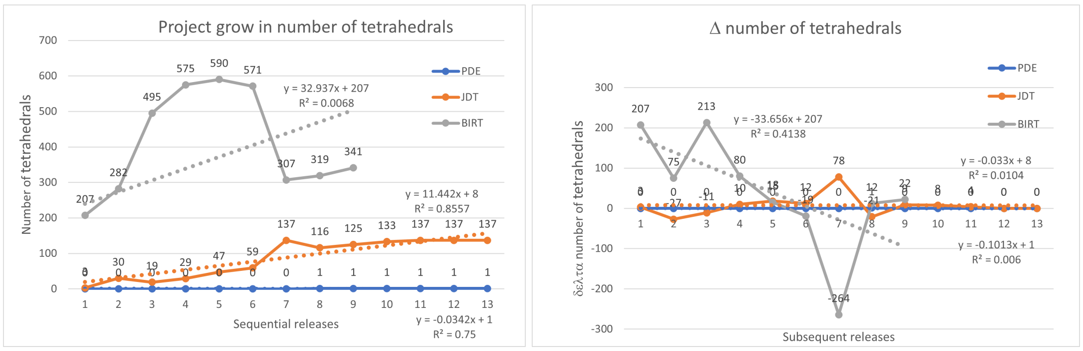

Figure 5: Project growth expressed by the growth of a) number of tethraedars, b) delta number of triangles

the project with the largest initial size in the 0D, BIRT, has rapid growth in the 3D dimension in the first three evolution cycles, then we may observe to stabilize and almost constant trend in the middle four versions of evolution and finally we may observe a high drop down almost to the initial size during the last three releases of evolution. It is interesting to observe how smaller projects JDT and PDE had relatively small or no variations in a growth change during the evolution while in the largest project, we may observe significant variations over the evolution.

4. Threats to validity

The current open-source development practice is still not matured to perform and develop clear measurement standards. In Software code analytics there has been numerous attempts to systematically collect source code metrics however the datasets are often criticised in sense of lack of systematic procedure. Besides numerous criticism, the SDP community have succeeded to develop some research results that improved not only the SDP research but also found applications in other areas of research and practice. However, recent studies have indicated on low reports from industrial case studies [11].

There are numerous issues one should address: remove test source files, identify relation of bug report and bug source (usually it is hard to match the exact location where the bug is

Table 1
Regression line parameters a, b (slope and intercept) for the analyzed Eclipse projects (PDE, JDT and BIRT) in different structural dimensions(0D, 1D, 2D, and 3D)

Project name	PDE	JDT	BIRT
0D	199.23, 616	307.05, 803	703.49, 2865
1D	787.58, 1517	1543.7, 2163	2627, 8889
2D	73.692, 25	294.65, 228	628.84, 2338
3D	-0.0342, 1	11.442, 8	32.937, 207
$\Delta 0D$	-52.376, 616	74.88, 803	-371.94, 2965
$\Delta 1D$	-110.38, 1517	-164.6, 2163	-1050.5, 8889
$\Delta 2D$	1.6435, 25	-8.663, 228	-303.72, 2338
$\Delta 3D$	0.1013, 1	0.033, 8	-33.656, 207

corrected in which code file and there exist numerous approaches to establish unique standards to address that problem.), remove duplicates (which may be hard).

Furthermore, in our study, we used the call graphs that are measuring dependencies among software parts as a source for grounding conclusions. The call graphs have been widely explored within the software engineering community aiming to develop better software design and analysis tools that would be able to perform and predict impact or vulnerability detection, analysis, risk assessment, etc. Using call graphs as a source to bring structural and quality conclusions have several threats. First of all, there are also numerous tools developed for call graph extraction from the source codes. One comparative study on the capabilities and effectiveness of various call graph extraction tools has been performed in [6]. The results indicate that each tool has its strengths and weaknesses. Furthermore, there are different views on software architecture. It may be seen from the runtime view, i.e. dependencies that are obtained from running code and capturing its processing flows or it may be obtained from dependencies that are obtained from statical code analysis by extracting all the possible calls among the system source files.

Despite numerous criticisms, some datasets become standard for various machine learning, deep learning and artificial intelligence approaches to software defect prediction. It turns positive to have at least some dataset, although not fully correct, to compare various algorithms on the unique base. The prerequisite for such an action is to have open datasets so the community can easily approach them and experiment. The conclusion is that it is more important to have a standard metric or dataset to bring valuable conclusions than to have a very precise and complex procedure. In that sense, we have developed tools for data collection and have systematically collected the data for all the datasets in the case study. Since we collected the data on the same Eclipse projects as the open datasets (PROMISE) we have the possibility to compare our results and analyze the differences. Moreover, our tools go beyond the PROMISE dataset because it connects two communities of software structure analysis and software defect prediction.

Here, we do not claim to have a completely ideal dataset. However, we do our best to collect thorough data and report on all possible misunderstandings. We did not use the usual datasets (PROMISE) because we do not have full control over the data in analysis. Although we have performed systematic comparisons with these datasets and reported on differences we identified.

5. Concluson

From our analysis, it is obvious that we can observe different trends in structure change when we observe the same software structure in different dimensions. Moreover, we may observe that this trend is somehow related to the project size of its initial release.

Projects that are initially smaller tend to stabilize already in lower-order dimensions and in fewer evolution steps. While for the projects with larger initial sizes in the zero and one dimensions, we may capture size growth stabilization at higher-order dimensions. Therefore, from the analyzed case we observed that initial project size may be in relation to trends observed at various structural dimensions.

All these imply that the evolution of structure growth has to be analyzed across various dimensions and a complete overview of structure growth behavior we may only understand by having such a multidimensional approach. One dimension of structural analysis may not be enough to understand structure evolution.

Furthermore, we concluded that the project's initial size is very important for its further evolution. Projects with initially larger sizes may have greater opportunities for future growth. If we observe this conclusion in the context of call graphs that are representing code dependencies we may say that projects with higher initial code complexity may have more evolution cycles before they stabilize. Also, it is interesting to observe the meaning of stabilization in higher-order dimensions. Projects with higher initial sizes have more fluctuations in higher dimensions and have a slower trend to stabilize. Also, we observe higher rising trends in higher dimensions. It is interesting to observe a significant downtrend in the third dimension of the largest project in our analysis. Probable, due to unpredictable complexity there was some redesign of software structure that might explain such a downtrend.

In analysing Eclipse projects we were able to find representations up to three-dimensional software structures. This implies that software structures are not much represented in higher dimensional structures and thus finding higher dimensional structures may not be a computationaly demanding task. However, we analysed just Eclipse projects and our further explorative analysis would involve a much wider study case.

6. Acknowledgments

This paper acknowledges the support of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No. 2020–1–PT01–KA203–078646: "SusTrainable - Promoting Sustainability as a Fundamental Driver in Software Development Training and Education" and the support of the Croatian Science Foundation under the project HRZZ-IP-2019-04-4216. The information and views set out in this paper are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

References

[1] Callahan, D., Carle, A., Hall, M.W., Kennedy, K.: Constructing the procedure call multigraph. IEEE Transactions on Software Engineering **16**(4), 483–487 (1990).

[2] Ryder, B.G. (May 1979). "Constructing the Call Graph of a Program". IEEE Transactions on Software Engineering. SE-5 (3): 216–226.

[3] Grove, David; DeFouw, Greg; Dean, Jeffrey; Chambers, Craig; Grove, David; DeFouw, Greg; Dean, Jeffrey; Chambers, Craig (9 October 1997). "Call graph construction in object-oriented languages". ACM SIGPLAN Notices. ACM. 32 (10): 108, 108–124, 124.

[4] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, Dimitris Mitropoulos, "PyCG: Practical Call Graph Generation in Python", 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp.1646-1657, 2021.

[5] Jean Petric, Tihana Galinac Grbac, Mario Dubravac, "Processing and Data Collection of Program Structures in Open Source Repositories". Proceedings of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA), pp. 57-66, 2014.

[6] G. Antal, P. Hegedus, Z. Tóth, R. Ferenc and T. Gyimóthy, "[Research Paper] Static JavaScript Call Graphs: A Comparative Study," 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), Madrid, Spain, 2018, pp. 177-186, doi: 10.1109/SCAM.2018.00028.

[7] Daniel S. Santos, Brauner R. N. Oliveira, Rick Kazman, and Elisa Y. Nakagawa. 2022. Evaluation of Systems-of-Systems Software Architectures: State of the Art and Future Perspectives. ACM Comput. Surv. 55, 4, Article 67 (May 2023), 35 pages.

[8] Eisenbarth, T.; Koschke, R.; Simon, D. (2001). "Aiding program comprehension by static and dynamic feature analysis". Proceedings IEEE International Conference on Software Maintenance. ICSM 2001: 602–611.

[9] Giray, G., Bennin, K.E., Köksal, Ö, Babur, Ö, Tekinerdogan, B.: On the use of deep learning in software defect prediction. Journal of Systems and Software **195**, Article no. 111537 (2023)

[10] Pandey, S.K., Mishra, R.B., Tripathi, A.K.: Machine learning based methods for software fault prediction: A survey. Expert Systems with Applications **172**, Article no. 114595 (2021)

[11] Szymon Stradowski and Lech Madeyski. 2023. Industrial applications of software defect prediction using machine learning: A business-driven systematic literature review. Inf. Softw. Technol. 159, C (Jul 2023).

[12] S. Pal and A. Sillitti, "Cross-Project Defect Prediction: A Literature Review," in IEEE Access, vol. 10, pp. 118697-118717, 2022, doi: 10.1109/ACCESS.2022.3221184.

[13] L. Gong, G. K. Rajbahadur, A. E. Hassan and S. Jiang, "Revisiting the Impact of Dependency Network Metrics on Software Defect Prediction," in IEEE Transactions on Software Engineering, vol. 48, no. 12, pp. 5030-5049, 1 Dec. 2022, doi: 10.1109/TSE.2021.3131950.

[14] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in Proc. Int. Conf. Softw. Eng., 2008, pp. 531–540.

[15] Ana Vranković, Tihana Galinac Grbac, Željka Car, Software structure evolution and relation to subgraph defectiveness. IET Softw. 13(5): 355-367 (2019)

[16] Milo, R., Shen-Orr, S., Itzkovitz, S., et al.: 'Network motifs: simple building blocks of

complex networks', Science, 2002, 298, (5594), pp. 824–827

[17] Petrić, J., Galinac Grbac, T.: Software structure evolution and relation to system defectiveness. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14. pp. 34:1–34:10. ACM (2014)

[18] Pita Costa, J., Galinac Grbac, T.: The topological data analysis of time series failure data in software evolution. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. pp. 25–30. ICPE '17 Companion, ACM, New York, NY, USA (2017).

[19] Valverde, S., Solé, R.V.: Network motifs in computational graphs: a case study in software architecture. Physical review. E, Statistical, nonlinear, and soft matter physics **72 2 Pt 2**, 026107–1, 026107–8 (2005)

[20] Sanja Grbac Babic, Tihana Galinac Grbac: Network analysis of evolving software-systems. SoftCOM 2017: 1-5

[21] Sanja Grbac Babic, Tihana Galinac Grbac, Jonatan Lerga: Community structure of a complex software-system in evolution. MIPRO 2018: 1467-1471

[22] N.E. Fenton and N. Ohlsson. 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Trans. Software Eng. 8 (Aug 2000.), 797–814.

[23] C. Andersson and P. Runeson. 2007. A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. IEEE Transactions on Software Engineering 33, 5 (2007), 273–286

[24] T. Galinac Grbac, P. Runeson, and D. Huljeni. 2013. A Second Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. IEEE Transactions on Software Engineering 39, 4 (2013), 462–476.

[25] Vrankovic, Ana, and Tihana Galinac Grbac. "Replication of Quantitative Analysis of Fault Distributions on Open Source Complex Software Systems." SQAMIA. 2018. CEUR Workshop proceedings, Vol. 2217, paper 22.

[26] Goran Mauša, Paolo Perković, Tihana Galinac Grbac, Ivan Stajduhar: Techniques for Bug-Code Linking. SQAMIA 2014, 47-55.

[27] Mauša, Goran et al. "Software defect prediction with Bug-Code analyzer - A data collection tool demo." 2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM) (2014): 425-426.