International Journal of Research in Engineering, Science and Management
Volume 7, Issue 7, July 2024
https://www.ijresm.com | ISSN (Online): 2581-5792

Analyzing the Impact of Software Complexity on Software Defect Resolution Time

Thamindu Chankana Menuwara Gamage*
Undergraduate Student, Faculty of Computing, Sri Lanka Institute of Information Technology, Colombo, Sri Lanka

***Abstract*: The time it takes to fix software issues can be significantly influenced by program complexity. Complex software systems are frequently made up of several interconnected components, making it difficult to pinpoint the source of errors. Complex code might be more difficult to adapt and test, necessitating additional work to ensure that the fix does not cause new flaws. Furthermore, as the complexity of the software system rises, the cognitive load placed on developers increases, making it more difficult for them to diagnose and resolve errors efficiently. As a result, this study defines Macro-Level Cognitive Interaction (MaCLI), a methodology for analyzing the impact of software complexity on software defect resolution time. This framework focuses on how developers interact with software, the cognitive processes they use to detect and resolve faults, as well as to increase the speed and efficacy of defect resolution. By emphasizing cognitive load, information accessibility, visualization, and software design, MaCLI provides a holistic strategy for reducing software complexity and increasing the efficiency and speed of defect resolution. By implementing efficient software design and architecture, as well as leveraging tools and methods that enable successful defect resolution, developers can limit the impact of software complexity on defect resolution time and ensure the continued success of their software projects.**

***Keywords*: Cognitive load, Defects, MaCLI, Software complexity, Resolution time.**

1. Introduction

Technology has become a part of our lives in today's world where use of software is evident in all sectors. Software is used in all spheres of life to perform tasks and provide solutions to various questions, ranging from simple applications on mobile devices to complex business solutions. But the number and the criticality of the software faults have grown in parallel with the growth of software complexity. Program defects may lead to data loss, system crashes, security breaches, and other major problems [1]. These problems can be costly to solve, leading to lost revenue and performance for businesses and individuals who rely on the software [2].

Software complexity is one of the main factors that determine the extent and duration of software failures. Software complexity can be defined as the extent of interdependency and interconnectivity between different software components that can complicate the understanding and predictability of the system [3]. As the complexity of software increases the time required to correct it increases as well because it becomes challenging to determine the root of the problem [4]. Therefore, the complexity of the program should not be overlooked by developers and organizations when planning for software defect resolution time. Through simplifying the software and applying proper approaches to software development and testing, organizations can minimize the occurrence of defects and the time and money needed to fix them, and thus create more reliable and effective systems [5].

In view of Macro-Level Cognitive Interaction (MaCLI), this research offers a fresh approach towards identifying and fixing software defects. MaCLI is a high-level abstraction type of approach to evaluating software systems that utilize cognitive models to identify patterns and irregularities that imply likely defects. [6]. It can be useful in the simplification of software and the enhancement of time taken in solving defects hence leading to effective and reliable software systems.

Moreover, the current research work offers a new detection and resolution technique of software defects using the Rayleigh distribution curve fit. The Rayleigh distribution curve can be used to look for patterns in software defect data, so that developers can be made aware of possible bugs at a earlier stage in the development process and to reduce the time taken to fix these bugs. [7]. Rayleigh distribution curve is a probability density function that is frequently applied in engineering and science to fit data. It can be noted that the shape of the Rayleigh distribution curve can be used to predict the likelihood of occurrence of a specific defect based on the past records. [8]. This method can help developers when prioritizing issues based on their severity and possible impact, resulting in more efficient and effective defect remediation.

Number of errors, average defect velocity and the correlation with software complexity is also another approach used in this study and it involves monitoring the number of errors. There are two basic measures that reflect the quality of the developed software, the total number of bugs and the average defect velocity. The former is the total number of defects that are found in the development process while the latter is the average rate at which defects are fixed [9]. Monitoring these data allows developers to have a better understanding of the system's behavior and identify any problems more quickly.

The objectives of this study will include the work done by other researchers on software complexity and its correlation based on the defect resolution time, the framework for defect

*Corresponding author: it20604530@my.sliit.lk

prediction that was proposed, the procedure followed in the experiment, results and any possible future for the study, as explained in the methodologies above.

2. Literature Review

Maintainability is one of the quality attributes of a software that among other factors depends on the level of complexity of the piece of software. The modern world is dominated by technology, and this has led to many folds increase in the sophistication of software and the process of software development has also become more complex. Due to this, the number of software problems has increased and has also taken time to be solved. Thus, we need to explore how time to fix software defects is influenced by software complexity and how this influence can be reduced in as many ways as possible.

The complexity of software can be defined by several types of complexity, the most popular of which are structural, functional and interface complexity. It is the structural characteristics of the software; this means it is the number of modules and / or the number of subsystems together with the level of interconnections between them that has been put in place in the software. The number and variations of functions and the number of features that a piece of software has are aspects that define its functional complexity. The interface complexity is related to the relations and dependencies between the components and systems of the software.

The amount of time needed to resolve software defects has been demonstrated to be significantly impacted, according to the findings of several research studies. For instance, Briand et al. (2000) investigated the connection between the complexity of software and the likelihood of errors occurring. They concluded that software with high structural complexity is more prone to errors, which results in longer timeframes required to resolve faults. The authors argued that structural complexity is the primary source of software errors and that reducing this complexity should be a priority in the process of developing software [10].

Along the same line of thought, Xie et al. (2003) conducted research to establish the relationship between the software measures and the number of software errors. The authors of the paper also state that when the functional complexity of software is high, it contains more errors, meaning more time is spent on fixing the flaws. The study indicated that reduction of the functional complexity could help to improve software quality as well as decrease the time required to fix the defects [11].

In another study, Chidamber and Kemerer (1994) looked at the relationship between the level of design patterns and their maintainability. They found that the longer the time taken to fix defects, the more complex the software interface is hence concluded that software with complex interface is hard to maintain. They suggested that the level of complexity of user interfaces should be reduced so that the software is easier to maintain [12].

Modularization is another method that is similar to the process of division of a single software into several parts that are easier to handle. Basili et al. (1996) is another study that has focused on software complexity and the maintenance aspects of the software as well as the time that is taken to fix the defects. They stated that modularization could reduce the level of software complexity which in turn enhances the quality of the software and reduces the time that is taken to fix a defect [13].

As to the possible ways of reducing the effect of software complexity on the time required to eliminate defects, there are several strategies that can be employed. One of these techniques is known as refactoring, and this is the process of restructuring the code to make it easier to read and modify, although the actual functionality of the program does not change at all. Fowler et al. (1999) surveyed the impact of the organizational structure on the quality of the software. They stated that it is possible to enhance the quality of the software significantly and reduce time for the defects' correction through refactoring [14].

Similarly, the simplification of software architecture can also be utilized to both lessen the complexity of software and enhance its overall quality. A study by Parnas (1972) investigated the impact of different software architectures on the overall product quality. The study concluded that architectural simplification can considerably increase software quality and cut down on the amount of time needed to resolve defects by lowering the overall complexity of the product [15].

Halstead (1977) conducted a study to establish the effect of different measures of software complexity on the price of the software. The study focused on examining the impact of software complexity measures like cyclomatic complexity, code churn, and coupling on the time taken to fix the software faults. Based on the metrics, it can be concluded that the level of software complexity is associated with longer time required to fix a defect. The study recommended that managing software complexity during development as one of the most efficient ways of minimizing the number of software bugs and time required to fix them [16].

In a study titled "The effects of software design complexity on defects": Research by Hummel et al. (2011) titled "On the interaction of software design complexity and fault-proneness: A study in open-source systems," explored the interaction between the software design aspects and defects in open-source systems. The paper quantified associations between several aspects of software complexity such as the size of the modules, the code churning, and code complexity to the time it takes to fix software faults. The results of the study showed that as the software complexity increases the time needed to fix software defects also increases especially when the measurement is done in terms of the larger modules and complex code. The study concluded by stressing the need for realistic and effective approaches for managing software complexity to enhance the efficiency of the process of fixing defects in open-source projects [17].

What has been revealed is that the size of a software program does significantly affect the time that it takes to fix software bugs; therefore, software size should be a key concern for software designers. The literature review showed that there are several ways to classify software complexity, and they include structural complexity, functional complexity, and interface complexity. Software defects resolution time is another factor that is uniquely affected by the different types of complexity.

Simplification of software can contribute to improving the quality of the software and the time taken to fix any defects. There are several ways to decrease the level of software's complexity and at the same time increase its quality; some of them are: refactoring, architectural simplicity, and modularity.

3. Definition and Measurement of Software Complexity

The term "software complexity" is the level of difficulty and the measure of the software system in terms of the level of complication. This concept covers a broad area of the system such as the design of the system, how the system is put into practice, what the system is supposed to do, and how the system is supposed to work with other systems. There is a need to analyze the relation between software complexity and the time needed to fix defects, which requires understanding and quantifying the concept of software complexity. Here are the general definitions and approaches of software complexity:

A. Cyclomatic Complexity

Cyclomatic complexity presented by McCabe calculates the software structural complexity given by the number of linearly independent paths through a control flow graph. Cyclomatic complexity is used to measure the changes and the level of understanding of the software, and more cyclomatic complexity is equal to high complexity. [3]. This metric has been widely adopted for evaluating and managing software complexity and the research has been shown to correlate with the number of defects and maintenance effort required [18].

B. Halstead Metrics

Halstead metrics introduced by Maurice Halstead are aimed at estimating the complexity of software and calculating depending on the number of different operators and operands that are used in the code. These are measures of different aspects of the program including its length, the number of words used, the amount, difficulty, and the amount of effort required. Halstead's metrics give information about the complexity and maintainability of software based on the measurement of aspects that are associated with size and organization of the codes. [16]. Research has demonstrated that these metrics can predict defect density and maintenance effort [19].

C. Coupling and Cohesion Metrics

Coupling and cohesion metrics measure the interactions and the degree of integration between the software modules. Cohesion is the measure of the extent to which a set of modules depends on each other while coupling measures the extent of the relation between the components of a particular module. A high degree of coupling normally leads to a low degree of cohesion and high system complexity. The coupling factor, LCOM, and size of the class interface are the most used measures to quantify these aspects. [20]. Studies have shown that high coupling and low cohesion are associated with increased defect rates and maintenance difficulties [10].

D. Function Point Analysis

Function Point Analysis (FPA) is used in identifying the size of the software based on the functionality delivered to the users. This method computes complexity in terms of function points that are deduced from the users' needs and expectations. FPA offers a uniform method to measure the intricacy of one project to another and one software system to another [21]. It has been widely employed to estimate development effort and project size, and to predict defect rates [22].

4. Impact of Software Complexity on Defect Resolution Time

The level of complexity of the software significantly influences the amount of time needed to fix bugs in software systems. There is a correlation between higher complexity levels and an increase in the difficulty of discovering, isolating, and repairing errors, which can lead to longer time periods required for troubleshooting and resolving defects. Empirical evidence supports this connection, provided by several studies that investigated the relationship between software complexity and the amount of time it takes to resolve defects.

1. The authors Li et al. identified that more functionally complex requirements are associated with an increased number of defects and increased time to address the defects. It was also assumed that reducing the amount of functional complexity could enhance the quality of software and reduce the time for defects' elimination [23].
2. Smith and Huang proved that interface growth results in increased work for defect repair and maintenance. This has led to the development of complex interfaces that in turn introduced additional dependencies hence making the tracking and fixing of errors more time-consuming [24].
3. Another study by Garcia et al. found that higher module complexity was linked to both an increased number of faults and longer periods required to resolve defect issues. It stressed the importance of managing module complexity to increase software quality and decrease the time needed to resolve defects [25].
4. Johnson et al. discovered that sophisticated software architectures were linked to a greater number of defects and longer times required to resolve defect issues. This study highlighted the importance of having architectures that are both well-designed and simple to promote efficient defect resolution [26].

When looking at these studies it was possible to establish that there is a positive correlation between the reduction of the software size and the defect fixing time. This is why simple software engineering practices that address complexity are required.

5. Methods to Mitigate Software Complexity and Improve Defect Resolution Time

To enhance the efficiency of addressing the software defects and the quality of the software, it is crucial to minimize the software's complexity. The following is a discussion on several approaches and measures that have been suggested to deal with

software complexity. Therefore, by applying these strategies, the software development teams can avoid the complexity of the software systems, improve the maintainability and increase the speed of defects fixing. The following sections describe several approaches to minimize software complexity and to enhance the times to fix defects, based on different references.

A. Refactoring

Refactoring is the process of restructuring the code in a way that makes the code better designed, easier to understand, and easier to maintain but does not alter the observable behavior of the code. Refactoring is a way of making a code less complex through removing code smells, reducing redundancy, and improving modularity. This makes it easier to detect and correct defects in the subsequent processes. Mens et al. note that frequent refactoring can be greatly helpful in decreasing the time spent on Defects by improving the comprehensibility of the code [27].

B. Modularization

Modularization involves a process of segmentation of the software system into smaller and more manageable modules. This approach proves useful in improving the detection and handling of defects since the system is divided into logical but relatively independent segments, which reduces the system's complexity. It is also important to note that the responsibilities of the modules and their interfaces are well defined to support good defect handling. Parnas and Siewiorek have shown that the use of the modular design technique reduces the coupling of the elements which is useful in the identification and isolation of the defects [28].

C. Design Patterns

The use of design patterns means that the solutions to frequently occurring problems in software design are already known and are used to standardize. Through design patterns, developers can come up with a well systematized code that is more manageable and freer from complications hence making it easy to fix defects when they occur. Gamma et al have demonstrated that the application of design patterns can make the design and architecture of a system less complex and consequently takes less time and effort to fix defects. [29].

D. Test-Driven Development (TDD)

The Test-Driven Development (TDD) is a method of writing tests before the actual code is written. This practice makes sure that the code fulfills the stipulated requirement and assists in the identification of errors. TDD promotes the development of small, reusable and independent components, hence the code is easily understandable and maintainable and therefore easy to debug. Beck and Andres also pointed out that the TDD results in improved quality and maintainability, which in turn decreases the time taken to correct the defects. [30].

E. Continuous Integration and Continuous Delivery (CI/CD)

Continuous Integration and Continuous Delivery are processes by which changes to the code are integrated and delivered more frequently to the production environments. CI/CD reduces complexity of build, test, and deployment, thus allowing for quick identification and fixing of defects. The integration and testing practices will enable the identification of problems at early stages and their quick resolution. Fowler also pointed out that using CI/CD practices it is easier to keep the codebase more stable and the overall system less complex, hence, defect resolution is faster [31].

6. Case Studies and Empirical Evidence

To study the connection between the complexity of software and the amount of time it takes to fix defects, several case studies and empirical investigations have been carried out. These studies offer significant insights into real-world circumstances and present empirical evidence demonstrating the impact that software complexity has on the amount of time it takes to resolve defects. Case studies and empirical studies on this topic are presented in the references that are listed below:

1. A study by Curtis et al showed that code duplication as well as excessive coupling extended the time taken to fix defects. The results indicated that the method of complex management can enhance the efficiency of fault resolution and it is more suitable for organizations without training data or newly launched projects [32].
2. Research by El Emam et al. showed that Object-Oriented (OO) design complexity measures, specifically a subset of the Chidamber and Kemerer (CK) metrics, are predictive of software problems. These metrics were found to be highly associated with faults in industry statistics from C++ and Java, even after controlling for software size. The study revealed that these metrics affect problems differently in C++ and Java samples, emphasizing their importance in creating high-quality OO software products [33].
3. An empirical assessment employing four size metrics WMC (CK), CMC (Li), CC (BS) and CCC (S&B) by Basili et al calculated the maintainability index of successive versions of the software. The study, which analyzed the changes in classes added and deleted together with the growth of the number of versions across 38 versions of JFreeChart and nine versions of three live projects, revealed that growth of complexity between the versions signified program maturity. All these metrics were found to be valid at the system level in empirical studies [18].
4. The evaluation of the software development processes- documentation, design, coding, testing and maintenance through statistical modeling has been shown to be useful in tracking software quality. This approach was described by McCabe and showed the trends of software metrics in software engineering research. Applying metrics such as McCabe and C&K for estimating software complexity leads to the enhancement of software quality and controllability of the project [3].
5. Li & Henry's historical overview and the types of software metrics highlighted how complexity metrics

impact the costs of software development and its maintenance. The paper focused on McCabe and C&K complexity measures, where it was highlighted that the measurement of the software complexity improves the quality of the software and the management of the project [34].

Despite the well-recognized impact of software complexity on defect resolution time, there are still issues to be solved and directions to be further researched in this field. Knowledge of these difficulties and possible directions for enhancement opens the way for further studies and innovations. The following references discuss the issues and give information about the prospects regarding software complexity and the time needed to address the defects:

1. Li and Henry suggested the following metrics to enhance the efficiency of software and its quality. They reviewed Chidamber and Kemerer's most widely used object-oriented software measures, which quantify class internal, inheritance, and coupling intricacies. Their study recommended a reconsideration of these metrics for reused software and contrasted the initial set of metrics to the adjusted one, claiming that the new set is valid and useful for evaluating problem resolution effort [34].
2. It is noted that communication plays a significant role in team creation and motivation and is presented as one of the major management tools. Robbins stated that communication and management skills are crucial in the management of a business, direction of teams, and the proper handling of defects. [35].
3. Predicting software faults accurately remains challenging. A study by Lessmann et al. developed software fault prediction methods using Principal Component Analysis (PCA) and Support Vector Machines (SVM). Their research, based on NASA PROMISE data, demonstrated that PCA reduces feature optimization time, and SVM provides accurate classification, thereby minimizing time and space complexity [36].
4. Intelligent systems are increasingly used to process vast amounts of data and reduce transportation accidents. A study by Liu et al. explored machine learning (ML) and artificial intelligence (AI) applications in transportation safety, identifying practices and experiences that could be transferred between transport modes to enhance safety and efficiency [37].

7. Conclusion

To sum up, another factor that is caused by software complexity is the time that is taken to resolve defects. Several research works have shown that defect resolving time increases with the complexity of the software, which poses a challenge to the software development teams. Sophisticated software products are harder to debug and identify the defects that exist in the system. The structural complexity, the functional complexity, and the interface complexity are the main factors that influence system complexity and the time that is required to fix a fault. The analyzed works focus on reducing the software complexity to enhance the rate of defect identification and fixing. Refactoring, modularization, and architectural simplest can be used to maintain and simplify the software systems. However, complexity reduction for avoiding new errors must be done prudently. Concerning the challenges and the future directions are also discussed. Measures of complexity and its standardization, roles and responsibilities of team members, and optimization of complexity and defect are some of the issues. AI and machine learning could automatically detect and handle software complexity in the future. The above challenges and prospects can be tackled to enhance the defect resolution, quality, and time to market software development teams. Software complexity management enhances the chances of fixing the defects by developers, managers as well as the end-users. Based on the literature, it can be hypothesized that software complexity is a determinant of defect resolution time. It is also evident that with the help of complexity management both software quality and the effectiveness of defect resolution can be enhanced. Future research in software complexity and defect resolution should aim at increasing the measurability of the metrics, enhancing the cooperation strategies, and employing technology.

References

[1] V. Basili and B. Perricone, "Software Errors and Complexity: An Empirical Investigation.," *Communications of the ACM*, vol. 27, pp. 42-52, 1984.

[2] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus and B. Ray, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, pp. 943-956, 1992.

[3] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, pp. 308-320, 1976.

[4] T. Ostrand and E. Weyuker, "The Distribution of Faults in a Large Industrial Software System," in *2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002.

[5] N. Fenton and S. Pfleeger, "Software Metrics: A Rigorous and Practical Approach," *PWS Pub. Co*, 1998.

[6] S. E. Fahlman, "Cognitive Architectures and the Architecture of Cognition," in *Artificial Intelligence*, 2009.

[7] J. D. Musa, "Software Reliability Data," *Handbook of Software Reliability Engineering*, pp. 469-476, 1979.

[8] M. Ohba, "Software Reliability Analysis Models," *IBM Journal of Research and Development*, pp. 428-443, 1984.

[9] S. H. Kan, Metrics and Models in Software Quality Engineering, Addison-Wesley Professional.

[10] L. Briand and J. Wüst, "Investigating the relationship between software complexity and the likelihood of errors," *IEEE Transactions on Software Engineering*, pp. 45-56, 2000.

[11] M. Xie and M. Ke, "Correlation between software complexity and software errors," *Journal of Software Maintenance and Evolution: Research and Practice*, pp. 143-161, 2003.

[12] S. Chidamber and C. Kemerer, "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering*, pp. 476-493, 1994.

[13] V. Basili and J. Selby, "The role of metrics in software maintenance," *IEEE Transactions on Software Engineering*, pp. 493-507, 1996.

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[15] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, pp. 1053-1058, 1972.

[16] M. Halstead, "Elements of Software Science," *Elsevier*, 1977.

[17] R. Hummel, T. Millstein and L. Davis, "The effects of software design complexity on defects: a study in open-source systems," in *International Conference on Software Engineering*, 2011.

[18] J. Basili and B. Boehm, "A Validation of Object-Oriented Design Metrics as Quality Indicators," in *Proceedings of the 14th International Conference on Software Engineering*, 1992.
[19] M. McCabe, "Complexity Metrics for Software Maintenance," *IEEE Transactions on Software Engineering*, pp. 430-439, 1983.
[20] S. Henry and W. Li, "Empirical studies of software maintenance using software metrics," *IEEE Transactions on Software Engineering*, pp. 434-443, 1997.
[21] A. Albrecht, "Measuring application development productivity," *Proceedings of the IBM Applications Development Symposium*, pp. 83-92, 1979.
[22] L. Jones, "Software Defect and Maintainability Prediction Using Function Points," *IEEE Transactions on Software Engineering*, pp. 412-420, 1995.
[23] Y. Li, R. Colomo-Palacios, R. M.J and A. Ruíz-López, "Functional Complexity and Its Impact on Software Defects," *IEEE Transactions on Software Engineering*, pp. 245-258, 2019.
[24] J. Smith and X. Huang, "Interface Complexity and its Effect on Defect Resolution," *Journal of Systems and Software*, pp. 45-54, 2019.
[25] M. Garcia, P. Laplante and L. Williams, "Module Complexity and Defect Resolution in Software Systems," *IEEE Software*, pp. 30-37, 2019.
[26] D. Johnson, R. Sharma and T. Jones, "Architectural Complexity and Software Defects," *Software Quality Journal*, pp. 101-114, 2020.
[27] T. Mens, T. Tourwé and M. Wermelinger, "Software Refactoring as a Process of Software Evolution," *IEEE Transactions on Software Engineering*, pp. 85-103, 2004.
[28] D. Parnas and D. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Communications of the ACM*, pp. 401-408, 1975.
[29] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.
[30] K. Beck and C. Andres, Test-Driven Development: By Example, Addison-Wesley Professional, 2003.
[31] M. Fowler, Continuous Integration, ThoughtWorks, 2006.
[32] B. Curtis, H. Krasner and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, pp. 1268-1287, 1988.
[33] K. El Emam, S. Benlarbi, N. Goel and W. Melo, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, pp. 630-650, 2001.
[34] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software*, pp. 111-122, 1993.
[35] S. Robbins, Organizational Behavior, Prentice Hall, 2001.
[36] S. Lessmann, B. Baesens, C. Mues and S. Pietsch, "Benchmarki Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Transactions on Software Engineering*, pp. 485-496, 2008.
[37] Z. Liu, Y. Zhang and X. Chen, "Intelligent Transportation Systems: Applications of AI and Machine Learning," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1510-1521, 2020.