Structural Complexity Analysis to Evaluate Technical Risk in Defense Acquisition

Pugliese, Antonio; Nilchiani, Roshanak; Vierlboeck, Maximilian

Monterey, California. Naval Postgraduate School

SYM-AM-21-071

EXCERPT FROM THE PROCEEDINGS OF THE EIGHTEENTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

Structural Complexity Analysis to Evaluate Technical Risk in Defense Acquisition

May 11–13, 2021

Published: May 10, 2021

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF DEFENSE MANAGEMENT
NAVAL POSTGRADUATE SCHOOL

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Defense Management at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF DEFENSE MANAGEMENT
NAVAL POSTGRADUATE SCHOOL

Structural Complexity Analysis to Evaluate Technical Risk in Defense Acquisition

Dr. Antonio Pugliese—Cornell University [pugliese@cornell.edu]

Dr. Roshanak Nilchiani—Stevens Institute of Technology [rnilchia@stevens.edu

Maximilian Vierlboeck—Stevens Institute of Technology [mvierlbo@stevens.edu]

Abstract

The study of "-ilities" in systems engineering has been fundamentally connected to the evaluation of system complexity in recent years. Complexity has been inherent in all defense acquisition programs where technology and human organizations interface. Complexity can be inherent in design of a defense system/System-of-systems, at the organizational layers of defense systems, and in the environment, every now and then imposing its unpredictability or non-linearity to an acquisition program. Increased knowledge and understanding of defense systems complexity can shed light on some various unknown and emergent behavior of such systems, as well as guiding us to better solution sets when facing major decisions or challenges.

The goal of our research is to identify, formulate, and model complexity in technical segment of defense acquisition programs, as the increased level of complexity contributes to increased fragility and potential failure of the system. In another word, complexity measure is an indirect measure of risk in complex systems. The future direction of our research aims at replacing a large portion of subject matter experts' opinions on technical systems risk assessment, with actual complex risk measures and therefore improve the decision-making process more objective.

Introduction

Defense acquisition programs are essential and fundamental to the goals of the United States in terms of defense and peace-keeping activities. The 2016 report on Performance of the Defense Acquisition System states that long-time issues such as large cost growth, heavy changes in requirements, and responsiveness in initiating new programs, which have been addressed in years of research in acquisition management, are now under control (Kendall, 2016). The same report warns future leaders to not neglect system "-ilities" when evaluating a system, claiming that well-engineered systems are more often effective. Reliability, availability, and maintainability are prerequisites to the system performing its function (Kendall, 2016).

The study of "-ilities" in systems engineering has been fundamentally connected to the evaluation of system complexity in recent years (Enos et al., 2019; Fischi et al., 2017; Pugliese et al., 2018; Pugliese & Nilchiani, 2017; Salado & Nilchiani, 2013). Complexity has been inherent to defense acquisition programs where technology and human organizations interface. Complexity can be inherent to design of a defense system/system-of-systems, at the organizational layers of defense systems, and in the environment, occasionally imposing its unpredictability or non-linearity to an acquisition program. System "-ilities" such as flexibility, reliability, modularity, etc. are most successful when they are embedded in large-scale programs where a fundamental understanding of the complex structure and behavior of such systems exists. Therefore, it is necessary and urgent to better understand, model, measure, and formulate such defense programs considering their complex behavior. Increased knowledge and understanding of defense systems complexity can shed light on various unknown and emergent behavior of such systems, as well as guide us to better solution sets when facing major decisions or challenges.

The goal of our research is to identify, formulate, and model complexity in technical segments of defense acquisition programs, as the heightened level of complexity contributes to

increased fragility and potential failure of the system. In other words, complexity measure is an indirect measure of risk in complex systems. The future direction of our research aims at replacing a large portion of subject matter experts' opinions on technical systems risk assessment with actual complex risk measures and therefore improve the decision-making process by enabling it to be more objective.

In software systems, complexity can be defined as "a measure of the resources expended by a system while interacting with a piece of software to perform a given task" (Basili, 1980). From this general definition many can be derived depending on the choice of the specific system interacting with the software under study (Mens, 2016). If the interacting system is a computer, we are looking at theoretical complexity, which can be of two types: algorithmic complexity, if the focus is on the time and storage space required to execute the computation, or computational complexity, if the focus is on the complexity of the problem at hand, regardless of the algorithm used to solve it. Efficient algorithms will have an algorithmic complexity that is close to the computational complexity of the problem at hand (Mens, 2016). If the interacting system is the user of the software system, then the corresponding complexity is complexity of use, usually referred to as a common system characteristic: usability (Mens, 2016). If the interacting system is a software developer, the type of complexity is structural complexity (Darcy et al., 2005).

Software structural complexity focuses on the software architecture, defined as the organization of the components of the software and how they relate to each other. A structural complexity analysis is performed by looking at the source code of the software under study, and is therefore dependent on the programming language and on a specific implementation of the solution. Depending on the level of granularity at which the software is analyzed, this static analysis, as it is also known among computer scientists, can consider as atomic units of the system the modules or files, inner constructs, such as classes and functions, or single instructions. A finer level of granularity can lead to a more detailed understanding of the dependencies, but requires the software to be completed before this analysis can be carried out.

Literature Review and State of the Research

When looking at software architecture (SA) in its general form and where the architectural aspects originated from, the history shows that the first approaches that are now all combined in SA can be traced back all the way to the early 1970s. Especially over the last 30 years, software architecture emerged as an important field for both research and practice (Shahin et al., 2014). On a general level, SA can be defined as the representation and definition of software and the software system. Such a representation includes descriptive elements which cover the relationships between elements and sub-elements (Angelov et al., 2009; Avci et al., 2020; Garlan & Shaw, 1993).

Early on in the 1960s and 1970s, research emerged that addressed data and data structures, which lead to an accentuation of certain structural elements above the level of the software code itself. This accentuation led to an abstraction and organizational understanding, and as a result, software architecture emerged in the following decades (Garlan & Shaw, 1993). The first appearances and mentions of SA can be found in the publication of Parnas in 1972. In this work, the author described the concept behind the module decomposition structure. Specifically, Parnas describes criteria that can be used to decompose the structure of systems into modules. Throughout the 1970s, Parnas published various other papers that outlined additional aspects of structures, and over time, the field of SA progressed and more nuances were added to differentiate between various forms of structures (Bass et al., 2012).

From the aforementioned time till around 1990, architecture in scientific fields was mostly related to systems (Kruchten et al., 2006). Yet, SA as a separate discipline in research and science emerged in the 1990s (Kruchten et al., 2006; Perry & Wolf, 2000) and has been flourishing

since then, also including empirical research approaches (Qureshi et al., 2013). The first book about SA was also published during these beginning times in 1994 (Witt et al., 1994).

Because of the pace increase, numerous approaches were developed in the 1990s in academia but also by companies, such as Lockheed Martin and IBM. Kruchten et al. (2006) lists various approaches that resulted from these efforts: Software Architecture Analysis Method (Kazman et al., 1994), the 4+1 view (Kruchten, 1995), Siemens's four views (Soni et al., 1995), and numerous other patterns that address the design of SA (Buschmann et al., 1996) as well as Architecture Description Languages (ADLs; Shaw & Clements, 2006).

Building upon the momentum, more companies started to participate in SA and its methodologies since the beginning of the third millennium. Two notable approaches for general architecture were standardized to unify certain efforts: RM-ODP (ISO/IEC, 1995; Linington, 1995; Putman, 2000) and IEEE 1471 (IEEE, 2000). Overall, a lot of pre-made platforms and architectures ready to use have been developed and are today available. Open-source software adds to this abundance. It is thus safe to say that SA has reached what Shaw and Clements (2006) describe as "popularization." Therefore, new trends and explorations also must be considered since they are a natural continuation of the described state.

Looking at the last 5 years, a few trends in SA emerge that are currently being pursued. The first of these trends is cloud and service related and addresses the question how SA is connected to such fields and how it can be utilized (Amal et al., 2018; Bahsoon et al., 2017; Hästbacka et al., 2019; Malavolta & Capilla, 2017). Second, a focus on intelligent architecture can be seen, which introduces topics such as machine learning into the field of SA and enables phenomena such as emergent architectures that only appear during runtime and are not pre-managed or set (Woods, 2016). This trend also increases the reliance of SA on data and algorithms, which will require rethinking of previously mentioned approaches, such as the 4+1 View, which did not originally include any views for data or underlying information (Kruchten, 1995; Woods, 2016). Third, also related to the previous one, the use of SA in agile environments has become more and more important and has thus moved into the focus of research as well (Dingsøyr et al., 2018; Venters et al., 2018). Agile and SA propose different viewpoints with the former advocating for flexible as well as iterative implementation of changes and the latter standing for fundamental decisions that might even be deferred until they can be made in the most informed manner if they are not already defined up front (Dingsøyr et al., 2018; Hasselbring, 2018). Hence, the integration of architecture into agile environments has been seen as a trend as well (Dingsøyr et al., 2018). Lastly, a focus on sustainability also in relation to longevity and scalability can be seen. Since scalability can be an issue with integrated databases due to their high coherence (Hasselbring, 2002), the applicability and longevity of SAs can become problematic if they are tightly vertically integrated. Thus, approaches such as Microservices (Francesco et al., 2017; Newman, 2015; Taibi et al., 2017) and other solutions to these problems (Capilla et al., 2017), which then also address sustainability (Cabot et al., 2019; Venters et al., 2018), are being pursued.

Lastly, for the research at hand, a categorization approach and characterization within SA is critical to allow for a methodological analysis. Thus, the most frequently used and applied structures were researched and are described hereinafter. On an overarching level, structures in SA can be seen as threefold (Bass et al., 2012): decomposition structure, use structure, and class structure. Each of these three categories can again be subdivided into more nuanced categories, but such detailed subdivisions can be strongly dependent on the case of application. Thus, for the work at hand, three of the subcategories of the module structure shall be outlined as they are directly related to the research presented as depicted in Figure 1: decomposition structure, use structure, and class structure.

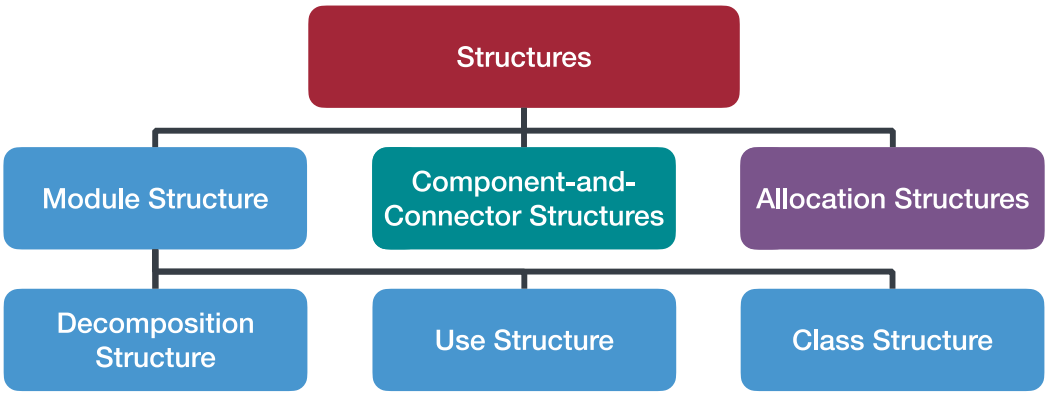

Figure 1. Classification of Relevant Architectural Structures for Software Systems

In this paper we analyze the source code of an open-source Python library, Snorkel. This is a static analysis that focuses on the module structure. In particular, the codebase is parsed to generate a class structure, which includes details about modules, classes, and methods. A series of relationships between these entities allow us to define a particular case of a use structure, which will be used as the basis of the static analysis.

Methodology

This paper presents a static analysis of the source code of a software package developed using the Python 3 programming language. The source code is parsed using the Abstract Syntax Tree (AST) module in the Python Standard Library. This module is based on the parser used in the native Python compiler and is continuously updated with any grammar change in the language. This parsing process leads to the creation of a graph where functions and classes are nodes and inheritance and functional calls are edges.

The resulting graph is known as a *module dependency graph* and has been a subject of a number of graph-theoretical research efforts (MacCormack et al., 2006). The module dependency graph is a particular case of a use structure. In this research, the module dependency graph will be analyzed with a series of complexity metrics based on the eigenvalues of various representations of the graph (Pugliese & Nilchiani, 2019). These metrics are based on other metrics, such as graph energy (Gutman, 2001) and natural connectivity (Jun et al., 2010).

The module dependency graph is built using an ad hoc model of Python objects and interdependencies. This version introduces function-level granularity, from file-level of the previous one, and is based on the Python AST module instead of simply parsing the code. The graph is built using the following rules:

- A file that imports code from another file is dependent on that file.
- A class that inherits from another class is dependent on that class.
- A function that calls another function is dependent on that function.
- A file that contains a class is dependent on that class.
- A file that contains a function is dependent on that function.
- A class that contains a function is dependent on that function.

Figure 2 shows the types of dependencies among the elements of the graph.

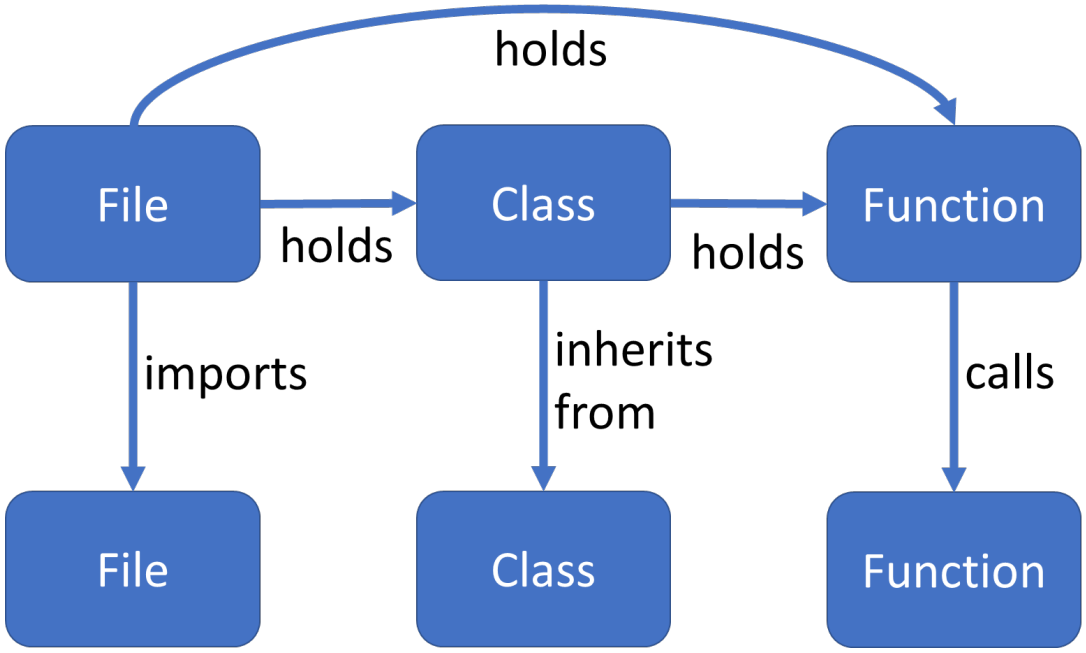

Figure 2. Types of Dependencies Among Graph Elements

The analysis of the module dependency graph is carried out using a set of spectral complexity metrics developed by our research group and represented using the following formula:

$$C(S) = f\left(\gamma \sum_{i=1}^{n} g\left(\lambda_i(M) - \frac{tr(M)}{n}\right)\right)$$

where $f_1(x) = x, g_1(y) = |y|, f_2(x) = \ln x\,, g_2(y) = e^y$ are the possible values for the functions f and g, the coefficient γ can be $\gamma_1 = 1, \gamma_2 = n^{-1}$, and the matrix representation of the graph can be either $M_1 = A, M_2 = L, M_3 = \mathcal{L}$, which have been defined in our previous publication (Nilchiani & Pugliese, 2016).

Table 1 shows the metrics that can be derived from this formula through combinations of the described parameters. Two sets of functions, two values for the coefficient γ, and three matrices yield 12 possible metrics. Throughout this paper, the metrics are referred to using acronyms: graph energy (GE), Laplacian graph energy (LGE), normalized Laplacian graph energy (NLGE), natural connectivity (NC), Laplacian natural connectivity (LNC), normalized Laplacian natural connectivity (NLNC). Where the acronym has a trailing n, such as in (GEn), the factor $\gamma = 1/n$.

Table 1. Twelve Examples of Spectral Structural Complexity Metrics

	Adjacency Matrix	Laplacian Matrix	Normalized Laplacian Matrix
$\boldsymbol{\gamma = 1}$	$GE = \sum_{i=1}^{n} \lvert\lambda_i\rvert$	$LGE = \sum_{i=1}^{n} \left\lvert\mu_i - \frac{2m}{n}\right\rvert$	$NLGE = \sum_{i=1}^{n} \lvert\nu_i - 1\rvert$
	$NC = \ln\left(\sum_{i=1}^{n} e^{\lambda_i}\right)$	$LNC = \ln\left(\sum_{i=1}^{n} e^{\mu_i - \frac{2m}{n}}\right)$	$NLNC = \ln\left(\sum_{i=1}^{n} e^{\nu_i - 1}\right)$
$\boldsymbol{\gamma = \frac{1}{n}}$	$GEn = \frac{1}{n}\sum_{i=1}^{n} \lvert\lambda_i\rvert$	$LGEn = \frac{1}{n}\sum_{i=1}^{n} \left\lvert\mu_i - \frac{2m}{n}\right\rvert$	$NLGEn = \frac{1}{n}\sum_{i=1}^{n} \lvert\nu_i - 1\rvert$
	$NCn = \ln\left(\frac{1}{n}\sum_{i=1}^{n} e^{\lambda_i}\right)$	$LNCn = \ln\left(\frac{1}{n}\sum_{i=1}^{n} e^{\mu_i - \frac{2m}{n}}\right)$	$NLGEn = \ln\left(\frac{1}{n}\sum_{i=1}^{n} e^{\nu_i - 1}\right)$

Results

This section presents the results of analysis on the module dependency graph for the Snorkel project published on GitHub. The project was selected due to its relatively small size of ~2,600 commits and less than 300MB of code as of March 2021, which allows us to run our analytical programs on a laptop. The number of contributors (50), the history of commits, and the prevalence of Python code were other attributes that affected this choice. Future and optimized versions of the code will aim at analyzing larger codebases.

The evolution of the graph at indicated time stamps is depicted in Figure 3. In these plots, the nodes are colored according to their type: file (blue), library (black), class (red), and function/method (green). These images suggest how even a relatively small project, such as Snorkel, can become eminently complex to manage and architect.

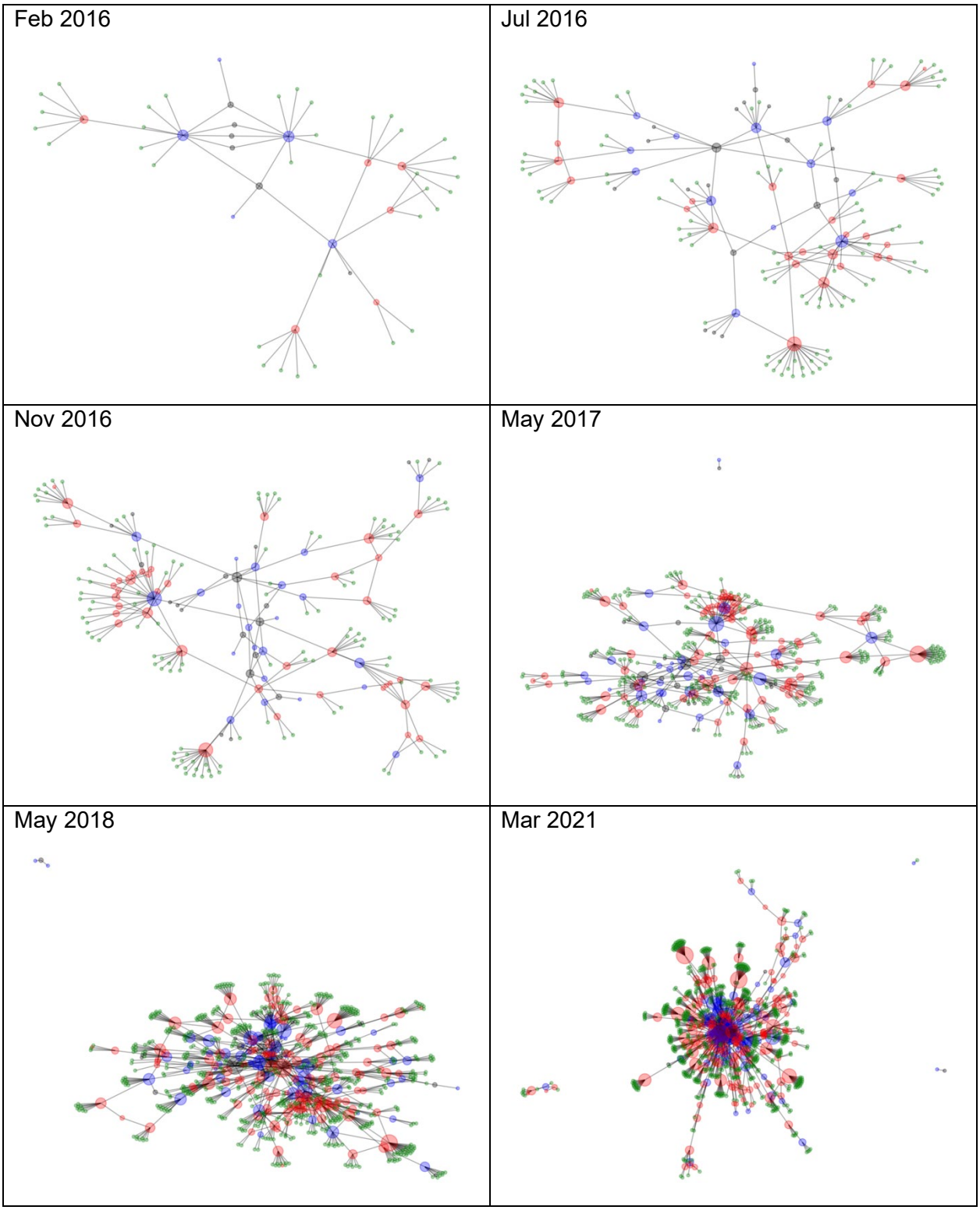

Figure 3. Evolution of the Module Dependency Graph at Select Points in Time for the Snorkel Project. *Snapshots are taken at intervals of approximately 530 commits.*

Linear Correlation Analysis

A linear correlation analysis of the metrics is described hereinafter. Using the Pearson correlation coefficient (r), it is possible to see if any of the metrics evaluated for the dependency graph are linearly co-dependent. These dependencies can provide insights regarding characteristics of the Snorkel code base.

As shown in Figure 4, the following group of metrics show $r > .99$ in all pairwise comparisons: GE, LGE, NLGE, n, m.

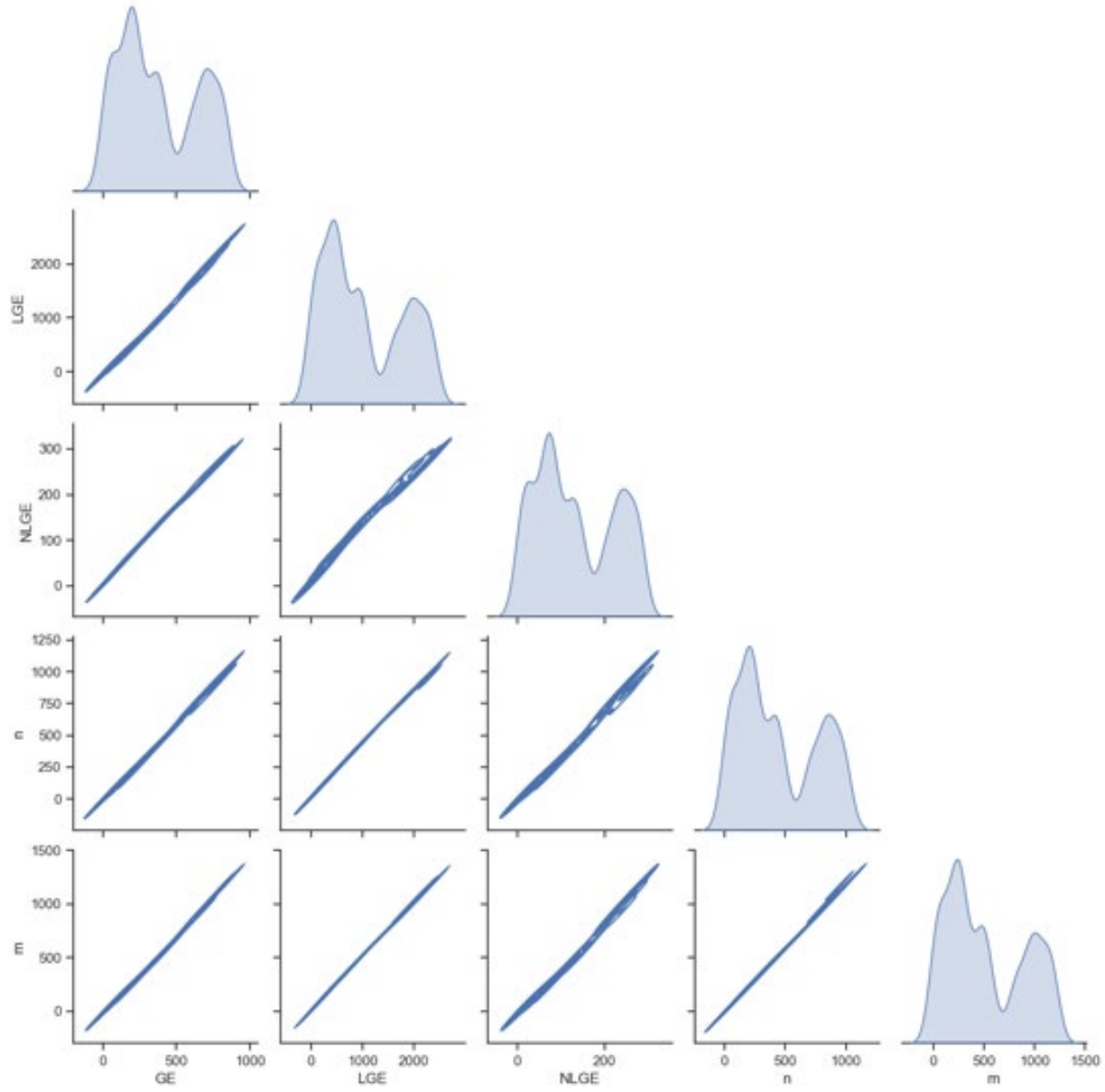

Figure 4. Comparison of GE, LGE, NLGE, Number of Nodes, and Number of Edges

The linearity between number of nodes (n) and number of edges (m) can be seen as a symptom of localized development. The addition of a module to the source code is followed by the connection of this module to one or more others. If for each additional module a low number

of connections are made, it means that the module is only being used in that specific part of the code. While a percentage of additions are justifiably of this type, most modules might also be reused in other locations and therefore should create more additional connections. A long-lasting linear relationship between n and m suggests a need for refactoring.

The linear relationship between GE and LGE is common in graphs with a close to uniform distribution of node degrees. In star graphs, GE would grow superlinearly with the number of nodes while LGE's behavior would converge to linear. The dissimilarity between the current dependency graphs and graphs with highly skewed distribution of node degrees is also seen in NLGE, which would be zero for star graphs.

Figure 5 shows a linear relationship ($r > .99$) in three pairwise comparisons between LNC, LNCn, and the maximum node degree. A linearity between LNC and LNCn is a characteristic of star graphs and wheel graphs. For graphs with more uniform degree distribution, the value of LNCn plateaus quickly with the number of nodes, while LNC's growth slows down more gently. This result is in contrast with the insights found in Figure 4, and adds a new research question regarding the relationship between these metrics and fundamental graph characteristics.

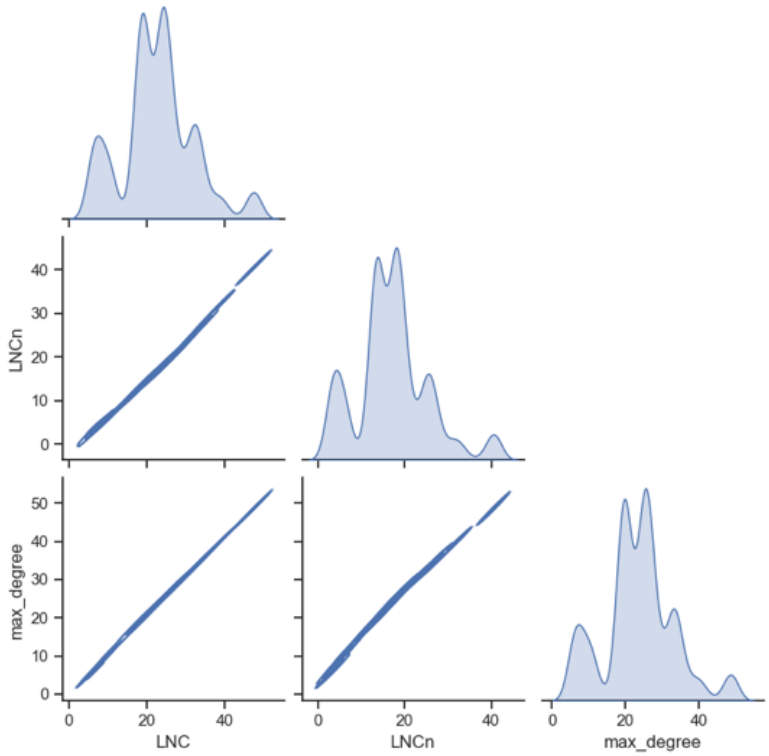

Figure 5. Comparison of LNC, LNCn, and Maximum Node Degree

The linear relationships of LNC and LNCn with the maximum node degree of the graph indicate that these metrics are connected to the size of the largest hub in the graph. This linearity is also found in star graphs, while in complete graphs, where there are no hubs by definition, and each node is equivalent to all the others, LNC would grow with a descending rate, and LNCn would plateau asymptotically towards 1.

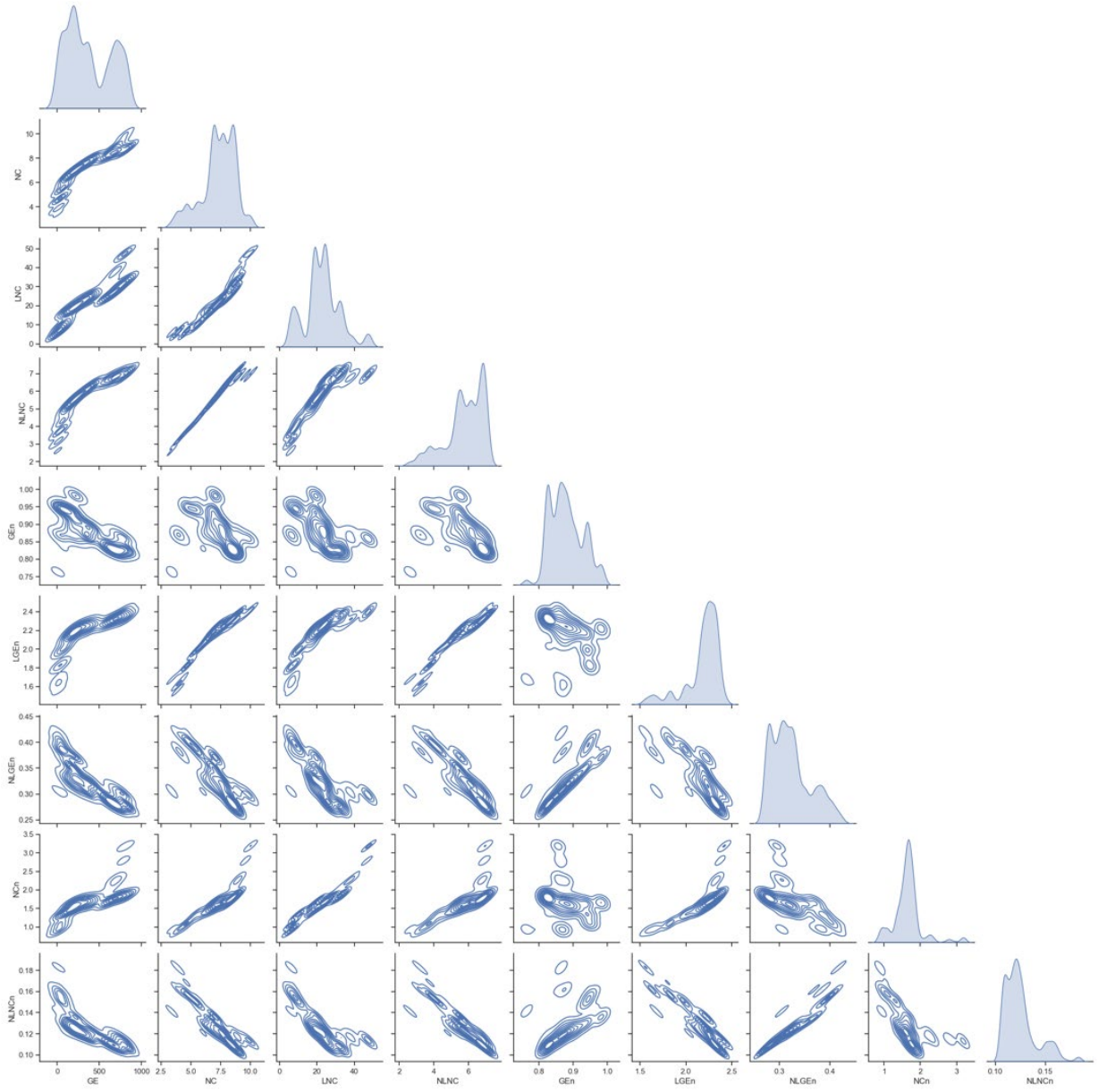

Figure 6. Comparison of Uncorrelated Metrics

Figure 6 shows the pairwise comparisons of all the metrics that do not present a clear linear correlation in the Snorkel code base. Some of these relationships are planned to be analyzed in subsequent research efforts, but an effort in narrowing the pool of metrics and towards a more purposeful metric design will be necessary to measure meaningful characteristics of software architectures.

Trends Over Time

The linear correlation analysis allows the connection of different metrics, in an effort to characterize the topology of the dependency graph. The actual development and creation of the codebase over the 5-year period can be analyzed by plotting some of these metrics over time. The evolution of the dependency graph presented in Figure 3 is depicted by the values of four of the metrics shown in Figure 7: GE, NC, GEn, and NCn.

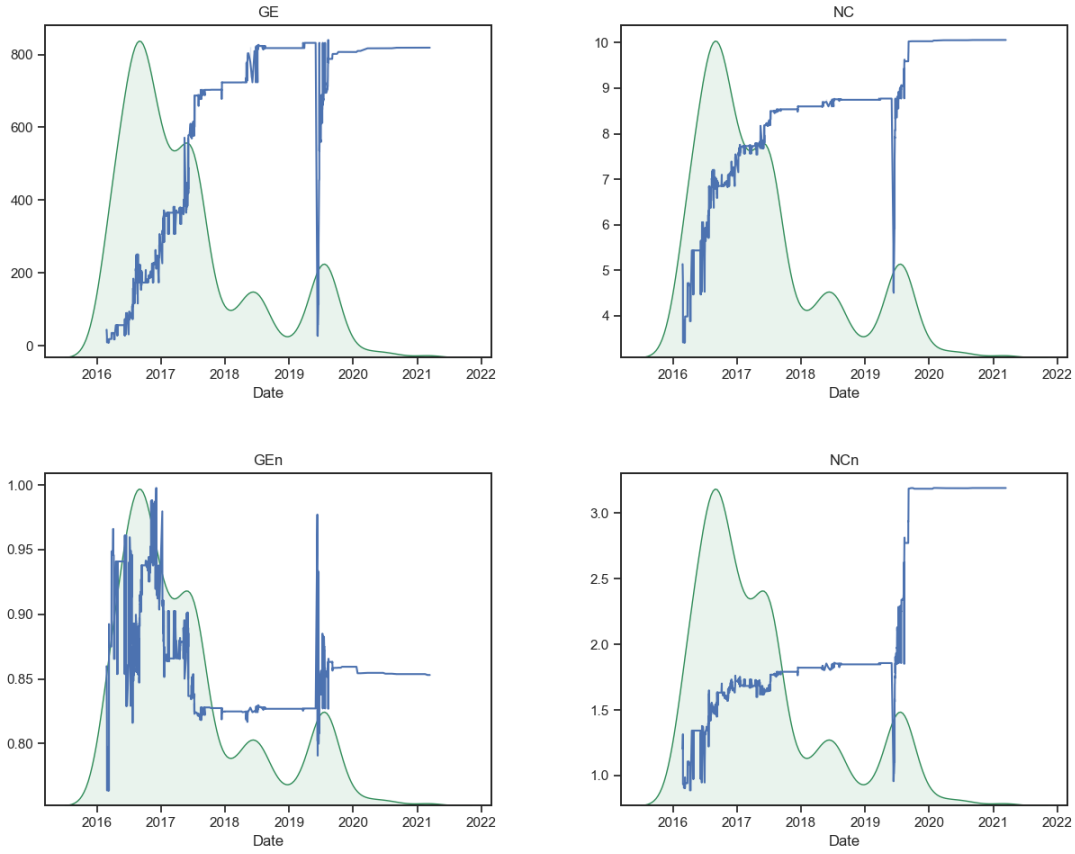

Figure 7. Trends for GE, NC, GEn, and NCn Over 6 Years of Project Development

Figure 7 presents a series of time plots for this select subset of metrics. For each metric, the green shaded area represents the frequency of commits in the project at a specific point in time. This frequency is not connected to the values on the y-axis. The plots show that the development of the project was very active in 2016 and 2017, with a smaller spike of activity in 2019, when, according to the commits, the project underwent a small overhaul, with frequent additions and removals of code. This allows us to better contextualize the changes in each metric and see how they react when the codebase is changed.

Graph energy (GE) quickly rises during the initial development, and fluctuates significantly during the overhaul, only to settle at essentially the same level afterwards. Natural connectivity (NC), on the other hand, rises also after the overhaul, suggesting that the changes made to the codebase in 2019 increased the cohesion of the whole project, without unnecessarily increasing coupling.

The comparison between GE and GEn shows the effect of the normalization factor $\gamma = \frac{1}{n}$, which was introduced to allow a comparison of graphs of different size (number of nodes). In this case, this normalization affects GEn to the point that the metric only seems to capture the frequency of the commits, and not the growth of the graph (as expected). This behavior is not the case when this normalization is applied to NC, as NCn still seems to be affected by the graph growth.

Conclusion

This paper presented a methodology to study the behavior of complex software systems in terms of their structural complexity with a focus on the modifiability of the code base. This approach is based on the parsing of the code and the creation of a dependency graph, a particular case of architectural structure that focuses on the dependency between software modules and the various ways they can call each other.

The dependency graph has been analyzed through the evaluation of a series of spectral metrics, which have shed light on some characteristics of the graph and given insights on the quality of the development effort. It is important to note that this approach forgoes the analysis of the actual lines of code and the dynamic effects that they will have at runtime and is therefore to be considered limited in scope and applicability.

In parallel to this analysis being carried out, the behavior of each metric is also being discovered, thus bootstrapping their applicability to the metrics. Behind the scenes, the metrics have been applied to conventional graphs, but the use case of a real software project is necessary to gauge the limitations of this approach.

Future research will continue the effort of connecting these and other metrics to important attributes of software code bases. Improvements to our own software tools will allow for analysis of projects with larger repositories, and with a longer development time frame, where the effects of technical debt might be more pronounced. Additional improvements are also planned for the visual representation of modifiability in software systems.

References

Amal, A., Sliman, L., Kmimech, M., Bhiri, M. T., & Raddaoui, B. (2018, September 26–28). *Towards a formal verification approach for cloud software architecture* [Paper presentation]. New Trends in Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 17th International Conference on New Trends in Intelligent Software Methodology, Tools and Techniques (SoMeT18), Granada, Spain.

Angelov, S., Grefen, P. W. P. J., & Greefhorst, D. (2009). A classification of software reference architectures: Analyzing their success and effectiveness. *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 141–150. https://doi.org/10.1109/WICSA.2009.5290800

Avci, C., Tekinerdogan, B., & Athanasiadis, I. N. (2020). Software architectures for big data: A systematic literature review. *Big Data Analytics*, *5*(1), 5. https://doi.org/10.1186/s41044-020-00045-1

Bahsoon, R., Ali, N., Heisel, M., Maxim, B., & Mistrik, I. (2017). Software architecture for cloud and big data. In I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, & B. Maxim (Eds.), *Software architecture for big data and the cloud*. Morgan Kaufmann.

Basili, V. R. (1980). Qualitative software complexity models: A summary. *Tutorial on models and methods for software management and engineering.*

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice* (3rd ed.). Addison-Wesley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture*. John Wiley & Sons.

Cabot, J., Capilla, R., Carrillo, C., Muccini, H., & Penzenstadler, B. (2019). Measuring systems and architectures: A sustainability perspective. *IEEE Software*, *36*(3), 98–100. https://doi.org/10.1109/MS.2019.2897833

Capilla, R., Nakagawa, E. Y., Zdun, U., & Carrillo, C. (2017). Toward architecture knowledge sustainability: Extending system longevity. *IEEE Software*, *34*(2), 108–111. https://doi.org/10.1109/MS.2017.54

Darcy, D. P., Kemerer, C. F., Slaughter, S. A., & Tomayko, J. E. (2005). The structural complexity of software an experimental test. *IEEE Transactions on Software Engineering*, *31*(11), 982–995. https://doi.org/10.1109/TSE.2005.130

Dingsøyr, T., Moe, N. B., Fægri, T. E., & Seim, E. A. (2018). Exploring software development at the very large-scale: A revelatory case study and research agenda for agile method adaptation. *Empirical Software Engineering*, *23*(1), 490–520. https://doi.org/10.1007/s10664-017-9524-2

Enos, J. R., Farr, J. V., & Nilchiani, R. R. (2019). Identifying and quantifying criticalilities in the acquisition of DoD system. *Defense Acquisition Research Journal*, *26*(1), 18–43. https://doi.org/10.22594/dau.18-799.26.01

Fischi, J., Nilchiani, R., & Wade, J. (2017). Dynamic complexity measures for use in complexity-based system design. *IEEE Systems Journal*, *11*(4), 2018–2027. https://doi.org/10.1109/JSYST.2015.2468601

Francesco, P. D., Malavolta, I., & Lago, P. (2017, April 3–7). *Research on architecting microservices: Trends, focus, and potential for industrial adoption* [Paper presentation]. 2017 IEEE International Conference on Software Architecture.

Garlan, D., & Shaw, M. (1993). An introduction to software architecture. *Advances in software engineering and knowledge engineering* (pp. 1–39).

Gutman, I. (2001). *The energy of a graph: Old and new results.* Berlin, Heidelberg.

Hasselbring, W. (2002). Web data integration for e-commerce applications. *IEEE MultiMedia*, *9*(1), 16–25. https://doi.org/10.1109/93.978351

Hasselbring, W. (2018). Software architecture: Past, present, future. In V. Gruhn & R. Striemer (Eds.), *The essence of software engineering* (pp. 169–184). Springer Open.

Hästbacka, D., Halme, J., Larrañaga, M., More, R., Mesiä, H., Björkbom, M., ... Hoikka, H. (2019, October 27–30). *Dynamic and flexible data acquisition and data analytics system software architecture* [Paper presentation]. 2019 IEEE SENSORS.

IEEE. (2000). Recommended practice for architectural description of software-intensive systems. In *IEEE 1471:2000*.

ISO/IEC. (1995). Reference model of open distributed processing (RM-ODP). In *ISO/IEC 10746:1995*.

Jun, W., Barahona, M., Yue-Jin, T., & Hong-Zhong, D. (2010). Natural connectivity of complex networks. *Chinese Physics Letters*, *27*(7), 078902. https://doi.org/10.1088/0256-307x/27/7/078902

Kazman, R., Bass, L., Webb, M., & Abowd, G. (1994). *SAAM: A method for analyzing the properties of software architectures* [Paper presentation]. Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy.

Kendall, F. (2016). *Performance of the defense acquisition system*. DoD.

Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Software*, *12*, 45–50.

Kruchten, P., Obbink, H., & Stafford, J. (2006). The past, present, and future for software architecture. *Software, IEEE*, *23*, 22–30. https://doi.org/10.1109/MS.2006.59

Linington, P. F. (1995). RM-ODP: The architecture. In K. Raymond & L. Armstrong (Eds.), *Open distributed processing: Experiences with distributed environments. Proceedings of the third IFIP TC 6/WG 6.1 international conference on open distributed processing, 1994* (pp. 15–33). Springer U.S.

MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, *52*(7), 1015–1030. https://doi.org/10.1287/mnsc.1060.0552

Malavolta, I., & Capilla, R. (2017, April 5–7). *Current research topics and trends in the software architecture community: ICSA 2017 workshops summary* [Paper presentation]. 2017 IEEE International Conference on Software Architecture Workshops.

Mens, T. (2016). *Research trends in structural software complexity*. https://arxiv.org/abs/1608.01533

Newman, S. (2015). *Building microservices*. O'Reilly.

Nilchiani, R. R., & Pugliese, A. (2016). *A complex systems perspective of risk mitigation and modeling in development and acquisition programs*. Stevens Institute of Technology.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM, 15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Perry, D., & Wolf, A. (2000). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*. https://doi.org/10.1145/141874.141884

Pugliese, A., Enos, J., & Nilchiani, R. (2018). *Acquisition and development programs through the lens of system complexity*. Naval Postgraduate School.

Pugliese, A., & Nilchiani, R. (2017). *A systems complexity-based assessment of risk in acquisition and development programs*.

Pugliese, A., & Nilchiani, R. (2019). Developing spectral structural complexity metrics. *IEEE Systems Journal*, *13*(4), 3619–3626. https://doi.org10.1109/JSYST.2019.2912368

Putman, J. (2000). *Architecting with RM-ODP*. Prentice Hall.

Qureshi, N., Usman, M., & Ikram, N. (2013). *Evidence in software architecture, a systematic literature review*. Paper presented at the Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, Porto de Galinhas, Brazil.

Salado, A., & Nilchiani, R. (2013). *Using requirements-induced complexity to anticipate development and integration problems: Analysis of past missions* [Paper presentation]. AIAA SPACE 2013 Conference and Exposition.

Shahin, M., Liang, P., & Babar, M. A. (2014). A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, *94*, 161–185. https://doi.org/10.1016/j.jss.2014.03.071

Shaw, M., & Clements, P. (2006). The golden age of software architecture. *IEEE Software*, *23*(2), 31–39.

Soni, D., Nord, R., & Hofmeister, C. (1995). Software architecture in industrial applications. *17th International Conference on Software Engineering*, 196–196.

Taibi, D., Lenarduzzi, V., Pahl, C., & Janes, A. (2017). *Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages* [Paper presentation]. Proceedings of the XP2017 Scientific Workshops, Cologne, Germany.

Venters, C. C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., ... Carrillo, C. (2018). Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software*, *138*, 174–188. https://doi.org/10.1016/j.jss.2017.12.026

Witt, B., Baker, T., & Merritt, E. (1994). *Software architecture and design: Principles, models, and methods*. Van Nostrand Reinhold.

Woods, E. (2016). Software architecture in a changing world. *IEEE Software*, *33*(6), 94–97. https://doi.org/10.1109/MS.2016.149

Acquisition Research Program
Graduate School of Defense Management
Naval Postgraduate School
555 Dyer Road, Ingersoll Hall
Monterey, CA 93943

www.acquisitionresearch.net