Comprehending Scenario-level Software Evolution using Calling Context Trees

Sarishty Gupta
National Institute of Technology
Jalandhar, India
sarishty.gupta@gmail.com

Paramvir Singh
National Institute of Technology
Jalandhar, India
singhpv@nitj.ac.in

Abstract— **Software evolution can be better understood if analyzed at user scenario level. Understanding the behavior of participating classes across a set of software versions could be helpful in scenario-level program comprehension. In this paper, we empirically investigate whether Calling Context Tree (CCT) based structural metrics provide new insights into comprehending the evolution of program scenarios. We analyze a set of four static, three dynamic, and four CCT metrics to comprehend the evolution of eight scenarios across four open source java applications. Correlation analysis and principal component analysis are used to analyze the relationship among the selected set of metrics. The results reveal that two of the CCT metrics have high correlation with selected static and dynamic metrics. The empirical results also suggest that CCT metrics are capable of providing additional valuable information for comprehending scenario-level software evolution.**

Keywords- software evolution; calling context tree; dynamic analysis; design and analysis of algorithm

I. INTRODUCTION

Software maintenance and evolution are vital to the success of modern-day software development. Software evolution is defined as "the dynamic behaviour of software systems as they are maintained and enhanced over their lifetimes" [4]. Software systems have to be continuously updated after their deployment in order to remain useful. Imminent changes in client requirements render software evolution inevitable. Software companies need their products to have the capacity to change as per the innovative demands of increasing software users and changing working environments. Incorporating software changes for successful maintenance and evolution requires better understanding of software functionalities, which can be achieved through effective program comprehension [22].

Software systems are becoming large and complex in order to accommodate advanced user requirements. Large software systems contain several millions of lines of code and voluminous documentation, which makes such systems quite difficult to comprehend [18]. Static program comprehension techniques face both data accuracy and performance concerns as they cannot provide the complete understanding of such large scale systems. On the other hand, although dynamic analysis techniques [3] may help in resolving the accuracy issues of static techniques, the former are still believed to cause even higher performance bottlenecks raising maintenance costs.

One possible solution is to reduce the scope of comprehension to the user scenario level. Scenario level analysis defines the behaviour of a software system from a user centric perspective [10]. Hence, the software engineers view user scenarios as a powerful way to determine user needs, and to discover the behaviour of the system. The advantages of such an approach could be threefold. Firstly, it enables the application of dynamic analysis techniques leading to more accurate program comprehension results. Secondly, it reduces the scope of program comprehension to target only those code regions that are implemented by a particular user scenario, incurring lesser performance overhead. Thirdly, scenarios represent the users of the system, making the code regions covered by them the most important for program comprehension.

Calling context profiling is a dynamic analysis technique that is used to capture the dynamic inter-procedural control flow of software applications [5]. CCTs have been used for program understanding (or comprehension), runtime optimization, and performance analysis in past [17, 24]. Due to their structural characteristics, CCTs are capable of providing the complete runtime information about a program's scenario-level behaviour. Hence, they are also expected to assist in comprehending the software systems at scenario-level.

This work aims to perform an empirical investigation on scenario-level software evolution using static and dynamic data collected from stable released versions of four open source java systems. The empirical investigation is divided into two parts: i) design and implementation of a scenario level metric collection process for aggregating CCT and dynamic metrics; ii) correlation analysis among a set of static, dynamic and CCT metrics. The overall contribution is a CCT based dynamic analysis approach that helps comprehend how a particular user scenario evolves across multiple software versions.

II. RELATED WORK

Belady and Lehman [4, 13] postulated the laws of program dynamics. They identified a set of programming process parameters, and discussed statistical models of programming process regarding system's life cycle. Based on these studies, Lehman *et al*. [14] proposed the laws of software evolution. Kemerer and Slaughter [11] conducted longitudinal empirical research on software evolution to focus on how software development cost and effort change over time. They then empirically evaluated the existing Lehman Laws of software evolution.

Mens *et al*. [16] provided an overview of the approaches that use metrics to analyze, understand, predict and control software evolution. Lee *et al*. [12] presented a software metric based analysis of open source software evolution. They concluded that size, cohesion and coupling metrics can be used to assess the software quality and understand the evolution behavior of software systems. Briand *et al*. [6] performed a comprehensive empirical investigation on a number of object-oriented (OO) design measures to explore the relationship between existing OO design measures and software quality.

Salah *et al.* [19] proposed a program comprehension approach using dynamic analysis to extract views of a software system at three different levels – use case views, module interaction views and class interaction views. Xie *et al*. [23] conducted an empirical study on the evolution of seven open source systems to investigate the Lehman's laws of software evolution. Cornelissen *et al*. [8] conducted a systematic survey to classify and structure the research on program comprehension through dynamic analysis.

To the best of our knowledge, CCTs have not been previously employed to comprehend the scenario-level evolution of software systems.

III. CALLING CONTEXT TREE METRICS

A. Calling Context Tree (CCT)

The Calling Context Tree (CCT) [1] is a data structure that represents all distinct calling contexts of a unique program execution. Each node in a CCT corresponds to a method call (or simply method signature), and a path from any node to the tree's root represents that method's calling context. The parent of a CCT node corresponds to the caller's context, while the child nodes represent the callee methods. CCT maintains dynamic metrics, such as method invocations, CPU time etc. for each calling context, and provides the complete information related to dynamic program behavior. Following is a simple C++ example code, whose CCT representation is shown in Fig. 1.

B. CCT Metrics

Various CCT structural metrics considered in this study are defined as follows.

Height: The height of a CCT is the height of the method representing the root node of the CCT. The height of a method is the maximum height of any sub-tree within the CCT with an instance of the method as its root [15]. Let M be the set of methods, representing related nodes in CCT.

Height $H(m)$ of any method $m \in M$ is:

$$H(m) = \max_{c \in child(m)} 1 + h(c)$$

Average Height(AH): The average height of a CCT is defined as the average taken over heights of the child nodes of the root node. Average height H is computed as:

$$AH = \frac{\sum_{i=1}^{n} h(i)}{out\ (r)}$$

Here r is the root node, n is the number of children of root node and *out* denotes the outdegree.

Example 1

```
1.  void main() {
2.      a();
3.      b();
4.      b();
5.  }
6.  void a() {
7.      b();
8.      x();
9.  }
10. void x() {.....}
11. void b() {
12.     y();
13.     y();
14.     z();
15. }
16. void y() {.....}
17. void z() {.....}
```

Number of nodes (NON): The NON is the count of the total number of method call nodes present in a given CCT representing a scenario-level execution.

Number of leaf nodes (NLN): The number of leaf nodes is the count of all CCT method nodes having no children i.e. a leaf method is one which doesn't call any other method.

For example the CCT shown in Fig. 1, the height of CCT is the height of the method main, i.e. 3, and AH is 1.5. The value of NON is 9, and NLN is 5.

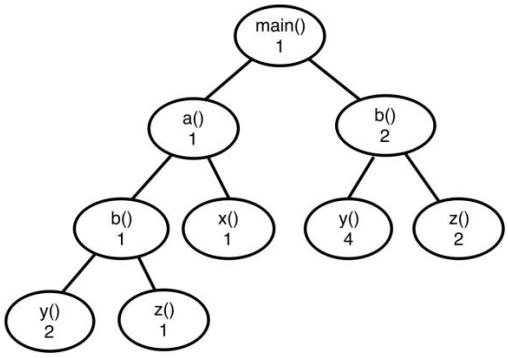

Figure 1. CCT representation

IV. EXPERIMENTAL STUDY

This section explains research questions, sample applications under study, selected metrics, tools used, data analysis techniques and the methodology adopted for this study.

A. Research Questions

In our study, we addressed the following two research questions (RQs):

RQ1: Can CCT metrics be helpful in understanding scenario-level software evolution? This research question investigates how CCT metrics are helpful in comprehending software evolution at scenario-level. We aim to investigate whether CCT structural metrics can provide useful information that may assist the program comprehension of software evolution at scenario-level.

RQ2: How do CCT metrics relate to static and dynamic design metrics? This research question aims to analyze the relationship among selected set of CCT, static and dynamic metrics across the evolution of multiple versions of sample software applications. It helps in understanding whether CCT metrics are strongly or loosely correlated with the other stable software metrics.

B. Sample Applications

The sample applications of varying sizes, in terms of both Lines of Code (LOC) and number of classes, are selected. We selected four open source GUI based applications from similar domains, i.e. drawing and image rendering namely DrawSWF[1] (Versions 1.2.5- 1.2.9), JHotDraw[2] (Versions 7.1-7.6) and Art of Illusion[3] (Versions 2.9.1-3.0.2) and Sunflow[4] (Versions 6.1-7.2). Two simple user scenarios of each application are considered in order to perform the empirical analysis of scenario-level software evolution. For applications DrawSWF, JHotDraw and Art of Illusion, Scenario 1 involves creating a figure in a file and saving that file on disk; whereas Scenario 2 comprises opening an existing file, editing the file and saving it on disk. For application Sunflow, Scenario 1 is opening a scene file, building it and then applying rendering; whereas Scenario 2 comprises performing interactive photorealistic rendering (IPR) of a scene file.

C. Selected Static and Dynamic Metrics

For each version of the selected sample applications, we computed a set of static and dynamic metrics described in Table I. It provides the traditional descriptions of these metrics. However, we have implemented the following improvisations to these metrics in order to suite the context of our research problem: 1) For each scenario (across all versions), we first identify the participating classes, and then calculate all static and dynamic metrics for each of these participating classes (i.e. class-level); 2) We then sum (or average – in case of EC, IC and CBO) these class-level metric values to get the scenario-scoped system-level metric values for each version of selected sample applications.

Dynamic EC and IC metrics are calculated for each participating class using our custom java DOM parser [25]. Java DOM parser is implemented to parse the CCT XML files generated by JP2. We derive our dynamic metric collection approach from [21]. Following a similar dynamic metrics collection approach, Sarvari et al. [21] followed a cloud based approach to calculate a set of dynamic metrics. The major focus of their study was to evaluate the performance of the proposed dynamic metric collection approach in terms of speed-up and scale-up over cloud. However, in this work, there is no such emphasis on the time or space overhead of metric collection. Our java project source code for the calculation of EC and IC metrics is publicly available here [25].

D. Tools Used

JP2 [20] is a java based open source calling context profiler, which collects accurate and complete program execution profiles. We used JP2 to generate CCT profiles. JP2 collects various instruction-level static and dynamic metrics, and associates metric values to the corresponding CCT node. JP2 generates output in both textual and XML formats. We utilize the XML output files to compute metrics of our interest. FindBugs[5] and EclEmma[6] Eclipse plug-ins are used to measure NOB and NPC metrics. CodePro AnalytiX[7] is used to compute LOC and CC metrics. STAN[8] is employed to calculate CBO metric.

TABLE I. SELECTED SET OF DYNAMIC AND STATIC METRICS

Metric Name	Description
Dynamic Metrics	
Import Coupling (IC)	Counts the number of distinct server classes used by all methods of all objects of a class [2].
Export Coupling (EC)	Counts the number of distinct client classes used by all methods of all objects of a class [2].
Number of Participating Classes (NPC)	Counts the number of classes that participate in a given scenario.
Static Metrics	
Lines Of Code (LOC)	Counts the number of lines that contain characters other than white space and comments.
Cyclomatic Complexity (CC)	Measure of the number of distinct paths of execution within the method [9]. We have calculated the sum of the cyclomatic complexity of each of the methods defined in the target elements.
Coupling Between Object (CBO)	Count of the number of classes to which it is coupled [7].
Number Of Bugs (NOB)	This measure counts the bug patterns in a program.

E. Methodology

Fig. 2 shows the overview of the experimental methodology followed in this work. Firstly, an executable jar file is created for each sample application. Then, this application jar file is loaded to JP2 profiler. JP2 executes each application and dumps the collected CCT profile to disk in XML format. An XML file storing the relevant CCT is generated for each of the selected scenarios (for all applications). The CCT XML files generated in this study are publicly available [25]. Next, a java DOM parser [25] is used to parse and examine the structure and contents of the XML files. The java DOM parser is used to calculate the CCT metrics as well as the selected dynamic metrics at scenario-scoped system-level. For example, NON metric was calculated by counting the number of nodes with a node type of ELEMENT_NODE in any given XML file. Also, the selected static metrics for the participating classes of each application are calculated using appropriate software tools as mentioned in Section IV.D. Next, the computed set of metric values is studied for descriptive statistical analysis. The evolution of CCT metrics is studied across various versions of each sample application for both the scenarios. Finally, in order to determine the relationship among the selected set of metrics various, correlation based statistical techniques are applied. Correlation analysis involving

[1] https://sourceforge.net/projects/drawswf/files/
[2] https://sourceforge.net/projects/jhotdraw/files/
[3] https://sourceforge.net/projects/aoi/files/
[4] https://sourceforge.net/projects/sunflow/files/

[5] http://findbugs.cs.umd.edu/eclipse/
[6] http://www.eclemma.org/
[7] https://marketplace.eclipse.org/content/codepro-analytix
[8] http://stan4j.com/

Pearson's correlation coefficient is computed to investigate how strongly the metrics are related, whereas Principal Component Analysis (PCA) is calculated to analyze the covariance structure of the collected metric data.

V. Results and Analysis

This section presents the evolution of CCT metrics, descriptive statistics, correlation analysis, PCA, and answer to the research questions.

A. Evolution of CCT Metrics

We discovered a number of trends in the values [25] of all four CCT metrics under study along the scenario-level evolution of the sample java applications, as presented in Fig. 3. Here, the subsequent versions of each selected sample application are denoted using numbers 1, 2, 3 and so on. It can be observed that the evolution of NLN is very similar to the evolution of NON metric across both the scenarios of all four applications, indicating toward a high correlation between these two metrics. Also, it can be observed that where values of AH metric do vary across software versions, the values of the height metric do not show any substantial variations across multiple versions of sample applications. We hence do not consider the height metric for further statistical analysis in this study.

B. Descriptive Statistics

We collected the descriptive statistics related to the selected set of metrics for both user scenarios of all four sample applications. The descriptive statistics comprise mean, standard deviation (SD), minimum value (Min), median, and maximum (Max) values for each measure. As already briefed in Section IV.C, the metric values [25] are calculated at the scenario-scoped system-level for each software version. Table II shows the descriptive statistics (for all the selected metrics) collected over metric values across multiple versions of JHotDraw for a given scenario. The descriptive statistics related to the other three applications are available at [25]. Analyzing and presenting the distribution of measures is useful for explaining the results of subsequent analysis.

C. Correlation Analysis

Tables III shows the correlation matrix for the selected set of metrics (for both scenarios) across multiple versions of JHotDraw. Correlation analysis data for the other three metrics are available at [25]. Overall results of correlation analysis can be summarized as follows. NON and NLN metrics are strongly correlated with each other. NON and NLN metrics have a high correlation with IC, EC, NOB and CBO metrics. NPC metric has a high correlation with LOC and CC metrics. AH metric has no definite correlation with any of the other metrics across all four sample applications.

D. Principal Component Analysis

The number of principal components is decided based on the amount of variance explained by each component.

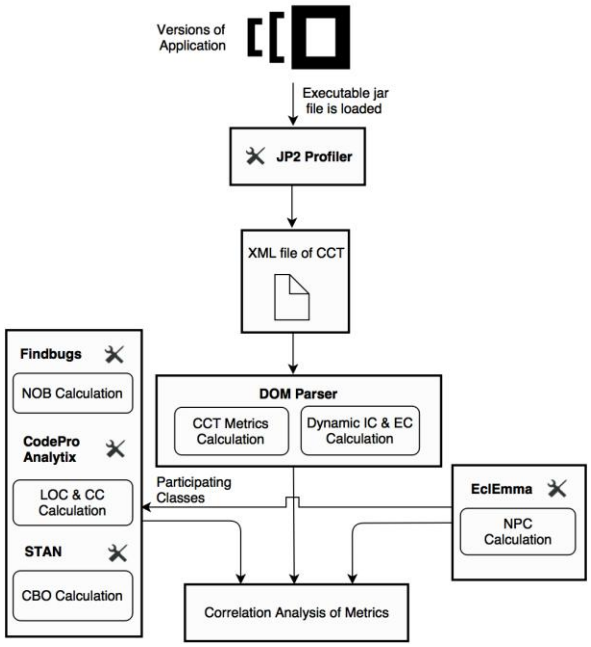

Figure 2. Overview of methodology.

A typical threshold for PCA is considering principal components with variance greater than 1.0. Using this criterion for each sample application, two principal components are considered. The values above 0.5 are considered to interpret principal components. Table IV shows the PC values for JHotDraw for Scenarios 1 and 2. PC values for other three metrics are available at [25].

A significant amount of variance is captured by the NON and NLN metrics. Such a strong variance is however not accounted for AH metric, as it is weakly or moderately correlated with the rest of the metrics. By analysing the metrics having high values of PC1 and PC2 for each application, the PCA results can be summarized as follows. PC1 measures NON, NLN, IC, EC and CBO metrics. This shows that all these measures are related to each other. As NON and NLN belong to the same principal component (PC1), and correlation matrix also shows that these two metrics are strongly correlated, it may be sufficient to employ only one of them. PC2 measures AH metric. It indicates that the average height of CCT captures additional information about the scenario-level evolution of a software system.

E. Answer to Research Questions

This subsection discusses the answers to the research questions, presented in Section IV, with the help of the experimental results analyzed in the previous subsection.

Answer to RQ1: Yes, CCT metrics are helpful in providing both replicative as well as useful additional scenario-level evolution information. It can be observed from Fig. 3 that the height of CCT remains constant across multiple versions of sample applications (per scenario). This trend can be attributed to the fact that increasing or amending the number of participating classes during runtime would very rarely change the call stack depth for a particular scenario across multiple

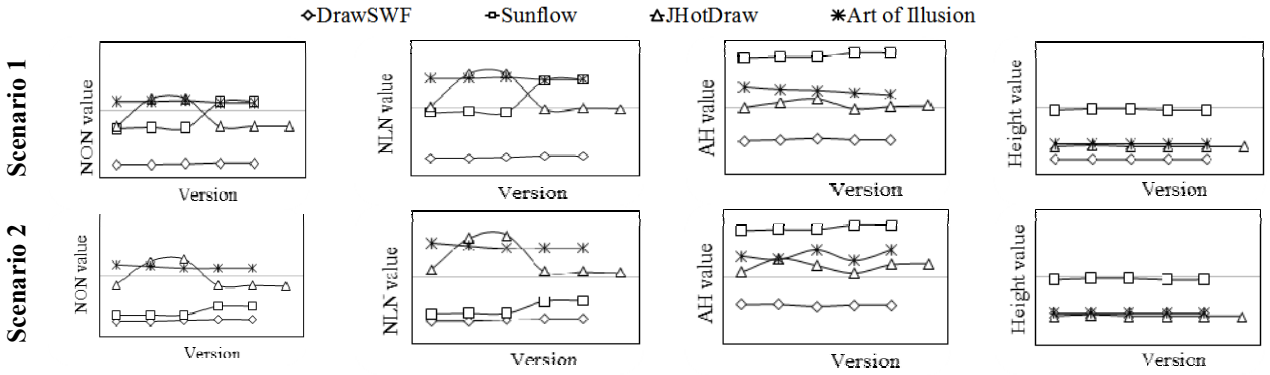

Figure 3. Evolution of NON, NLN, AH and Height metrics across multiple versions of sample applications

TABLE II. DESCRIPTIVE STATISTICS FOR JHOTDRAW

Variable	Mean	SD	Min	Median	Max
Scenario 1					
NON	44956.2	10320.9	38040	38428.0	58421
NLN	23343.0	5306.9	19724	20066.0	30272
AH	10.3862	0.5319	9.833	10.2130	11.250
Avg IC	3.190	0.183	2.96	3.155	3.49
Avg EC	2.968	0.160	2.75	2.970	3.22
LOC	19148.8	2180.0	15339	19787.5	20902
CC	3810.2	315.7	3221	3932.0	4071
NOB	28.0	7.1	20	26.5	37
NPC	115.2	8.4	100	117.0	125
Avg CBO	5.173	0.514	4.56	5.090	6.11
Scenario 2					
NON	48970.3	10473.3	41548	42603.0	63368
NLN	22103.7	11010.8	2178	22182.0	32776
AH	11.598	0.826	10.58	11.775	12.77
Avg IC	3.057	0.046	3.00	3.045	3.13
Avg EC	2.910	0.056	2.83	2.905	2.99
LOC	21773.8	2180.0	17964	22412.5	23527
CC	4323.2	315.7	3734	4445.0	4584
NOB	33.0	7.1	25	31.5	42
NPC	129.2	8.4	114	131.0	139
Avg CBO	5.247	0.539	4.74	4.995	6.02

TABLE III. CORRELATION MATRIX FOR JHOTDRAW

Coefficient	NON	NLN	AH	Avg IC	Avg EC	LOC	CC	NOB	NPC	Avg CBO
Scenario 1										
NON	-	1.000	0.925	0.857	0.791	-0.217	-0.056	0.929	-0.072	0.758
NLN	1.000	-	0.924	0.863	0.798	-0.239	-0.078	0.936	-0.093	0.763
AH	0.925	0.924	-	0.938	0.879	-0.108	0.027	0.885	-0.057	0.906
Avg IC	0.857	0.863	0.938	-	0.988	-0.378	-0.255	0.941	-0.337	0.975
Avg EC	0.791	0.798	0.879	0.988	-	-0.435	-0.327	0.925	-0.414	0.967
LOC	-0.217	-0.239	-0.108	-0.378	-0.435	-	0.976	-0.481	0.937	-0.300
CC	-0.056	-0.078	0.027	-0.255	-0.327	0.976	-	-0.351	0.979	-0.180
NOB	0.929	0.936	0.885	0.941	0.925	-0.481	-0.351	-	-0.390	0.845
NPC	-0.072	-0.093	-0.057	-0.337	-0.414	0.937	0.979	-0.390	-	-0.276
Avg CBO	0.758	0.763	0.906	0.975	0.967	-0.300	-0.180	0.845	-0.276	-
Scenario 2										
NON	-	0.721	0.556	0.919	0.859	-0.238	-0.071	0.935	-0.085	0.987
NLN	0.721	-	0.163	0.657	0.614	-0.407	-0.260	0.610	-0.213	0.713
AH	0.556	0.163	-	0.220	0.127	0.200	0.195	0.527	0.087	0.441
Avg IC	0.919	0.657	0.220	-	0.983	-0.383	-0.192	0.896	-0.173	0.967
Avg EC	0.859	0.614	0.127	0.983	-	-0.517	-0.335	0.884	-0.300	0.930
LOC	-0.238	-0.407	0.200	-0.383	-0.517	-	0.976	-0.481	0.937	-0.346
CC	-0.071	-0.260	0.195	-0.192	-0.335	0.976	-	-0.351	0.979	-0.171
NOB	0.935	0.610	0.527	0.896	0.884	-0.481	-0.351	-	-0.390	0.954
NPC	-0.085	-0.213	0.087	-0.173	-0.300	0.937	0.979	-0.390	-	-0.172
Avg CBO	0.987	0.713	0.441	0.967	0.930	-0.346	-0.171	0.954	-0.172	-

versions of sample applications. It can also be observed from Tables III-IV (and from [25]) that AH metric has weak correlations with the rest of the metrics, and is measured by a different principal component (PC2) as well. Hence, AH metric can capture additional comprehension information about scenario-level software evolution.

Answer to RQ2: There are relationships among the selected set of metrics (i.e. static, dynamic and CCT metrics). It can be observed from Table IV that CCT metrics NON and NLN are highly correlated with IC, EC, NOB and CBO metrics. As already mentioned, AH is not correlated with any of the metrics. Dynamic metric NPC has a high correlation with LOC and CC metrics.

VI. THREATS TO VALIDITY

There are a number of threats to the validity of this work. Firstly, we only sampled the metrics for the participating classes for each software system. We are aware that these classes may not represent the whole software system; however this work objectively targets to utilize execution scenarios exhibiting user interaction rather than focusing on the full code coverage. We obtained results based on the data collected from five to six versions of a set of four medium-sized software systems. If the data can be collected over a period of several years with tens of versions available for each application, it could affect the generalization of our findings.

Moreover, we limited this study to only four Java open source systems and that too from a similar domain, as we wanted the two scenarios to be as closely defined and executed as possible. The overall work of evaluating 21 software versions, with each version to be executed for two scenarios, the metric data to be handled was quite overwhelming as indicated by the NON values of CCTs, which are in tens of thousands on average.

TABLE IV. PCA STATISTICS FOR JHOTDRAW

	JHotDraw			
	Scenario 1		*Scenario 2*	
PC ► Metric ▼	***PC1***	***PC2***	***PC1***	***PC2***
NON	0.897	-0.311	0.913	-0.404
NLN	0.905	-0.290	0.746	-0.070
AH	0.012	-0.882	0.335	-0.524
Avg IC	0.981	-0.094	0.927	-0.222
Avg EC	0.965	-0.004	0.934	-0.063
LOC	0.489	-0.851	-0.609	-0.790
NOB	-0.361	-0.930	-0.458	-0.877
CC	0.983	-0.013	0.964	-0.127
NPC	-0.417	-0.895	-0.454	-0.844
Avg CBO	0.919	-0.138	0.953	-0.296

VII. CONCLUSION AND FUTURE WORK

This paper presents an empirical study conducted to assist program comprehension through investigating the scenario-centric features of software evolution at runtime. The experiments were performed on four open source java projects namely, DrawSWF, JHotDraw, Sunflow and Art of Illusion. A selected set of scenario-level metrics comprising CCT, static, and dynamic metrics was extracted to understand the interrelationships among various metrics, and to find useful trends across the evolution of CCT metrics along the evolution of multiple releases of subject applications at scenario-level. Empirical results obtained from correlation analysis and PCA show that CCT metrics are related to selected static and dynamic metrics. CCT metrics NON and NLN have high correlation with two dynamic metrics (IC, EC) and two static metric (NOB, CBO). AH CCT metric can capture some extra comprehension information related to scenario-level software evolution. Also, the height of CCT remains constant for the user scenarios concerning each sample application. The results show that CCT metrics provide some additional insights into the scenario-level software evolution.

In future, we aim to extend the study by exploring other CCT metrics such as calling context size, CCT balance, etc. Also, it would be interesting to analyze the relationship between CCT metrics and other object oriented external quality metrics such as maintainability, testability, etc., and how different measures can provide better program comprehension support to software developers. Finally, we plan to evaluate the CCT metrics used in this study on industrial systems so that more generalized and practically applicable results may be obtained.

REFERENCES

[1] G. Ammons, T. Ball, and J.R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," ACM Sigplan Notices, 32(5), pp. 85-96, 1997.

[2] E. Arisholm, L.C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," IEEE Transactions on Software Engineering, 30(8), pp. 491-506, 2004.

[3] T. Ball, "The concept of dynamic analysis," In Software Engineering—ESEC/FSE'99, Springer Berlin Heidelberg, pp. 216-234, 1999.

[4] L.A. Belady, and M.M. Lehman, "A model of large program development," IBM Systems journal, 15(3), pp. 225-252, 1976.

[5] W. Binder, D. Ansaloni , A. Villazón, and P. Moret, "Parallelizing calling context profiling in virtual machines on multicores," In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, ACM, pp. 111-120, 2009.

[6] L.C. Briand, J. Wüst, J.W. Daly, and D.V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," J. Sys. & Soft., 51(3), pp. 245-273, 2000.

[7] S.R. Chidamber, and C.F. Kemerer, "A metrics suite for object oriented design. IEEE Trans. on softw. engg.," 20(6), pp. 476-493, 1994.

[8] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," IEEE Trans. on Softw. Engg., 35(5), pp. 684-702, 2009.

[9] G.K. Gill, and C.F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," IEEE transactions on software engineering, 17(12), pp. 1284-1288, 1991.

[10] M. Hertzum, "Making use of scenarios: a field study of conceptual design," International Journal of Human-Computer Studies, 58(2), pp. 215-239, 2003.

[11] C.F. Kemerer, and S. Slaughter, "An empirical approach to studying software evolution," IEEE Transactions on Software Engineering, 25(4), pp. 493-509, 1999.

[12] Y. Lee, J. Yang, and K.H. Chang, "Metrics and evolution in open source software," In Seventh International Conference on Quality Software (QSIC), pp. 191-197, 2007.

[13] M.M. Lehman, and L.A. Belady, "Program evolution: processes of software change, Academic Press Professional, Inc., 1985.

[14] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M Turski, "Metrics and laws of software evolution-the nineties view," In Software Metrics Symposium. Proceedings Fourth International, pp. 20-32, 1997.

[15] D. Maplesden, E. Tempero, J. Hosking, and J.C. Grundy, "Subsuming methods: Finding new optimisation opportunities in object-oriented software," In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, pp. 175-186, 2015.

[16] T. Mens, and S. Demeyer, "Future trends in software evolution metrics," In Proceedings of the 4th international workshop on Principles of software evolution, pp. 83-86, 2001.

[17] P. Moret, W. Binder, and A. Villazon, "CCCP: Complete calling context profiling in virtual execution environments," In Proceedings of the ACM SIGPLAN workshop on Partial evaluation and program manipulation, pp. 151-160, 2009.

[18] G. Penny, and T. Armstrong, "Software Maintenance: Concepts and Practice," SE, ISBN, pp. 978-981, 2003.

[19] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta, "Scenario-Driven Dynamic Analysis for Comprehending Large Software Systems," In CSMR (Vol. 6), pp. 71-80, 2006.

[20] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini, "JP2: Call-site aware calling context profiling for the Java Virtual Machine," Science of Computer Programming, 79, pp. 146-157, 2014.

[21] S. Sarvari, P. Singh, and G. Sikka, "Efficient and Scalable Collection of Dynamic Metrics using MapReduce," In Asia-Pacific Software Engineering Conference (APSEC), pp. 127-134, 2015.

[22] A. Von Mayrhauser, and A.M. Vans, "Program comprehension during software maintenance and evolution," Computer, 28(8), pp. 44-55, 1995.

[23] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," In ICSM (Vol. 9), pp. 51-60, 2009.

[24] X. Zhuang, M.J. Serrano, H.W. Cain, and J.D. Choi, "Accurate, efficient, and adaptive calling context profiling," In ACM Sigplan Notices (Vol. 41, No. 6), pp. 263-271, 2006.

[25] Additional Information: 2016. http://www.pvsingh.com/s_gupta. Accessed: 2016-09-19.