Processing and data collection of program structures in open source repositories

JEAN PETRIĆ, TIHANA GALINAC GRBAC AND MARIO DUBRAVAC, University of Rijeka

Software structure analysis with help of network analysis showed promising results in software engineering community. Some previous studies have presented potential of representing software structure as subgraph frequencies, that are present in software structure using network graphs, for effective program characterization and differentiation. One of the prerequisites for exploration of this potential is collecting large dataset with number of different software projects. Nowadays, there are plenty of open source code repositories which not only they contain source code, but also provide a lot of crucial information useful in guiding software engineering actions. However, systematical building of such large dataset is highly challenging and time consuming process. Therefore, our approach is to automate data collection process for open source repositories aiming to provide as large as possible dataset that could provide us reliable conclusions. In this paper we present software structure tool analyzer based on subgraph frequencies, discuss its automation opportunities and illustrate preliminary results obtained by its usage. Some potential research directions are discussed.

Categories and Subject Descriptors: H.3.0 [**Information storage and retrieval**] General; D.2.8 [**Metrics**] Product metrics

General Terms: Experimentation

Additional Key Words and Phrases: open-source repositories, automatic tool, software analysis

1. INTRODUCTION

The use of network analysis in software analysis has been widely explored recently. In [Zimmermann and Nagappan 2008] authors showed that using of network graphs metrics can provide better results in the SDP (software defect prediction) than with classical software metrics. This fact motivated our analysis of software as network graphs.

One way to approach the network graphs is to analyze its basic building blocks as subgraphs and motifs. In our previous work [Petrić and Galinac Grbac 2014] we analyzed three Eclipse systems and got some interesting preliminary results. We realized that with help of subgraph frequencies we may differentiate different software. This is an important finding because it may be useful in many software engineering tasks - from planning software design and system modeling to planning the verification activities. In fact, variations of k–node subgraph frequencies, that in sum have probability one in the analyzed graph structure, are bounded. In order to discover these bounds in software projects we wanted to empirically investigate as much as possible software structures. We are interested what are the bounds and are they related to some other software metrics. Also we want to explore these bounds across the software domains, i.e. the software written for different purposes.

Nowadays, there are plenty of open source software repositories. Source code repositories and all supporting software development repositories contain a lot of potentially useful information. However, those information are weakly explored and used for guiding and executing software projects although

Author addresses: J. Petrić, e-mail: jean.petric@riteh.hr; T. Galinac Grbac, e-mail: tihana.galinac@riteh.hr; M. Dubravac, e-mail: mario.dubravac@riteh.hr; Faculty of Engineering, Vukovarska 58, HR-51000 Rijeka, Croatia

posses huge potential for improving the software engineering tools and practices. On the other hand, there are serious obstacles to this problem. Firstly, data are not usually available in the form that easily generate useful information, there is a lack of systematically collected data from diverse projects that could produce sound and general theories, and development of systematic and standard data collection procedures is limited due to huge diversity of repositories data structure. For example, in the SDP area, there was a huge number of papers which did not satisfy proper collection of datasets, leading in questionable models for SDP [Hall et al. 2012]. One of the main reason was inadequately defined data collection procedure. A process of collecting proper datasets for analysis in empirical software engineering is always a tough job to do. Empirical data are occasionally exposed to bias. First problem is that many studies are done with small datasets and small number of datasets, which may represent a serious threat to validity and lead to weak generalization of conclusions. Another problem is lack of freely available data from commercial software. When some research has been done on closed software it is impossible to repeat this research on same datasets, so everything remains on trust. Third problem is a limited number of investigated software domains for same model, e.g. a model for defect prediction. Fourth problem is in validity of information collected retroactively from the project repositories. Some useful information are stored by software developers in project repository and that process may be biased. Collecting the network subgraphs is free of the data collection bias introduced by software developers. Software structure is readable directly from the source code files. A huge potential for future evolution of software engineering discipline is in development of systematic data collection procedures that would be able to collect data consistently from diverse project repositories [Wnuk and Runeson 2013]. The challenge is in developing tools and procedures that would automatically collect and explore this information over different projects and repositories. The process of reading the software structure is not dependent on the repository, therefore we think there is a potential to automatize this data collection.

In this paper we want to explore opportunities to develop a generic tool that would automatically and continuously collecting live data from huge number of software project repositories just by simple searching the web. More precisely, we want to investigate opportunities to develop a live software structure analyzer. Such tool could be of great importance to the software development community since software structure has huge effect on its reliability and evolution [Galinac Grbac et al. 2013; Galinac Grbac and Huljenic 2014; Petrić and Galinac Grbac 2014]. An another important thing which motivated us to start analyze open-source repositories and to think about an automation process is our previous work [Petrić and Galinac Grbac 2014]. Namely, we did our analysis on limited datasets which included three different Eclipse plugins with more than 30 releases in total for all three systems. A process of collecting and processing data was very long and ineffective. Thus, we have started to think about some improvements. From the above observations, a better understanding of open-source repositories structure is mandatory in sense of collecting needed knowledge for an automating process, and also for developing such tool which would be able to process retrieved data.

Two main things will be covered. Firstly, we will discuss about diversity of open-source repositories, and their properties. Some basic information about few popular repositories will be explored, like their adequacy for a generic data collection. Secondly, based on findings we will define our new developed tool for purposes of automatic collecting datasets from different repositories. The tool is modular and very easy for extending with new functions. We briefly discuss its current main functions. Finally, we observe some difficulties in this live data collection process. We succeed to collect software structures for 234 open source software projects and based on simple observations we discuss our future work and future research directions.

The paper is organized as follows - in Sect. 2 more about open-source repositories will be said, in Sect. 3 the software structure collection tool will be described in detail, in Sect. 4 we describe our experiences of using the tool and provide some preliminary results of the collected data. Finally in Sect. 5 we will make a conclusion and say something about our future work.

2. OPEN-SOURCE REPOSITORIES

This section will describe how we approached to the open-source repository analysis, and how we determined for which repositories an automatic collecting process of data should be done. Open-source repositories are collection of data which are related to some project. Most often, such repositories are used for storing source code files, but this is not their only purpose. Sometimes, open-source repositories may contain vast number of data, which if they are processed in a proper way, may give crucial information about particular project. Today, many open-source repositories are available, and differences between them are properties they offer. For example, some repositories offer possibility for having a bug report system, some of them offer an unlimited number of projects, etc. In this research we have included only few repositories which show best overall properties, in terms of information they offer, a number of projects they have, etc. Information on many repositories over the Internet can be found in the comparative table available on Wikipedia [1].

From the objective parameters from that table it is obvious that many repositories share same or similar properties, so we also employed subjective parameters which should help us in making an easier decision. Thus, for subjective parameters we have introduced two main categories: number of prominent projects and how easy is to implement automatic retrieval of data for those repositories. In easy to implement category we have looked only how well repositories are organized, i.e. how briefly are links to the source code files provided, and how good search filters are implemented, i.e. is it easy to separate Java projects from C++ projects. Also, this category contains only poor, average and well keywords for determine how good organization and search filters are. For each keyword we gave the grade, thus poor is equal to 0, average is equal to 1 and well is equal to 2.

From the analysis of a number of open source projects we made following conclusions:

—Not all projects contain a link to the source code repository to directly load the project, but have a link just to the repository web page from where the user should manually search for the project
—Most of the projects store the source code just into the GIT repository, or at some point during development switch to GIT repository, without providing direct link to the source code for each version of the software
—Number of projects are very old and the repository is not maintained more or is not available
—Some repositories does not alow search by project name and have not list of all projects storing the source code. Project search should be done manually by searching for specific keywords and thus are hardly to automate
—Some repositories are limited to specific communities, e.g. CodePlex is limited to the Microsoft community and contains projects that are based on their technology

After we performed analysis on several repositories, we decided that we should include GitHub, SourceForge and Eclipse repositories in our automation process, because they got best overall grades according to objective and subjective parameters.

3. SOFTWARE STRUCTURE COLLECTION TOOL

To face some of the problems we discussed in previous sections, we decided to implement a modular tool which will be able to automate some processes in data retrieval. Because we have found that some

[1] http://en.wikipedia.org/wiki/Comparison_of_open-source_software_hosting_facilities

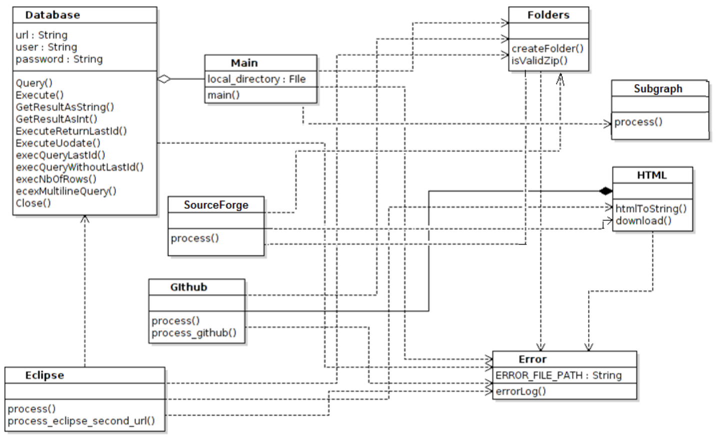

Fig. 1. Class diagram of the software structure collection tool

repositories have a good structure, i.e. a structure which is easy to be processed by some script tools, logical move was making a single tool which will be able to collect all needed data by themselves, and also to perform some additional tasks which will save time for further analysis. In that case, there is no need for manual collecting datasets from repositories, which is an important thing, because except that manual collecting data is a very time consuming process, it is also the process in which human error may have a significant impact. An automatic tool should collect all data in the same manner, which excludes occurrence of mistakes. Except aforementioned, such automatic tool is able to collect vast number of projects, so further analysis can be done on bigger datasets. Also, many software from different domains can be analyzed.

The class diagram of the implemented software structure collection tool is given on Figure 1. The software structure collection tool was mainly written in Java, but it also contains few external tools, which are rFind and JHawk. The rFind is a tool for searching relations between classes in a Java code, which is our previous work for transforming object-oriented software code into its graph representation described in [Petrić and Galinac Grbac 2014]. The software structure collection tool has few purposes, which can be divided in next phases:

—**Phase 1:** automatic retrieval of a Java source code from repositories

—**Phase 2:** uncompression of source code files

—**Phase 3:** transformation of source code files in its graph representation with the help of the rFind tool

—**Phase 4:** count of subgraphs occurrence from software representation of a graph

—**Phase 5:** a software metric collection

3.1 Phase 1: automatic data retrieval

Automatic retrieval was at first the main purpose of the software structure collection tool. After the analysis we had initially chose two repositories which satisfied our criteria. Additionally, we added and third repository which was the Eclipse. Finally, we also included a built-in option in the tool. The built-in option of the tool is able to get the list of the links as an input, which leads to different source code files. After the list is provided, the tool only process those source code files. Because the process

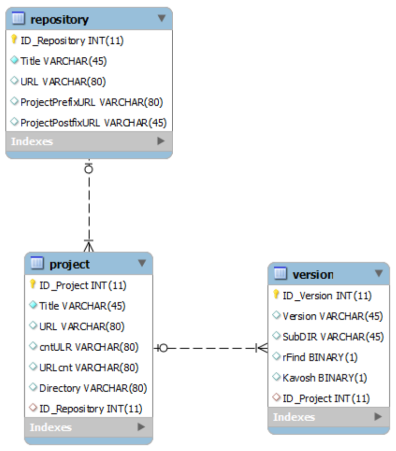

Fig. 2. An entity-relationship diagram for the software structure collection tool

can be interrupted by some unwanted events, e.g. a loss of power, we provided a recovery. The recovery enables us to be sure that already processed files will not be retrieved and processed again.

Because the software structure collection tool downloads many different types of software we have also built database for storing as much as possible information about retrieved software. The entity-relationship diagram of a database is given on Figure 2. The database contains important information about every project, such as a URL of the project, a version, etc. After each project is downloaded, the software structure collection tool continues on the phase 2. The phase 1 is repeated every time after the last phase is finished, and until there are non-processed projects in the repository.

3.2 Phase 2: uncompression of files

Immediately after the phase 1 is finished, uncompression of files begin. Repositories occasionally compress projects to preserve space on their servers, so before we can continue to the next phase, we must handle this step. Each repository uses different type of compressing files, but in most cases there are a limited number of used formats. In the software structure collection tool we have implemented support for uncompressing four different formats, which includes: zip, gz, bz2 and tar. We found that those formats are most occurring ones. If some different format is used for a compression, than this project will be discarded for the further analysis.

3.3 Phase 3: getting graph representation

This phase starts an external tool when is appropriate. After the phase 2 is finished, the rFind is called from the software structure collection tool. As is mentioned before, the rFind transforms a source code into its graph representation. To do this, it parses a code line by line and seeks for relations between classes. In that case, Java classes are nodes, and communication links are edges. A communication link means any relation between two classes. For example, if some class A tries to send a message to the class B through some method, then the edge is directed from the class A to the class B. For better understanding, the process is shown on Figure 3. Similar works also if one class tries to instantiate another class, or if it tries to communicate through parameters or a return value. For the current version of rFind, the communication in terms of class inheritance is not covered, so we do not record such relation. The rFind generates two files which are subject for further analysis. Those files are *graph* and *classlist*. Graph files contain information about communication links between classes, presented

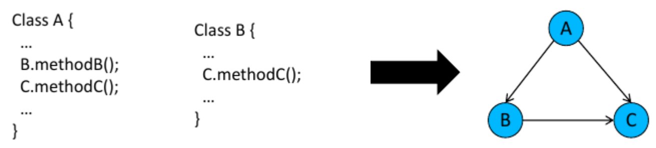

Fig. 3. Example of graph generated by rFind for given code

with IDs of the classes. To map those IDs with the corresponding classes we have a classlist in key-value format, where the key is an ID and the value is a class name.

3.4 Phase 4: subgraph frequency

The fourth phase of the software structure collection tool is a subgraph frequency counter. Its purpose is to count occurrences of all 3-node subgraphs in the given graph. We did not find any separated implementation of the subgraph counter, so we decided to use existing libraries, and adjust them to work in manner we expected. For this purpose we found the S-Space Package, which is a collection of algorithms for building Semantic Spaces as well as a highly-scalable library for designing new distributional semantics algorithms [2]. The S-Space has already implemented algorithms for processing subgraphs, but without a subgraph counter. Thus, because the S-Space is an open-source, we used few its algorithms to implement an expected behavior.

Non-separated versions of the subgraph counter can be found in motif tools. For example, mFinder [3] is a tool which provides getting of all n-node subgraphs for the given graph. The main problem is that this tool primarily searches for motifs, which is a very time-consuming process. So, our implementation of the subgraph counter has accelerated this process.

An another important thing is that our implementation of the subgraph counter can process any subgraph size, and any types of the subgraph, as long as it has correct formatted list of the subgraphs provided to the input. This means that our implementation is highly flexible. For example, if we want to find subgraph frequencies of subgraphs 1, 2 and 3 shown on Figure 4 we need to put this list as an input to the software structure collection tool:

```
1:1-2,1-3
2:1-2,3-1
3:1-2,1-3,3-1
```

According to the list above, we can put any type of the subgraph and count them in the given graph. Many motif tools does not have similar option.

3.5 Phase 5: software metrics collection

Currently, the last phase in the software structure collection tool is a software metric collection. For this purpose we used the external commercial tool JHawk [4]. JHawk is a static code analysis tool, i.e. it takes the source code of your project and calculates metrics based on numerous aspects of the code, e.g. volume, complexity, relationships between class and packages and relationships within classes and packages. After this process is finished all data are stored, and the process continues with the first phase until all projects are processed.

[2]https://github.com/fozziethebeat/S-Space/
[3]http://www.weizmann.ac.il/mcb/UriAlon/NetworkMotifsSW/mfinder/MfinderManual.pdf
[4]http://www.virtualmachinery.com/jhawkprod.htm

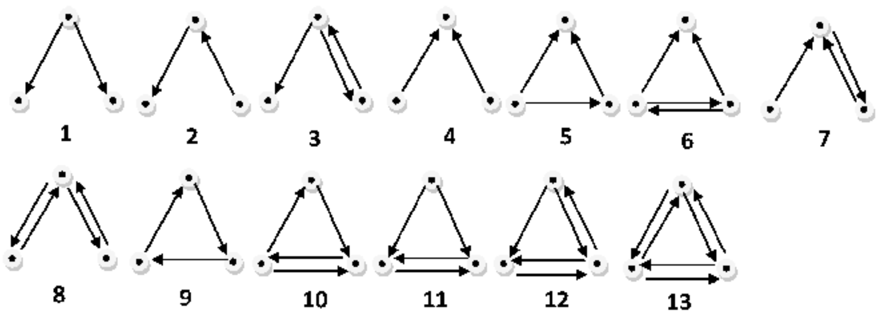

Fig. 4. All 3-node subgraphs

Table I. Descriptive statistics of the dataset

	No of 3-node subgr.	No of subgr. 6	No of subgr. 12	No of subgr. 14	No of subgr. 36	No of subgr. 38	No of packages	No of classes	No of methods	No of lines of code
Mean	80318	1593	368	0,02	78449	34	0,02	1703	10979	112511
Stdev	374760	3296	872	0,14	374529	77	0,13	2865	19737	210433
Min	3	3	0	0	0	0	1	4	66	818
Max	3291827	21343	6460	1	3287902	478	1646	16234	121811	1445451

4. EVALUATION OF EXPERIENCES AND PRELIMINARY RESULTS

In this section we discuss benefits and limitations of using the tool for the purpose of automatic data collection. Furthermore, we present some preliminary results obtained by using the tool.

4.1 Experiences of using the tool

We start data collection with help of the tool described in section 3. The total time needed for collecting data varied for different phases implemented in tool, but also from the project to project. For example, the first phase of data collection highly depends on the project and size of the source code that may vary from several MB to 200 MB. Also, data collection time in the first phase is very much depended on the Internet link throughput available to server running the data collection tool. Subsequent phases, from two to four were executed relatively fast (in few minutes) and are not so sensitive on the project size. The last phase may be very time consuming, from several minutes for smaller projects to few hours for larger ones.

In the last phase we use JHawk tool for collecting metrics on the source code files. In most cases JHawk tool performs excellent, but for some projects we experienced some problems that block our data collection process. This happens for small and large projects and we could not eliminate that issue because the JHawk is a commercial tool and we do not have insight in it. In such cases we skipped that project and continue with the next one.

4.2 Descriptive statistics for datasets collected

As a result of data collection process we succeeded to collect data for 233 projects. For each project we counted subgraph frequencies for all 3–node subgraphs and collected all metrics on the class and system level (provided by JHawk tool). Descriptive statistics for the collected projects are given in Table I.

Figure 5 depicts the relative subgraph frequencies of 3–node subgraphs (6, 12, and 36) that are present in all analyzed projects. We did not provide the figure of relative subgraph frequencies for

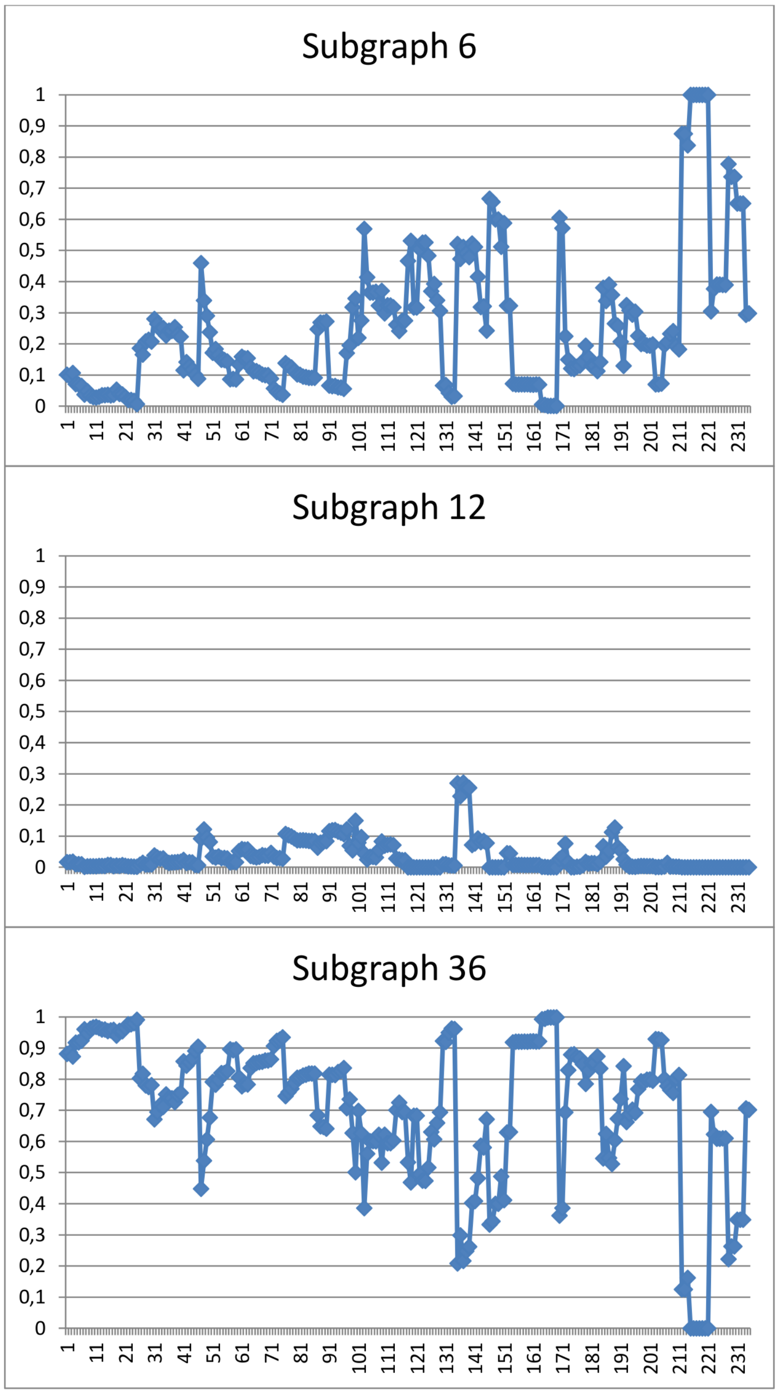

Fig. 5. Relative subgraph frequencies (3–node: 6, 12, 36) per software project. Note that software projects are ordered with respect to number of classes

Table II. Distribution of projects with respect to number of classes

No. of classes	0–200	201–400	401–600	601–800	801-1000	1001–1200	1201–1400
No. of projects	78	35	19	14	12	13	2
No. of classes	1401–1600	1601-1800	1801–2000	2001–2200	2201–2400	2401–2600	2601–2800
No. of projects	3	6	0	5	1	1	0
No. of classes	2801–3000	3001–4001	4001–5000	5001–6000	6001–7000	7001–8000	8001–9000
No. of projects	6	6	4	3	4	8	8
No. of classes	10001–11000	11001–12000	12001–13000	13001-14000	14001–15000	15001–17000	
No. of projects	0	1	1	2	1	0	

nodes 14 and 38 because are neglected, like for all other subgraphs because they are not present in neither of the analyzed projects. In all subfigures on Figure 5 the projects are ordered with respect to the number of classes present in the project source code. In the analyzed sample we have very well represented source codes with number of classes ranging from 0 to 8000. We conclude this from the distribution of projects collected over the categories with amount of classes given in Table II.

As it can be observed from the graph the frequencies of subgraph 6 are increasing with number of classes present in the project. On the other hand, the frequencies of subgraph 36 are decreasing.

We analyzed the occurrences of each 3–node subgraph (see Figure 4) and presented results in the separate subfigures for each subgraph that is represented in the analyzed projects. We found that all projects contain only subgraphs 6, 12, 14, 36, 38, 46, and 74. However, subgraphs 46 and 74 are very rarely represented in all of the analyzed projects, so we did not provide separate figure here. The most represented subgraph in all projects is the subgraph 38.

4.3 Limitations and future work

Future work should consider to improve the data collection tool with repositories that contain bigger projects. Since, the GIT repositories become a norm in the open source community the future work should consider how to automatically read structure data from all open projects stored in GIT.

Based on the observations while running the tool aiming to collect the dataset, we concluded that some phases may be computationally demanding. Future work should consider to eliminate execution stops due to data quality (e.g. problems with JHawk) and how to run the tool within the Cloud environment and without human intervention.

We aim collecting more datasets in the future that would cover wider spectra of projects than it is currently covered by the collected dataset. Some conclusions are limited in generalization due to relatively small projects that are collected. In our initial dataset mean subgraph frequencies of all analysed projects is 80k and 112 kLOC and for example some Eclipse projects that we collected have more than million 3-node subgraphs and about million lines of code. From this initial analysis we observed that the probability of particular subgraph occupancy in software may depend on the amount of classes, more packages, and more lines of code. We want to further study this observation and we are interested how we can define and develop models that would be useful for designing and verifying software systems.

5. CONCLUSION AND FURTHER WORK

In this paper we have presented a basic analysis of software structure that is collected from the open-source repositories and we have also introduced our modular software structure data collection tool.

With the software structure collection tool we have remarkably boosted our process of collecting datasets. Except that, we have also achieved an another important thing, which is avoiding of human errors in the collection process. Moreover, we identified weaknesses of our tool and discuss its future extensions.

From the collected datasets we performed a preliminary analysis and discuss future research directions in data collection. Such automation process will help us in future to collect a significant number of projects, that could truly represent a software from different domains on which we are going to perform additional analysis in terms of the subgraph frequencies and software metrics as continuation of our previous work [Petrić and Galinac Grbac 2014].

REFERENCES

Tihana Galinac Grbac and Darko Huljenic. 2014. On the Probability Distribution of Faults in Complex Software Systems. *Information and Software Tchnology* (2014). DOI:http://dx.doi.org/10.1016/j.infsof.2014.06.014

Tihana Galinac Grbac, Per Runeson, and Darko Huljenic. 2013. A Second Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. *Software Engineering, IEEE Transactions on* 39, 4 (April 2013), 462–476. DOI:http://dx.doi.org/10.1109/TSE.2012.46

Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on* 38, 6 (2012), 1276–1304.

Jean Petrić and Tihana Galinac Grbac. 2014. Software structure evolution and relation to system defectiveness. In *EASE 2014 (May 13–14, 2014)*. ACM Digital Library, 10. DOI:http://dx.doi.org/10.1145/2601248.2601287

Krzysztof Wnuk and Per Runeson. 2013. Engineering Open Innovation a Framework for Fostering Open Innovation. (2013). http://dx.doi.org/10.1007/978-3-642-39336-5_6

Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 531–540. DOI:http://dx.doi.org/10.1145/1368088.1368161