Call Graph Evolution Analytics over a Version Series of an Evolving Software System

Animesh Chaturvedi

Indian Institute of Information Technology (IIIT), Dharwad

Dharwad, Karnataka, India

animesh.chaturvedi88@gmail.com

ABSTRACT

Software evolution analytics can be supported by generating and comparing call graph evolution information over versions of a software system. Call Graph evolution analytics can assist a software engineer when maintaining or evolving a software system. This paper proposes *Call Graph Evolution Analytics* to extract information from a set of *Evolving Call Graphs* ECG = $\{CG_1, CG_2, ...CG_N\}$ representing a Version Series VS = $\{V_1, V_2, ...V_N\}$ of an evolving software system. This is done using *Call Graph Evolution Rules* (CGERs) and *Call Graph Evolution Subgraphs* (CGESs). Similar to association rule mining, the CGERs are used to capture co-occurrences of dependencies in the system. Like subgraph patterns in a call graph, the CGESs are used to capture evolution of dependency patterns in evolving call graphs. Call graph analytics on the evolution in these patterns can identify potentially affected dependencies (or procedure calls) that need attention. The experiments are done on the evolving call graphs of 10 large evolving systems to support dependency evolution management. This is demonstrated with detailed results for evolving call graphs of Maven-Core's version series.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Information systems → Data mining**.

KEYWORDS

Software evolution, Graphs, Software repository, Data Mining

1 INTRODUCTION

Four phases in a software project includes: requirements gathering, development, testing, and maintenance/evolution [1]. In the maintenance phase, a software repository holds plenty of information about the dependency evolution in call graphs. Software

repositories are of various kinds; this study focuses on dependency repositories (e.g. Maven and GitHub), which contain jars, APIs, and libraries. Source code or executables are retrieved from a repository, and used in a software project. Growth of software - in the numbers of repositories and in the size of repositories - makes mining of a dependency repository a challenge. Existing techniques for mining software repositories [2] and search based software engineering [3] commonly analyse a single software version.

A call graph [4][5] represents procedure calls (or dependency) relationships in the form of a directed graph, where nodes represent procedures (method or function) and directed edges represent dependencies from caller to callee procedures. Earlier call graph analysis techniques have proven to be helpful in representing software component interactions [6], program comprehension of large software systems [7], impact analysis based on mutation testing [8], auto-completion of the procedure calls [9], and anomaly detection for monitoring system calls [10]. Graph or network theory includes e.g., temporal networks [11], evolving networks [12], dynamic networks [13], and multilayer networks [14]. The evolution of procedure calls can be exploited using techniques such as change mining [15] and evolution mining [16][17].

This paper proposes two forms of Call Graph Evolution mining techniques to uncover interesting and useful information: Call Graph Evolution Rules (CGERs) and Call Graph Evolution Subgraphs (CGESs). For software evolution analytics, apply graph evolution mining on multiple call graphs representing multiple versions of a software repository. Use call graph evolution that provides information about the software evolution. Call graph evolution information enables tools to support and manage the evolution of dependencies. Section II presents an abstract of the two proposed techniques. Section III demonstrates results (Stable CGERs and CGESs), and also discusses a **Research Question** "How does the results correspond to the Lehman's law of software evolution?".

2 CALL GRAPH EVOLUTION ANALYTICS

In an evolving software, each version (say V_i) of the software has its call graph, which can be built by capturing the procedure call relationship between the dependencies of version V_i. This preprocessing involves various steps (e.g., parsing) that depend upon the evolving software. The number of procedures is constant for a version, however different versions have different numbers of procedures over time. The series of software versions can be represented as a series of call graphs, which is referred to as Evolving Call Graphs. Here, "Evolving" denotes call graphs belonging to multiple versions of an evolving software. A series of call graphs are Evolving Call Graphs ECG = $\{CG_1, CG_2, ...CG_N\}$ that represents a Version Series VS = $\{V_1, V_2, ...V_N\}$ of an evolving software.

On a fine-grained level, the rule mining retrieves co-occurrences of dependencies in a call graph of a version, and then counts stability of retrieved rules over the version series. On a coarse-grained level, the subgraph mining extracts frequently occuring structural patterns of dependencies in a call graph of a version, and then aggregates frequencies of retrieved patterns over the version series.

2.1 Stable Call Graph Evolution Rules (CGERs)

Initially, perform Call Graph Rule (CGR) mining for each version of an evolving software in two steps. First, given a version of the system, the procedures of the system are assumed to be divided up into modules, which is referred to as the "procedures-module membership information". Transform this information into a set of "call pairs", which form the core or the Call Graph Database CG_Db. Then, pre-process a version V_i to create a call graph database (CG_Db_i) as shown in Fig. 1. Second step uses this database with two rule mining thresholds, minimum support (minSup) and minimum confidence (minConf), to generate a collection of Call Graph Rules (CG_Rules_i) as shown in Fig. 1. The Call Graph Rule $X \rightarrow Y$ can be interpreted as "if the procedure(s) of set X appear in a call pair, then the procedure(s) of set Y are likely to appear in the module with support and confidence above minSup and minConf, respectively". In other words, the presence of the caller procedure(s) in X implies the presence of the callee procedure(s) in Y with sufficiently high frequency to surpass the two thresholds minSup and minConf. This formally defines the Call Graph Rule with its support and confidence, and when a rule is interesting in a version V_i.

Next, defining the Call Graph Evolution Rules (CGERs) mining over a version series. Out of the retrieved CGRs, select a collection of distinct CGRs as CGERs. Out of the CGERs, retrieve Stable CGERs. Then, there will be a subset relationship between CGRs, CGERs, and SCGERs as shown in Fig. 2. The CGR is a rule derived from a call graph of a software version. The CGER is a 'distinct' CGR from a collection of CGRs in multiple versions. Count of a reoccurring CGR in multiple versions depicts its stability over a version series. The CGERs accompany the count (as stability). Each CGER has its 'stability', which is the count of versions in which CGR appears interesting (i.e., its support and confidence is above minSup and minConf). Earlier, we defined minimum Stability (minStab) [34][35]. A CGER is defined as a Stable CGER in VS, if (a) the support and confidence are greater than minSup and minConf in a version V_i, and (b) the 'stability' ($\geq$) minStab (i.e. stability of CGERs is greater than the minStab). This means a CGER is stable in VS if it exceeds the three user-specified thresholds: minSup, minConf, and minStab. These CGERs can also be referred to as call graph prediction rules (temporal rules, or episode rules). This takes the general form "if a certain dependency(ies) occurs, then another dependency(ies) is(are) likely to occur in the module". The Stable CGERs Mining steps are shown in Fig. 3.
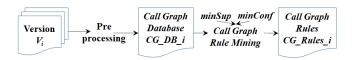

Figure 1: Call Graph Rule mining on version (V_i).
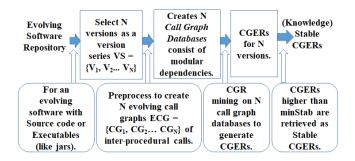

Figure 2: Venn diagram of CGRs, CGERs, and Stable CGERs.

Figure 3: Overview of the Stable CGER mining steps.

Figure 4: Call Subgraph Mining for Version V_i.

2.2 Call Graph Evolution Subgraphs (CGES)

The CGES mining retrieves the evolution of subgraphs in call graphs. Software version V_i is first pre-processed to build its call graph, which is used as the input for subgraph mining to retrieve subgraph patterns with their frequencies as shown in Fig. 4. Apply subgraph mining to each of the call graphs in a set of N call graphs. This iteratively retrieves the graphlets information for all versions of a software to extract the Call Graph Evolution Subgraphs and their frequencies. These subgraphs are of two types *Call Graph Evolution Graphlets* (CGEGs) and *Call Graph Evolution Motifs* (CGEMs).

Further, the Call Graph Evolution Graphlet information (subgraphs and their frequencies) are used to calculate a Call Graph Complexity (CG-Cx) of a single version and the overall Evolving Call Graph Complexity (ECG-Cx) for a version series. The Fig. 5 shows the steps to detect CGEGs, CGEMs, CG-Cx, and ECG-Cx by following the guidelines of knowledge discovery and data-mining (KDD). Our approach discovers hidden dependency evolution patterns, which is captured in CGEGs and CGEMs. The frequencies of graphlets are meaningful quantities that are used to calculate complexity for a call graph.
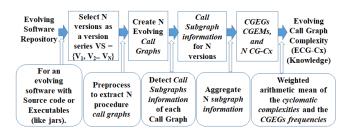

Figure 5: Steps of the CGES mining.

3 EVOLVING CALL GRAPHS ANALYTICS

This section presents study on 10 large evolving software given in the Table 1. The table provides the version series used to perform experiments, number of procedures in those versions, and average number of neighbours of each procedure. Firstly, for each one of them, make a series of evolving call graphs. Secondly, apply the call graph evolution analytics based on the two techniques: Stable CGERs mining and CGESs mining. These two techniques retrieved CGRs, CGERs, Stable CGERs, CGEGs, and CGEMs. The experimentation results of the two techniques are further used to study the persistence and complexity of dependencies in evolving call graphs. Out of ten evolving software systems, detailed demonstration of results for one evolving software i.e., Maven-Core is presented.

Table 1: Information about Evolving Call Graphs

Evolving software	Evolving Call Graphs		
	Version series	# proce-dures	Average # neighbours
Commons Codec	{1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10}	162	1.995
Guava	{12, 13, 14, 15, 16, 17, 18, 19}	1281	2.471
Hadoop HDFS	{2.2.0, 2.3.0, 2.4.0, 2.4.1, 2.5.0, 2.5.1, 2.5.2, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.7.0, 2.7.1, 2.7.2}	3129	2.166
HTTP Client	{4.3.1, 4.3.2, 4.3.3, 4.3.4, 4.3.5, 4.3.6, 4.3.0, 4.4.1, 4.4.0, 4.5.1, 4.5.2, 4.5.0}	276	2.020
JMeter Core	{1.8.1, 1.9.1, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13 }	806	2.632
Joda Time	{2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8.0, 2.8.1, 2.8.2, 2.9.0, 2.9.1, 2.9.2, 2.9.3}	418	2.886
JUnit	{4.1, 4.6, 4.7, 4.8, 4.9, 4.11, 4.12 }	171	1.628
Log4J	{1.1.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 1.2.11, 1.2.12, 1.2.13, 1.2.14, 1.2.15, 1.2.16, 1.2.17}	307	2.320
Maven Core	{3.1.0, 3.1.1, 3.2.1, 3.2.2, 3.2.3, 3.2.5, 3.3.1, 3.3.3, 3.3.9}	594	2.385
Storm Core	{0.9.1, 0.9.2, 0.9.3, 0.9.4, 0.9.5, 0.9.6, 0.10.0, 0.10.1}	373	1.686

\# Number of

- The Fig. 6 shows the comparison between the number of CGRs, CGERs, and Stable CGERs for Maven-Core. The figure shows nine experiments, in each experiment we identified that interesting CGERs and Stable CGRs are fewer than CGRs. The 16 Stable CGERs retrieved for the Maven-Core are given in the first column of Table 2. Some of the transitivities and lattices formed based on these
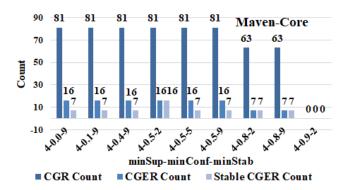

Figure 6: Compare counts of CGRs, CGERs, & Stable CGERs.

Stable CGERs are shown in the Fig. 7, which shows relationships between the various procedures in the Maven-Core.

- Various CGEGs are retrieved from the evolving call graphs of the Maven-core as shown in the Fig. 8, which shows the CGEGs with their frequencies over a version series. These frequencies of CGEGs along with their cyclomatic complexities are used to retrieve varying complexity of each call graph (shown with a varying time-series in Fig. 9) and an aggregated complexity of all evolving call graphs (shown with a constant time-series in Fig. 9). Out of many CGEGs, few statistically significant CGEMs patterns with their frequencies above a certain threshold are shown in the second column of Table 2.

Table 2: Experimental results on Maven-Core.

Stable CGERs	CGEMs
{getGroupId → getArtifactId} {getGroupId → getArtifactId, getId} {getArtifactId → getGroupId} {getArtifactId → getGroupId, getId} {getGroupId → getId} {getId → getGroupId} {getId → getGroupId, getArtifactId} {getKey → getGroupId} {getKey → getGroupId, getId} {getKey → getGroupId, getArtifactId} {getKey → getGroupId, getArtifactId, getId} {getArtifactId → getId} {getId → getArtifactId} {getKey → getArtifactId} {getKey → getArtifactId, getId} {getKey → getId}	$(M_{198}, 39.55)$ $(M_0, 34.25)$ $(M_{178}, 13.85)$

The two kinds of experiments to retrieve Stable CGERs and CGESs made us infer that changes in an evolving software lead to several versions, whose dependency patterns also evolve in the versioning process to represent a state of the evolution information. Then, aggregate the state information over a version series to achieve evolution information about a whole centralized repository. This evolution information is helpful to manage and track evolution happening in a version series of an evolving software.
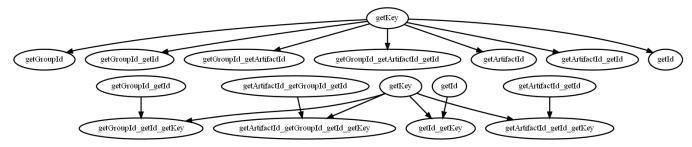
Figure 7: Transitivity and Lattice graphs respectively for given Stable CGERs in Table 2 for the Maven-Core (evolving software).
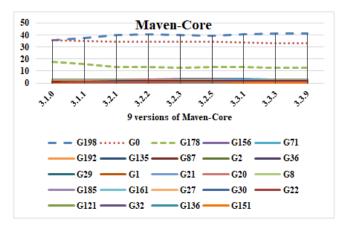

Figure 8: Frequencies of various CGEGs of evolving call graphs representing the version series of Maven-core.
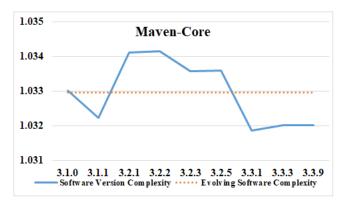

Figure 9: The varying complexities for each version and a constant aggregated complexity over multiple versions.

Answer to Research Question: For the call graph evolution, we found three inferences with Lehman's law of software evolution (in 1980) [18]. We found two similar inferences. An evolving software undergoes continuous upgrading to make better versions (in Table I). The stability of an evolving software remains almost constant with time (in Fig. 8). We found one dissimilar inference. The complexity of evolving software keeps on changing with time, but not necessarily increasing (in Fig. 9). This is because, now there are better software repository management techniques and systems (e.g. GitHub, Maven) are easily accessible as compared to the era when Lehman's law was introduced.

Like Vasa and Schneider [19] found patterns having low cyclomatic complexity occur with high percentage in a version. We also found recurring subgraph patterns with low complexity occur in high percentage over a version series. These subgraphs are the CGEMs $\{M_{198}, M_0, M_{178}\}$ with cyclomatic complexity = 1 in Table 2, which has high frequencies of CGEGs $\{G_{198}, G_0, G_{178}\}$ in Fig. 8.

4 RELATED WORKS AND DISCUSSIONS

On one hand, 1970-2010 was the era of control flow graph analysis [20] and dependency analysis [21], which are exploited to do the software analysis [22], program slicing [23][24], and source code analysis [25]. On the other hand, in 1993 association rule mining proposed by Agrawal et al. [26], which was used in the context of software engineering. Ying Annie et. al. [27] (in 2004) predicted a set of possible future dependencies across files using support metrics to mine history of software changes. Zimmerman et al. [28] (in 2005) applied association rule mining on software version histories to predict and suggest changes, co-changes, and prevent errors. Works like [27][28] made the field of Mining Software Repositories.

Vasa et al. [29] presented multiple releases of object-oriented classes and interfaces for stability and complexity. Pang et al. [30] proposed N-gram analysis and prediction of vulnerable components for features like sequences in source code files, and Java class files. Kikas et al. [31] discussed the structure and evolution of dependency graphs collected from the package. Honsel [32] retrieved interesting evolution patterns in the dependency graphs. Decan et al. [33] presented empirical analysis of Dependency Network Evolution on seven packages. In comparison to these works, this paper presented a novel way to use call graphs evolution mining for studying structural patterns, stability, and complexity.

5 CONCLUSIONS

This paper introduced two proposed techniques, which are used to analyze 10 evolving software systems (including Maven-Core described in detail). Looking forward to publishing a complete study of Stable CGERs mining and CGESs mining, which will provide intrinsic details of the approaches and demonstrate detailed study of 10 evolving software systems. Other related works by the author on System Evolution Analytics for Hadoop-HDFS repository [34]-[41] and Cloud Service Evolution Analytics [42]-[45].

ACKNOWLEDGMENTS

REFERENCES

[1] V. Rajlich. "Software evolution and maintenance". *Future of Software Engineering Proceedings*. 2014. 133-144.

[2] A. E. Hassan. "The road ahead for mining software repositories". *In Frontiers of Software Maintenance*, 2008. FoSM 2008. pp. 48-57. IEEE, 2008.

[3] M. Harman. "The current state and future of search based software engineering". *Future of Software Engineering*. IEEE Computer Society, 2007.

[4] B. G. Ryder. "Constructing the call graph of a program". *IEEE Transactions on Software Engineering* 3 (1979): 216-226.

[5] D. Grove, and C. Chambers. "A framework for call graph construction algorithms". *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.6 (2001): 685-746.

[6] N. L. Hashim, and N. Ismail. "Usage of Call Graph for Representing Software Component Interactions". *Knowledge Management Int. Conf. (KMICe)* 2012.

[7] S. C. Shaw, et al. Moralising the call graph as a means of program comprehension. *Tech. Rep., Department of Mathematical Sciences, University of Durham*, 2002.

[8] V. Musco, M. Monperrus, and P. Preux. "A large-scale study of call graph-based impact prediction using mutation testing". *Software Quality Journal* 25.3 (2017): 921-950.

[9] D. Garbervetsky, E. Zoppi, and B. Livshits. "Toward full elasticity in distributed static analysis: the case of callgraph analysis". *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.

[10] D. Gao, M. K. Reiter, and D. Song. "Gray-box extraction of execution graphs for anomaly detection". *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004.

[11] P. Holme, and J. Saramäki. "Temporal networks". *Physics reports* 519.3 (2012): 97-125.

[12] C. Aggarwal, and K. Subbian. "Evolutionary network analysis: A survey". *ACM Computing Surveys (CSUR)* 47.1 (2014): 10.

[13] J. E. Aronson. "A survey of dynamic network flows". *Annals of Operations Research* 20.1 (1989): 1-66.

[14] M. De Domenico, M. A. Porter, and A. Arenas. "MuxViz: a tool for multilayer analysis and visualization of networks". *Journal of Complex Networks* 3.2 (2015): 159-176.

[15] B. Mirko, F. Höppner, and M. Spiliopoulou. "On exploiting the power of time in data mining". *ACM SIGKDD Explorations Newsletter* 10.2 (2008): 3-11.

[16] T. Mansoureh, et al. "Community evolution mining in dynamic social networks". *Procedia-Social and Behavioral Sciences* 22 (2011): 49-58.

[17] W. Bin, et al. "Resume mining of communities in social network". Seventh *IEEE International Conference on Data Mining Workshops ICDMW*. IEEE, 2007.

[18] M. M. Lehman. "Programs, life cycles, and laws of software evolution". *Proceedings of the IEEE* 68.9 (1980): 1060-1076.

[19] R. Vasa, and J.G. Schneider. "Evolution of cyclomatic complexity in object oriented software". *Proceedings of 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'03)*, Darmstadt, Germany. 2003.

[20] F. E. Allen, "Control flow analysis". *ACM SIGPLAN Notices* 5.7 (1970): 1-19.

[21] A. Sharma, P. S. Grover, and R. Kumar. "Dependency analysis for component-based software systems". *ACM SIGSOFT Software Engineering Notes* 34.4 (2009): 1-6.

[22] D. Jackson, and M. Rinard. "Software analysis: A roadmap". *Proceedings of the Conference on the Future of Software Engineering*. 2000.

[23] M. Weiser. "Program slicing". *IEEE Transactions on Software Engineering* 4 (1984): 352-357.

[24] D. W. Binkley, and K. B. Gallagher. "Program slicing". *Advances in computers* 43 (1996): 1-50.

[25] D. Binkley. "Source code analysis: A road map". *Future of Software Engineering*. IEEE Computer Society, 2007.

[26] R. Agrawal, T. Imieliński, and A. Swami. "Mining association rules between sets of items in large databases". *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 1993.

[27] A. TT Ying, et al. "Predicting source code changes by mining change history". *IEEE Transactions on Software Engineering*, 30.9 (2004): 574-586.

[28] T. Zimmermann, et al. "Mining version histories to guide software changes". *IEEE Transactions on Software Engineering*, 31(6), (2005): 429-445.

[29] R. Vasa, J.-G. Schneider, and O. Nierstrasz. "The inevitable stability of software change". *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007.

[30] Y. Pang, X. Xue, and A. S. Namin. "Predicting vulnerable software components through n-gram analysis and statistical feature selection". *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015.

[31] R. Kikas, et al. "Structure and evolution of package dependency networks". *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017.

[32] V. Honsel, et al. "Mining software dependency networks for agent-based simulation of software evolution". *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 2015.

[33] A. Decan, T. Mens, and P. Grosjean. "An empirical comparison of dependency network evolution in seven software packaging ecosystems". *Empirical Software Engineering* 24.1 (2019): 381-416.

[34] A. Chaturvedi, A. Tiwari, and N. Spyratos. "minStab: Stable Network Evolution Rule Mining for System Changeability Analysis". *IEEE Transactions on Emerging Topics in Computational Intelligence* 5.2 (2019): 274-283.

[35] A. Chaturvedi, A. Tiwari, and N. Spyratos. "System Network Analytics: Evolution and Stable Rules of a State Series". *IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2022.

[36] A. Chaturvedi and A. Tiwari. "System Network Complexity: Network Evolution Subgraphs of System State Series". *IEEE Transactions on Emerging Topics in Computational Intelligence* 4.2 (2018): 130-139.

[37] A. Chaturvedi, A. Tiwari. "System Evolution Analytics: Evolution and Change Pattern Mining of Inter-Connected Entities". *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2018: 3877-3882.

[38] A. Chaturvedi and A. Tiwari. "System Evolution Analytics: Deep Evolution and Change Learning of Inter-Connected Entities". *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2018: 3075-3080.

[39] A. Chaturvedi and A. Tiwari. "SysEvoRecomd: Graph Evolution and Change Learning based System Evolution Recommender". *IEEE International Conference on Data Mining Workshop (ICDMW)*, 2018, 1499-1500.

[40] A. Chaturvedi, A. Tiwari, and S. Chaturvedi. "SysEvoRecomd: Network Reconstruction by Graph Evolution and Change Learning". *IEEE Systems Journal* 14.3 (2020): 4007-4014.

[41] A. Chaturvedi, et al. "System Neural Network: Evolution and Change based Structure Learning". *IEEE Transactions on Artificial Intelligence* 3.3 (2022): 426-435.

[42] A. Chaturvedi. "Subset WSDL to access Subset Service for Analysis". *In 2014 IEEE 6th International Conference on Cloud Computing Technology and Science* (pp. 688-691), 2014, IEEE.

[43] A. Chaturvedi. "Automated Web Service Change Management AWSCM-A tool". *In 2014 IEEE 6th International Conference on Cloud Computing Technology and Science* (pp. 715-718), 2014, IEEE.

[44] A. Chaturvedi and D. Binkley. "Web Service Slicing: Intra and Inter-operational Analysis to Test Changes". *IEEE Transactions on Services Computing* 14.3 (2018): 930-943.

[45] A. Chaturvedi, et al. "Service Evolution Analytics: Change and Evolution Mining of a Distributed system". *IEEE Transactions on Engineering Management* 68.1 (2020): 137-148.