On the Measurement of Software Complexity for PLC Industrial Control Systems using TIQVA

Adnan Muslija, Eduard Enoiu,
Email: muslija.adnan@gmail.com, eduard.enoiu@mdh.se
Mälardalen University, Västerås, Sweden.

ABSTRACT

In the safety-critical domain (e.g. transportation, nuclear, aerospace and automotive), large-scale embedded systems implemented using Programmable Logic Controllers (PLCs) are widely used to provide supervisory control. Software complexity metrics, such as code size and cyclomatic complexity, have been used in the software engineering community for predicting quality metrics such as maintainability, bug proneness and robustness. However, since there is no available approach and tool support for measuring software complexity of PLC programs, we developed a tool called TIQVA in an effort to measure complexity for this type of software. We show how to measure different software complexity metrics such as lines of code, cyclomatic complexity, and information flow for a popular PLC programming language named Function Block Diagram (FBD). We evaluate the tool using data provided by Bombardier Transportation from a Train Control Management System (TCMS). In addition, we report some empirical and industrial evidence showing how TIQVA can be used to provide some experimental evidence to support the use of these metrics to estimate testing effort for an industrial control software. The results from this evaluation indicate that other specific dimensions of PLC programs (e.g., function block relationships, block coupling and timing) could be used to improve the measurement of complexity for industrial embedded software.

1 INTRODUCTION

Industrial control software is a type of software typically used in industries such as transportation, chemical, automotive, and aerospace to provide supervisory and regulatory control. This type of software has different characteristics [39] that differ from traditional software. Programmable Logic Controllers (PLCs) [10] are computer devices used for controlling industrial equipment and they are often the primary components in smaller control systems used to provide operational process control of such systems as trains, car assembly lines and power plants. The semantics of software running on a PLC [20] is characterized by the execution in a cyclic loop where each cycle contains three phases, read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs). In addition, inputs and outputs in a PLC program correspond to internal signals, sensors, or actuators. IEC 61131-3 is a popular programming language standard for PLCs used in industry because of its simple textual and graphical notations and its digital circuit-like nature. As shown in Figure 1, blocks in an IEC 61131-3 program can be represented in a Function Block Diagram (FBD). These diagrams form the basis for composing applications. There is a need for tools measuring complexity for FBD programs. Software engineering studies and textbooks often used software complexity metrics to predict extra-functional metrics such as faults proneness and maintainability (e.g., [9, 17, 36]). However, no tools and studies have looked at how code complexity of FBD industrial control software is measured.

This paper presents TIQVA, a tool for measuring the complexity of industrial control software written in the FBD programming language. In addition, we present the results of applying this tool to an industrial embedded software project from Bombardier Transportation Sweden AB. Data used in this study was created by experienced engineers for an industrial safety-critical system already in use. Even if there are many aspects of a tool evaluation that can be taken into account, we present here an experimental evaluation to gather information that will help define problems and suggest hypotheses. For example, one might expect that more, more complex, and larger requirements would increase software measures as well as testing effort.

The paper makes the following contributions to this kind of investigations:

- A tool for measuring code complexity for embedded software written in IEC 61131-3 Function Block Diagram (FBD), a popular programming language in the safety critical domain. There is a need to investigate how existing software complexity metrics can be tailored to FBD software.
- Empirical and industrial evidence showing the applicability of this tool for measuring the complexity of an industrial system using real-world PLC software.
- A discussion on the use of TIQVA, as well as the use of a linear regression model to estimate the test effort using software complexity measured by TIQVA are shown. In addition, the empirical results suggest that there is a moderate correlation between software complexity and the number of test cases created by experienced industrial engineers as well as the execution time of these test cases.

2 BACKGROUND

In this section, we explain the concepts necessary for understanding TIQVA and how the tool is used to obtain the results by covering software complexity metrics and industrial control software. The

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373914

```
    if (a > b) {
        sum = a + c;
} else {
        sum = b + c;
}
```

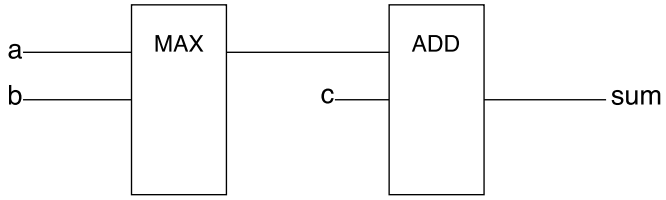

Figure 1: Two equivalent simplified programs written in two different program languages: a Java example (left) and a PLC program written using the FBD language (right).

work presented in this paper focuses on the measurement of software complexity for industrial embedded software written in the FBD language.

2.1 Industrial Control Software

In specific domains (e.g. transportation, nuclear, aerospace and automotive), embedded systems implemented using Programmable Logic Controllers (PLCs) are widely used to provide supervisory control [11]. For example, this supervisory control can be used for opening and closing doors or controlling the temperature in a furnace. PLC software differentiate from its general-purpose counterpart in several ways, including the way that they are written and tested. PLC programs are usually created using one of the IEC 61131-3 programming languages [20]. IEC 61131-3 is a international standard that describes the programming language rules and requirements used for creating PLC programs [20]. IEC 61131-3 has a number of programming language implementations: *Structure Text* (ST), *Instruction List* (IL), *Ladder Diagram* (LD), *Function Block Diagram* (FBD). Two of these languages, FBD and LD, are graphical programming languages and do not use a textual source code notation. Since the IEC 61131-3 programming languages are used in domain-specific applications, the resulting software is organized and operates using Program Organization Units (POUs) [20] containing functions (i.e., procedure-like program code), function blocks (i.e., stateful functions) and a top-level program code that has access to the IO ports. FBDs contain variables, data types, functions. However, conditional statements and loops are implemented differently in FBDs. As shown in Figure 1, the IF statement is encapsulated in the MAX function. In this study we used programs developed in the IEC 61131-3 standard and FBD programming language by industrial engineers describing a safety-critical system used in the train domain.

2.2 Software Complexity

A software complexity metric is a quantitative value that describes a certain dimension of the software and depends on the type of the artifact used for measurement [30]. Even if multiple software dimensions can be used, it is not easy to use such measures on multiple software artifacts (e.g., program source code and the software architecture). Nevertheless, there are a number of software complexity metrics that have been successfully used in software engineering domain [8]. Source Lines of Code (SLoC) is a simple size metric that measures the logical and the physical size of a source file. It is a size metric, since it can only describe the size dimension of a software artifact (e.g. the program source code). Weyuker et al. [41] have shown how to formalize, evaluate and compare different complexity metrics including SLoC. Although the motivation for measuring a specific dimension of a software varies in practice [38], several studies [21, 23] have indicated that complexity metrics may be good at predicting quality and development effort. For example, software complexity measurements [27] can be used to indicate the number of test cases needed to cover the logic of a particular artifact or can be used to show that a software architecture has a high levels of coupling [19]. Kumar et al. [25] proposed the use of source code metrics for PLC programs written in the ladder diagram (LD) programming language.

Even if the literature on measuring software complexity for industrial control software for PLCs has been scarce, other graphical programming languages have been the focus of research on complexity measurements. Olszewska et al. [33] tailored software complexity metrics to the component-based syntax of Simulink models. This study also performed a correlation analysis between complexity and fault data obtained from a car fuel program created using Simulink and found a positive correlation between components with high complexity and the number of faults found. The data was validated using three domain experts. At the time of writing, no other study have presented their method in a tool that can be used by both researchers and practitioners. In addition, we consider the use of such a tool for a correlation evaluation between test effort and software complexity metrics in the industrial control software domain.

3 COMPLEXITY MEASUREMENT OF PLC PROGRAMS USING TIQVA

Since there is no approach and tool support for measuring software complexity on PLC programs written in the FBD language, we developed a tool called Tiqva [32] in an effort to create a complexity measurement method for FBD software. Practically, we create an FBD data structure model based on the Abstract Syntax Trees (AST) used for parsing and processing source code. The data model is organized hierarchically and works in several iterations. We use the FBD XDS Schema rules in creating the data model (as shown in Figure 2) that contains all relevant program information from the FBD representation and maintains the hierarchy and relationships between different FBD elements.

A significant number of software complexity metrics have been proposed in the literature [12, 18, 19, 27, 29]. We chose to adapt and implement the following complexity metrics, since these are among the most popular and well-researched metrics [30] in the software engineering literature: Source Lines of Code (SLoC), Cyclomatic

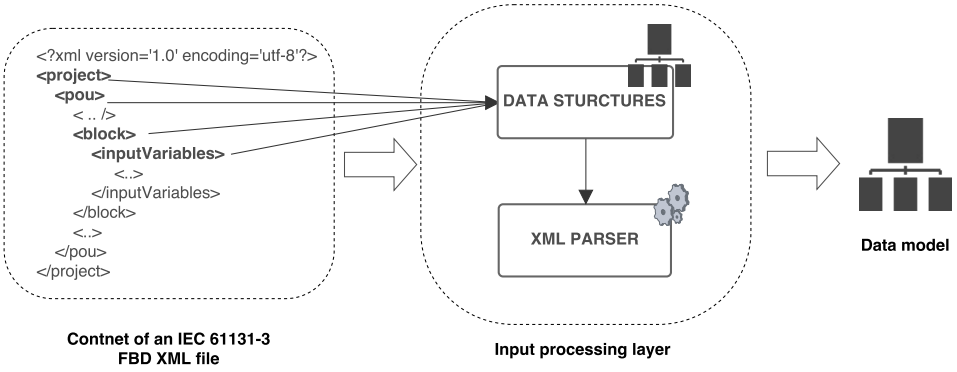

Figure 2: A high-level view of the input processing layer of Tiqva for parsing the FBD programs.

complexity (CC), Halstead complexity (HC) and Information Flow complexity (IFC). Both SLoC and HC are code size metrics and can abstract the size or the length of a program artifact. The HC metric is suggested to be good at providing information about software maintenance [18]. CC is a direct measure of the amount of decisions programmed in the software [27] and is used for determining the number of tests achieving basic path coverage [40]. The IFC metric was proposed by Henry and Kafura [19] and is mainly used for measuring the complexity of software architecture designs, since the metric computes the amount of coupling and cohesion between different software modules. We use IFC for FBD programs since some of these architecture models and FBD programs are both using basic component-based modelling concepts [20].

Even if the utility of using HC and SLOC metrics is in doubt [16, 35], there is some evidence (e.g., [34]) suggesting that the size of the software is the strongest individual predictor to identify programs likely to contain the largest numbers of faults. These kinds of relations need to be investigated further for other software quality attributes (e.g., testing effort, maintainability) and our tool supports these endeavours for PLC industrial software.

3.1 Number of Elements (NoE)

In the IEC 61131-3 FBD programming language, the notion of a program statement is very different compared to other general-purpose programming languages. While in Java, a line of code can be a function call, in FBDs functions are encapsulated inside block components. Therefore, we mapped the SLoC metric to FBDs. If the function calls and other program statements are abstracted via blocks, and the order of their execution is controlled via connections, then we can assume that the SLoC metric for FBDs would measure the number of elements including the blocks and connections in an FBD program [20]. We propose the use of *Number of Elements (NoE)* in the context of the FBD programming language by counting the number of declarations, blocks and connections. When initialized in the graphical programming environment, FBD variables and their data types are represented as component blocks (e.g., input, output and local component blocks). For example, for the program shown in Figure 3, the NoE score is 13 and is computed as the sum of all 3 variables, 2 function and 3 variable blocks and 5 connections.

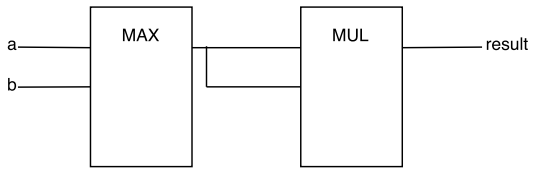

Figure 3: An example of an FBD program that outputs the square of the maximum between two input parameters.

3.2 Cyclomatic Complexity (CC)

In the original paper [27], Thomas McCabe proposed a software measurement technique for computing the number of linearly independent paths through a program code. This metric is based on graph theory and can be applied to a wide range of software artifacts (from simple program functions to architectures [28]). The CC metric can be measured using the following equation [27]:

$$M = \Pi - S + 2, \tag{1}$$

where Π is the number of decision points of a program and S is the number of exit points. This equation 1 points out that the CC score is directly influenced by the number of conditional statements in the program. Since CC is influenced by the decision points of a program, CC can directly be used for the FBD programming language. Using equation 1 on the FBD example in Figure 3 we can compute a CC value of 2.

3.3 Halstead Complexity

HC metric [18] is computing multiple software dimensions based on the measurement of operands and operators. We assume that a set of operators are represented using different mathematical and logical operations and programming language functions and syntax, while the set of operands are variables and values used in the operations. HC metric defines the following measurements: program vocabulary, program length, calculated program length, volume, difficulty, effort, time, and delivered bugs. These measurements are computed based on both the unique and total number of operators and operands. In FBDs, program variables and their definition are separated from the logic itself, so operators and operands are created in a different fashion compared to a Java or C program.

Halstead measurements	Equation	Result
Number of unique operators	η_1	6
Number of unique operands	η_2	7
Total number of operators	N_1	8
Total number of operands	N_2	11
Program vocabulary	$\eta = \eta_1 + \eta_2$	13
Program length	$N = N_1 + N_2$	19
Calculated program length	$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$	35
Volume	$V = N \times \log_2 \eta$	70
Difficulty	$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$	4
Effort	$E = D \times V$	331
Time	$T = \frac{E}{S} s, S = 18$	18s
Delivered bugs	$B = \frac{E^{\frac{2}{3}}}{3000}$	0.02

Table 1: Halstead complexity measurement equations [18] and the results for the example shown in Figure 3.

Functions and function blocks are representing *operations* such as comparison and multiplication. In addition, FBD connections are used for the flow of data through connections and thus we use them to calculate the number of *operands* in an FBD program. The results of calculating the Halstead values are shown in Table 1.

3.4 Information Flow Complexity

Henry and Kafura proposed the use of a software complexity metric [19] that could be applied at earlier stages of software development (e.g., during the software architecture modelling). As shown in Figure 4, a software module (i.e., module D) depends on other software modules. IFC can be used to measure the information flow between procedures or functions of a single program unit. Although IEC 61131-3 POUs (i.e., programs, functions and function blocks) are used as independent program units, these can be represented as software modules in the overall software architecture of a PLC software system. IFC can be tailored by measuring the number of defined inputs and outputs of an FBD functions or function blocks. This provides a baseline IFC score of an FBD POU, and that value can only increase when the POU is used in other FBD programs. In addition, we compute fan-in value using the number of output parameters and fan-out value using the number of input parameters. The SLoC value was measured using the FBD-equivalent NOE metric already defined in TIQVA.

TIQVA computes an IFC value based on (*fan-in*, *fan-out* and NoE:

$$c = NoE \times (\text{fan-in} \times \text{fan-out})^2 \quad (2)$$

For example, for the FBD program shown in Figure 3, the information flow complexity score can be calculated as follows:

$$c = SLOC \times (\text{fan-in} \times \text{fan-out})^2 = 13 \times (1 \times 2)^2 = 52 \quad (3)$$

4 IMPLEMENTATION AND EXPERIMENTAL METHODOLOGY

In this section we present the experimental roadmap used including the subject software, how we measured software complexity on

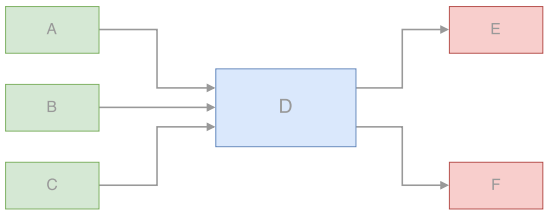

Figure 4: The architectural view of an examined module *D*, which is used by *A, B, C* and is using modules E, F.

such software as well as a use case for TIQVA in the form of a correlation analysis of the test effort for manual testing.

4.1 Implementation of TIQVA

IEC 61131-3 IDEs (i.e., the open-source Beremiz IDE [2] and the enterprise IDE Multiprog [3]), are lacking the support of a tool for measuring software complexity measurements that some general-purpose IDEs usually offer (e.g., Eclipse [1] or IntelliJ [4]). We developed TIQVA using Java programming language and the tool works directly with the PLCOpen XML file supported by IEC 61131-3 IDEs.

In Figure 5 we show the high-level architecture view of TIQVA containing three essential components:

- *Processing of Input Programs* by reading XML files containing FBD programs and generating a model.
- *Software Complexity Metrics Selection* by measuring different software complexity metrics using the defined techniques and generated models.
- *Results Writer*: Reporting the results in a .csv file format.

To increase modularity and reusability of TIQVA, we created a Java interface, *ComplexityMetric*, that defines methods for measuring software complexity of an FBD program. The interface contains two methods, one for measuring the complexity of a single POU, while the other measures the complexity of an entire FBD project containing multiple POUs. Once the software complexity measurement scores are computed, the collected results are reported. The *SoftwareComplexity* interface methods return the *HashMap* objects which contain the measurement results.

Maven [6] was used for building and handling the dependencies of the tool. TIQVA depends on standard Java packages found on the Maven repository. We used *DOM4J* framework to do the initial XML parsing. Since XML parsing is slow, compared to other components of TIQVA, we use multi-threading for certain parts of the tool to compensate for the overhead introduced by the parsing. If multiple XML files are provided to the tool, each file is assigned an individual thread for XML parsing, model building, and complexity measurement. Up to 100 FBD XML projects can be parsed at a given time. Once all XML files have been parsed by TIQVA, the results are collected and reported using the .csv results writer.

4.2 Subject Software

The safety-critical industrial control software used in this paper is part of the Train Control Management System (TCMS). TCMS is a system developed and used by Bombardier Transportation

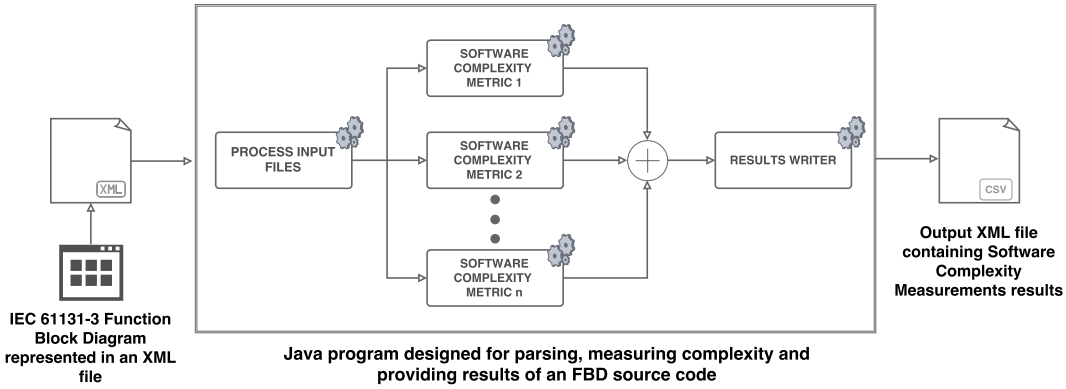

Figure 5: Architecture and modules of TIQVA.

Sweden AB for high speed trains. TCMS is an embedded software running on PLCs and used for handling a wide variety of operation-critical and safety-critical functions of a train. TCMS is written in IEC 61131-3 FBD programming language using a combination of IEC 61131-3 function and function blocks and in-house built function blocks. We used 82 FBD programs part of train control management system that perform supervisory operations and are developed independently of each other.

4.3 Test Effort

In this paper we use test cases for the individual TCMS FBD programs, which have been manually created and used for thorough testing performed by experienced industrial engineers. Practically, we used 82 test suites created by industrial engineers in Bombardier Transportation from a TCMS project delivered already to customers. A test case created for an FBD program contains a set of test cases containing inputs, expected and actual outputs and timing information. Data about these test cases was collected by using a post-mortem analysis [15]. In testing FBD programs in TCMS, the testing processes of software assurance are performed according to safety standards and regulations. Requirement-based testing is mandated by the EN 50128 standard [13] to be used to design test cases with each test case contributing to the requirement satisfaction. In addition, testers are required to create test cases based on multiple goals such as their experience, negative test cases as well as coverage-based test cases. Executing test cases on TCMS is supported by a test automation framework. The test cases collected in this study were based on functional requirements expressed in a natural language and achieved 100% requirement coverage for each program.

Test cases used in this paper are developed by experienced industrial engineers and were used for testing of a critical system already deployed in which the development has been finished. In this case strict requirements on both specification-based testing and code coverage typically are met with rigorous manual testing. Test suites were created to meet a rather good level of adequacy and are covering 100% of the functional and extra-functional requirements.

Many factors affect the effort needed to test an FBD program. According to Leung and White [26], testing involves direct and indirect costs. A direct cost includes the time needed for testing activities and the machine resources such as the test infrastructure used. Indirect costs could include the management of the testing process and the test tool development. Ideally, the test effort is captured by measuring the time required for performing all the different testing activities. Since this is a post-mortem study of a deployed TCMS system and the testing process was performed a few years back, we used proxy measures capturing the context that directly affects testing effort. We note here that the number of test cases depends on the testing strategy used but also on other contextual factors. A test strategy that requires that every branch in the program to be executed generally needs more tests than one which only requires all statements of the program to be executed. in this paper we assume that higher the number of tests cases, the higher is the respective test effort. Practically, this is a measure of the test effort of industrial engineers (working at Bombardier Transportation Sweden AB testing the programs used in our study) to perform thorough testing. The intuition is that a complex program will require more effort for testing, and also more tests than a simple program. Thus, the investigated hypothesis is that the test effort is related to the same factor— the complexity of the software which will influence the number of test cases. In addition, we use the time needed to execute the test cases as another proxy measure of test effort. This was measured directly when executing the test case on the actual test system configuration.

5 RESULTS

In this section, we quantitatively evaluate TIQVA. As Section 3 explained, we collected data and computed complexity measures for the FBD programs considered; and by collecting the number of test cases in each test suite as well as the test execution time we aimed at investigating the relation between the measured complexity scores and test effort.

5.1 Complexity Measurements

Figure 6 shows some of the data collected for all FBD programs, with each data point representing the number of test cases in each test suite as well as the test case execution time. Table 2 gives the descriptive statistics of the test data. We can observe that the average test execution time is 32 seconds while the test suite with the largest execution time takes 900 seconds. In addition, the average

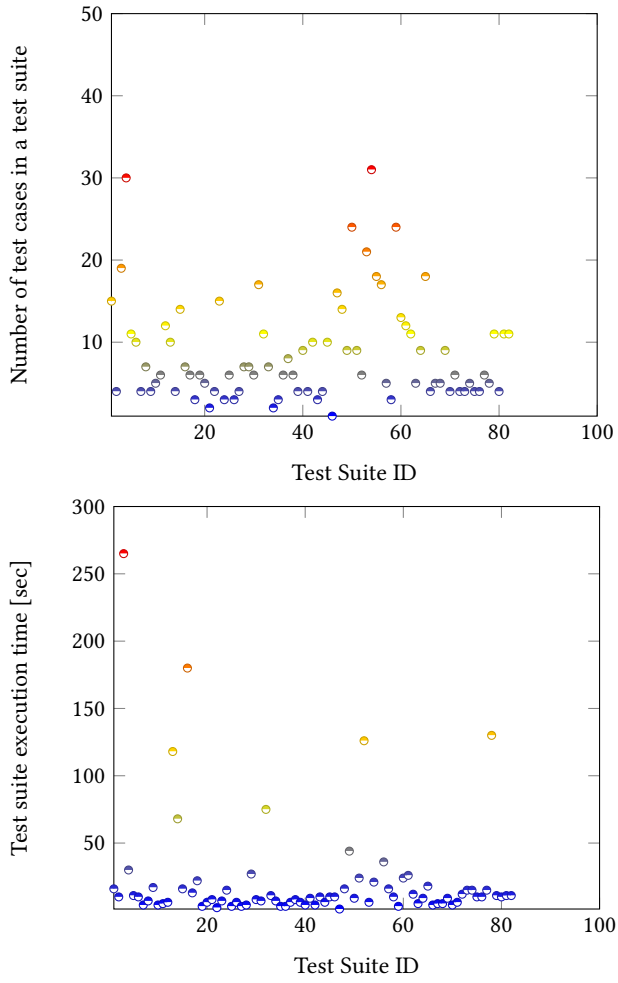

Figure 6: The number of test cases and the test execution time plotted for each individual test suite created for all programs.

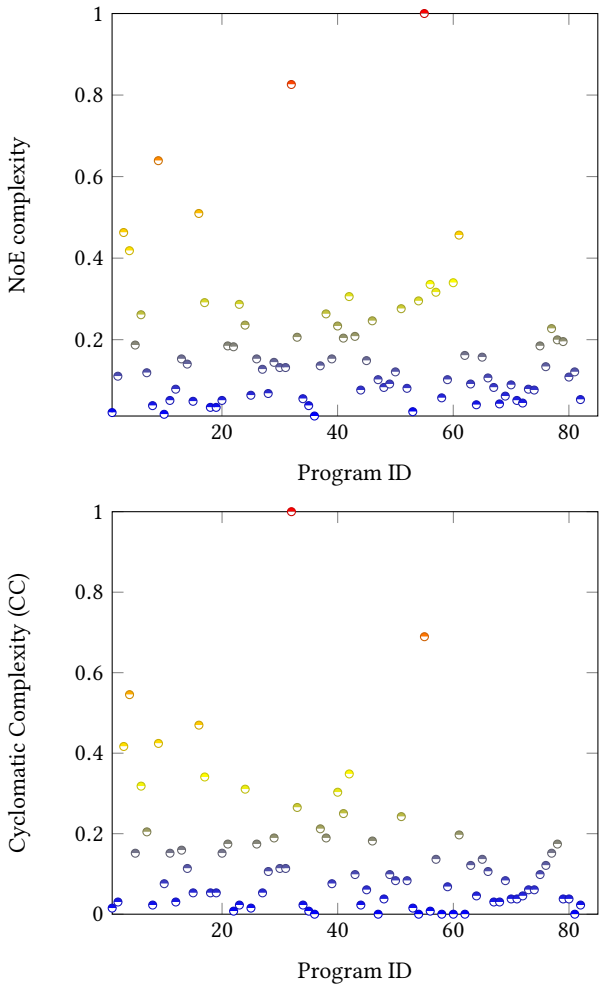

Figure 7: Normalized scores for the Number of Elements (NoE) complexity and Cyclomatic Complexity (CC) for all considered programs.

number of test cases in a test suite is 8.5 with the largest test suite containing 31 test cases.

We measured the software complexity of each FBD program using the Tiqva tool. This resulted in a total of 15 measurements for all software complexity measures used in this paper (NoE, CC, HC, IFC). Tiqva used these 15 measurements for the selected 82 programs.

In Table 3 we show the results of measuring the complexity of all FBD industrial control software. The different software complexity measures cannot be directly compared with each other. However, one program and its IFC score stands out with a high value of IFC complexity score. Actually, four FBD programs from TCMS showed high software complexity scores (i.e., Program 9, Program 60, Program 32 and Program 55). In particular, Program 32 achieved high complexity scores for NoE, CC, and Halstead (HC). Program 55 achieved the highest NoE score, Program 60 the highest IFC score and Program 9 the highest Halstead Difficulty score. Upon closer inspection, Program 32 has a very high number of input and outputs parameters as well as elements and can be considered, based on all complexity scores, the most complex program considered in this paper.

In order to get a better view on the distribution of individual metrics, we normalized the reported values (i.e., with 0 representing the lowest complexity score while 1 showing the highest complexity score.

The plots in Figure 7 show the distribution of NoE and CC metrics. We can observe that two outliers (Programs 32 and Program 55) show high NoE and CC scores. The rest of the programs are scattered below the 0.5 threshold score. A similar result can also be seen when considering IFC and Halstead Difficulty metrics. In Figure 8 we observe that IFC scores are very much polarized with quite low scores for most of the programs. In Figure 9 we show an area plot for two Halstead complexity measures (i.e., Halstead Difficulty and Halstead Volume) which are used to construct the

Measure of Test Effort	Min	Max	Average	Median	Standard Deviation
Number of test cases	1	31	8.5	6	6.2
Test execution time (sec.)	1	900	32.2	10	105.2

Table 2: Results for both test effort metrics: the number of test cases in each test suite and the test execution time.

Software complexity measure	Min	Max	Average	Median	Standard Deviation
Variables	4	85	22	18.5	14.7
Connections	3	216	33.8	25.5	32.5
Blocks	5	228	39	29.5	35.1
Number of elements (NoE)	*12*	*483*	*94.8*	*74.5*	*80*
Cyclomatic Complexity (CC)	*1*	*133*	*18.9*	*13*	*21.8*
Information flow complexity (IFC)	*12*	*57065472*	*2690651*	*68506*	*8478907.8*
Unique operators	8	19	11.4	11	2.6
Unique operands	9	262	59.5	47.5	47
Total operators	12	229	59.1	49.5	39.8
Total operands	12	402	86.3	69	71.1
Halstead Program vocabulary	17	278	71	60	48.5
Halstead Program length	24	599	145.5	117	110.1
Halstead Calculated Program Length	52.5	2168.7	413.2	309.1	385.6
Halstead Volume	98.1	4844.3	942.5	691.5	886.8
Halstead Difficulty	5.3	14.1	8.1	7.9	1.8
Halstead Effort	523.2	59747.4	8756.5	5470.9	10925.8
Halstead Time	29	3319.3	486.4	303.9	606.9
Halstead Delivered Bugs	0.02	0.5	0.1	0.1	0.09

Table 3: Results for all software complexity metrics (i.e., Number of Elements (NoE), Cyclomatic Complexity (CC), Information Flow Complexity (IFC) and Halstead) together with the basic measures used to calculated these metrics.

rest of the other Halstead metrics (i.e., Effort, Testing Time and Delivered Bugs). Both of the metrics shown in Figure 9 have a similar distribution for all the programs considered in this study.

5.2 Correlation Analysis

Is the test effort (i.e, number of test cases and test execution time) influenced by the software complexity of the programs considered in this study? Table 4 shows the Kendall correlation coefficients [22] we computed to answer this question. Kendall rank correlation is used as a measure of correlation between software complexity scores and the test effort proxy scores. Since the data is not normally distributed we use Kendall correlation to not introduce unnecessary assumptions about the collected data. We try to determine the possible statistical relationship between software complexity and the test effort scores. We used Kendall's rank correlation coefficient to calculate the statistical relationship between the scores with a significance level of *0.05* since a statistically significant correlation does not necessarily mean that the strength of the correlation is strong. Here we use the Cohen scale [14], in which correlations with absolute value less than 0.3 are described as weak, 0.3 to 0.5 as moderate, 0.5 to 0.9 as strong and very strong.

The two test effort proxy measures required the computation of the correlation coefficients using R [5]. Table 4 shows τ coefficients and p-values for the two proxy measures (i.e., E stands for test suite execution time and N stands for the number of test cases in a test suite). A positive correlation can be observed for four software complexity metrics (i.e., Halstead is shown as three separate complexity measures: Difficulty, Volume and Effort). We should note here that the p-value$_N$ for Halstead Difficulty is 0.1, thus showing that for this measure the correlation is not strong. Overall, the results show that all coefficients, except for Halstead difficulty measure, are significant. Table 4 gives the correlation between the different complexity scores and the test suite execution

Software complexity metrics	τ_E	p-value$_E$	τ_N	p-value$_N$
Number of Elements	0.342	$8.192e^{-6}$	0.368	$2.315e^{-6}$
Cyclomatic Complexity	0.225	0.003	0.252	0.001358
Information Flow Metric	0.264	0.0005	0.345	$9.116e^{-06}$
Halstead Volume	0.328	$1.878e^{-5}$	0.351	$6.25e^{-6}$
Halstead Difficulty	0.208	0.006	0.125	0.1061
Halstead Effort	0.320	$2.882e^{-5}$	0.320	$3.876e^{-5}$

Table 4: Kendall correlation coefficient (τ) and p-value between software complexity metrics and the test effort. The test effort was expressed by two proxy scores: test suite execution time (E) and the number of tests in a test suite (N).

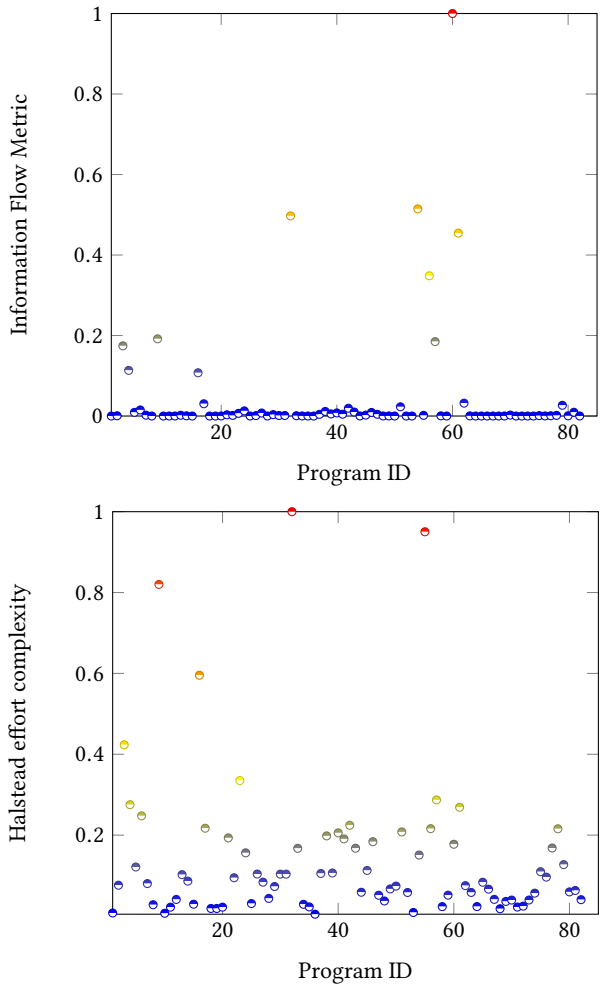

Figure 8: Normalized scores for the Information Flow Metric (IFC) and Halstead Effort Complexity for all considered programs.

time and the number of test cases measures of test effort. For all programs, we see a low to moderate correlation between software complexity and the number of test cases created as well as for the test suite execution time (with a highest correlation coefficient of 0.368 for the Number of Elements (NoE) metric). These results show that there is a statistical relationship between software complexity measures and test effort measures for FBD programs and test data for a real industrial system engineered by Bombardier Transportation Sweden AB. The NoE metric achieved the highest correlation score. Interestingly enough, the cyclomatic complexity metric, a structure metric, obtained a lower correlation than NoE size metric. This can be taken as an argument in favor of not measuring the structure of FBD programs in this way.

Our results suggest that, for the industrial programs considered in this study, TIQVA is an applicable tool and can be used for correlation analysis between different software metrics. Our results suggest that there is a low to moderate correlation between the test

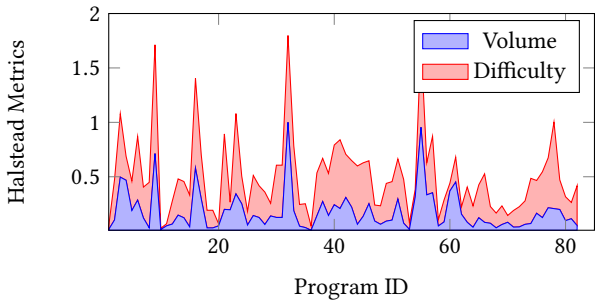

Figure 9: A normalized area plot showing two Halstead metrics (i.e., Volume and Difficulty) with a similar distribution of scores across all programs in TCMS.

effort (i.e., the number of created test cases and the test execution time) and the software complexity of a program. The size of the software (i.e., number of elements measure) provides the highest correlation with the test effort.

5.3 Building a Prediction Model using TIQVA's Complexity Metrics

We examined a train control software, which is a valid and representative case for industrial software and IEC 61131-3 FBD software used in the embedded system domain. By using the complexity measurement results and the test effort dedicated for testing the software (i.e., the number of tests cases of a test suite and the execution time of a test suite), we indicated that there is a statistical relationship between software complexity and test effort. However, the correlation is low to moderate (i.e, 0.368 Kendall's τ coefficient is the highest correlation score). There are indications that higher FBD software complexity does not imply a higher test effort.

As an effort to implement a test effort prediction model, we designed a linear regression model of the test effort using several software complexity metrics. The idea is to predict a dependent variable using correlated independent variables. We use a linear regression [31] (i.e., multiple linear regression (multiple independent variable)) to show how the results and the TIQVA tool can be used predict the test effort required for adequate testing. In practice, we used a linear regression model of the test effort measure using the following measures:

$$\beta_1 C_1 + \beta_2 C_2 + \cdots + \beta_{n-1} C_{n-1} + \beta_n C_n = T_{\text{measure}}, \quad (4)$$

where the test effort measure T_{measure} is a linear function of different weighted software complexity scores C_ns. The linear regression model shown in equation 4 could be used to predict the test effort for an industrial control software after determining the weight values (βs), and it requires an existing set of software complexity measurements and a test effort measure to be solved.

Using the previously measured software complexity scores for the 82 FBD programs as the input data set and the two test effort proxy measures as the output data set, we determined the weights using a Python machine learning library [7]. We used the *Linear Regression* module to determine the weights as well as to assign a variance score (i.e., the amount of correct predictions of the model

Test Effort Measures	β_{NoE}	β_{CC}	β_{HEC}	β_{IFM}	MSE	Score
Number of Test Cases	$1.118e^{-7}$	$4.69e^{-2}$	$-6.528e^{-4}$	$1.36e^{-7}$	17.31	0.55
Test Suite Execution Time	1.92	-1.4	$-3.82e^{-3}$	$-5.56e^{-6}$	5077.68	0.54

Table 5: Complexity weights and mean square error of predictions based on the weights and the prediction variance score.

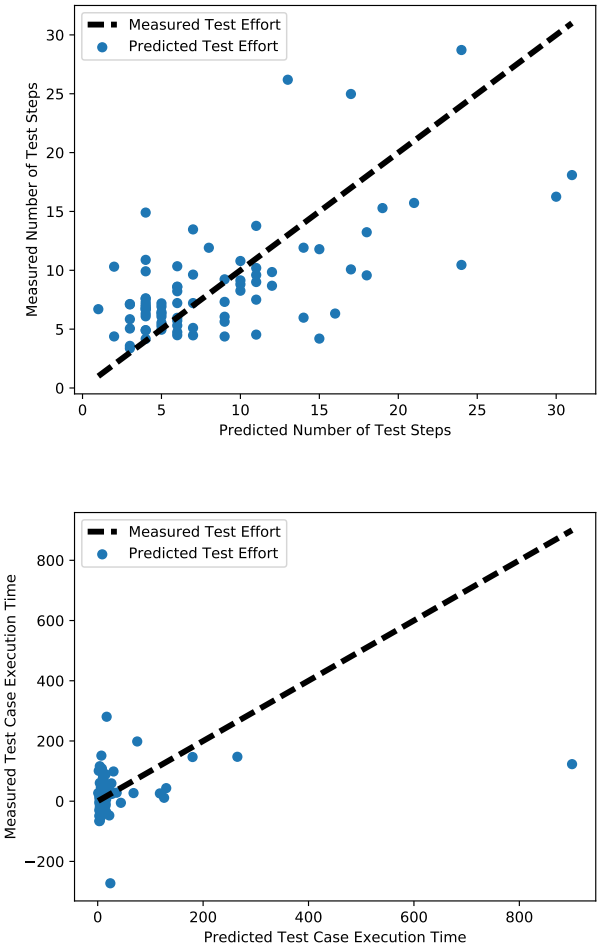

Figure 10: Graphs showing the predictions of the trained linear regression model for the test effort proxy measures (i.e., number of test cases and test suite execution time).

with 1.0 being the highest score). Two regression models have been developed based on the previous equation (i.e., Equation 4) and several complexity metrics for FBD programs:

$$\beta_{\text{NoE}}C_{\text{NoE}} + \beta_{\text{CC}}C_{\text{CC}} + \beta_{\text{HEC}}C_{\text{HEC}} + \beta_{\text{IFM}}C_{\text{IFC}} = T_{\text{N}} \quad (5)$$

$$\beta_{\text{NoE}}C_{\text{NoE}} + \beta_{\text{CC}}C_{\text{CC}} + \beta_{\text{HEC}}C_{\text{HEC}} + \beta_{\text{IFM}}C_{\text{IFC}} = T_{\text{E}}, \quad (6)$$

where NoE is the Number of Elements, CC is the Cyclomatic Complexity, HEC is the Halstead Effort Complexity, IFC is the Information Flow Complexity, N and T stand for the number of test cases and test suite execution time respectively. Only Effort has been taken into account from the different HC metrics.

Based on our results, we assumed that the linear regression model will not have a high prediction accuracy considering the low to moderate correlation between software complexity metrics and test effort. After examining the trained linear regression model using the full data set of 82 programs for training and for testing the model, we report the results in Table 5. These results show that only half of the test effort predictions were accurate. The mean squared value was low when the model tried to predict the number of test cases and significantly higher for the test suite execution time. This can be explained by the non-linear and irregular values of the execution time test effort shown in Figure 6. Another characteristic of the model is the achieved higher β weight value for the Number of Elements (NoE) messure in both models.

Figure 10 shows the predictions (in blue scatter points) of the test effort in contrast to the measured test effort (black line). Although the predictions of the test execution time are similar to the predictions for the number of test cases, these are clustered around one area (i.e., lower test execution time), while the number of test cases is distributed across the complete test effort scores spectrum. This shows that a test effort estimation can be made using software complexity measurement scores. Since test cases in industry are designed using different information sources (e.g., in the safety critical domain using functional specifications and human domain knowledge), one would need to include other metrics for predicting the test effort. The results are promising, but we only achieved a rough estimator. However, researchers and practitioners should fine tune this kind of estimations by taking into account other metrics for software artifacts (e.g., specification, test knowledge) which are heavily influencing the overall test effort.

6 THREATS TO VALIDITY

Industrial control software used in PLCs can be programmed in a wide variety of programming languages, such as IEC 61131-3 FBD and ST languages. In this paper, we examined how software complexity metrics can be applied on industrial control software developed using FBDs in one company (i.e., Bombardier Transportation Sweden AB), thus narrowing the scope of the study. However, we argue that the examined software (TCMS) shows general characteristics of the safety-critical industrial domain. In addition, the set of software complexity metrics chosen to be used on FBD programs is not complete by any means. We have not used complexity measures such as entropy [37], Kolmogorov complexity [24], since the purpose of this paper is to explore the test effort relation with software complexity. This is a first step in this endeavour. Also, the test effort is not straightforward to measure and requires knowledge of the multiple phases performed during software testing including test creation. In this study we focus on two proxy measures for

test effort. More studies are needed to generalize the results of this paper.

7 CONCLUSION

TIQVA is a tool that automatically produces complexity measurements scores for FBD control programs. TIQVA implements several measures adapted to the nature of the FBD language. This paper presents also an exploration on how software complexity can be applied on industrial domain-specific software written in the FBD graphical programming language. We used four, well known, software complexity metrics. We studied the relationship between test effort (i.e., number of test cases and test execution time of a program's test suite) and the program complexity scores. From the 82 industrial FBD programs we studied, we drew the following conclusions: TIQVA is applicable to measure the complexity of real industrial FBD programs, there is a low to moderate correlation between the effort needed to test a program and its complexity, and the size of the software in terms of the number of elements provides the highest correlation with the test effort. The results from this study also indicate that other aspects than code complexity should be taken into account to better capture the relationship between the implemented and specified software artifacts and test effort. Also, other complexity dimensions of the FBD programs (e.g., function block relationships, block coupling and timing) could be used to improve the measurement of complexity for an FBD program.

The results of applying TIQVA on an industrial system are useful for both researchers and industrial engineers and show its applicability when applied on a safety-critical industrial software for high speed trains. We argue that we went further and showed experimental evidence on how this tool can be used for estimating one aspect of software development (i.e., test effort). Given that we have used well-known complexity metrics in TIQVA, we believe that our results (even if negative for the correlation between test effort and well-known software complexity metrics) are especially important in engineering of PLC systems. Since experimentation should drive research forward, for pragmatic and methodological reasons, it is important to report both positive and negative experimental results like the ones described in our paper. Finding and acknowledging the actual use of this tool and these complexity metrics should be regarded as a benefit for our field of research.

ACKNOWLEDGMENTS

This research was supported by the ECSEL Joint Undertaking under grant agreement No. 737494 and the Swedish Innovation Agency, Vinnova (MegaM@Rt2 and XIVT projects).

REFERENCES

[1] [n. d.]. Eclipse - The Eclipse Foundation open source community website. https://eclipse.org/. (Accessed on 05/15/2017).
[2] [n. d.]. Home page of Beremiz. http://www.beremiz.org/. (Accessed on 05/15/2017).
[3] [n. d.]. Index - KW-Software | EN. http://www.kw-software.com/en/. (Accessed on 05/15/2017).
[4] [n. d.]. IntelliJ IDEA the Java IDE. https://www.jetbrains.com/idea/. (Accessed on 05/15/2017).
[5] [n. d.]. Kendall Rank Coefficient | R Tutorial. http://www.r-tutor.com/gpu-computing/correlation/kendall-rank-coefficient. (Accessed on 04/27/2017).
[6] [n. d.]. Maven âĂŞ Welcome to Apache Maven. https://maven.apache.org/. (Accessed on 05/15/2017).
[7] [n. d.]. scikit-learn: machine learning in Python âĂŤ scikit-learn 0.18.1 documentation. http://scikit-learn.org/stable/index.html. (Accessed on 04/29/2017).
[8] Alain Abran. 2010. *Software metrics and software metrology*. John Wiley & Sons.
[9] Victor R Basili and Barry T Perricone. 1984. Software errors and complexity: an empirical investigation0. *Commun. ACM* 27, 1 (1984), 42–52.
[10] William Bolton. 2015. *Programmable logic controllers*. Newnes.
[11] William Bolton. 2015. *Programmable logic controllers*. Newnes.
[12] David N Card and Robert L Glass. 1990. *Measuring software design quality*. Prentice-Hall, Inc.
[13] CENELEC. 2001. 50128: Railway Application: Communications, Signaling and Processing Systems, Software For Railway Control and Protection Systems. In *Standard Official Document*. European Committee for Electrotechnical Standardization.
[14] Jacob Cohen. 1988. Statistical power analysis for the behavioral sciences. 2nd.
[15] Reidar Conradi and Alf Inge Wang. 2003. *Empirical methods and studies in software engineering: experiences from ESERNET*. Vol. 2765. Springer.
[16] NE Fenton. 1996. eeger, Software Metrics, A Rigorous and Practical Approach.
[17] John E Gaffney. 1984. Estimating the number of faults in code. *IEEE Transactions on Software Engineering* 4 (1984), 459–464.
[18] Maurice Howard Halstead. 1977. *Elements of software science*. Vol. 7. Elsevier New York.
[19] Sallie Henry and Dennis Kafura. 1981. Software structure metrics based on information flow. *IEEE transactions on Software Engineering* 5 (1981), 510–518.
[20] Karl-Heinz John and Michael Tiegelkamp. 2010. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media.
[21] Dennis Kafura and Geereddy R. Reddy. 1987. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering* 3 (1987), 335–343.
[22] Maurice George Kendall. 1948. Rank correlation methods. (1948).
[23] Kostas Kevrekidis, Stijn Albers, Peter JM Sonnemans, and Guillaume M Stollman. 2009. Software complexity and testing effectiveness: An empirical study. In *Reliability and Maintainability Symposium*. IEEE, 539–543.
[24] Andrei N Kolmogorov. 1963. On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A* (1963), 369–376.
[25] Lov Kumar, Raoul Jetley, and Ashish Sureka. 2016. Source code metrics for programmable logic controller (PLC) ladder diagram (LD) visual programming language. In *International Workshop on Emerging Trends in Software Metrics*. IEEE, 15–21.
[26] Hareton KN Leung and Lee White. 1991. A cost model to compare regression test strategies. In *Proceedings. Conference on Software Maintenance*. IEEE, 201–208.
[27] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.
[28] Thomas J McCabe and Charles W Butler. 1989. Design complexity measurement and testing. *Commun. ACM* 32, 12 (1989), 1415–1425.
[29] Carma L McClure. 1978. A model for program complexity analysis. In *Proceedings of the 3rd international conference on Software engineering*. IEEE Press, 149–157.
[30] Tom Mens. 2016. Research trends in structural software complexity. *arXiv preprint arXiv:1608.01533* (2016).
[31] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. 2015. *Introduction to linear regression analysis*. John Wiley & Sons.
[32] Adnan Muslija. 2019. Tiqva - FBD complexity measurement tool. https://github.com/amuslija/fbd-complexity-tool
[33] Marta Olszewska, Yanja Dajsuren, Harald Altinger, Alexander Serebrenik, Marina Waldén, and Mark GJ van den Brand. 2016. Tailoring complexity metrics for simulink models. In *Proccedings of the 10th European Conference on Software Architecture Workshops*. ACM, 5.
[34] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering* 31, 4 (2005), 340–355.
[35] Shari Lawrence Pfleeger, Ross Jeffery, Bill Curtis, and Barbara Kitchenham. 1997. Status Report on Software Measurement. *IEEE Software* 14, 2 (1997), 33–43.
[36] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. 2009. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 367–377.
[37] Claude E Shannon. 1951. Prediction and entropy of printed English. *Bell Labs Technical Journal* 30, 1 (1951), 50–64.
[38] Martin Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal* 3, 2 (1988), 30–36.
[39] Keith Stouffer, Joe Falco, and Karen Scarfone. 2011. Guide to industrial control systems (ICS) security. *NIST special publication* 800, 82 (2011), 16–16.
[40] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. 1996. *Structured testing: A testing methodology using the cyclomatic complexity metric*. Vol. 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology.
[41] Elaine J. Weyuker. 1988. Evaluating software complexity measures. *IEEE transactions on Software Engineering* 14, 9 (1988), 1357–1365.