Automated Software Maintainability Assessment for SOFL Formal Specifications

A thesis submitted in partial fulfillment for the
degree of Master of Informatics and Data Science

Yu Du

Under the supervision of
Professor Shaoying Liu

Dependable Systems Laboratory,
Informatics and Data Science Program,
Graduate School of Advanced Science and Engineering,
Hiroshima University, Higashi-Hiroshima, Japan

January 2024

Abstract

Software quality is directly impacted by software maintainability, a vital component of the program quality model, over the whole lifecycle of a software product. Assessment and analysis of software maintainability is one of the focuses of software quality assurance efforts.

Formal specifications written in the SOFL specification language have been used in several industrial projects to help enhance software reliability. Experience suggests that specifications need to be frequently changed in different ways during the specification construction process and the specification-based implementation process. To make the changes easy to conduct without introducing potential faults, specifications must be ensured to have a high maintainability. To this end, we must first be able to assess the maintainability of formal specifications, but the problem we are facing is the lack of appropriate metrics for this purpose.

The majority of previous software maintainability research has focused on assessing it from implementation source code or project documentation rather than other artifacts such as specifications and designs. Several predictive metrics or assessment models currently exist to measure software maintainability in code. Despite the obvious application of assessing software maintainability using a variety of clear criteria, maintainability models are still challenging to implement effectively. Furthermore, the main disadvantage of such indicators is that they do not provide project developers with early feedback since they cannot be calculated until after major development work has been completed. SOFL has various special concepts that we need to factor into the maintainability measures of the specification so that maintainers can deal with them effectively.

To address this problem, we put forward new metrics for assessing the maintainability of formal specifications in this thesis. It is based on eight clearly calculable metrics: line of expressions (LOE), number of processes (NOP), number of control data flows (NOCDF), cyclomatic complexity (CC), module Halstead volume (MHV), number of data stores used (NODSU), extensiveness of comments (EOC), and extensiveness of blank lines (EOBL). Compared with metrics for code maintainability assessment, our metrics deal with the features of formal specifications that are distinct from code. Most rules in the proposed metrics can be applied to model-based formal specifications in general and only a small part is specific for the SOFL language. Secondly, we discuss the principle and the assessment rules of our metrics. Thirdly, we construct an automated assessment support tool to help efficiently evaluate large and complex projects written in the SOFL specification language. Finally, we experiment with several formal specification cases to demonstrate how the metrics work in practice and verify the effectiveness of the method and tool.

Acknowledgements

Completing this master's thesis has been a challenging, growing, and valuable journey. I would like to express my deepest gratitude to those who have played an important role in my master's academic endeavor.

First and foremost, I would like to express my heartfelt gratitude to my thesis advisor, Professor Shaoying Liu. Your guidance, expertise, and unwavering support have been instrumental in shaping the direction and quality of this research. Your spirit of keeping thinking, pursuing excellence, and passion for the research topic have always inspired me. I am very grateful to my co-supervisors, Professor Tadashi Dohi and Professor Tooru Nakanishi, as well as the teachers and fellow students of the Dependable Systems Laboratory of the Informatics and Data Science Program at Hiroshima University, for creating a stimulating academic environment. The insightful discussions, feedback, and encouragement from my peers throughout the research process have greatly enriched my understanding and perspective.

I would also like to extend a special thank you to my family and friends for their unfailing support and understanding. Your belief in my abilities and encouragement in times of doubt has been my moral support. Across the expansive world, our paths may lead us to different peaks, yet in the end, we shall meet at the summit of our individual aspirations. Your love and companionship provide me with the motivation for my studies so that I can stand at the crossroads of my life, be firm on the path I am facing, and walk on it earnestly. When the wind rises, echo your call once more, run with the wind, and pursue the longings of your heart. For this, I am deeply grateful.

Finally, I would like to thank Hiroshima University for creating an atmosphere of intellectual curiosity and academic pursuit. The pursuit of peace and knowledge, diversity of thought, and commitment to academic excellence have been an integral part of my academic journey.

Thank you all for your support.

Contents

Chapter 1

Introduction

1.1 Research Background and Purpose

Maintainability is defined as the degree of effectiveness and efficiency with which the intended maintainers can modify a product or system. Software maintenance refers to the process of making updates to software after it has been delivered to the customer. The goal is to keep the software operational while it goes through the changes that occur over its lifespan. Maintenance activities account for the majority of software lifetime costs. Nowadays, software products have become an essential component of our daily lives. Each software product is created to satisfy one or more industry or user needs. However, these needs may alter over time due to various influencing variables such as changing market circumstances or customer behavior. The costs of maintenance activities account for the majority of software lifetime costs and are a major concern for both software producers and customers. As a result, developing and designing software systems with maintainability in mind is critical. We can predict which software components will be complex, prone to errors, and difficult to modify early in the development life cycle, particularly during the specification phase. By assessing the software maintainability at the specification phase, we can also obtain earlier feedback to help improve software maintainability and thus reduce the increasingly high cost of software maintenance and upcoming expenditures in significant personnel and time for software maintenance.

It is well recognized that formal methods are a potentially effective approach to the development of safety-critical systems [1]. Particularly, formal specification can be beneficial for assisting developers in making sense of ambiguous requirements and potentially avoiding a lot of faults in the early stages. Comprehensible formal specifications can be written using a formal notation called Structured Object-Oriented Formal Language (SOFL) [2]. By promoting the idea of building a formal specification based on the experience of writing an informal specification and honing it into a semi-formal specification, SOFL offers a three-step specification approach that makes it easier to write a high-quality formal specification.

Based on previous experiences, it appears that modifications to specifications are frequently necessary in different ways for both the specification construction process and the specification-based implementation process. Specifications need to be ensured highly maintainable in order to facilitate the updates without creating potential errors.

Furthermore, the majority of software lifecycle costs are attributed to maintenance activities, which are a significant source of concern for both software developers and customers. As a result, it is crucial to develop and design software systems with maintainability in mind. It would be ideal if we can determine whether the current specification has a good maintainability in the early phase of the development life cycle, particularly during the specification phase. We can also get early feedback to help enhance software maintainability and so lower the rising cost of software maintenance and future investments in significant personnel and time for software maintenance by evaluating the software maintainability during the specification process.

1.2 Research Status

The majority of previous software maintainability research has focused on assessing it from implementation source code or project documentation rather than other artifacts such as specifications and designs. Several predictive metrics or assessment models currently exist for the measurement of software maintainability in code. Zhuo et al. [3] present and contrast seven software maintainability assessment models to investigate the use of metrics in assessing software maintainability. Ash et al. [4] describe mechanisms for automated software maintainability assessment and apply those techniques to industrial software systems. Mittal et al. [5] propose a fuzzy logic-based, precise, and easy approach to quantify the maintainability of software. They focused on the average number of live variables, the average life span of variables, the average cyclomatic complexity, and the comment ratio as the four main software-related factors. The objective of Momeni et al. [6] is to show that the Adaptive Neuro-Fuzzy Inference System (ANFIS) can more accurately predict maintainability as compared to other models such as fuzzy logic. Braeuer et al. [7] identified a reference study that may be used to validate the Consortium for IT Software Quality (IT-CISQ) approach and classified the maintainability of eight open-source Java projects. The experimental findings demonstrate that the IT-CISQ assessment method is incapable of accurately determining the quality of projects since project size metrics are not taken into account.

Effective application of maintainability models remains a challenge, despite the apparent benefits of evaluating software maintainability using several explicit indicators. Instead of analyzing the specifications directly, most existing models concentrate on determining maintainability from implementation source code or project documentation. It is hard to evaluate the maintainability of a software system using this method because it is hard to derive quantitative maintainability information from a requirement description expressed in natural language. Moreover, since these indicators cannot be computed until essential development work has been finished, their primary drawback is that they do not offer project developers early feedback. It is expensive to modify the system based on its measurements if poor software maintainability is recommended. The Maintainability Index was put forward by Oman et al. [8] to measure the maintainability of software systems objectively based on the current state of the source code. Their experiments on various software systems served as the basis for this measurement, and these findings were modified in response to input from the engineers responsible for system maintenance. The MI has been used frequently as a comparison model in later maintainability research and has been verified many times for several procedural programming languages (C, Pascal, FORTRAN, and Ada) [9]. However, because of their unique characteristics from programs, traditional software maintainability metrics are not suitable for formal specifications, although fundamental ideas and principles of program maintenance can still be utilized as a point of reference.

While they can be used to a particular extent, current work and notions on code maintainability are insufficient for assessing specification maintainability. This is mostly because the formal notation used in SOFL is very different from programming languages in many aspects. For instance, in SOFL, the functionality of a process is defined by pre- and post-conditions rather than a sequence of commands. Furthermore, SOFL has various special concepts that we need to factor into the maintainability measures of the specification so that maintainers can deal with them effectively.

In this thesis, we make three major contributions. We present a method for assessing the maintainability of SOFL formal specifications, based on the criteria of formal specifications [2] and the maintainability index model widely used in code maintainability metrics [3, 10], which provides useful assistance to developers when maintaining formal specifications. A suite of metrics has been developed to measure the maintainability of SOFL formal specifications. In addition, we developed an automated support tool that enables efficient software maintainability assessment of SOFL formal specifications. To validate the effectiveness of the proposed metrics and the automated support tool, we also conduct a case study to check the implementation of our maintainability metrics and support tool.

1.3 Structure of This Thesis

The structure of the paper is outlined as follows:

Section 2 delves into the background, exploring the relevant context of software maintenance evaluation. This section encompasses an examination of prior research and methods in the field and outlines the challenges and limitations associated with formal specifications in the realm of software maintenance. Additionally, it discusses the unique characteristics of SOFL specifications, such as condition data flow diagrams, three-step modeling approaches, and rigorous review and testing.

Section 3 discusses the software maintainability assessment methods for SOFL specifications. This is followed by a discussion of the maintainability subfactors of the specifications and an in-depth description of the SOFL characteristics and maintainability assessment methods.

Analysis and design of an automated software maintainability assessment tool are covered in Section 4. An overview of the development environments and tools utilized is introduced in this section. A requirements analysis of the support tool comes next. The general design architecture of the support tool is presented. Furthermore, Section 4.3 explores the automated tool's functional implementation, outlining the technical details of its development and the algorithms used.

Section 5 transitions into a practical application of the developed methods through a case study. This section details the application of the automated tool within these environments and presents the empirical analysis results.

Finally, Section 6 serves as the conclusion and future work. The section concludes by summarizing the research findings, delineating the contributions and limitations of the thesis, and outlining directions for future research.

Chapter 2

Research Background

This section contains a brief background of software maintainability assessment methods and the metrics, followed by a brief discussion of formal methods, formal engineering methods, the SOFL specification, and its characteristics.

2.1 Software Maintainability Assessment

2.1.1 Software Maintainability

Software maintainability is a component of overall software quality. A software quality model is defined by the ISO/IEC 25010 standard as a set of attributes including efficiency, usability, suitability, compatibility, security, reliability, portability, and maintainability. ISO/IEC 25010:2011 defines maintainability as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [11]. Software with high maintainability is easier and less expensive to keep up-to-date and to extend with new functionality. Consequently, maintainability is an essential aspect of software quality and is recognized as one of the most challenging assessments owing to the difficulty in predicting future activities [12].

And the factors in a software quality model are too abstract to be measured directly. Therefore, researchers usually hierarchize the high-level quality characteristics into lower-level quality factors, which are measured with metrics. The hierarchy of maintainability will be described in detail in Section 3.

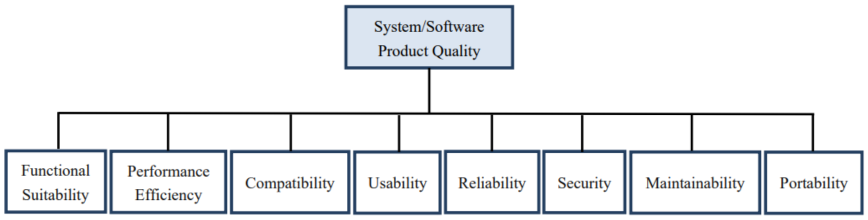


Figure 2.1: ISO/IEC 25010 SQuaRE - System and software quality model

2.1.2 Software Maintainability Assessment Metrics

Software maintainability assessment is a systematic analysis of the ease with which a software product can execute subsequent maintenance. Software maintainability could be divided into several sub-characteristics and attributes that serve as formalized maintainability indicators. Researchers have proposed various software maintainability prediction or assessment models in previous studies on software maintainability assessment in an attempt to quantify the maintainability of a software system. The Maintainability Index(MI) is the most extensively used metric for quantifying software maintainability based on the current state of the source code. The Maintainability Index was constructed at the University of Idaho by Oman and Hagemeister [8]. It is a maintainability model consisting of a variety of easily computed metrics. The MI is a polynomial expression that provides a specific value representing the overall system maintainability.

The typical MI exists in two variants, which only differ in the last component [13]:

$$MI3 = 171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g') - 16.2 \times \ln(aveLOC) \quad (2.1)$$

$$\begin{aligned} MI4 = & 171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g') - 16.2 \times \ln(aveLOC) + \\ & 50 \times \sin \sqrt{2.4 \times perCM} \end{aligned} \quad (2.2)$$

In these formulas, aveV is the average Halstead Volume per module, aveV(g') is the average extended cyclomatic complexity per module, aveLOC is the average number of lines of code per module, and perCM is the average percent of lines of comments per module.

At the module level, the components are determined and then averaged. The word "module" is used here to refer to the smallest functional unit. Depending on the programming language, this might be a function, process, method, subroutine, or section. Halstead volume is a quantifiable measure of the density of operators and operands. In other words, it indicates the number of variables and their usage.

Cyclomatic complexity is a measurable indicator of the complexity of a system's logic. It reveals the number of possible code execution pathways. The number of lines of code displays the size of the program. And a quantitative metric of human insight is the percentage of lines of comments. Comments in source code are a double-edged sword when considering their role in software maintenance. Accurate, up-to-date comments and additional insights not already apparent in the source code are usually useful when making changes to the program in the future. However, comments that are ambiguous, meaningless, and not continually updated with the development process can actually become an obstacle to the software maintenance process. Only people can judge whether comments in source code are helpful. More comments in the source code do not mean that it is easier to maintain. Therefore, expression (2.2) which contains the components of the percentage of lines of comments, should be used only when a human being judges that the comments are valid, not when blocks of program code are commented out. Otherwise, expression (2.1) should be used. The higher the MI value, the better the maintainability. With slightly various metrics, metric combinations, and weights, several MI variations have evolved. Each of them possesses the universal traits of the fundamental MI equations and principles.

$$\begin{aligned} Maintainability\ Index = &\mathrm{MAX}(0, (171 - 5.2 * \ln(HV) - \\ &0.23 * CC - 16.2 * \ln(LOC)) * 100/171) \end{aligned} \tag{2.3}$$

The expression above is the definition of the MI calculation method in Visual

Studio code analysis, where the maintainability index has been reset to be between 0 and 100. MI values are shown as follows: values above 20 indicate high software maintainability and the marker is displayed in green; values between 10-20 indicate moderate maintainability, and the marker is displayed in yellow; values below 10 indicate difficulty in maintaining the system, and the marker is displayed in red [14].

Table 2.1: MI values and display in Visual Studio code analysis

Maintainability	**MI Value**	**Marker**
High	$20 \leq MI < 100$	Green
Moderate	$10 \leq MI < 20$	Yellow
Low	$0 \leq MI < 10$	Red

2.2 Formal Methods

Formal methods are an approach based on mathematical theory that uses formal specification as well as verification to provide reliability assurance for software development. There are various languages in formal methods, such as VDM-SL, B. Method, Z, etc., and include corresponding techniques and tools, etc. [15]. Formal methods are extracting and refining the specification, using set theory and logic to describe the software specification, allowing the system functionality to be progressively clarified, resulting in a more precise and unambiguous requirements analysis of the software functionality. Since the whole process of refining requirements analysis is difficult to automate, there is a possibility of errors. Therefore, it is also necessary to model the system with formal methods to detect errors in the specification through rigorous mathematical reasoning and to verify that the design and implementation of the system satisfy the requirements. The principle of formal methods is as follows:

Formal methods are built based on basic mathematical theories, such as set theory, logic, algebra, etc. They can be used in all phases of software de-

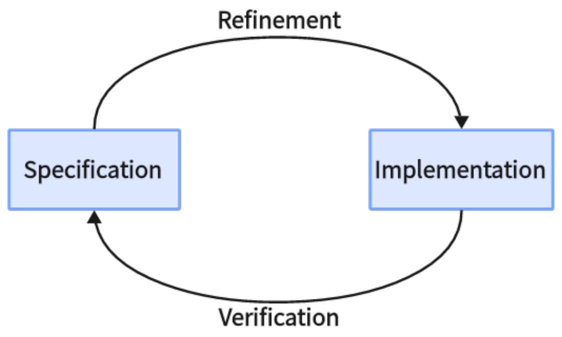


Figure 2.2: The principle of formal methods

velopment, and are one of the techniques to improve the quality of software development, with two main techniques: formal specification and formal verification.

2.3 Formal Engineering Methods

Nowadays, developing software is a large-scale complex project. The ultimate goal of software design is to describe how a set of models interact with each other clearly and concisely. Although formal methods applied in the software field can improve the quality of software and enhance the performance of software systems, its application requires a high level of mathematical abstraction and proof, which hinders its widespread use in software development. There are some disadvantages of formalization methods. For large-scale software programs, formal specifications are more difficult to read and write than informal specifications, which aggravates software complexity for developers, makes development more difficult, and reduces software readability. Formal verification techniques are not easy to master for general developers, and applying them to the development of actual software systems will increase the development cost, especially for large-scale industrial software systems or systems with high requirements for safety performance. The application of formal verification technology requires more theoretical foundation and professional skills, and it is difficult for general software product developers to meet the requirements of the application of this technology. In addition, in the system analysis and design phase, using formal methods will require more time and labor, making the development efficiency lower.

To address these shortcomings, Prof. Liu Shaoying proposed the concept of Formal Engineering Methods (FEM) at the First International Conference on Formal Engineering Methods (ICFEM) in Japan in the 1990s [16]. Formal Engineering Methods are the methods that support the application of formal methods to the development of large-scale computer systems [2]. They are a description method based on mathematical logic that is typically used to describe a complete system, analyze the behavior in the system, assist in the implementation of the system, and validate key parts of the system through meticulous and effective logical tools. Formal engineering methods are neither equivalent to the application of formal methods nor formal methods themselves. They provide a series of techniques and methods to build a bridge between formal methods and their application in real development and provide techniques to incorporate formal methods into the entire software engineering process, aiming to promote the application of formal methods in industrial software development [17].

In general, the formal engineering methodology has several features: first, the formal engineering methods use a suitable specification language, which is a suitable combination of natural language, graphical symbols, and corresponding notations. The entire framework of the system can be described in terms of graphical symbols, and the definitions corresponding to these graphical symbols can be created in terms of formal notations, while at the same time, the definitions are interpreted in terms of natural language to make them easy to understand. Integrating these three with a suitable specification language can take full advantage of both graphical symbols and formal notations.

Second, the formal engineering methods employ a stepwise evolutionary technique rather than a program refinement technique to create the specification and develop the program [17]. The point of this evolutionary technique is how to control, support, and verify changes in the specification during the software development process to meet changing system requirements.

Moreover, formal engineering methods adopt rigorous but effective techniques to inspect and verify the system such as testing, reviewing, model checking, etc., and do not use formal proofs, which saves cost, saves time and improves the efficiency of the system development, which can be used in industrial software development. Therefore, formal engineering methods for the creation and transformation of specifications, provide theoretical guidance and technical support for software system inspection and promote its practical application in the process of software product development. Structured Object-Oriented Formal Language (SOFL), as an outstanding representative of formal engineering methods, has been widely used in the field of software engineering. It is widely used in the field of software engineering.

2.4 SOFL Specification Overview

SOFL, standing for Structured Object-Oriented Formal Language, which was first started in 1989 at the University of Manchester, UK, and is both a formal method and a specification language. Through more than three decades of development and popularization, SOFL is increasingly used in software system development. As a specification language, SOFL effectively integrates traditional graphical symbols and formalized notations. As a methodology, SOFL adopts a unique three-step mechanism for building formal specifications and supports specification-based review techniques, inspection, and verification techniques. By adopting structured processes and object-oriented thinking, SOFL can fully utilize the advantages and avoid the disadvantages of both. Previous studies have shown that the SOFL approach is useful for improving software reliability and promoting the design of maintainable software systems [18], [19], [20].

In general, SOFL has the following characteristics:

2.4.1 Condition Data Flow Diagram

SOFL specification language integrates Data Flow Diagrams, Petri nets, and VDM-SL (Vienna Development Method - Specification Language). The graphical notation Data Flow Diagrams are adopted to describe comprehensibly the architecture of specifications; Petri nets are primarily used to provide operational semantics for the data flow diagrams; and VDM-SL is employed, with slight modification and extension, to precisely define the components occurring in the diagrams. A formalized Data Flow Diagram, resulting from the integration, is called Condition Data Flow Diagram, or CDFD for short.

In SOFL, a specification is composed of a set of modules. Each module has a Condition Data Flow Diagram that represents the behavior of the module and contains all the essential data, including types and variables and the processes utilized in the CDFD. Each process is specified using pre- and post-conditions. In semantics, the CDFD associated with a module describes the behavior of the module, while the module is an encapsulation of data and processes, with an overall behavior represented by its CDFD. Furthermore, the use of a natural language, such as English, is facilitated to provide comments on the formal definitions in order to improve the readability of formal specifications [2].

2.4.2 Three-Step Modelling Approach

SOFL supports a three-step approach to developing formal specifications. Typically, an informal specification is first created, then the informal evolves into a semi-formal, and finally, a formal specification is developed. Such development is an incremental process, with different tasks at each step.

The purpose of the informal specification in the first stage is to obtain abstract requirements for system functionality, usually written in natural language. The developers can communicate better with the users to obtain their requirements and clarify the problems that the system is trying to solve. In this phase, the specification consists of three parts: 1) the functional description section, which is a hierarchical structure connecting the upper and lower levels; 2) the data resources section, which shows the data information necessary to realize the system functionality, and also describes the relationships between the data: and 3) the conditional constraints section, which describes the conditions between the operations and the data resources. The second stage is the refinement of the informal specification to create a semi-formal specification. The goal of the semi-formal specification is to write the functionality, data information, constraints, etc. in the informal specification in the SOFL specification language, which will help to improve the readability and accuracy of the specification. In this phase, the process uses pre- and post-conditions, and logical equations of invariant types are still written informally.

The third stage is to convert the semi-formalized specification into a formal specification. In this phase, the CDFD is designed to represent the structure of the system, the pre-conditions, and post-conditions of the process are also written in formal expressions, and the logical expressions in the invariant types are close to being formalized. Since requirements and design specifications serve different purposes, SOFL advocates that requirements specifications should be written in a semi-formal way while design specifications must be completely formal. The obvious explanation for that is that, while design specifications serve mainly to provide a clear foundation for implementation, requirements specifications are frequently used for communication between users and developers, demanding the document's comprehensibility. Furthermore, studying the demands described in the requirements specification is necessary to develop design specifications, and formalization can be quite helpful in this process.

In conclusion, developing a formal specification using a three-step approach can assist in achieving several goals, including reducing the specification's complexity, increasing its readability, and preventing unintended errors during the evolutionary process.

2.4.3 Rigorous Review and Testing

SOFL uses rigorous review techniques for specification verification and validation. The purpose of specification verification is to detect errors in the specification. Rigorous review techniques are derived from a combination of formal proof and fault tree analysis as a method of safety analysis. Reviews must be done on a precise ground and supported by rigorous mechanisms [21]. They are usually less formal than formal proof, but easy to conduct. The use of rigorous review techniques in the software development process, especially for complex systems, will increase the efficiency of system development, reduce costs, and minimize software risks. The purpose of specification verification is to detect errors in the specification to ensure that the specification accurately meets the demands of the requirements analysis.

2.4.4 Maintenance of SOFL

In practice, maintaining formal specifications is often regarded as a challenging task. This is primarily since some maintainers may not be familiar with the formal notation used, especially SOFL, and may find the logical relationships of some of the processes to be complicated. In fact, tool support is crucial. The process of maintaining the specification will be made much easier if a tool can be used to assess modules with complex logic conditions and provide suggestions for maintaining the specification. Moreover, SOFL has various special concepts that need to be taken into account in maintainability assessment, which we will discuss in detail in Section 3.2.

Chapter 3

Software Maintainability Assessment Methods for SOFL Specifications

ISO/IEC (International Organization for Standardization/International Electrotechnical Commission) published “Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and software quality models”, ISO/IEC 25010:2011 in March 2011. This standard outlines an eight-characteristic software quality model that can be further broken down into a subset of qualities. Modifiability, reusability, analyzability, modularity, and testability are the five sub-characteristics that constitute maintainability, as illustrated in Figure 3.1. Modularity is the degree to which a system or computer program consists of discrete components, i.e., a change in one component that minimizes the impact on other components. The ability of an asset to be utilized in multiple ways or to construct other assets is known as reusability. Analyzability is the degree of effectiveness and efficiency in assessing the impact on a system of anticipated changes to one or more components of the system, diagnosing the cause of system defects or failures, or identifying components that require modification. Modifiability is the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality. Testability is the degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component, and tests can be performed to determine whether those criteria have been met [11].

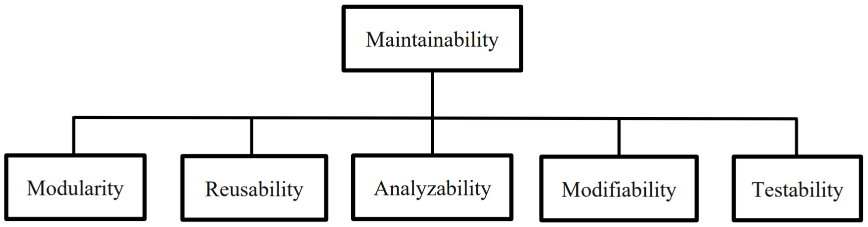


Figure 3.1: ISO/IEC 25010 SQuaRE: Sub-characteristics of maintainability

The qualities of the customer-oriented quality factor are listed in the SQuaRE model. High-level quality qualities are further broken down into several internal and external sub-characteristics because they are challenging to quantify and evaluate directly. These quantifiable quality subfactors have greater significance for software developers. Next, we can choose formal quantitative definitions of maintainability metrics that are appropriate for assessing software features that influence the quality subfactors of a certain quality attribute. In our work, which is described below, we create our SOFL maintainability metrics hierarchy using the SQuaRE model.

3.1 Maintainability Subfactors of the Specifications

Identifying the appropriate subfactors that affect software maintainability is a foundation for constructing a SOFL formal specification maintainability metric hierarchy. The first step in the software maintenance process is identifying the maintenance objects. This level of software maintainability is influenced by the modularity and analyzability of the SOFL formal specification. The most crucial elements of the SOFL specification are its modules. A hierarchy-based collection of related modules utilizes a specification. Each module is a functional abstraction: it has a behavior represented by a graphical notation, known as a CDFD and a structure to encapsulate the data and processes used in the CDFD. The current research focuses exclusively on the analyzability of a SOFL specification due to the high degree of modularity in the SOFL definition.

The analyzability of a specification is significantly impacted by its complexity because analysis of the specification necessitates a precise comprehension of it. Because modification is a common procedure during the development process, we also focus on the issue of subfactor modifiability. Furthermore, as it is vital to test the program or modified specification against the specification to verify whether the system maintains the same level of reliability as the original version, specification testability is an important component of software maintainability. Moreover, since the subfactor reusability is more concerned with how the specification, or a portion of it, might be used in another specification or project than it is with the dependability of the system that is currently in development, we have not addressed it in our study. According to the analysis of the software maintenance process, four subfactors—analyzability, complexity, modifiability, and testability—are crucial for the maintainability of SOFL formal specifications.

3.2 SOFL Features

Based on the experience gained from the literature discussed in Section 2 and our analysis of SOFL specifications, we believe that the following major features of a SOFL specification have an impact on the four maintainability subfactors:

- Total size. The size of a SOFL specification has a direct and simple impact on its complexity. In general, the more complex the specification is, the more maintenance effort is required. For example, the number of lines of expressions a module has is one of the size metrics of a SOFL module. It affects the complexity of the internal processes of the SOFL module. If a module includes a lot of expressions, it could be difficult to analyze and could affect additional testing. As a result, the SOFL formal specification's overall size has an impact on its complexity, analyzeability, and modifiability.

- Number of units. A module in a software system can be decomposed into "units". The unit is the smallest piece of code that can be executed individually. In Java or C#, a unit is a method, in C a unit is a procedure. The system components that can be the target of "unit testing" using automated unit testing frameworks such as JUnit (for Java) or NUnit (for .net) are the units of code. The SOFL definition states that a module can have more than one process. The procedure, which has inputs, outputs, and certain pre- and post-conditions, can be thought of as a unit of the formal specification. Thus, in SOFL specifications, the quantity of processes serves as a maintainability indication. This measure offers a basic approximation of cohesiveness and coupling.

- Process coupling. A metric used to quantify the level of interdependence between processes is called process coupling. To lessen the impact of process modifications, we generally think that process coupling should be maintained at a reasonable degree. The movement of data from one process to another is referred to as data flow. The SOFL formal specification distinguishes between two types of data flows: control data flow and active data flow. The primary function of an active data flow is to transfer the actual data that is expected to be used by another process, while a control data stream is usually used to describe the dependencies between processes, i.e. the requirement to execute a process after the execution of the previous one without receiving any useful data flow. This indicates that "special data" that won't be utilized by another operation is transmitted via the control data flow. Therefore, the primary determinants of process coupling complexity are these two types of data flows and the possible frequency of contact between the two processes. By describing the data flow between processes, the process coupling metric assesses the complexity, analyzability, and modifiability of the SOFL specification.

- Logical structure of conditions. When describing process functionality using pre- and post-conditions, SOFL enhances conciseness and precision. This type of specification works adequately for high-level design and requirement definition because it allows the developer to focus on the relationship between inputs and outputs rather than how that relationship should be executed. Nonetheless, an algorithmic solution utilizing sequence, choice, and iteration is typically required when deriving a comprehensive design specification from a high-level design. Therefore, the logical structure complexity of the conditions of a SOFL module is another factor affecting its maintainability. Logical structure metrics are used to measure the complexity and testability of the processes of the SOFL specifications.

- Data Usage. A variable that holds data at rest is referred to as a "data store" or "store" in the definition of SOFL formal specification. A data store is passive. Instead of actively sending any data item to any process, it always makes its value available for any related processes to read and write. The SOFL formal specification defines a data store that can be either read or written (updated) by a process, the complexity of data usage within a SOFL process is a factor affecting its analyzability and modifiability.

- Specification style. Specification style refers to the typography, naming, and comments in the specification. It is distinct from other SOFL specification features such as size and logical structure. Many previous studies have found that programming style is a factor that affects the psychological complexity of programs. In traditional software maintainability, programming style has been regarded as a significant factor affecting the comprehensibility of a program. Consequently, one crucial component of specification maintainability that should not be overlooked is specification style. Specification style is considered to be an important factor influencing specification complexity and modifiability.

The framework for assessing the maintainability of a SOFL formal specification is shown in Figure 3.2. Analyzability, complexity, modifiability, and testability are the four subfactors that compose maintainability. The major SOFL features that impact the four subfactors are subsequently determined.

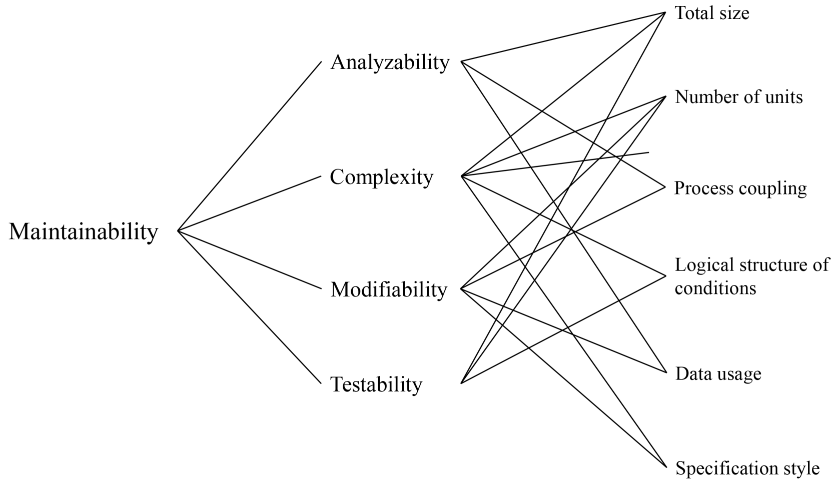


Figure 3.2: The assessment framework for the maintainability of SOFL

3.3 Maintainability Assessment Metrics for Specifications

Based on the identified SOFL features explained previously that affect the four maintainability subfactors, this thesis proposes the SOFL maintainability metric model. Lines of expressions, number of processes, number of control data flows, cyclomatic complexity, module Halstead volume, number of data stores used, extensiveness of comments, and extensiveness of blank lines are the metrics we advocate applying in this model. The developer can use this approach to identify complex and error-prone modules and components in SOFL formal specifications. This will assist in determining how the developer can enhance maintainability. We will focus our discussion below on certain measures that are included in our metric model.

3.3.1 Lines of Expressions

The lines of expressions (LOE) metric is often used to evaluate the total size of a system. The LOE metric is similar to the lines of code (LOC) metric in the program. LOC metrics have several definitions. Comment lines are included in certain LOC definitions but not in others. While some specify counting the logical lines of a program (source instruction lines ended by a logical delimiter, such as a semicolon), others define counting the physical lines of a program. We propose a basic LOE metric to evaluate the maintainability of the SOFL formal specification. This metric counts all lines of expressions that are neither comments nor blank lines. It is a measure of the number of expressions that are actually doing something. Comments and blank lines are ignored, so it gives a more accurate picture of the total size of the system.

As an example, the formal specification for the Current_Show_Balance process in an ATM system is given below. We define the seventh and eighth lines as comment lines and we can see that the number of lines with comments is 2, the number of blank lines is 0, and the number of LOE is 7.

```
1   process Current_Show_Balance(current_inf3: CustomerInf)
2                                balance: nat0
3   ext rd current_accounts
4   post balance = current_accounts(current_inf3).balance
5   explicit
6   balance := current_accounts(current_inf3).balance
7   comment
8   display the balance of the customer's current account
9   end_process;
```

Figure 3.3: Example of Current show balance process

3.3.2 Number of Processes

The number of processes (NOP) in a module determines the number of units in the SOFL specification. The more processes in a module that perform the same function, the more independent those processes are, and the easier it is to maintain the module. Fig. 3.4 displays the CDFD of the Hotel Reservation

System's Reserve module as an illustration. The example contains 3 processes, namely Check_Vacancy, Make_Reservation, and Issue_Reservation_Number.

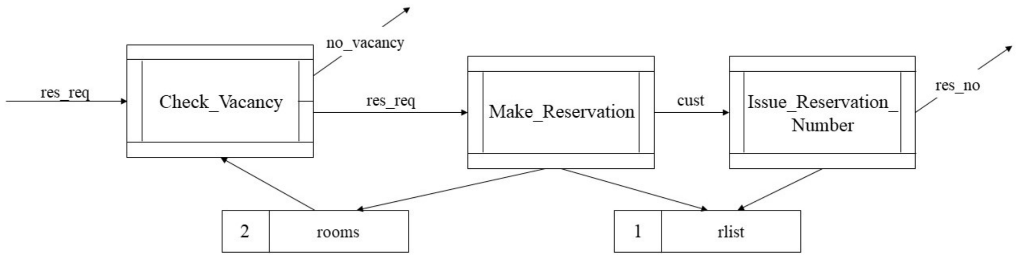


Figure 3.4: The CDFD of the Reserve module

3.3.3 Number of Control Data Flows

According to the SOFL specification, the active data flow is from the input data of one process to the output data of another process. The degree of connectedness of this data coupling is limited and unavoidable. Consequently, we do not consider the number of active data flows as a metric of the complexity of the process coupling. We consider the number of control data flows (NOCDF) in a module to be the primary factor of the process coupling complexity. In CDFD, the control data flow is represented by a dashed directed line. As a result, the number of control data flows for a module is defined as a measure of the level of process coupling for that module.

3.3.4 Cyclomatic Complexity

A pre-conditional and a post-conditional compose the conditions component in the SOFL formal specification. The necessities that the input data flows need to meet in order for the process to be carried out correctly are known as the pre-conditions. In other words, there is no guarantee of the correct output data flows if the pre-conditions are not met by the input data flows. Post-conditions show the conditions that must be satisfied by the output data flows after the process has been executed. Typically, the relationship between the input data flow and the output data flow is defined in the post-condition. As a result, it is evident how the output data flow can be produced using the input data flow. Since both are predicate expressions and the SOFL process specification has similar sequential statements in the conditions section, we propose to use McCabe's cyclomatic complexity metric [22] to measure their logical structure complexity. In addition to the fact that the cyclomatic complexity measure can be derived from the program control flow graph, McCabe observes that it is equivalent to the number of predicates plus one.

Cyclomatic complexity is a metric for software quality developed by Thomas J. McCabe Sr. in 1976 and refers to the number of linearly independent paths through a program's source code [22]. The greater the cyclomatic complexity, the greater the number of potential execution paths through the code, and the more difficult the code is to test and maintain. Mathematically, the following formula is used to calculate the cyclomatic complexity:

$$V(G) = E - N + P. \tag{3.1}$$

A flow graph notation for a program defines several nodes connected through the edges. Where E refers to the number of edges, N refers to the number of nodes, and P refers to the number of connected components. For a traditional program flow graph, we can simply find the number of edges, nodes, and connected components, but for SOFL specifications these metrics are not clearly defined. Consequently, we have considered an alternative method for assessing cyclomatic complexity for SOFL specifications.

One alternative way to measure the cyclomatic complexity of a module is to count the number of decision points, such as if-statements and while-statements. The above method would obtain the cyclomatic complexity rating by multiplying the number of decision points, or keywords, by one. Both methods yield the same result. In the SOFL specifications, we calculate an approximation of cyclomatic complexity by counting the keywords in conditional expressions, multiple choice expressions, block expressions, and while expressions. The cyclomatic complexity keywords of SOFL formal specifications include the keywords "if", "else if", "case", and "while". For each keyword in the specification, it is in-

cremented by 1. The following specification fragment consists of one decision point, thus yielding a cyclomatic complexity of 2.
if deposit_amount ≤ maximum_deposit_once
then . . .
else . . .

Because there are some operators and quantifiers in SOFL that have the same function as decision statements and contribute to the logical complexity of the conditionals. Thus, the logic structure complexity metric of each process in a module, denoted by V(G), is defined as the sum of the following add one:

- V(GK): the total number of cyclomatic complexity keywords,
- V(GQ): the total number of quantifiers provided by SOFL,
- V(GO): the total Cyclomatic Complexity of operators.

$$V(G) = V(GK) + V(GQ) + V(GO) + 1, \tag{3.2}$$

where the quantifiers provided by SOFL predicate logic include the universal quantifier "forall," and the existential quantifier "exists." Because the definition of some operators contains keywords or quantifiers, such as "subset," "get," and "power," their logic structure complexity needs to be calculated. Table 3.1 illustrates the definition of the V(GO) for the operators on sets, sequences and map types in the SOFL formal specification.

The average cyclomatic complexity per module is the definition of the cyclomatic complexity component in the MI metric. Systems constructed with object-orientation techniques generally yield low average complexity per module, meaning that these programs consistently score highly on this criterion. To determine which specification fragments are particularly difficult to maintain, we computed the metric for cyclomatic complexity in the SOFL formal specification maintainability assessment as the sum of the cyclomatic complexity of all the processes in the module.

Table 3.1: The V(GO) for the operators on sets, sequences and map types in SOFL

Operator	**Name**	**V(GO)**	**Operator**	**Name**	**V(GO)**
inset	Membership	0	hd	Head	1
notin	Non-membership	0	tl	Tail	0
card	Cardinality	0	elems	Elements	1
subset	Subset	1	inds	Indexes	1
psubset	Proper subset	1	conc	Concatenation	2
get	Member access	1	dconc	Distributed concatenation	1
union	Union	0	dom	Domain	1
inter	Intersection	0	rng	Range	0
diff	Difference	0	domrt	Domain restriction to	1
dunion	Distributed union	0	rngrt	Range restriction to	1
dinter	Distributed intersection	0	domrb	Domain restriction by	1
power	Power set	1	rngrb	Range restriction by	1
len	Length	0	inverse	Map inverse	1
s	Sequence application	1	override	Override	2
s(i, j)	Subsequence	1	comp	Map composition	2

3.3.5 Module Halstead Volume

Halstead complexity measures (HCM) is a series of metrics known as Software Science and was first presented by Halstead in 1977 [23]. It serves in the construction of software to identify several quantitative laws. Halstead complexity uses a set of basic metrics that are often calculated after the design phase of the program is complete. Halstead believed that this statistic was indicative of the size of any algorithmic implementation and also counted the number of mental comparisons required to develop a program. However, Halstead just gives a brief example without more discussion in his book rather than stating specifically what should be regarded as operators and what should be considered operands. In general, the most common types of operators include reserved words, function calls, mathematical operators, and related separators, among others. Operands can be identifiers like variables and constants. When assessing the maintainability of the SOFL specification, we compute the Module Holstead volume as:

$$\mathrm{MHV} = 10.4 \times \ln\left(\mathrm{N} \times \log 2(\mathrm{n})\right), \tag{3.3}$$

where n is the number of different operators indicating the specification vocabulary and N is the total number of operators appearing in the module indicating the specification length. Operators contain reserved words, function calls, mathematical operators, and can also include related separators, constants and variables, etc. Table 3.2 displays the operators in the SOFL formal specification.

Table 3.2: Operators in the SOFL formal specification

Numeric types\Character types\Enumeration types		**Boolean types**	**Set types**		**Sequence and string types**
+	mod	and	inset	diff	len
−	**	or	notin	dinter	s
*	<	not	card	power	hd
/	>	=>	subset		tl
abs	<=	<=>	psubset		elems
floor	>=		get		inds
div	=		union		conc
rem	<>		inter		dconc

The lexical complexity of the formal specification is revealed in the Module Holstead Volume. It functions as a metric for evaluating the logical structure of conditions and specification style of SOFL formal specifications.

3.3.6 Number of Data Stores Used

The data usage metrics are used to capture the amount of data input to, processed in, and output from a SOFL process. According to the SOFL formal definition, data store or store is a variable holding data at rest. A data store is passive. It does not actively send any data item to any process, but rather always makes its value available for reading and writing by any associated processes. A data store can be either read or written (updated) by a process, which is represented by a directed line pointing to the process from the store or pointing to the store from the process in CDFD. The currently used maintainability measures were determined to be inadequate. Given the growing usage of databases these days, it is imperative to give database access the same consideration. We suggested that the SOFL formal specification maintainability metrics be expanded to include the number of data stores used metrics that indicate database accesses in order to address this deficiency. The higher the value of this metric, the lower the maintainability of the specification. This is because a high value denotes a more frequently accessed database by the module and a more complex system, making it more challenging to modify the specification without creating new defects.

3.3.7 Extensiveness of Comments

In general, we can determine the characteristics of two specification style features. The first feature is computed by the software analyzer and is expressed quantitatively. The "human-assessable characteristics" are the other characteristic. We simply take into account aspects that can be evaluated by a computer and that affect the maintainability of the specification. Since features that can be evaluated by humans necessitate expert subjective opinions that are not accessible to quantitative expression. The extensiveness of comments (EOC) determines the comment extension feature. The EOC expressed in percentage for a module is further defined as the number of lines with comments divided by the number of total lines of expressions (including lines with comments and blank lines) in a SOFL specification.

$$EOC = 100 \times \frac{Lines\ with\ Comments}{Total\ Lines\ of\ Expressions}. \tag{3.4}$$

3.3.8 Extensiveness of Blank Lines

Typographic features are another aspect we take into account when characterizing specification style, similar to comment features. The following are the proposed metrics along with their definitions.

Extensiveness of blank lines (EOBL). The EOBL expressed in percentage for a module is further defined as the number of blank lines divided by the number of total lines of expressions (including lines with comments and blank lines) in a SOFL specification.

$$EOBL = 100 \times \frac{Blank\ Lines}{Total\ Lines\ of\ Expressions}. \tag{3.5}$$

3.3.9 Maintainability Metrics Hierarchy

The maintainability metrics hierarchy for a formal specification written in SOFL is summarized as shown in Figure 3.5.

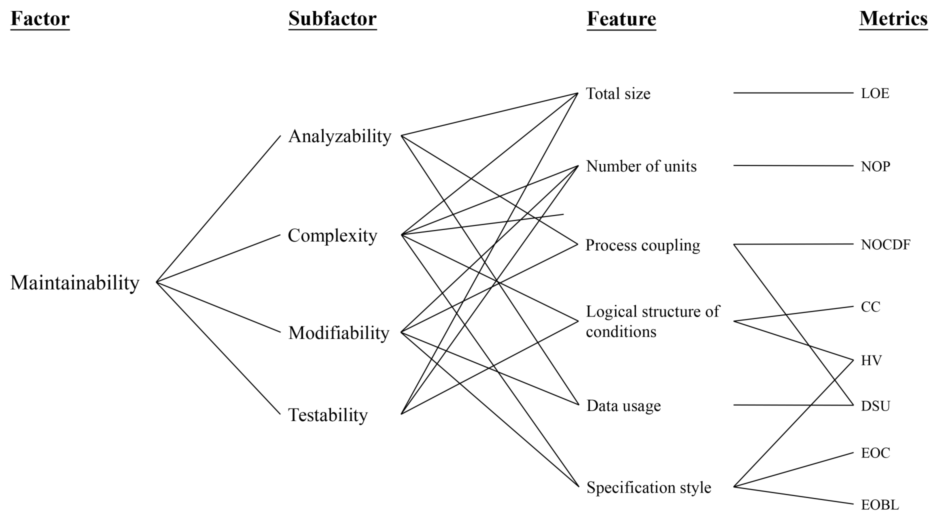


Figure 3.5: SOFL formal specifications maintainability metrics hierarchy

Chapter 4

Analysis and Design of Automated Support Tool

We have built a prototype tool, called SOFL-ASMAT (Automated Software Maintainability Assessment Tool for SOFL formal Specification), to support our research methodology for software maintainability assessment. The support tool was designed using the SOFL method and implemented based on WPF and C# languages. This section will analyze the key information that needs to be extracted from the requirements of the support tool, identifying valuable information and defining the scope of the effort of the tool. After that, the overall design architecture of the support tool will be presented. The implementation of metrics assessment methodologies in the support tool, along with the design of the tool's functionality for evaluating software maintainability based on SOFL features, will be the main areas of focus. Finally, the design of the processing approaches for software maintainability ratings of the support tool will be described in detail.

4.1 Requirements Analysis of the Support Tool

Requirements analysis is a clear description of the functional and performance aspects of a software system that should be met before the software product is designed and implemented. Only with clear requirements on the functional and performance aspects of the system can a compliant software product be accomplished at the time of implementation. The research objective of this paper is to apply the software maintainability assessment method mentioned in the previous section to automate the assessment of software maintainability of the SOFL formal specification, and upon completion, to rate and analyze each software maintainability metric of the specification as a whole and for each module, and to export the generated maintainability rating results. Maintainability metrics assessment is an important module of the tool to maintain consistency between requirements analysis and design, improve software quality, and increase reliability. With its excellent practicality, this support tool may significantly increase the user's efficiency while assessing the software maintainability of the SOFL formal specification. The results generated in the support tool also provide the basis for research into formalized module-based specification maintainability assessment and other related SOFL technology explorations. To accomplish the entire application of software maintainability assessment metrics, there are the following functional requirements:

- SOFL formal specification analysis. Users can import SOFL specification files through the interface or menu. Text file formats for SOFL specification files are supported by the tool. The tool can parse the SOFL specification file to extract key information, including but not limited to modules, processes, expressions, etc. At the same time, for documents that do not meet the SOFL specification, the tool will provide appropriate error messages and processing mechanisms.

- SOFL formal specification maintainability assessment calculations. The tool supports the calculation of maintainability assessment metrics for each module, including the following eight metrics: line of expressions (LOE), number of processes (NOP), number of control data flows (NOCDF), cyclomatic complexity (CC), module Halstead volume (MH), module size (MH), module size (MH), module size (MH), module size (MH), and so on. , module Halstead volume (MHV), number of data stores used

(NODSU), extensiveness of comments (EOC), and extensiveness of blank lines (EOBL). For every module, the support tool offers precise computed values and rating outcomes. The tool should generate an overall maintenance assessment for the system by synthesizing the maintenance assessments for each module of the import specification in addition to computing the metrics for each module.

- The generation of a visual report for the SOFL formal specification maintainability assessment. To visualize the characteristics of the system and the assessment findings, the tool will generate a chart based on the results of the maintainability assessment. Additionally, it offers a data table that includes the raw data and computation results for each module's maintenance assessment results. Users can export the data table to a text file via a menu or button so that they can retain and review their analysis results again.

The automated software maintainability assessment support tool for SOFL specification needs to fulfill not only the most basic functional requirements but also the performance requirements. Functional requirements describe the functions of each part to be accomplished by the support tool, and performance requirements describe the operational characteristics of the system from the following aspects:

To guarantee a positive user experience, the tool should react to user actions on time. The tool should precisely calculate and rate the specification maintainability because it is a crucial support tool for the growth of SOFL formal applications. Furthermore, to be able to handle the processing of large SOFL specifications to guarantee that there is no speed bottleneck in the analysis, the system's fault tolerance and reliability need to be improved in order to ensure that the maintainability assessment methods are applied appropriately.

4.2 Architecture of the Support Tool

Based on the previous requirements analysis, a structural division of the automated software maintainability assessment tool for SOFL formal specifications is derived. Three major modules comprise the support tool. These are the SOFL file preprocessing module, the main function module, and the visualization result generation and output module. The application begins from the start menu. Users can interact with the software and initiate various tasks using its main interface. The main function of the SOFL file preprocessing module is to preprocess the text file of the SOFL formal specification. This involves traversing through the specification file, extracting essential information from such module processes, and constructing the specification structure tree. It can visualize the hierarchy and relationship of each process in the SOFL formal specification. The main function module is the core function module responsible for processing the application. The calculation logic for the metrics specified in the application is handled and carried out by this module. It takes data out of the SOFL formal specification and runs the appropriate calculations to figure out the metric values and assign a rating to all of them. The display pages are then refreshed with the computed ratings and values. The visualization results generation and output module are responsible for generating visual analysis charts based on calculated data and predefined thresholds for each metric. Users will find it easier to analyze and comprehend the analysis results according to the charts, which demonstrate the information visually. Additionally, it generates a comprehensive data table that users can export for further comparative maintainability study.

The structure chart of the entire automated software maintainability assessment tool for SOFL formal specification is shown in Figure 4.1.

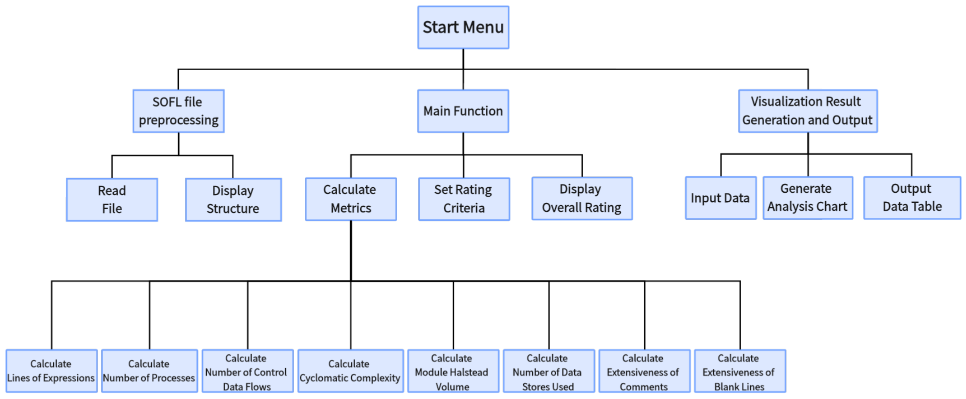

Start Menu
SOFL file preprocessing
Main Function
Visualization Result Generation and Output
Read File
Display Structure
Calculate Metrics
Set Rating Criteria
Display Overall Rating
Input Data
Generate Analysis Chart
Output Data Table
Calculate Lines of Expressions
Calculate Number of Processes
Calculate Number of Control Data Flows
Calculate Cyclomatic Complexity
Calculate Module Halstead Volume
Calculate Number of Data Stores Used
Calculate Extensiveness of Comments
Calculate Extensiveness of Blank Lines

Figure 4.1: The structure chart of SOFL-ASMAT

The programming language for this support tool is C#, and the development environment is VS2022. C#, as an object-oriented programming language derived from C and C++ by Microsoft, is simple, stable, and safe. At the same time, it bears a strong resemblance to Java, thus programmers of all stripes can quickly and simply become proficient in this language. The effectiveness and usability of the tool also played a role in the decision to adopt C# as the programming language. Windows Presentation Foundation (WPF) is a framework for creating Windows applications. Comprehensive user interface design, including 2D and 3D visuals, animations, styles, templates, and more, is supported by WPF's powerful UI technology. It facilitates the creation of interactive, highly visual apps by developers. WPF provides a powerful data binding mechanism that allows UI elements to be directly connected to data sources for synchronization and automatic updating. That improves development efficiency and streamlines data handling and presentation. Furthermore, WPF utilizes XAML as the declarative language for the user interface, which facilitates the separation of the logic and design of the UI. WPF allows developers to use styles and templates to customize the look and feel of the application and provides a flexible layout framework that enables programmers to create intricate layout structures that satisfy the requirements of SOFL support tool design by utilizing a range of panels (such as Grid, Stack Panel, etc.). In terms of integration, WPF can readily interact with other .NET components and services(such as ASP.NET, Windows Services, etc.). WPF also promotes the Model-View-View Model (MVVM) architecture, which divides an application's logic, data, and interface. This allows developers to better organize the code and enhances testability and maintainability. All things considered, SOFL-ASMAT chose WPF because it offers a powerful and flexible toolkit that lets programmers create stunning and effective applications for Windows.

The organization chart of the application in Figure 4.2 shows the page hierarchy in the support tool. The index is the top-level page which serves as the application's entrance point. Level 2 menus include the specification menu and the function menu, which offer options related to different functions and specifications. Level 3 is divided into three sections: Overview Rating, Maintainability Assessment, and Analysis Report. These pages, which are subsections of the function menu, provide an overview of the ratings or scores assigned to the module for maintainability assessment. The analysis report that is produced based on the evaluation findings is shown. The measures page and the summary page contribute to Level 4. These pages are subsections of the maintainability assessment page that offer comprehensive metrics associated with the assessment.

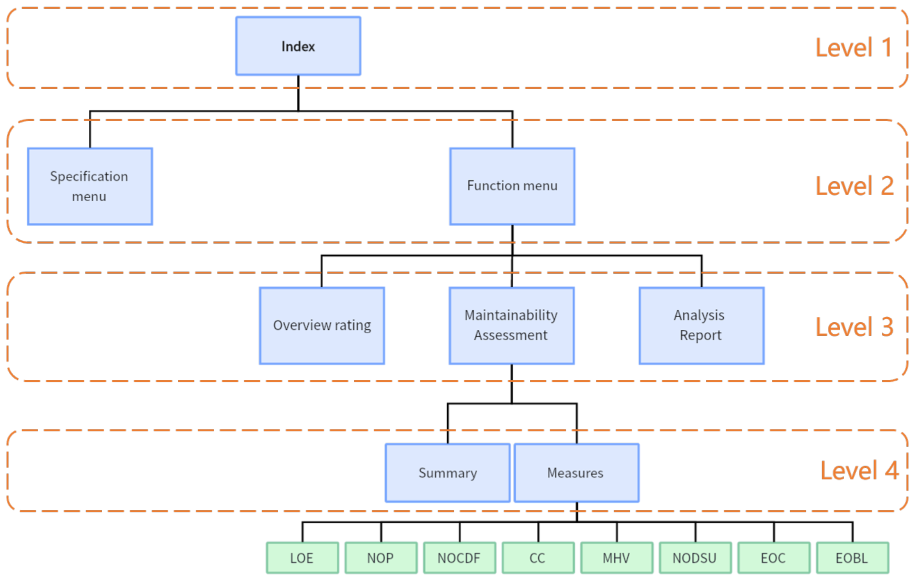


Figure 4.2: The organization chart of SOFL-ASMAT

The interface of the SOFL-ASMAT index page is illustrated in Figure 4.3, which provides a visual representation of how the pages are arranged and connected. The SOFL formal specification file's structure and the hierarchical structure containing its modules and processes are displayed in the specification menu, which is located on the left side of the main screen. Using the top menu bar or interface buttons, the user can import the formal specification file for parsing through the function menu, which occupies a large portion of the screen. For the benefit of new users in becoming acquainted with how to use the tool quickly, the top menu bar of the interface also includes a user guide and an introduction to maintainability metrics.

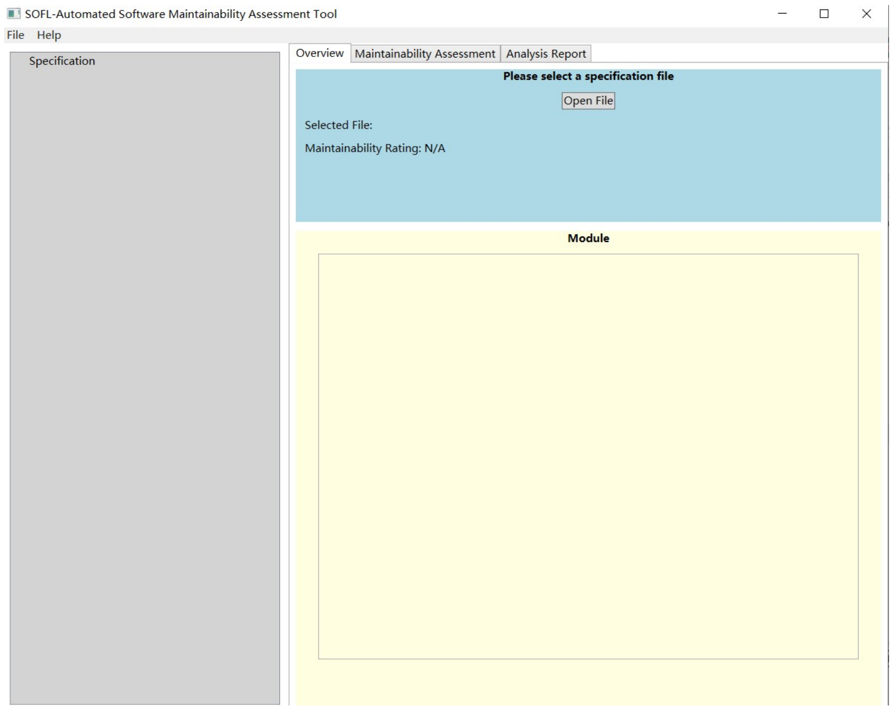

SOFL-Automated Software Maintainability Assessment Tool
File
Help
Specification
Overview
Maintainability Assessment
Analysis Report
Please select a specification file
Open File
Selected File:
Maintainability Rating: N/A
Module

Figure 4.3: The interface of SOFL-ASMAT index page

4.3 Functional Implementation of the Support Tool

In order to read the information to be processed from the SOFL formal specification file, we need to extract the names of the modules and processes, obtain the specification's general structure, and calculate the precise values of each maintainability metric that will be recorded in the list.

4.3.1 Implementation of the Specification Key Information Parsing Method

In a SOFL formal specification, a module is a document written in the SOFL language that encapsulates the data and processes that appear in the CDFD associated with the module and defines the components that appear in the CDFD. In general, a module has the structure depicted in Figure 4.4:

module *ModuleName / ParentModuleName*;
const *ConstantDeclearation*;
type *TypeDeclearation*;
var *VariableDeclearation*;
inv *TypeandStateInvariants*;
behave *CDFD_no*;
InitializationProcess;
process_1;
process_2;
…
process_n;

Function_1;
Function_2;
…
Function_m;
end_module

Figure 4.4: Module structure

The name of the module is used after the keyword "module" to indicate where a module starts. If the module is a submodule of a procedure in another module, the module name should be followed by the "/" identifier, and then the name of the parent module. Constant declarations are denoted by the keyword "const", type declarations by the keyword "type," and variable declarations by the keyword "var". The CDFD related to the module can be found by using the keyword "behav". Some operations and functions are defined by the keywords "process" and "function," which are used to identify the relevant process part and function part respectively. The end of the module is indicated with the keyword "end_module". The specific implementation of identifying the module and its name in C# is shown in the following code:

```
List<ModuleMetrics> moduleMetricsList = new List<ModuleMetrics>();
// Convert content in the specification to lowercase for easy
identification
string fileContent = File.ReadAllText(filePath).ToLower();
string[] moduleFragments = fileContent.Split(new string[] { " module " },
StringSplitOptions.RemoveEmptyEntries)
                        .Select(fragment => "module " + fragment)
                        .ToArray();
//Since the identified fragments are the content after the
```

```
    keyword,
add "module" to each fragment

foreach (var moduleText in moduleFragments)
{
    int endIndex = moduleText.IndexOf("end_module");
    if (endIndex != -1)
    {
        string moduleContent = moduleText.Substring(0,
            endIndex + "end_module".Length);
        string[] lines = moduleContent.Split(new string[] {
            Environment.NewLine }, StringSplitOptions.
            RemoveEmptyEntries);

        // Check if the module content contains "module" to
            ensure that only valid module fragments are
            processed
        if (moduleContent.Contains("module "))
        {

            if (lines.Length > 0)
            {
                // Extract module name
                string firstLine = lines[0].Trim();

                // Identifies the content after "module " as
                    the module name
string moduleName = firstLine.Substring("module ".Length);

                // Handling cases where the module name is a
                    submodule in another module
                int slashIndex = moduleName.IndexOf('/');

                if (slashIndex != -1)
                {
                    moduleName = moduleName.Substring(0,
                        slashIndex).Trim();
                }

                // Handling module names ending in a semicolon
                if (moduleName.EndsWith(";"))
                {
                    moduleName = moduleName.Substring(0,
                        moduleName.Length - 1).Trim();
                }
                ModuleMetrics moduleMetrics =
                    CalculateModuleMetrics(moduleContent);
                moduleMetrics.ModuleName = moduleName;
                moduleMetricsList.Add(moduleMetrics);
            }
        }
    }
}
return moduleMetricsList;
```

One of the most essential components of a module is the process. A process performs an action, task, or operation that takes input and produces output [2]. The typical structure of a process is shown in Figure 4.5:

process *ProcessName(input) output*
ext *ExternalVariables*
pre *PreCondition*
post *PostCondition*
decom *LowerLevelModuleName*
explicit *ExplicitSpecification*
comment *InformalExplanation*
end_process

Figure 4.5: Process structure

The process name and the input and output flows appear following the keyword "process," which indicates the beginning of the process. The keyword "ext" designates external variables that are required for the process. The submodules related to a multi-level process's lower-level CDFDs are written following the keyword "decom." Comments are written after the keyword "comment" to improve the readability of the specification. The general execution flow of the process is that the input and output data flows are first examined, and when the input data flows are available and the output data flows are unavailable, then the input data flows can be accepted and prepared for execution. Accept the input data flow, check whether it meets the pre-conditions, and meet the conditions of the process. At last, the process is carried out by producing the output data flow and checking to see if it satisfies the post-conditions. If it does, the process is considered valid. Similar to the module identification method, the implementation of identifying the process and its name in C# is shown in the following code:

```
    else if (lowercaseLine.StartsWith("process "))
{
    string processLine = lowercaseLine.Substring("process ".
        Length);

    if (processLine.Contains("("))
    {
        // If parentheses are included, the original logic is
            followed
        int openingParenthesisIndex = processLine.IndexOf('(')
            ;
        string processName = processLine.Substring(0,
            openingParenthesisIndex).Trim();

        Process process = new Process();
        process.Name = processName;
        processes.Add(process);

        // Get the last module added and add the process to
            the module's Processes list
        if (modules.Count > 0)
        {
            Module lastModule = modules.Last();
            lastModule.Processes.Add(process);
        }

        // Create a ModuleProcessData object and set the
            process name
        tableData.Add(new ModuleProcessData
        {
            ProcessName = processName,
        });
        nopValue++;
    }
    else
    {
        // If no parentheses are included, the entire line is
            treated as processName
        string processName = processLine.Trim();

        Process process = new Process();
        process.Name = processName;
        processes.Add(process);

        // Get the last module added and add the process to
            the module's Processes list
        if (modules.Count > 0)
        {
            Module lastModule = modules.Last();
            lastModule.Processes.Add(process);
        }

        // Create a ModuleProcessData object and set the
            process name
        tableData.Add(new ModuleProcessData
        {
            ProcessName = processName,
        });
        nopValue++;
    }
}
```

4.3.2 Implementation of Lines of Expressions (LOE) Calculation Method

This section details the implementation of the Lines of Expressions (LOE) calculation method in the context of SOFL formal specifications. The total size and complexity of a software system are critical factors in determining its maintainability. LOE is an essential metric for assessing the overall size and complexity of a software system represented using SOFL language. Developers can understand and grasp the scope of the SOFL formal specifications by utilizing the LOE metric, which offers information on the total number of lines with meaningful expressions. With the exclusion of comments and blank lines, the algorithm described here aims to count the lines that contain genuine expressions precisely. Figure 4.6 displays the primary components of the LOE algorithm's calculation:

```
Input: fileLines : An array of strings containing the lines of a SOFL module fragment
       trimmedLine : Trim the line to remove leading and trailing spaces
Output: totalLines : The number of expression lines in the SOFL module fragment
totalLines = 0
for each line in fileLines do
    trimmedLine = line.Trim()
    if trimmedLine is not empty then
        if rimmedLine does not start with //, /*, or comment then
            totalLines = totalLines + 1
        end
    end
end
return totalLines
```

Figure 4.6: The Lines of Expressions (LOE) calculation method

The algorithm counts the number of expression lines by iterating through each line of the file, checking to see if it is an expression line (non-commented and non-empty), and accumulating counters. It ensures accurate counting by excluding comments and blank lines, focusing on the lines contributing to the logical structure of the SOFL formal specification. The LOE calculation method provides a quantitative measure of the number of lines containing actual expressions in the SOFL formal specifications. This metric aids developers in comprehending the scale of the specification system, contributing to better-informed decisions on system maintenance and improvement.

4.3.3 Implementation of Number of Processes (NOP) Calculation Method

Number of Processes (NOP) is a crucial metric reflecting the number of processes within a module, which is indicative of the module's independence and complexity. The algorithm is implemented in C# and is triggered when a line in the SOFL specification starts with "process." The process identification part of this algorithm in performing key information extraction has been demonstrated. The name of the identified process is used in the program for constructing a process object. The process will be added to the global list of processes. The identified process will be added to the process list of the last module. For storing the process name, a ModuleProcessData object is created. Following that, the NOP value that was determined is increased.

4.3.4 Implementation of Number of Control Data Flows (NOCDF) Calculation Method

The SOFL definition states that the requirement that a process be carried out following the completion of its preceding process without requiring the receipt of any useful data flow is typically described by the control data flow. Each control data flow variable needs to be declared using a unique type identified as the signal, which is represented by the sign symbol. The exclamation mark , which indicates that there is only one value in this type, and serves as a signal to specify the associated control data flow variable. That is, a data flow variable of type sign is defined if it is bound to the value ; otherwise, the variable is undefined. Formally,

sign = !

There is no operator on this type. It is important to note that type sign cannot be used to declare any active data flow variables. In the SOFL formal specification, we can thus identify inputs of type sign and compute the number of control data flows. Number of Control Data Flows (NOCDF) is an essential metric for assessing the complexity of process coupling within a module of SOFL formal specifications. It measures the number of control data flows. Figure 4.7 keeps track of how many times the ': sign' pattern appears in every line of the SOFL specification.

```
Input: fileLines : An array of strings containing the lines of a SOFL module fragment
       signCountInLine : The number of times a signal type appears in a one-line expression
Output: totalSignCount : The number of control data flows in the SOFL module fragment
totalSignCount = 0
for each line in fileLines do
    signCountInLine = Counts the occurrences of the ': sign' pattern
    totalSignCount += signCountInLine
end
return totalSignCount
```

Figure 4.7: The Number of Control Data Flows calculation method

4.3.5 Implementation of Cyclomatic Complexity (CC) Calculation Method

Cyclomatic Complexity (CC) is a metric employed to gauge the logical structure complexity of software processes. In the context of SOFL formal specifications, it considers keywords, quantifiers, and operators to evaluate the complexity of conditions. Figure 4.8 shows the main components of the CC calculation algorithm.

```
Input: fileLines : An array of strings containing the lines of a SOFL module fragment
       trimmedLine : Trim the line to remove leading and trailing spaces
Output: ccValue : The cyclomatic complexity in the SOFL module fragment
Initialize keywordCount, quantifierCount, operatorComplexity
List<string> keywords = new List<string> { "if", "else", "while","case"}
for each line in fileLines do
    trimmedLine = line.Trim()
    if trimmedLine is not empty then
        for each keyword in keywords do
            if trimmedLine.Contains(keyword) then
                keywordCount ++
            end
        end
    if trimmedLine.Contains(quantifier) then
        quantifierCount ++
end
operatorComplexity += CalculateOperatorComplexity(trimmedLine)
ccValue = keywordCount + quantifierCount + operatorComplexity + 1
return ccValue
end
```

Figure 4.8: The Cyclomatic Complexity calculation method

According to the computational method presented in subsection 3.3.4, counters for keyword occurrences, quantifier occurrences, and operator complexity are first initialized. Then keyword and quantifier identification is performed by traversing each line in the SOFL formal specification and checking for the presence of predefined keywords and quantifiers. For each non-empty line, a separate CalculateOperatorComplexity method is used to calculate the operator complexity. The relevant operator complexity has been defined in Section 3 and finally the ccValue is calculated according to the cyclomatic complexity formula.

4.3.6 Implementation of Module Halstead Volume (MHV) Calculation Method

Module Halstead Volume (MHV), derived from Halstead Complexity Measures, is employed to evaluate the lexical complexity of SOFL formal specifications. The Module Halstead Volume (MHV) serves as a vital metric in software maintainability, offering valuable insights into the size and complexity of the vocabulary within SOFL specifications. The MHV algorithm provides a quantitative measure of the logical structure of SOFL formal specifications. It is a valuable tool for developers to assess and enhance the maintainability of their software by identifying areas of potential complexity within the formal specifications. Figure 4.9 shows the main components of the MHV calculation algorithm.

```
Input: fileLines : An array of strings containing the lines of a SOFL module fragment
       words : The content of the file is split into individual words
Output: MHV : The Module Halstead Volume in the SOFL module fragment
List<string> words = SplitIntoWords(filelines);
n = CalculateDistinctKeywordsCount(words)
N = CalculateTotalKeywordsCount(words)
MHV = 10.4 * Math.Log(N * Math.Log(2, n))
    if N > 0 and n > 0 then
        return MHV
    end
    else
        return 0
    end
```

Figure 4.9: The Module Halstead Volume calculation method

The MHV calculation begins by splitting the read SOFL formal specification into individual words using the SplitIntoWords method, taking into account spaces and common punctuation marks. Then the number of distinct keywords (n) and the total number of occurrences of all keywords (N) are calculated using the CalculateDistinctKeywordsCount and CalculateTotalKeywordsCount specialized methods. The quantitative values are then calculated using the formula for MHV defined in Section 3. Finally, exceptions are made for cases where N or n may be invalid.

4.3.7 Implementation of Number of Data Stores Used (NODSU) Calculation Method

According to the SOFL definitions, there are two ways to connect a store to a process: read and write. Each method denotes a distinct means for the process to access the data in the store. The data store that is connected to a process is regarded as an external variable by users. In actuality, it is a state variable of the related module whose CDFD contains the store. The keywords rd or wr must be used to correctly identify the access method when providing a formal specification of a process that accesses a store. Rd is an abbreviation for ”read” and wr is an abbreviation for ”write”. When a data store variable is designated as external to the process, the process can read all or a portion of its value, but it won’t be modified while the process is executing. A store variable declared as external to wr means that the variable may be updated during process execution, and it does not eliminate the possibility of the process reading from the variable. In the Number of Data Stores Used (NODSU) algorithm, we identify ”rd” and ”wr” to obtain the amount of data input, processed, and output in the SOFL process.

```
Input: fileLines : An array of strings containing the lines of a SOFL module fragment
Output: nodsuValue : The Number of Data Stores Used in the SOFL module fragment
Initialize wrCount, rdCount
for each line in fileLines do
    wrOccurrences = CountSubstringOccurrences(line, "wr")
    wrCount += wrOccurrences
    rdOccurrences = CountSubstringOccurrences(line, "rd");
    rdCount += rdOccurrences
end
nodsuValue  = wrCount + rdCount
return nodsuValue
```

Figure 4.10: The Number of Data Stores Used calculation method

A quantitative measure of data store utilization in the SOFL process is offered by the NODSU algorithm. Developers can obtain a more nuanced view of data flow in the SOFL specification by employing the CountSubstringOccurrences method to count the occurrences of writes and reads. This approach helps developers understand the interactions between processes and data stores.

4.3.8 Implementation of Extensiveness of Comments (EOC) and Extensiveness of Blank Lines (EOBL) Calculation Methods

The EOC and EOBL calculation involves similar steps to the Calculating LOE algorithm. In the EOC algorithm, the total lines of expressions and the number of lines with comments are first determined. The percentage of lines with comments to the total total lines of expressions is calculated according to the definitions in Section 3. In the EOBL algorithm, first, determine the total lines of expressions and the number of blank lines. Calculate the percentage of blank lines to the total lines of expressions based on the definitions in Section 3. Finally, format the results by rounding the percentage to two decimal places for readability.

The algorithms for EOC and EOBL provide developers with valuable metrics for evaluating the documentation and visual structure of SOFL specifications. These metrics contribute to a more nuanced understanding of the qualitative aspects of SOFL formal specifications, facilitating improved readability and maintainability.

4.4 Rating Methodology of the Support Tool

This section outlines the methodology for rating various maintainability metrics within SOFL formal specifications. The evaluation incorporates a Weighted Z-score approach to provide a nuanced rating, considering the diverse units of each metric.

Z-score, also known as a standard score, is a measure that represents the deviation of a variable's value from its mean divided by the standard deviation. It is used to determine the relative position of a data point within a set of data. Z-scores are also frequently used in student educational assessments for grading, determining percentile ranks, and quantifying quality assessments, among other purposes.

The Z-score is calculated as

$$Z = \frac{X - \mu}{\sigma}. \tag{4.1}$$

In which, X is the metric value, μ is the mean of the set of metric values, and σ is the standard deviation of the set of metric values. The raw values were in different units in the different metrics. Z-scores are all in the same unit, that is, standard deviation [24]. Z-scores tell us how much the variable's values are above or below the mean and are expressed in standard deviation. A Z-score above 1 is farther from the mean than a Z-score below 1. Therefore, rating based on Z-scores identifies outliers in the data.

However, we can observe that judging ratings by Z-scores alone is inaccurate for SOFL formal specifications. When the size of the specification is small, the values of complexity, MHV, NODSU, etc. will also be small. This module performs comparatively better in terms of maintainability. At this moment, it is further from the average value. According to our previous definition, the bigger the Z-score, the lower the rating. To address this problem, the Weighted Z-score (Zw) is calculated as follows

$$Zw = \frac{X - \left(\frac{LOE_i}{\mathrm{avg}(LOE)}\right) \times \mu}{\sigma}. \tag{4.2}$$

Where X is the metric value, LOEi is the Lines of Expressions value for this module, avg(LOE) is the average LOE value in the entire specification, μ is the mean of the set of the specific metric values, σ is the standard deviation of the set of specific metric values. Note that since the LOE metrics represent the total size of the specification in the SOFL, this weighting model is used here for metrics in the specification other than the LOE metrics. For the LOE metric itself, we use the value of the NOPi divided by the average of the NOP metrics as the weights for the calculation of Zw. Table 4.1 illustrates the classification of differences using the absolute value of the weighted Z-score.

Table 4.1: The classification of differences using weighted Z-score

Value	**Degree**	**Grade**
$\lvert Zw \rvert < 1$	Small difference	A
$1 \leq \lvert Zw \rvert < 2$	Moderate difference	B
$2 \leq \lvert Zw \rvert < 3$	Large difference	C
$\lvert Zw \rvert \geq 3$	Very large difference	D

The Maintainability Metrics Rating Algorithm is implemented in C# and outlines the following key components: First, the CalculateOverallStatistics method is used to calculate weighted Z-scores for various maintainability metrics. The CalculateStandardDeviation method is called to calculate the standard deviation of a given metric. The GetGrade method is used to assign grades based on the magnitude of the weighted Z-scores and assign values to the grades, and finally, the GetOverallGrade method is used to obtain an overall grade for the SOFL formal specification based on the assigned grades. By following these steps, the support tool not only grades the different metrics for each module but also obtains an overall grade for the maintainability of the SOFL formal specification. The Weighted Z-score approach ensures a fair and comprehensive evaluation of maintainability metrics in SOFL specifications. This method provides a robust basis for identifying modules that significantly deviate from the average, allowing developers to prioritize areas requiring attention.

Chapter 5

Case Study

To exemplify the practical application of these metrics and the support tool, we present a case study involving the assessment and rating of the software maintainability of an Automated Teller Machine (ATM) system. The ATM system was selected as a case study mainly because of its business effect, critical nature, and availability of the SOFL formal specification. The specification defines the following six functions: authentication, operations on current accounts, operations on savings accounts, money transfer between accounts, operations on foreign currency accounts, and changing a password [25]. The example consists 6 modules, in which Manage_Current_Account_Decom, Manage_Savings_Account_Decom, Manage_Transfer_Decom, Manage_Foreign_Currency_Account, Change_Password_Decom are decompositions of the high-level processes defined in the top-level module SYSTEM_ATM. The toplevel Control and Data Flow Diagram (CDFD) of the ATM system is depicted in Figure 5.1.

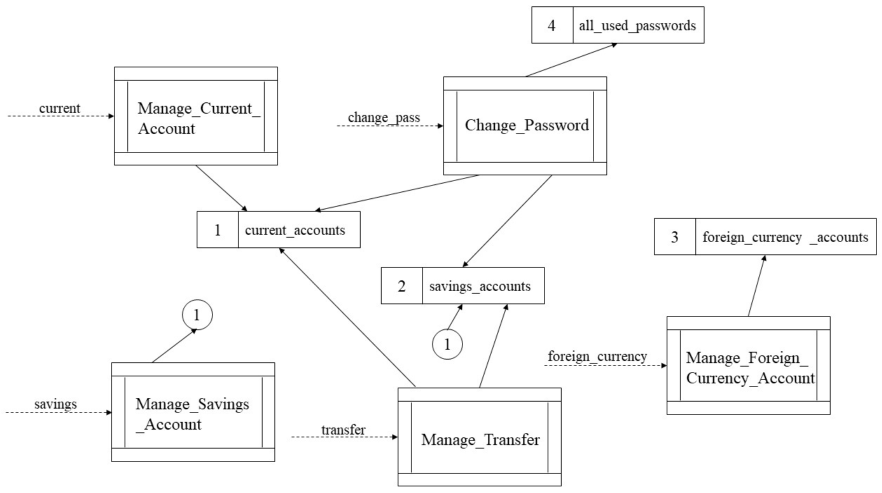
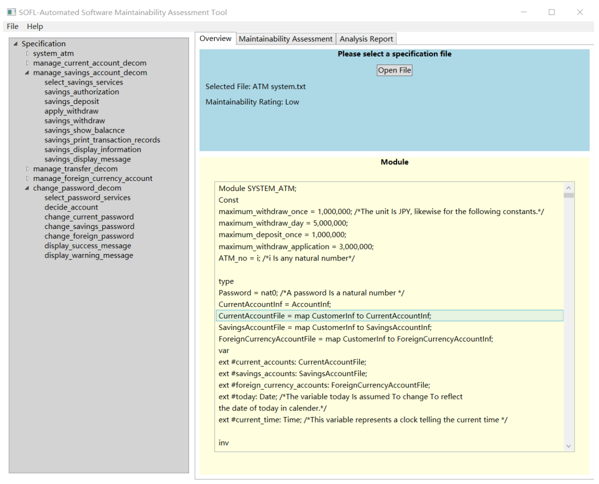


Figure 5.1: Top Level CDFD of the ATM system

Our tool guides the user through a series of simple actions that make up the assessment process. The user accesses the ”Overview” page in the first stage, where the designated specification file is available for analysis. By selecting ”File” from the top menu, the user can import the specification file. ”Help” offers a user guide and an overview of the software maintainability metrics assessment. After that, the user can carry out the specification analysis by following the instructions provided in the user guide. On the overview page, users can directly click the ’Open File’ button to import the ATM system specification in .txt format. The analysis starts with this easy yet important step. After importing the specification, the tool performs a maintainability calculation and generates an overview rating based on the complexity of the system, using the algorithm described in Section 4 of this thesis. This initial rating provides the user with an immediate understanding of the complexity of the system specification. The overall structure of the specification is well displayed on the left side of the tool interface. Figure 5.2 displays the main page that appears after the user imports the ATM system specification file.



Selected File: ATM system.txt

Maintainability Rating: Low

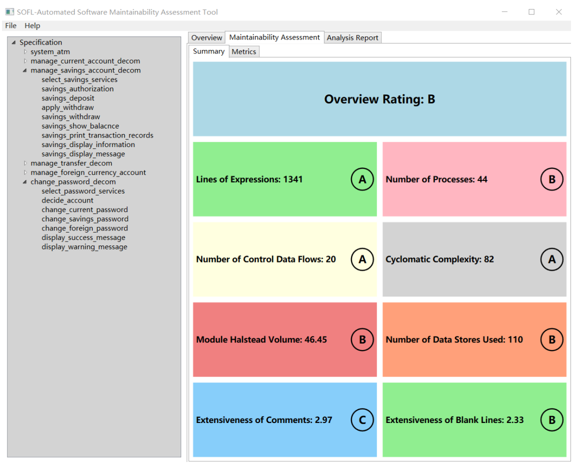
```
Module SYSTEM_ATM;
Const
maximum_withdraw_once = 1,000,000; /*The unit Is JPY, likewise for the following constants.*/
maximum_withdraw_day = 5,000,000;
maximum_deposit_once = 1,000,000;
maximum_withdraw_application = 3,000,000;
ATM_no = i; /*i Is any natural number*/

type
Password = nat0; /*A password Is a natural number */
CurrentAccountInf = AccountInf;
CurrentAccountFile = map CustomerInf to CurrentAccountInf;
SavingsAccountFile = map CustomerInf to SavingsAccountInf;
ForeignCurrencyAccountFile = map CustomerInf to ForeignCurrencyAccountInf;
var
ext #current_accounts: CurrentAccountFile;
ext #savings_accounts: SavingsAccountFile;
ext #foreign_currency_accounts: ForeignCurrencyAccountFile;
ext #today: Date; /*The variable today Is assumed To change To reflect
the date of today in calender.*/
ext #current_time: Time; /*This variable represents a clock telling the current time */

inv
```

Figure 5.2: A snapshot of the Overview page

In the Maintainability Assessment section, two crucial pages unfold the various layers of the assessment process, the Summary page and the Metrics page. The Summary page displays the composite value of each metric in the specification. The tool utilizes weighted Z-scores to calculate an average rating based on the module assessment. This process gives the user a nuanced view of the overall maintainability of the system. Figure 5.3 shows the summary page of the support tool.



Overview Rating: B

Lines of Expressions: 1341 A

Number of Processes: 44 B

Number of Control Data Flows: 20 A

Cyclomatic Complexity: 82 A

Module Halstead Volume: 46.45 B

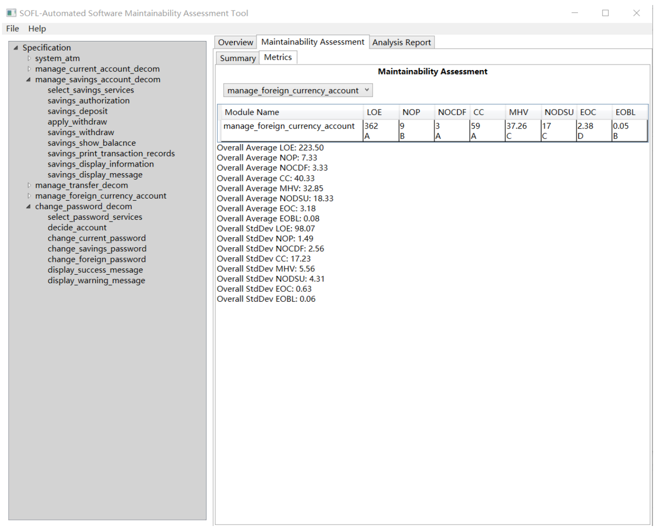
Number of Data Stores Used: 110 B

Extensiveness of Comments: 2.97 C

Extensiveness of Blank Lines: 2.33 B

Figure 5.3: A snapshot of the Summary page

Access to the Metrics page allows users to drill down into the specifics of each module. Here the user can select a specific module and view detailed values and ratings for each corresponding metric. The exact values of the mean and standard deviation for the entire specification file are also displayed below the table in the selection module, which facilitates user comparison and analysis. The component takes a synthesized approach, incorporating weighted Z-scores for each metric to provide granular insight into the maintainability of individual modules. Figure 5.4 shows the metrics page of the support tool.



Maintainability Assessment

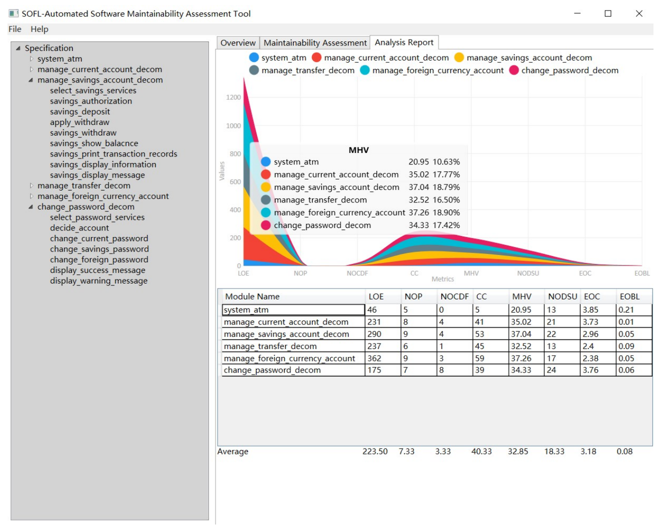
manage_foreign_currency_account

Module Name	LOE	NOP	NOCDF	CC	MHV	NODSU	EOC	EOBL
manage_foreign_currency_account	362 A	9 B	3 A	59 A	37.26 C	17 C	2.38 D	0.05 B

```
Overall Average LOE: 223.50
Overall Average NOP: 7.33
Overall Average NOCDF: 3.33
Overall Average CC: 40.33
Overall Average MHV: 32.85
Overall Average NODSU: 18.33
Overall Average EOC: 3.18
Overall Average EOBL: 0.08
Overall StdDev LOE: 98.07
Overall StdDev NOP: 1.49
Overall StdDev NOCDF: 2.56
Overall StdDev CC: 17.23
Overall StdDev MHV: 5.56
Overall StdDev NODSU: 4.31
Overall StdDev EOC: 0.63
Overall StdDev EOBL: 0.06
```

Figure 5.4: A snapshot of the Metrics page

The Analysis Report section utilizes integrated LiveCharts to effectively visualize data. Visually appealing charts are displayed to users, who can hover over individual data points to see particular metric values and their relative proportions to the entire system. Figure 5.5 shows the Analysis Report page of the support tool.


SOFL-Automated Software Maintainability Assessment Tool
File
Help
Overview
Maintainability Assessment
Analysis Report
Values
Metrics
LOE
NOP
NOCDF
CC
MHV
NODSU
EOC
EOBL

Specification
- system_atm
- manage_current_account_decom
- manage_savings_account_decom
  - select_savings_services
  - savings_authorization
  - savings_deposit
  - apply_withdraw
  - savings_withdraw
  - savings_show_balacnce
  - savings_print_transaction_records
  - savings_display_information
  - savings_display_message
- manage_transfer_decom
- manage_foreign_currency_account
- change_password_decom
  - select_password_services
  - decide_account
  - change_current_password
  - change_savings_password
  - change_foreign_password
  - display_success_message
  - display_warning_message

● system_atm ● manage_current_account_decom ● manage_savings_account_decom ● manage_transfer_decom ● manage_foreign_currency_account ● change_password_decom

MHV

● system_atm	20.95	10.63%
● manage_current_account_decom	35.02	17.77%
● manage_savings_account_decom	37.04	18.79%
● manage_transfer_decom	32.52	16.50%
● manage_foreign_currency_account	37.26	18.90%
● change_password_decom	34.33	17.42%

Module Name	LOE	NOP	NOCDF	CC	MHV	NODSU	EOC	EOBL
system_atm	46	5	0	5	20.95	13	3.85	0.21
manage_current_account_decom	231	8	4	41	35.02	21	3.73	0.01
manage_savings_account_decom	290	9	4	53	37.04	22	2.96	0.05
manage_transfer_decom	237	6	1	45	32.52	13	2.4	0.09
manage_foreign_currency_account	362	9	3	59	37.26	17	2.38	0.05
change_password_decom	175	7	8	39	34.33	24	3.76	0.06
Average	223.50	7.33	3.33	40.33	32.85	18.33	3.18	0.08

Figure 5.5: A snapshot of the Analysis Report page

Additionally, we have included a "Save" function, considering that documentation and comparison are necessary. The "Save Analysis Table" option in the "File" menu bar allows users to save the table with the findings of the maintainability assessment. This makes it easier to compare the results with the updated specification document. Table 5.1 shows a summary of the maintainability assessment in this case study.

Table 5.1: Maintainability Assessment Summary of the ATM system

No.	Module	LOE	NOP	NOCDF	CC	MHV	NODSU	EOC	EOBL
1	SYSTEM_ATM	46	5	0	5	20.95	13	3.85	0.21
		B	C	A	A	C	C	D	D
2	Manage_Current_Account_Decom	231	8	4	41	35.02	21	3.73	0.01
		A	A	A	A	A	A	A	B
3	Manage_Savings_Account_Decom	290	9	4	53	37.04	22	2.96	0.05
		A	A	A	A	B	A	B	A
4	Manage_Transfer_Decom	237	6	1	45	35.52	13	2.4	0.09
		A	B	A	A	A	B	B	A
5	Manage_Foreign_Currency_Account	362	9	3	59	37.26	17	2.38	0.05
		A	B	A	A	C	C	D	B
6	Change_Password_Decom	175	7	8	39	34.33	24	3.76	0.06
		A	A	C	A	B	C	C	A

A case study demonstrates the benefit of our metrics and tools for software maintainability assessment. Our method offers a way of comparing the maintainability of the measured object at the process, module, and system levels. Furthermore, our support tool provides a useful resource for areas of system improvement and user decision-making. The key factors influencing the maintainability of SOFL formal specifications have been covered by the SOFL maintainability metrics along with measurement features given in this work. However, our method is sufficiently flexible to enable users to measure relevant metrics and add additional characteristics to the model-based formal specifications. Additionally, the supporting tool makes it easier to apply the idea more broadly.

Chapter 6

Conclusion and Future Work

This thesis has delved into the crucial realm of software maintainability within the context of formal specifications, particularly emphasizing the SOFL formal specifications. Recognizing the importance of maintainability, we have proposed novel metrics and a comprehensive framework to assess and enhance the maintainability of SOFL specifications. And developed an automated software maintainability assessment tool to provide application support for our approach. The developed framework offers an organized method for maintaining processes during specification-based implementation, with analyzability, complexity, modifiability, and testability encapsulated as key subfactors. We calculate the maintainability metrics for each module utilizing a hierarchical dimensional assessment approach, and then we assess the entire maintainability of the specification. Through a hierarchical dimensional assessment, we've demonstrated the applicability of our metrics and support tool, showcasing their effectiveness in a real-world case study. The result shows that our approach addresses the need for specific metrics for assessing maintainability during the specification phase. In this research, we assigned individual relative grades to a range of metrics. These grades were assigned values to obtain an overall aggregated maintainability rating. This rating aims to holistically weigh individual metrics, offering a nuanced and accurate assessment of maintainability. This will help developers and organizations gain a better understanding of the maintainability status of the systems and take appropriate measures to improve and enhance maintainability. The automated software maintainability assessment tool for SOFL formal specifications also helps developers conduct assessments more efficiently and visualize the results.

We are still fully committed to moving the field forward. Our immediate focus is on validating the effectiveness of our approach through systematic evaluations with large-scale SOFL formal specifications. This step is pivotal in ensuring the broader applicability and robustness of our method in diverse software development scenarios. The next goal of our research is that by aggregating and weighing these maintainability metrics, it is possible to trace the modules in the specification that have low maintainability and provide targeted recommendations for improving maintainability for problematic specification fragments. This will help developers and organizations to better understand the maintainability status of their systems and take appropriate measures to improve and enhance maintainability.

Regarding the complexity of industrial projects and the growing magnitude of formal specifications, we have emphasized the requirement for automated assessment tools. The suggested tool will enable the application of software maintainability assessment to a larger range of domains and will not be restricted to SOFL specifications. Assessors will be able to rapidly and thoroughly assess a system's maintainability by using automated techniques. This aligns with our desire to broaden the scope of applicability and extend these methods to model-based formal specifications.

Bibliography

[1] E. S. Grant and S. P. Nanda, "A Review of Applications of Formal Specification in Safety-Critical System Development," Proceedings of the 2020 the 4th International Conference on Compute and Data Analysis, pp. 208–215, Mar. 2020.

[2] S. Liu, "Formal Engineering for Industrial Software Development – an Introduction to the SOFL Specification Language and Method," Lecture Notes in Computer Science, vol. 3308, pp. 7–8, Jan. 2004.

[3] F. Zhuo, B. Lowther, P. Oman, and J. Hagemeister, "Constructing and Testing Software Maintainability Assessment Models," [1993] Proceedings First International Software Metrics Symposium, pp. 61–70, 1993.

[4] Ash, Alderete, Yao, Oman, and Lowtber, "Using Software Maintainability Models to Track Code Health," Proceedings International Conference on Software Maintenance ICSM-94, 1994.

[5] H. Mittal and P. Bhatia, "Software Maintainability Assessment Based on Fuzzy Logic Technique," ACM SIGSOFT Software Engineering Notes, vol. 34, no. 3, p. 1, May 2009.

[6] H. Momeni and S. Zahedian, "Aspect-oriented Software Maintainability Assessment Using Adaptive Neuro Fuzzy Inference System (anfis)," Journal of Mathematics and Computer Science, vol. 12, no. 03, pp. 243–252, Oct. 2014.

[7] J. Braeuer, R. Ploesch, and M. Saft, “Measuring Maintainability of OO-Software - Validating the IT-CISQ Quality Model,” Advances in Intelligent Systems and Computing, vol. 511, pp. 283–301, Dec. 2016.

[8] P. Oman and J. Hagemeister, “Construction and Testing of Polynomials Predicting Software Maintainability,” Journal of Systems and Software, vol. 24, no. 3, pp. 251–266, Mar. 1994.

[9] F. R. Oppedijk, “Comparison of the SIG Maintainability Model and the Maintainability Index,” University of Amsterdam, 2008.

[10] M. Anan, H. Saiedian, and J. Ryoo, “An architecture-centric Software Maintainability Assessment Using Information Theory,” Journal of Software Maintenance and Evolution: Research and Practice, vol. 21, no. 1, pp. 1–18, Jan. 2009.

[11] ISO/IEC, ISO/IEC 25010 Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE). 2011.

[12] M. Dagpinar and J. H. Jahnke, “Predicting Maintainability with object-oriented Metrics -an Empirical Comparison,” Working Conference on Reverse Engineering, pp. 155–164, Nov. 2003.

[13] E. VanDoren, K. Sciences, and C. Springs, “Maintainability Index Technique for Measuring Program Maintainability,” Software Tech Review SEI, 2002.

[14] kexugit, “Maintainability Index Range and Meaning,” learn.microsoft.com, Nov. 20, 2007.

[15] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, “SOFL: a formal engineering methodology for industrial applications,” IEEE Transactions on Software Engineering, vol. 24, no. 1, pp. 24–45, 1998.

[16] M. G. Hinchey and S. Liu, “Formal Engineering Methods: Proc.,” 1st International Conference on Formal Engineering Methods (ICFEM’97), 1997.

[17] S. Liu, Formal Engineering for Industrial Software Development. Springer Science & Business Media, 2013.

[18] J. Luo, S. Liu, Y. Wang, and T. Zhou, “Applying SOFL to a Railway Interlocking System in Industry,” Structured Object-Oriented Formal Language and Method, vol. 10189, pp. 160–177, 2017.

[19] J. Li, S. Liu, A. Liu, and R. Huang, “Knowledge Graph Construction for SOFL Formal Specifications,” International Journal of Software Engineering and Knowledge Engineering, vol. 32, no. 04, pp. 605–644, Apr. 2022.

[20] Y. Xia and S. Liu, “A Framework of Formal Specification-Based Data Generation for Deep Neural Networks,” Proceedings of the 2023 12th International Conference on Software and Computer Applications, Association for Computing Machinery, Feb. 2023.

[21] S. Liu, “A rigorous approach to reviewing formal specifications,” 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings., pp. 75–81, Mar. 2004.

[22] T. J. McCabe, “A Complexity Measure,” IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[23] M. H. Halstead, Elements of Software Science (Operating and Programming Systems series). Elsevier Science Inc., 1977.

[24] C. Andrade, “Z Scores, Standard Scores, and Composite Test Scores Explained,” Indian Journal of Psychological Medicine, vol. 43, no. 6, p. 025371762110465, Oct. 2021.

[25] Y. Du and S. Liu, “Maintainability Assessment for SOFL Formal Specifications,” 2023 10th International Conference on Dependable Systems and Their Applications (DSA), pp. 680–687, Aug. 2023.