Comparison of Software Structures in Java and Erlang Programming Languages

ANA VRANKOVIĆ, TIHANA GALINAC GRBAC, University of Rijeka, Faculty of Engineering
MELINDA TÓTH, ELTE Eötvös Loránd University, Budapest, Hungary

Empirical studies on fault behaviour in evolving complex software systems have shown that communication structures among the software entities such as classes, modules, software units and communications among them, is significantly affecting the system fault behaviour. Therefore, we were motivated to further investigate software structures. One interesting question is to investigate software structures from software products written in different programming languages. In this work we present our preliminary study for which we developed tools to examine software structures of software products written in Java and Erlang programming language. We provide details on how we extract software structure from software product and provide preliminary results analyzing four Erlang software products and four Java software products.

Categories and Subject Descriptors: H.2.11 [**Software Engineering**]: Software Architectures—*Languages*

General Terms: Software structure

Additional Key Words and Phrases: Network graph, Subgraphs

1. INTRODUCTION

Today, network analysis is used in many scientific fields. It has proven to be useful in numerous problems. In medicine, physics, sociology, electrical engineering, it helped solve diverse issues. In computer science it is used as a tool to understand software behaviour by structuring software dependencies as a network graph. Milo in [Milo et al. 2002] proposed network motifs as patterns in complex networks with higher appearance than in random networks and elaborate its purpose as hidden structural property that can be used in characterizing various complex networks. In our previous study, we analyzed the software structure using network analysis on software written in the Java programming language. For the purpose of network graph extraction we developed the tool rFind [Petric et al. 2014a; Petric and Grbac 2014]. By extracting software graph structures we identified structural changes during the releases of evolving Eclipse software product. In an aim to better understand software behaviour in terms of structural changes, and influence of programming language on obtained conclusions we want to expand our study to other programming languages. In this study, we present preliminary study on the results obtained by analysing software structures in products implemented in Erlang. The subject of this work is to explore whether the programming language has any influence on software structure

This work has been supported in part by Croatian Science Foundations funding of the project UIP-2014-09-7945 and by the University of Rijeka Research Grant 13.09.2.2.16 and by the Hungarian Government through the New National Excellence Program of the Ministry of Human Capacities.
Author's address: A. Vranković, Faculty of engineering, Vukovarska 58, 51000 Rijeka, Croatia; email: avrankovic@riteh.hr; T. Galinac Grbac, Faculty of engineering, Vukovarska 58, 51000 Rijeka, Croatia; email: tgalinac@riteh.hr; M. Tóth, Faculty if Informatics, Pázmány Péter sétány 1/C, 1117, Budapest, Hungary; email: tothmelinda@elte.hu

Copyright ©by the paper's authors. Copying permitted only for private and academic purposes.
In: Z. Budimac (ed.): Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, 11-13.9.2017, Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

in terms of network graphs. Like we did in our previous work [Milo et al. 2002; Petric and Grbac 2014; Petric et al. 2014a], we used thirteen subgraph types to study software structure, as presented in Figure 1. These subgraphs cover all three-node connections. All subgraph types present directed graphs where each node is a class/module and every edge is a connection between them. This preliminary study is based on simple comparison of subgraph counts present in software structures obtained from different Erlang and Java software products.

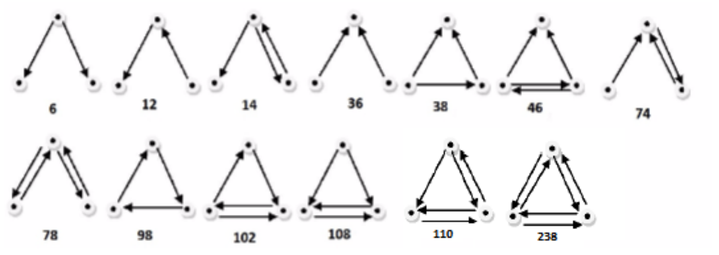

Fig. 1. Subgraph types

The rest of the paper is organized as follows. At first, in Section 2 is description of background, in Section 3 we present the used tools to extract the software structures in different programming languages. Then in Section 4 we present the preliminary results obtained by simple comparison of results obtained for Erlang and Java code. Finally, in Section 6 we conclude the paper.

2. BACKGROUND

To analyze software structure we can define different types of structures, modules, classes and different types of software unites. Graph theory is a field of study that looks into the formal description and analysis of graphs [Bullmore and Sporns 2009]. Part of graph theory study are also complex networks, graphs that are based on real world networks, they are discussed in [Simon 1991]. Analyzing system using network graphs and complex networks has been used in many scientific fields for a long time: in medicine, for protein analysis [Aristóteles Góes-Netoa and et al. 2010], in logistics [Carlos PaisMontes and Laxe 2013], crime analyses [Colladon and Remondi 2017], electrical system analyses [Alexandre P. Alves da Silva and Souza 2012] and many more. In computer science there has been a few ideas of using complex networks as a tool to better understand the software behavior. In [S.Jenkins and S.R.Kirk 2007] software architecture graphs were presented as a complex networks using Java written applications. Interesting finding was that as the software ages, more out-going calls than incoming calls are present. In paper [Chong and Lee 2015] complex networks are used as a tool for analyzing the complexity of software system based on object oriented approach. They used a weighted complex network on a system to help them understand its maintainability and reliability. They also managed to identify violations of common software design principals. In [Luis G. Moyanoa and Vargas 2011] the community structure of a real complex software network is explored. The results of this paper shows a significant dependence between community structure and internal dynamical processes. Relationships between Erlang processes have been discussed in [Bozó and Tóth 2016]. No work has been found on comparison between the Java and Erlang software structure. Since Java and Erlang are not similar in paradigm and are not usually used in the similar products, the comparison between them is not often explored. Therefore we wanted to compare them because of their differences to see if there is any variation in the way they communicate in terms of the subgraph type.

3. TOOLS

In this work we used four different tools.

For analyzing Java written applications we used the tool rFind [Petric and Grbac 2014; Petric et al. 2014a]. The input of rFind is an application code written in Java.

As an output we receive two files that allow us to see all calls between classes. One of those files is .classlist where a list of all classes is displayed using class ids for easier reading. Each class represents a node in the network. The other file is a .graph where all connections between class ids are presented. Every connection is viewed as an edge between nodes.

For getting the same information from Erlang applications we used the tool RefactorErl [Bozó et al. 2011]. RefactorErl is an open source static source code analyser and transformer tool for Erlang. RefactorErl supports dependency examination both on module and function level, and is able to present it as a graph to the user. The input of the tool were applications written in Erlang, and it was able to produce a textual representation of dependencies as an output. Although the presentation of the result was quite different from the output of rFind, but the main idea was the same: present communication between modules.

The SuBuCo tool [Petric et al. 2014b] is an application that expects an Rfind output as input: the .classlist and .graph. Then it searches for three-node subgraph structures inside .graph file. Its output is a list of all subgraphs that appear in the given code. The file created contains a list of all subgraphs separated by subgraph type and ids of every class/module contained in specific subgraph.

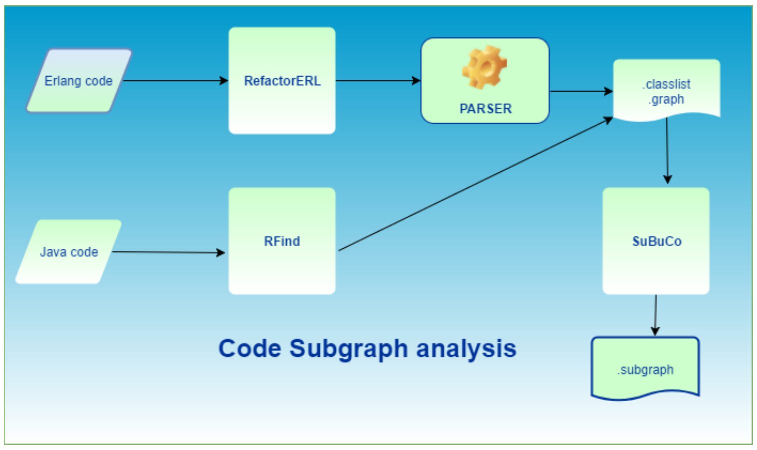

Fig. 2. Subgraph analysis process graph

Since the output of RefactorErl was not in the form for SuBuCo analysis, we wrote a parser to adjust the result so that it also contains .classlist and .graph. The parser was implemented in Java where the input files were files gathered from RefactorErl and the output files were the two needed files. The whole analysis process can be seen in Figure 2.

4. RESULTS

Our tests were conducted on four different software implemented in the Erlang programming language and four written in Java. The analysed Erlang software are: Mnesia for distributed telecommunications database; Dialyzer that allows static analysis for identifying software discrepancies; Cowboy which is a http server for Erlang/OTP; and RabbitMQ server that runs a multi-protocol messaging broker. The former two are part of the standard Erlang/OPT distribution, the latter two applications were taken from open git repositories. For analyzing software written in Java we used Java Development Kit (JDT) and Plug-in Development Environment (PDE) projects from Eclipse project, Open Microscopy Environment that is an open-source software and data format standards for the storage and manipulation of biological microscopy data, and Ultimate Android, development framework, from git repository.

Table I. Tested data

ERLANG PROJECT	NUMBER OF NODES	NUMBER OF EDGES	LOC
Mnesia	1914	6092	21417
Cowboy	510	948	4966
Dialyzer	1380	4089	14757
RabbitMq	3416	6648	23472
JAVA PROJECT	NUMBER OF NODES	NUMBER OF EDGES	LOC
OpenMicroscopy	3127	10775	438107
Ultimate Android	1893	9286	224442
JDT	3202	16923	606767
PDE	2542	9834	333390

Table II. Mnesia Results

ID	Appearance	Percentage
36	40999	67.24%
6	13053	21.407%
12	5994	9.83%
38	705	1.15623%
14	177	0.2903%
74	36	0.059%
46	6	0.00984%
98	2	0.00328%
78	2	0.00328%
102	0	0%
238	0	0%
110	0	0%
108	0	0%

Table III. Dialyzer Results

ID	Appearance	Percentage
6	24398	51.0664%
36	16466	34.4643%
12	5207	10.8986%
14	926	1.9382%
38	588	1.2307%
74	87	0.1821%
46	52	0.10884%
78	33	0.0691%
98	14	0.0293%
102	4	0.00837%
108	2	0.004186%
238	0	0%
110	0	0%

The number of edges and nodes for each tested software can be seen in Table I. Number of edges seems to be much larger in Java software, even where number of nodes is lesser then in Erlang software. In examples where the number of nodes are similar, Mnesia and Ultimate Android applications, number of edges is still much greater in Java application than in Erlang. Communication is far more common in Java written software. We can see that in all tested applications number of edges grows with the number of nodes.

Subgraph ids discussed in this section are referring to the network subgraphs in Figure 1. In three out of four applications in Erlang subgraph with id 36 was the most common. The same subgraph id also was the most present in both Java projects and in projects gathered from git repositories. It seems that the communication in which multiple classes/modules heavily use one library is the most frequent one. Only one had different results, Dialyzer. We can see from the Tables II-V that in all Erlang applications subgraph ids 36,6 and 12 are the most common ones, most often in that exact order. Subgraph with id 6 presents communications where one node needs multiple resources from other nodes and the subgraph with id 12 could be the situation where communication flows from one node to the other and when the second node is triggered he calls for the third node. In Tables VI-IX.

Table IV. RabbitMQ Results

ID	Appearance	Percentage
36	31870	58.0955%
6	14671	26.7436%
12	7759	14.144%
38	474	0.86405%
14	66	0.12031%
74	12	0.021875%
46	3	0.0054687%
108	2	0.003646%
102	1	0.001823%
98	1	0.001823%
238	0	0%
110	0	0%
78	0	0%

Table V. Cowboy Results

ID	Appearance	Percentage
36	1826	38.9755%
6	1388	29.6265%
12	1228	26.2113%
38	156	3.32968%
14	58	1.237994%
74	15	0.32017%
98	8	0.170758%
46	4	0.085379%
102	1	0.02135%
78	1	0.02135%
238	0	0%
110	0	0%
108	0	0%

we can see that Java projects behave similarly. In all projects it is the same order of frequency while in PDE id 38 is present more often than 12.

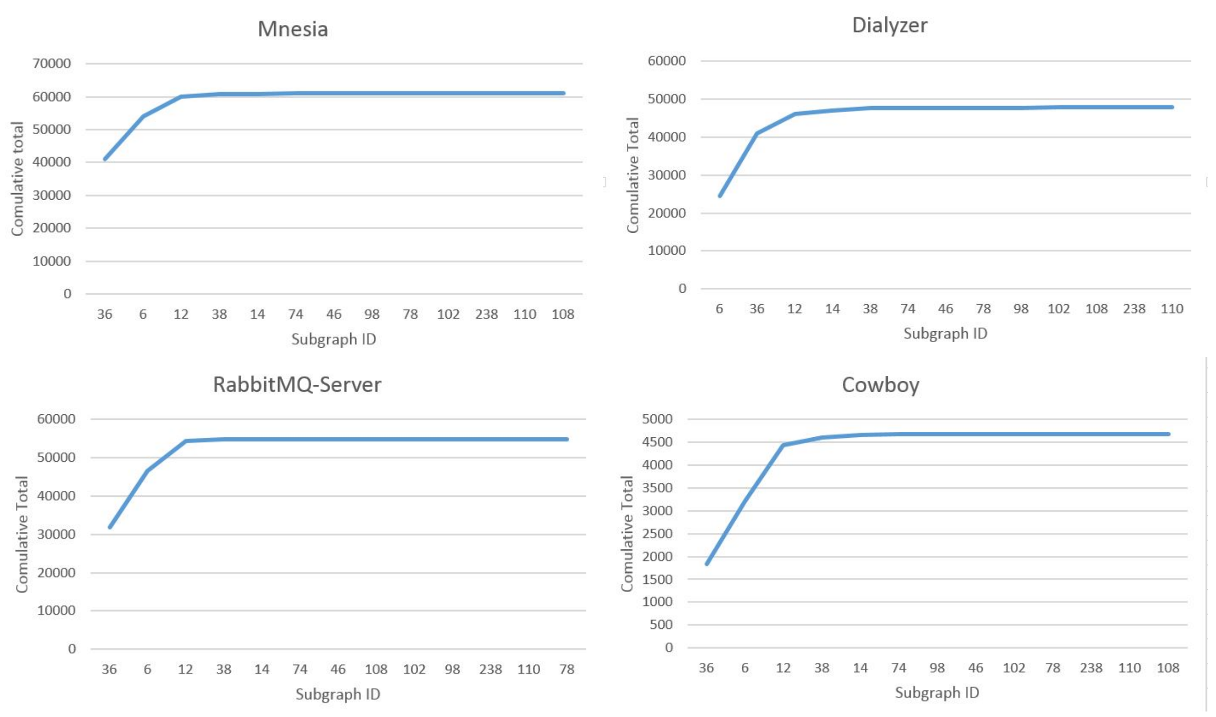

Fig. 3. Pareto graphs for open-source Erlang projects

In terms of subgraph id appearance, we can see that subgraphs with id 238 and 110 do not appear in any of Erlang application and neither in Java applications. Subgraph with ids 102 and 98 were found in Erlang application, but not in any of Java applications. We can see that applications written in Erlang and Java have similar behavior in terms of subgraph id appearance, even though Java software products are greater in class/module size.

Table VI. Ultimate Android Results

ID	Appearance	Percentage
36	463879	96.14989%
6	17338	3.5937105%
12	924	0.19152%
38	273	0.0565857%
14	30	0.00622%
74	6	0.001244%
46	4	0.00083%
102	0	0%
238	0	0%
98	0	0%
110	0	0%
108	0	0%
78	0	0%

Table VII. Java Development Tool Results

ID	Appearance	Percentage
36	1349284	89.5352%
6	132736	8.808%
12	18535	1.2299%
38	4469	0.29655%
74	952	0.0632%
14	951	0.0632%
78	45	0.00299%
46	44	0.00292%
102	0	0%
238	0	0%
98	0	0%
110	0	0%
108	0	0%

Table VIII. OpenMicroscopy Results

ID	Appearance	Percentage
36	802077	92.1518%
6	54724	6.2873%
12	10547	1.21176%
38	2792	0.32078%
14	167	0.01919%
74	45	0.00517%
46	17	0.00195%
108	14	0.00161%
78	4	0.00046%
102	0	0%
238	0	0%
98	0	0%
110	0	0%

Table IX. Plug-in Development Environment Results

ID	Appearance	Percentage
36	1064393	96.2593884%
6	36556	3.309%
12	3671	0.33199%
38	1042	0.09433%
14	81	0.007325%
74	11	0.0009948%
46	1	0.00009044%
102	0	0%
238	0	0%
98	0	0%
110	0	0%
108	0	0%
78	0	0%

Looking at Pareto diagrams on Figures 3 and 4 we can see that there is significant growth only for subgraph types 36,6,12 and 38 in both Java projects and Erlang projects.

5. THREATS TO VALIDITY

Data collection and analysis is possible on any code written in Erlang or Java. Erlang applications that were tested are server implementations, database and static analytic tool. Java software applications were frameworks for developing Java software and software for working with specific types of data. Software function is not the same in Erlang and Java applications. Since Erlang is a language used for scalable soft real-time systems and Java is general purpose programming language, comparing software applications written in each of them could not give us generalized conclusions. Comparing similar types of languages could be a better approach.

6. CONCLUSION

In this study our main focus was to analyze code structure on software written in Erlang and compare it to the software written in Java.

To do that we used several tools and combined them together to get the appropriate output that we can analyze. We represented class/module communication using thirteen subgraph types.

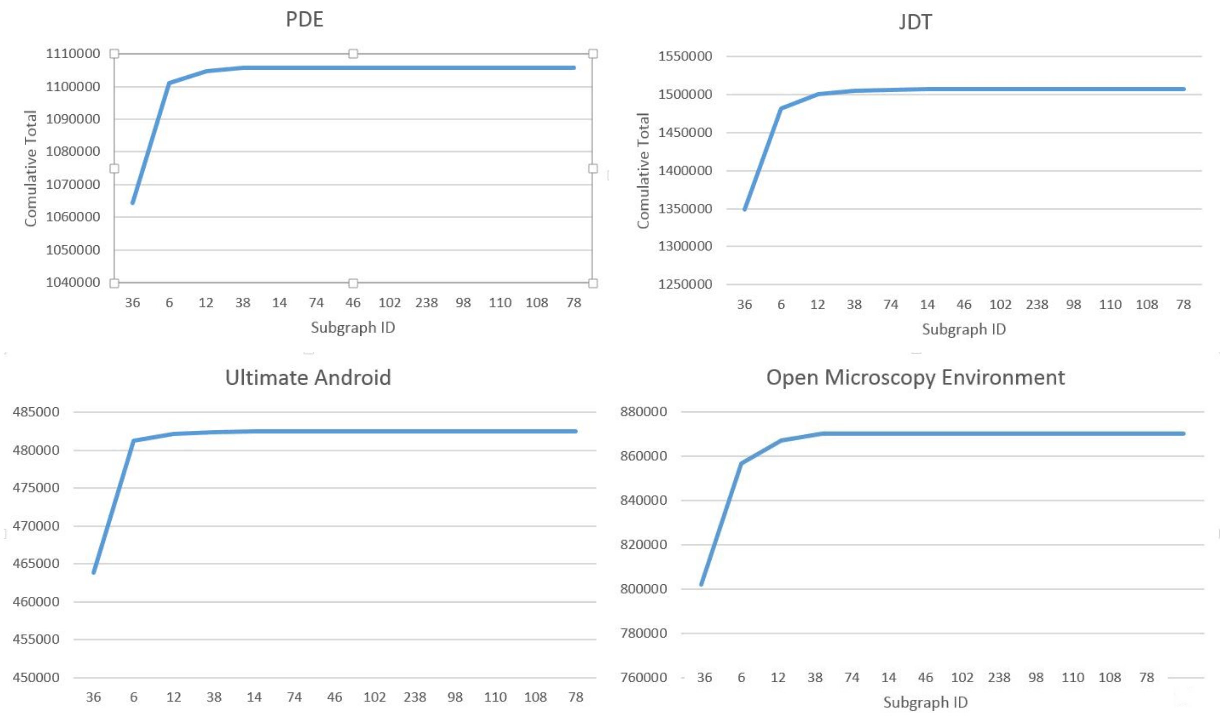

Fig. 4. Pareto graphs for open-source Java projects

In code written in Java, there was much larger number of communicating classes in comparison to communicating functions in Erlang code. Subgraph types 38,36,6,46,12,74 and 14 were present in all tested code. Types 108 and 78 were present in two software applications. Besides id 46, they are the only ones present that have the number of edges higher then 3. The one with the highest occurrence was subgraph type 36 in every tested application, followed by types 6 and 12. Subgraph ids 102,238,98 and 110 did not appear at all. There is no communication where more then four interactions between three classes are existent.

Unlike in Java applications, in Erlang applications subgraph id 98 occurred in all tested applications and id 102 appeared in two tested applications. Subgraph with id 98 is the only one where communication is circular, it starts and ends in the same node with just one interaction between each node. Just as in Java application, in Erlang applications ids 36,6 and 12 had the highest occurrence but in different percentage. While in Java applications subgraph id 36 occupied over 89% of all subgraphs, in Erlang the same id occupied between 58% and 68% while in Dialyzer it had appearance of only 34%. On tested java software, had a low appearance rate of under 10%. In Erlang applications result was different. Id 6 had a presence of around 20% in Mnesia and RabbitMQ. In Dialyzer, it had the largest number of appearance, 51%.

We can see that in Java written code, subgraph id 36 occupies more then 90% of all the communication while in Erlang code, ids 36 and 6 together occupy 80-90%.

Based on the code analysis, we can conclude that although there is similar behavior between languages, there are some differences. There are structures that appear in Erlang, but not in Java. Specifically structures where there is more communication edges between modules and id 98 where communication is circular. It is possible that those types are specific to that language. There is also a difference in percentage of the subgraph. While id 36 is in an extensive number of subgraph in Java, types of communication where one library is being heavily used by other classes, in Erlang that number is much lesser. We can see that the usage of libraries is greater in Java programs. There is also a big difference in number of communicating classes/modules. It seems that classes in Java programming language tend to communicate more often than Erlang modules. It is possible that those results are because of the fact that Java is an object oriented language and is based on object communication.

In our future work we aim to do the analysis on code written in other programming and scripting languages, both functional and object oriented. Doing that we can come to the determinant conclusion in aspect of which subgraph types are specific for individual programming languages or applications.

REFERENCES

Antonio C.S. Lima Alexandre P. Alves da Silva and Suzana M. Souza. 2012. Fault location on transmission lines using complex-domain neural networks. *Electrical Power and Energy Systems* 43 (Dec. 2012), 720–727. https://doi.org/10.1016/j.ijepes.2012.05.046

Marcelo V.C. Diniza Aristóteles Góes-Netoa and et al. 2010. Comparative protein analysis of the chitin metabolic pathway in extant organisms: A complex network approach. *BioSystems* 101, 1 (July 2010), 59–66.

I. Bozó, D. Horpácsi, R. Kitlei, Z.n Horváth, J. Kőszegi, M. Tejfel, and M. Tóth. 2011. RefactorErl-source code analysis and refactoring in Erlang. In *Proceeding of the12th Symposium on Programming Languages and Software Tools*. Tallin, Estonia.

István Bozó and Melinda Tóth. 2016. Analysing and Visualising Erlang Behaviours. *AIP Conference Proceedings* 1738 (June 2016). http://dx.doi.org/10.1063/1.4952023

E. Bullmore and O. Sporns. 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nat Rev Nurosci* 10 (April 2009), 186–198. DOI:http://dx.doi.org/10.1038/nrn2575

Maria Jesus Freire Seoane Carlos PaisMontes and Fernando Gonzalez Laxe. 2013. General cargo and containership emergent routes: Acomplex networks description. *Transport Policy* 24 (Nov. 2013), 126–140. https://doi.org/10.1016/j.tranpol.2012.06.022

Chun Yong Chong and Sai Peck Lee. 2015. Analyzing maintainability and reliability of object-oriented software using weighted complex network. *Journal of Systems and Software* 110 (Dec. 2015), 28–53. https://doi.org/10.1016/j.jss.2015.08.014

Andrea Fronzetti Colladon and Elisa Remondi. 2017. Using Social Network Analysis to Prevent Money Laundering. *Expert Systems With Applications* 67 (Jan. 2017), 49–58. https://doi.org/10.1016/j.eswa.2016.09.029

Mary Luz Mourontea Luis G. Moyanoa and Maria Luisa Vargas. 2011. Communities and dynamical processes in a complex software network. *Physica A* 390, 4 (Feb. 2011), 741–748. https://doi.org/10.1016/j.physa.2010.10.026

R. Milo, S. Shen-Orr, S. Itzkovitz, and et al. 2002. Network motifs: simple building blocks of complex networks. *Science* (Oct. 2002), 298:824–27.

Jean Petric and Tihana Galinac Grbac. 2014. Software structure evolution and relation to system defectiveness. *EASE* (May 2014). DOI:http://dx.doi.org/10.1145/2601248.2601287

Jean Petric, Tihana Galinac Grbac, and Mario Dubravac. 2014a. Processing and Data Collection of Program Structures in Open Source Repositories. In *Proceedings of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA 2014), Lovran, Croatia, September 19-22, 2014.* 57–66.

J. Petric, T. Galinac Grbac, and M. Dubravac. 2014b. Software structure evolution and relation to system defectiveness. In *Proceedings of SQAMIA 2014*. Lovran,Croatia, 57–66.

H. Simon. 1991. *The Architecture of Complexity, in: Facets of Systems Science* (1st ed.). Springer, Boston, MA, USA.

S.Jenkins and S.R.Kirk. 2007. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences* 177, 12 (June 2007), 2587–2601. https://doi.org/10.1016/j.ins.2007.01.021