SIT-SE-22-003

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Risk Quantification of Acquisition Programs Through Systems Complexity Measures

October 8, 2021

Dr. Roshanak Rose Nilchiani, Associate Professor
Stevens Institute of Technology

Dr. Antonio Pugliese, Assistant Professor,
Embry-Riddle Aeronautical University

Abstract

The objective of this research is to mathematically formulate and manage the relationship between the quantitative complexity level of an acquisition or engineering development program and its relationship to the increased technical and programmatic risk, respectively. This research builds upon the PIs previous research experience and grants (NPS BAA 14-002, NPS BAA 15-001, NPS FOA 16001). This research aims to discover and determine the relationship between the quantitative complexity value of an acquisition program (at various points in its lifecycle) as a measure of increased actual technical and programmatic risk respectively. The main goal is to improve the current inaccurate subjective practice of assessment of risk in different stages of a wide range of engineered system development programs as well as acquisition programs.

Currently lifecycle risk assessment methodologies such as color-coded risk matrix are heavily subjective in their nature and therefore weak in the assessment of the actual risk. As a result, acquisition programs frequently are exposed to unforeseen technical and programmatic risks and failures; cost and schedule overruns that are due to inaccurate risk identification and assessments. This research proposal focuses on expanding and examining the novel set of new complexity measures that are recently created by our team (with the PIs previous NPS research grants) as pre-indicators of emergence of risks at different stages of a systems development process and lifecycle. The detailed set of created complexity measures, will be modified and categorized based on their application category in physical/hardware/software systems as well as DoD System of Systems level studies. The refinement and categorization of the complexity/risk measures will be applied to and examine several historical case studies of engineered systems success or failures. The focus of this part of research will be on discovering the suitability of each of the 12 complexity/risk measures for application to the right type and category of subsystem/system/SoS of acquisition programs or complex engineered systems. The focus of the case studies chosen will be at refinement and choice of complexity/risk metric to appropriately fit a particular complex engineered system to various manifestation of increased (or

decreased) technical as well as some programmatic risks. Multiple historical and theoretical cases of design of complex engineered systems will be studied.

The results of this research project will have a broad public purpose in systems development community in various domains of engineering by improving the quantitative assessment of risk from the preliminary and critical design phase, manufacturing and testing, implementation, operation and the retirement of the system. The research result is expected to be applied to a variety of cyber-physical systems as well as DoD systems of Systems (SoS). The complexity-based risk assessment can be applied to various domains of applications such as telecommunication satellite design, regional power infrastructure design and operation, and the next generation of human spaceflight vehicle and many more. The suggested improved methodology can warn the program manager and the other stakeholders on assessing the alternative courses of action at each stage in systems lifecycle as well as reduction and management of the complexity content to mitigating some of the technical risk that a system is facing.

SIT-SE-22-003

ACQUISITION RESEARCH PROGRAM SPONSORED REPORT SERIES

Risk Quantification of Acquisition Programs Through Systems Complexity Measures

October 8, 2022

Dr. Roshanak Rose Nilchiani, Associate Professor
Stevens Institute of Technology

Dr. Antonio Pugliese, Assistant Professor,
Embry-Riddle Aeronautical University

THIS PAGE LEFT INTENTIONALLY BLANK

Table of Contents

THIS PAGE LEFT INTENTIONALLY BLANK

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 (Pugliese, Enos, & Nilchiani, Acquisition and Development Programs Through the Lens of System Complexity, 2018; Nilchiani & Pugliese, 2017; Fischi, Nilchiani, & Wade, 2015; Enos, Farr, & Nilchiani, 2019; 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, Kemerer, Slaughter, & Tomayko, 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.

Based on the above-described circumstances, the research project for which this report presents the results set out to develop a static analysis method for source code of software. As such, the following sections will first outline the tasks and scope of the work as well as the initially proposed steps. Following the scope, a literature review is presented that provides the current state of the research and relation to other publications. Subsequently, the applied methodology and process is outlined. Based on the applied methodology, the results will be presented and finally, a conclusion will be provided that also considers the initial tasks and scope.

THIS PAGE LEFT INTENTIONALLY BLANK

Tasks, Scope, and Proposed Work

At the proposal stage, five tasks were planned to be executed. Each task was divided into sub-tasks that were addressed successively over the course of the project. The exact task list is shown hereinafter and also reconsidered in the conclusion.

Task 1. Selection of metrics among the ones created. Some metrics have shown inaccurate prediction of complexity characteristics, and more data is needed to justify their elimination or presence in future research.

1. Analysis of developed metrics
2. Selection of metrics based on representation of complexity

Task 2. Selection of data. The data coming from GitHub is accurate and can be parsed with ad-hoc software. Interesting projects with public information available regarding their scale and overall success will be targeted for this research effort.

1. Collection of datasets
2. Selection of most informative datasets

Task 3. Development of architecture-generating software. The developed software allows for the generation of architectures based on functional dependencies, at the file level granularity. The generation of a finer architecture, or ones based upon different interface types need modifications to the code.

1. Selection of alternative interfaces
2. Development of architecture-generating software
3. Selection of alternative programming languages of the SOI
4. Development of architecture-generating software
5. Selection of deeper granularity levels
6. Development of architecture-generating software

Task 4. Evaluation of complexity metrics for the generated architectures. The selected complexity metrics will be evaluated based upon the architectural data.

1. Evaluation of complexity metrics
2. Evaluation of complexity metrics
3. Visualization of results

Task 5. Analysis of results. The complexity results will be compared to public data regarding the project scale, cost, and development issues.

1. Analysis of results

Based on this scope, the sections described in the introduction correspond to the tasks listed above. As such, Task 1 is addressed by the literature review, Tasks 2 and 3 are outlined as part of the methodology and process, and lastly, Tasks 4 and 5 are presented in form of the results and insights. Table 1 below shows the structure of the report regarding the tasks above:

Table 1 - Twelve examples of spectral structural complexity metrics

Task Number	**Report Section**
Task 1	Literature Review and State of the Research
Task 2	Methodology and Process
Task 3	
Task 4	Results and Insights
Task 5	

With this scope, the next section will begin with the literature and state of the research.

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 70s of the twentieth century. Especially over the last 30 years, software architecture emerged as an important field for both research and practice (Shahin, Liang, & Babar, 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, Grefen, & Greefhorst, 2009; Avci, Tekinerdogan, & Athanasiadis, 2020; Garlan & Shaw).

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). The first appearances and mentions of SA can be found in the publication of Parnas in 1972 (Parnas, 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, Clements, & Kazman, 2012).

From the aforementioned time till around 1990, architecture in scientific fields was mostly related to systems (Kruchten, Obbink, & Stafford, 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, Usman, & Ikram, 2013). The first book about SA was also published during these beginning times in 1994 (Witt, Baker, & Merritt, 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, for instance. Kruchten (Kruchten et al., 2006) lists various approaches that resulted from these efforts: Software Architecture Analysis Method (Kazman, Bass, Webb, & Abowd, 1994), the 4+1 view (Kruchten, 1995), Siemens' four views (Soni, Nord, & Hofmeister, 1995), and numerous other patterns that address the design of SA (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996) as well as Architecture Description Languages (ADLs) (Shaw & Clements, 2006).

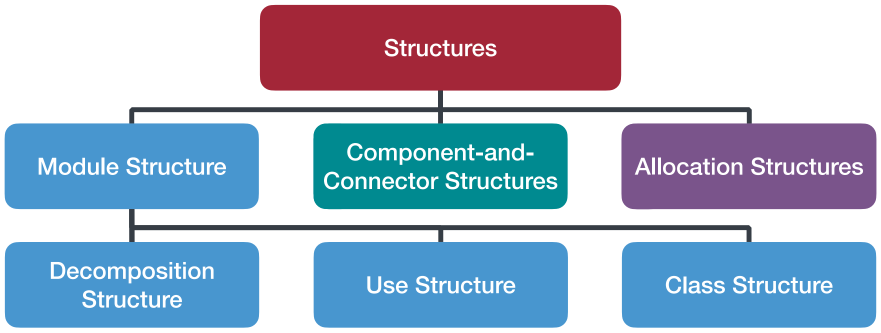

Figure 1 - Classification of relevant architectural structures for software systems

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 & Clements describe as "Popularization" (Shaw & Clements, 2006). Therefore, new trends and explorations also must be considered since they are a natural continuation of the described state.

Looking at the last five years, a few trends in SA emerge. 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, Sliman, Kmimech, Bhiri, & Raddaoui, 2018; Bahsoon, Ali, Heisel, Maxim, & Mistrik, 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, Moe, Fægri, & Seim, 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 an informed manner if they are not defined up-front (Dingsøyr et al., 2018; Wilhelm 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 (W. Hasselbring, 2002), the applicability and longevity of SAs can become problematic if they are tightly vertically integrated. Thus, approaches such as Microservices (Francesco, Malavolta, & Lago, 2017; Newman, 2015; Taibi, Lenarduzzi, Pahl, & Janes, 2017) and other solutions to these problems (Capilla, Nakagawa, Zdun, & Carrillo, 2017), which then also address sustainability (Cabot, Capilla, Carrillo, Muccini, & Penzenstadler, 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 sub-categories 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.

Based on the above-described literature and research, the tasks outlined in the previous section were approached. As such, the source code of an open-source Python library, Snorkel was analyzed. This analysis was conducted in a static manner which 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 was used as the basis of the static analysis.

Methodology and Process

As mentioned above, a static analysis of the source code of a software package developed using the Python 3 programming language was developed and conducted. 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, Rusnak, & Baldwin, 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 (A. Pugliese & Nilchiani, 2019). These metrics are based on other metrics, such as graph energy (Gutman, 2001) and natural connectivity (Jun, Barahona, Yue-Jin, & Hong-Zhong, 2010).

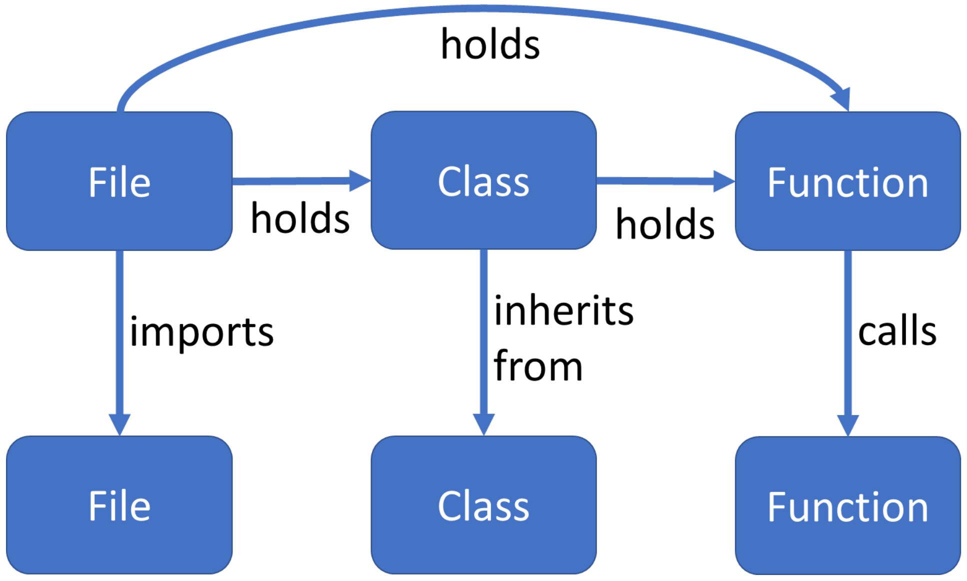

Figure 2 - Types of dependencies among graph elements

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 above shows the types of dependencies among the elements of the graph.

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 2 below 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 twelve possible metrics. Hereinafter, 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 2 - Twelve examples of spectral structural complexity metrics

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

THIS PAGE LEFT INTENTIONALLY BLANK

Results and Insights

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.

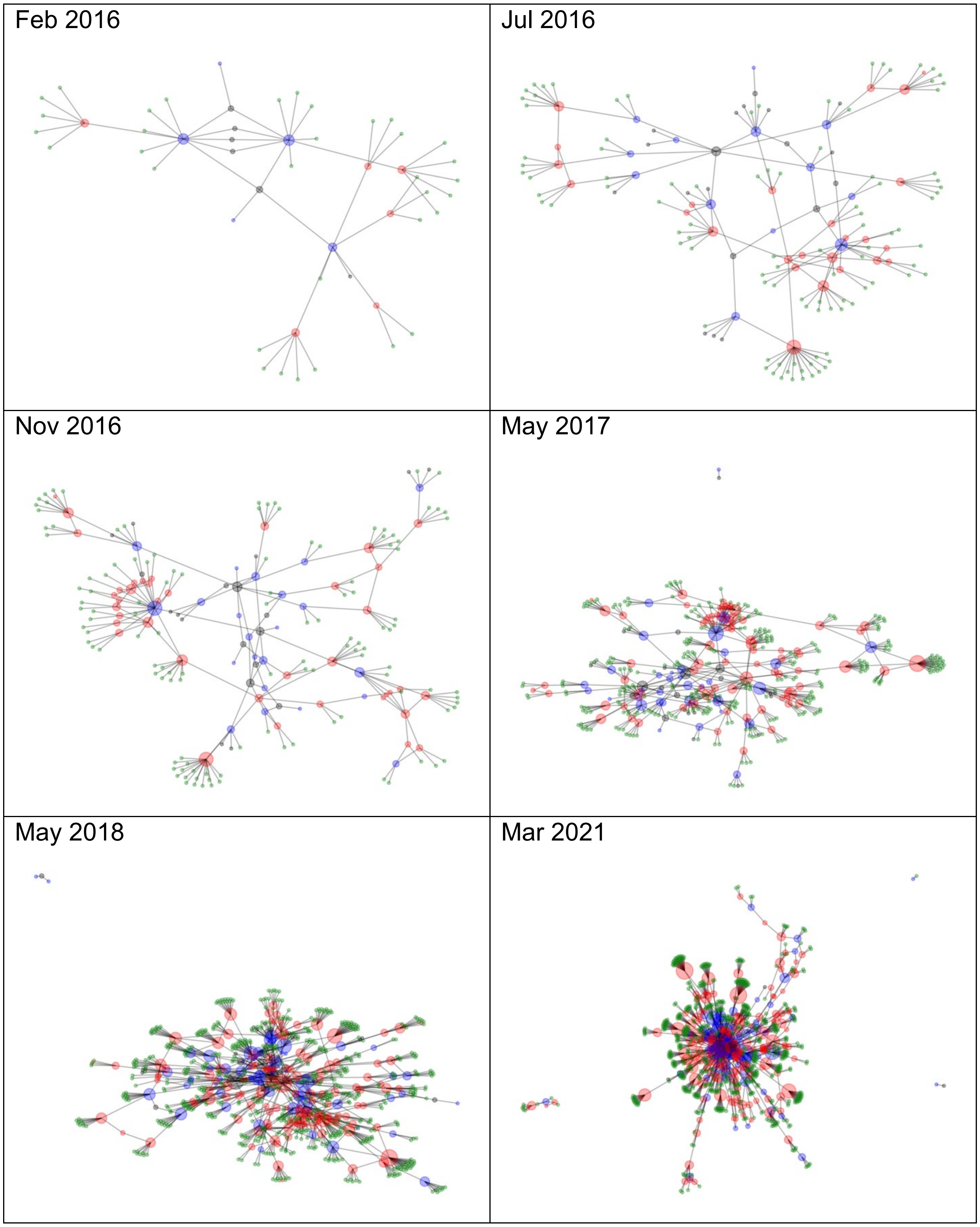

Figure 3 - Evolution of the module dependency graph at select points in time for the Snorkel project. Snapshots are taken at intervals of approximately 530 commits.

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 tt > .99 in all pairwise comparisons: GE, LGE, NLGE, n, m.

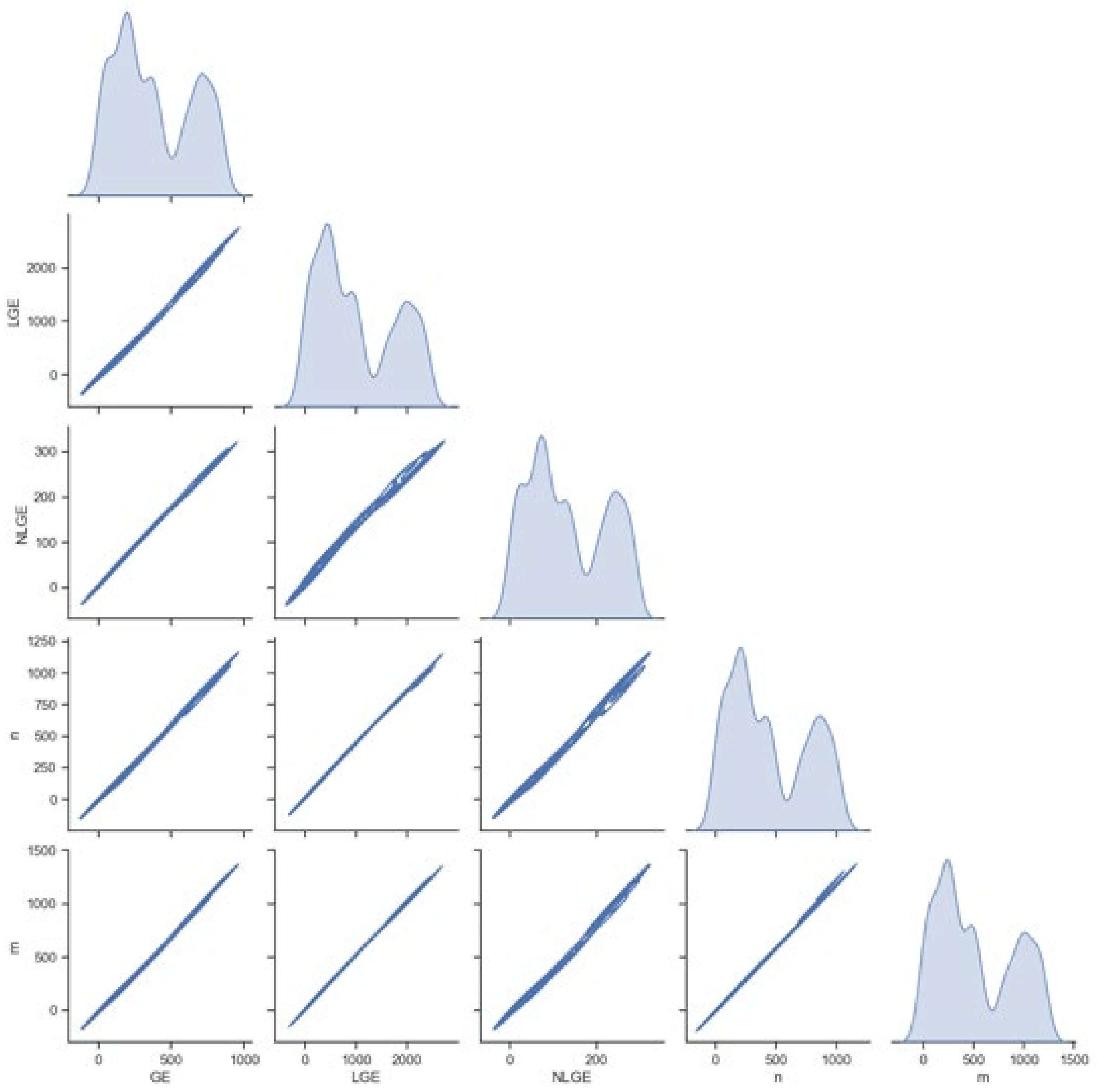

Figure 4 - Comparison of GE, LGE, NLGE, number of nodes, and number of edges

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

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 super linearly 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.

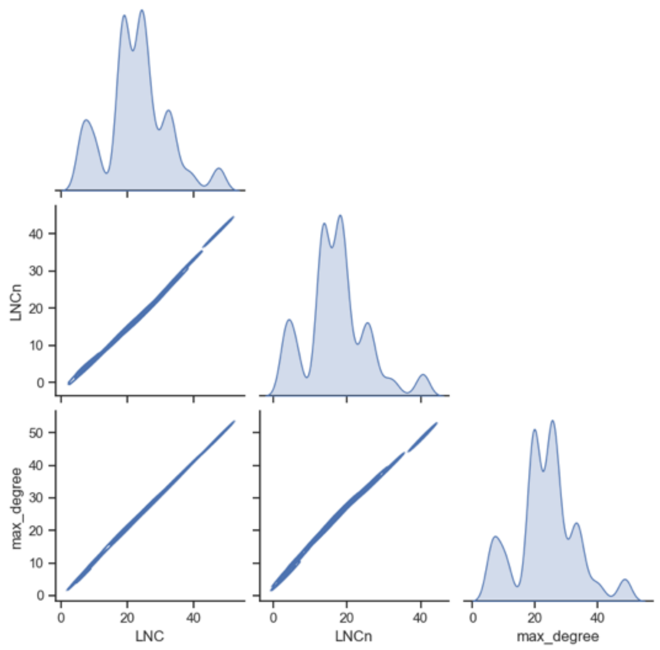

Figure 5 - Comparison of LNC, LNCn, and maximum node degree

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.

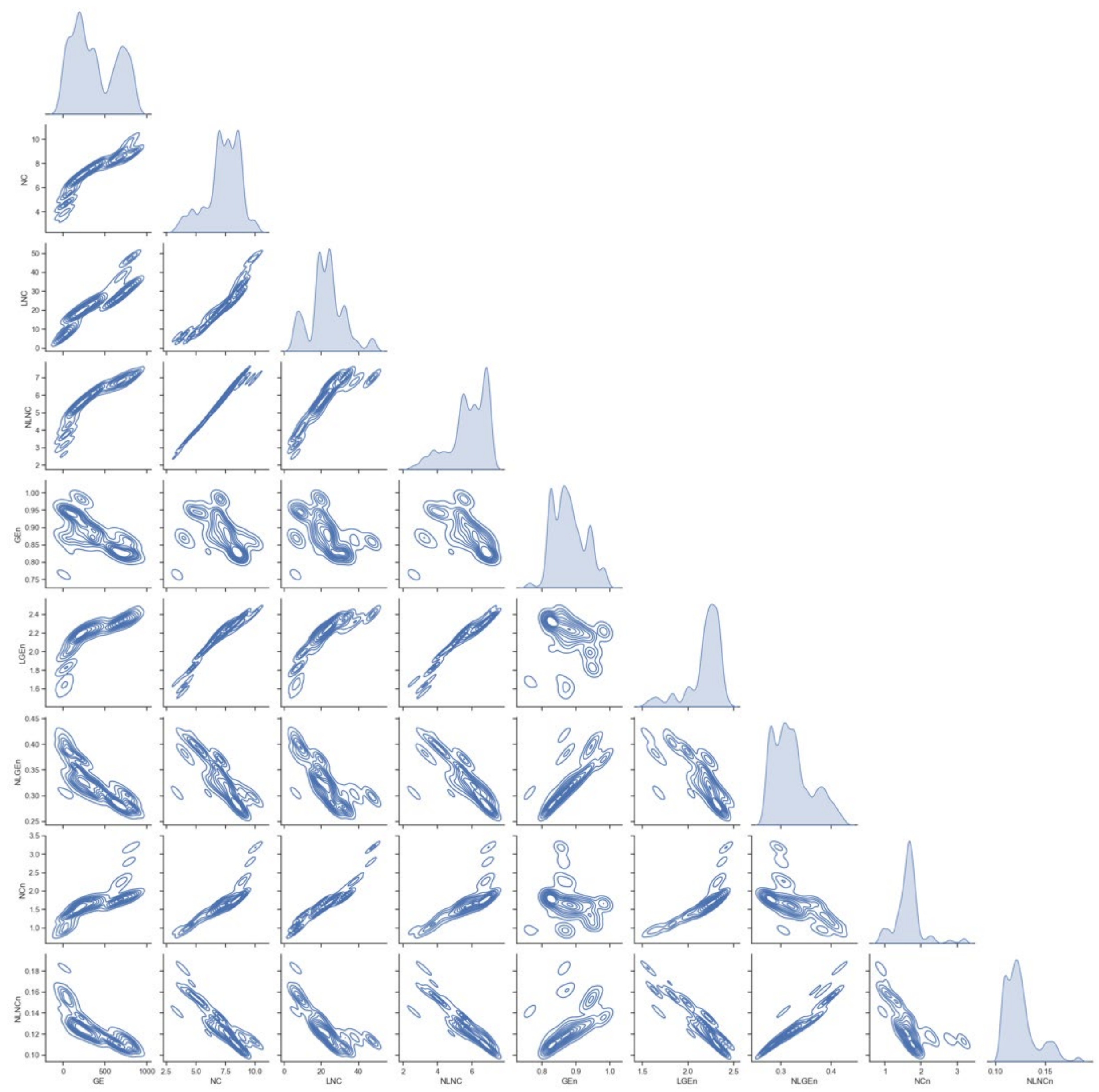

Figure 6 - Comparison of uncorrelated metrics

Figure 6 shows the pairwise comparisons of all the metrics which 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 five-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 below by the values of four of the metrics: GE, NC, GEn, and NCn.

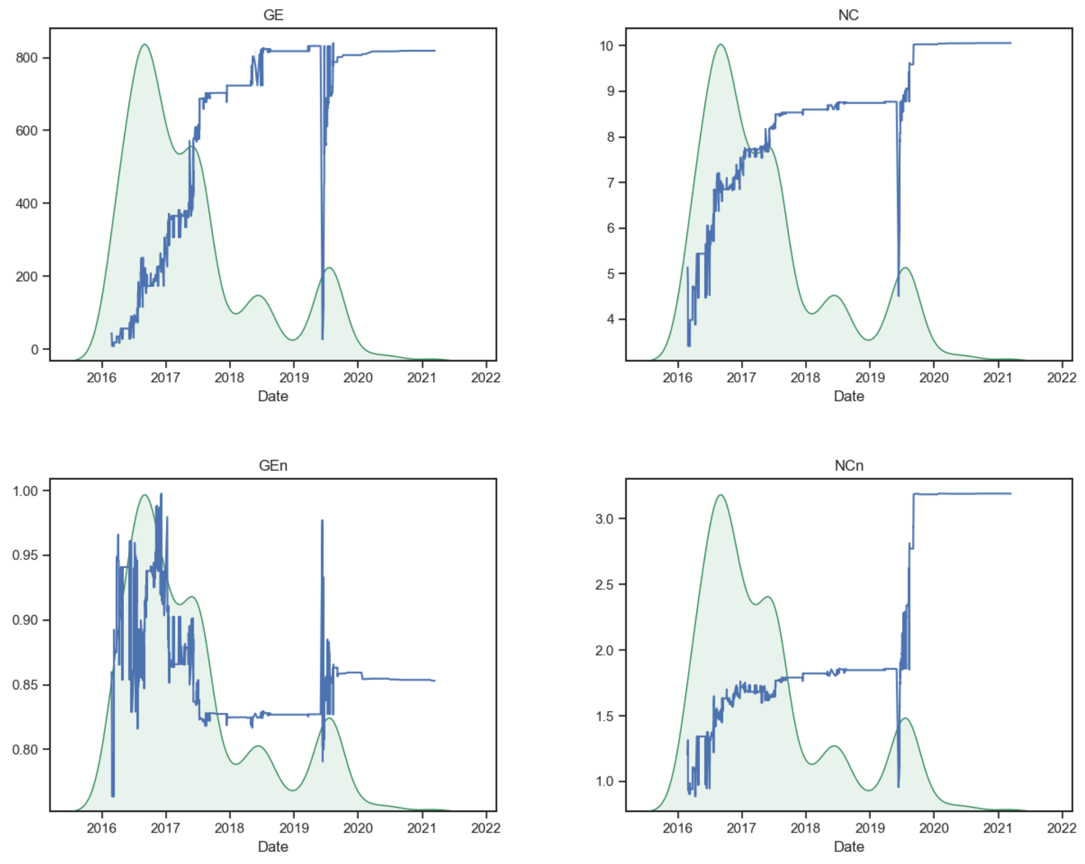

Figure 7 - Trends for GE, NC, GEn, and NCn over 6 years of project development

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

The report 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.

As a result of the proposal structure outlined in the second section, the following list shows the completed tasks:

Task 1. Completed including al sub-tasks
Task 2. Completed including all sub-tasks
Task 3. Completed except for sub-task 3, which turned out to be not feasible
Task 4. Completed including all sub-tasks
Task 5. Completed including all sub-tasks

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.

Bibliography

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

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*, 982-995.

Enos, J. R., Farr, J. V., & Nilchiani, R. R. (2019). IDENTIFYING AND QUANTIFYING Critical ilities in the ACQUISITION of DoD Systems. *Defense Acquisition Research Journal: A Publication of the Defense Acquisition University, 26*.

Fischi, J., Nilchiani, R., & Wade, J. (2015). Dynamic Complexity Measures for Use in Complexity-Based System Design. *IEEE SYstems Journal*.

Gutman, I. (2001). The energy of a graph: old and new results. In *Algebraic combinatorics and applications* (pp. 196-211). Springer.

Kendall, F. (2016). Performance of the Defense Acquisition System. Department of Defense.

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*, 1015-1030.

Mens, T. (2016). Research trends in structural software complexity. *arXiv preprint arXiv:1608.01533*.

Nilchiani, R. R., & Pugliese, A. (2016). *A Complex Systems Perspective of Risk Mitigation and Modeling in Development and Acquisition Programs.* Tech. rep., Stevens Institute of Technology Hoboken United States.

Nilchiani, R. R., & Pugliese, A. (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*.

Pugliese, A., Enos, J., & Nilchiani, R. (2018). Acquisition and Development Programs Through the Lens of System Complexity. 136-154.

Salado, A., & Nilchiani, R. (2013). Using Requirements-Induced Complexity to Anticipate Development and Integration Problems: Analysis of Past Missions. *AIAA SPACE 2013 Conference and Exposition*, (p. 5357).

Wu, J., Barahona, M., Yue-Jin, T., & Hong-Zhong, D. (2010). Natural connectivity of complex networks. *Chinese physics letters, 27*, 078902.