A Comparative Study of Manual and Automated Testing for Industrial Control Software

Eduard Enoiu, Adnan Čaušević, Daniel Sundmark, Paul Pettersson
Mälardalen University, Västerås, Sweden

***Abstract*—Automated test generation has been suggested as a way of creating tests at a lower cost. Nonetheless, it is not very well studied how such tests compare to manually written ones in terms of cost and effectiveness. This is particularly true for industrial control software, where strict requirements on both specification-based testing and code coverage typically are met with rigorous manual testing. To address this issue, we conducted a case study in which we compared manually and automatically created tests. We used recently developed real-world industrial programs written in the IEC 61131-3, a popular programming language for developing industrial control systems using programmable logic controllers. The results show that automatically generated tests achieve similar code coverage as manually created tests, but in a fraction of the time (an average improvement of roughly 90%). We also found that the use of an automated test generation tool does not result in better fault detection in terms of mutation score compared to manual testing. Specifically, manual tests more effectively detect logical, timer and negation type of faults, compared to automatically generated tests. The results underscore the need to further study how manual testing is performed in industrial practice and the extent to which automated test generation can be used in the development of reliable systems.**

I. Introduction

Testing is an important activity in engineering of industrial control software. To support developers in testing their software, researchers have proposed different approaches for producing good test cases. In the last couple of years a wide range of techniques for automated test generation [15] have been explored with the goal of complementing manual testing. Even though there is some evidence suggesting that automatically generated test suites may even cover more code than those manually written by developers [8], this does not necessarily mean that these tests are effective in terms of detecting faults. As manual testing and automated code coverage-directed test generation are fundamentally different and each strategy holds its own inherent limitations, their respective merits or demerits should be analyzed more extensively in comparative studies.

In this paper, we empirically evaluate automated test generation and compare it with test suites manually created by industrial engineers on 61 programs from a real industrial train control system. This system contains software written in IEC 61131-3 [10], a popular language in safety-critical industry for programming control software [12]. We have applied a state-of-the-art test generation tool for IEC 61131-3 software, CompleteTest [5], and investigated how it compares with manual testing performed by industrial engineers in terms of code coverage, cost and fault detection.

Our case study indicates that for IEC 61131-3 safety-critical control software, automated test generation can achieve similar code coverage as manual testing performed by industrial engineers but in a fraction of the time. The use of automated test generation in IEC 61131-3 software development can potentially save around 90% of testing time. Even when achieving full code coverage, automatically generated test suites are not necessarily better at finding faults than manually created test suites. In our case study, 56% of the test suites generated using CompleteTest found less faults than test suites created manually by industrial engineers. Overall, it seems that manually created tests are able to detect more faults of certain types (i.e, logical replacement, negation insertion and timer replacement) than automatically generated tests. Using the detection of these specific mutations in addition to structural properties as the coverage criterion used by an automated test generation tool, we could generate more effective test suites.

II. Related Work

In the context of developing industrial control software, IEC 61131-3 [12] has become a very popular programming language used in different control systems from traffic control software to nuclear power plants. Several automated test generation approaches [19], [5], [4] have been proposed in the last couple of years for IEC 61131-3 software. These techniques can typically produce test suites for a given code coverage criterion and have been shown to achieve high code coverage for different IEC 61131-3 industrial software projects. While high code coverage has historically been used as a proxy for the ability of a test suite to detect faults, recently Inozemtseva et al. [11] found that coverage should not be used as a measure of quality mainly because of the fact that it is not a good indicator for fault detection. In other words, the fault detection capability of a test suite might rely more on other test design factors than the extent to which the structure of the code is covered. There are studies investigating the use of both manual testing and automated coverage-based test generation of real-world programs. Several researchers have performed case studies [21], [13], [17] and focused on already created manual test suites while others performed controlled experiments [8] with human participants manually creating and automatically generating test suites. These results kindled our interest in studying how manual testing compares to automated coverage-based test generation in an industrial safety-critical control software domain. For such systems, there are strict requirements on both traceable specification-based and implementation-based testing. Is there any evidence on how these code coverage-directed automated tools compare with, what is perceived as, rigorous manual testing?

III. Method

From a high level view we started the case study by considering: (i) manual test suites created by industrial engineers and a tool for automated test generation named CompleteTest, (ii) a set of IEC 61131-3 industrial programs from a recently developed train control management system (TCMS), (iii) a cost model and (iv) a fault detection metric.

We aimed to answer the following research questions:

- *RQ1: Are automatically generated test suites able to detect more faults than tests suites manually created by industrial engineers for IEC 61131-3 control software?*
- *RQ2: Are automatically generated test suites less costly than tests suites manually created by industrial engineers for IEC 61131-3 control software?*

For each selected program, we executed the test suites produced by both manual testing and automated test generation and collected the following measures: code coverage in terms of achieved code coverage, the cost of performing testing and the mutation score as a proxy for fault detection.

A. Case Description

The studied case is an industrial system actively developed in the safety-critical domain by Bombardier Transportation, a leading, large-scale company focusing on development and manufacturing of trains and railway equipment. The system is a train control management system (TCMS) that has been in development for several years and is engineered with a testing process highly influenced by safety standards and regulations. TCMS is a distributed control system with multiple types of software and hardware components, and is in charge of much of the operation-critical, safety-related functionality of the train. TCMS runs on Programmable Logic Controllers (PLC) which are real-time controllers used in numerous industrial domains, i.e., nuclear plants and avionics. An IEC 61131-3 program running on a PLC [12] executes in a loop where every cycle contains the reading of input values, the execution of the program without interruption and the update of the outputs. IEC 61131-3 programs contain particular types of blocks, such as timers that provide the same functions as timing relays and are used to activate or deactivate a device after a preset interval of time. For more details on this programming language we refer the reader to the work of John et al. [12].

B. Test Suite Creation

We used manual test suites created by industrial engineers in Bombardier Transportation from a TCMS project delivered already to customers. Manual test suites were collected by using a post-mortem analysis [3] of the test data available. In testing programs in TCMS, the engineering processes of software development are performed according to safety standards and regulations. Specification-based testing is mandated by the EN 50128 standard [2] to be used to design test cases. Each test case should contribute to the demonstration that a specified requirement has indeed been covered and satisfied. Executing test cases on TCMS is supported by a test framework that includes the comparison between the expected output with the actual outcome. The test suites collected in this study were based on functional specifications expressed in a natural language.

In addition, we used test suites automatically generated using an automated test generation tool. For the programs in the TCMS system, EN 50128 recommends the implementation of test cases achieving a certain level of code coverage on the developed software (e.g., decision coverage which is also known as branch coverage). To the best of our knowledge, CompleteTest [5] is the only available automated test generation tool for IEC 61131-3 software that produces tests for a given coverage criterion. It should be noted that in case CompleteTest is unable to achieve full coverage for a given program (which may happen since some coverage items may not be reachable, or that the search space is too large), a cutoff time is required to prevent indefinite execution. Based on discussions with engineers developing TCMS regarding the time needed for CompleteTest to provide a test suite for a desired coverage, we concluded that 10 minutes was a reasonable timeout point for the tool to finish its test generation. As a consequence, the tests generated after this timeout is reached will potentially achieve less than 100% code coverage.

C. Subject Programs

We used a number of criteria to select the subject programs for our study. We investigated the industrial library contained in TCMS provided by Bombardier Transportation. Firstly, we identified a project containing 114 programs. Next, we excluded 32 programs based on the lack of possibility to automatically generate test cases using CompleteTest, primarily due to the fact that those programs contained data types or predefined blocks not supported by the underlying model checker (i.e. string and word data types). The remaining 82 programs were subjected to detailed exclusion criteria, which involved identifying the programs for which engineers from Bombardier Transportation had created tests manually. This resulted in 72 remaining programs, which were further filtered out by excluding the programs not containing any decisions or logical constructs (since these would not be meaningful to test using logic criteria). A final set of 61 programs was reached. These programs contained on average per program: 18 decisions (i.e., branches), 10 input variables and 4 output variables.

For each of the 61 programs, we collected the manually created test suites. In addition we automatically generated test suites using CompleteTest for covering all decisions in each program. As a final step we generated additional test cases for all 61 programs using random test suites.

D. Measuring Fault Detection

Fault detection was measured using mutation analysis. For this purpose, we used our own tool implementation to generate faulty versions of the subject programs. To describe how this procedure operates, we must first give a brief description of mutation analysis. Mutation analysis is the technique of creating faulty implementations of a program (usually in an automated manner) for the purpose of examining the fault detection ability of a test suite [1]. A *mutant* is a new version of a program created by making a small change to the original program. For example, in an IEC 61131-3 program, a mutant is created by replacing a block with another, negating a signal, or changing the value of a constant. The execution of a test suite on the resulting mutant may produce a different output as the original program, in which case we say that the test suite *kills* that mutant. A mutation score is calculated by automatically seeding mutants to measure the mutant detecting capability of the written test suite. We computed the mutation score using an output-only oracle (i.e., expected values for all of the program outputs) against the set of mutants. For all programs, we assessed the fault-finding capability of each test suite by calculating the ratio of mutants killed to the total number of mutants.

In the creation of mutants we rely on previous studies that looked at commonly occurring faults in IEC 61131-3 software [14], [18]. We used these common faults in this study for establishing the following mutation operators:

- *Logic Block Replacement Operator (LRO).* Replacing a logical block with another block from the same function category (e.g., replacing an AND block with an OR block).
- *Comparison Block Replacement Operator (CRO).* Replacing a comparison block with another block from the same function category (e.g., replacing a Greater-Than (GT) block with a Greater-or-Equal (GE) block).
- *Arithmetic Block Replacement Operator (ARO).* Replacing an arithmetic block with another block from the same function category (e.g., replacing a maximum (MAX) block with a subtraction (ADD) block).
- *Negation Insertion Operator (NIO).* Negating an input or output connection (e.g., an input variable in becomes NOT(in)).
- *Value Replacement Operator (VRO).* Replacing a value of a constant variable connected to a block (e.g., replacing a constant value ($const = 0$) with its boundary values (e.g., $const = -1$)).
- *Timer Block Replacement Operator (TRO).* Replacing a timer block with another block from the same function category (e.g., replacing a Timer-On (TON) block with a Timer-Off (TOF) block).

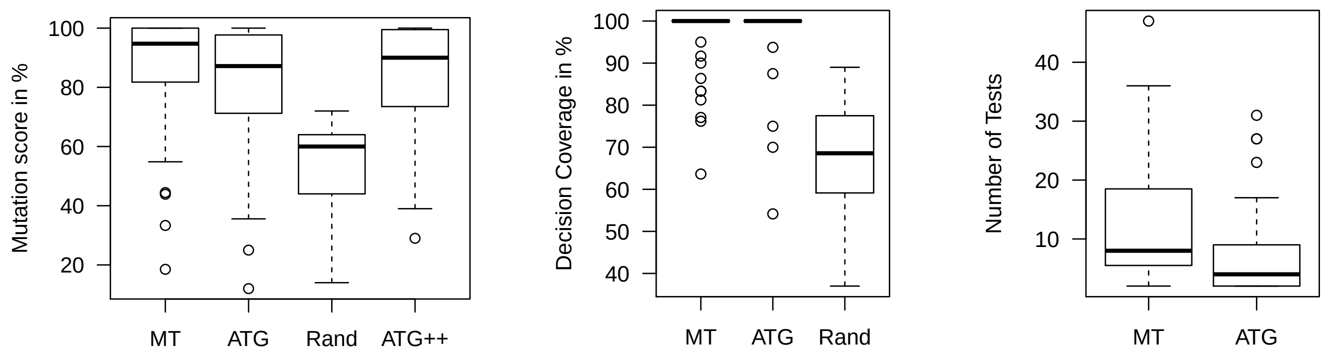
To generate mutants, each of the mutation operators was systematically applied to each program element wherever possible. In total, for all of the selected programs, 5161 mutants (faulty programs based on ARO, LRO, CRO, NIO, VRO and TRO operators) were generated by automatically introducing a single fault into the original implementation.

E. Measuring Efficiency

Ideally, the test effort is captured by measuring the time required for performing different testing activities. However, since this is a post-mortem study of a now-deployed system and the development was undertaken a few years back, this was not practically possible in our case. Instead, efficiency was measured using a cost model that captures the context that affects the testing of IEC 61131-3 software. We focused on the unit testing process as it is implemented in Bombardier Transportation for testing the programs selected in this case study. In the cost model, we concentrated on the following components: the cost of writing the necessary test suite, the cost of executing a test suite, the cost of checking the result of the test suite, and the cost of reporting a test suite. The cost estimation does not include the required tool preparation. However, preparation entails exporting the program to a format readable by COMPLETETEST and opening the resulting file. This effort is comparable to that required for opening the tools needed for manual testing. To formulate a cost model incorporating each cost component, we must measure costs in identical units. To do this, we recorded all costs using a time metric. For manual testing all costs are related to human effort. Practically, we measured the costs of these activities directly as an average of the time taken by three industrial engineers (working at Bombardier Transportation implementing some of the IEC 61131-3 programs used in our case study) to perform manual testing. For automated test generation the cost of checking the test result is related to human effort with the other costs measured in machine time needed to compute the results. For more details on this cost model we refer the reader to the extended technical report of Enoiu et al. [6].

IV. RESULTS

This section provides an analysis of the data collected in this case study. For each program and each testing technique considered in this study we collected the produced test suites. The overall results of this study are summarized in the form of boxplots in Figure 1. In Table I we present the mutation scores, coverage results and the number of test cases in each collected test suite (i.e., *MT* stands for manually created test suites and *ATG* is short for test suites automatically generated using COMPLETETEST). This table lists the minimum, median, mean, maximum and standard deviation values. As our observations are drawn from an unknown distribution, we evaluate if there is any statistical difference between *MT* and *ATG* without making any assumptions on the distribution of the collected data. We use a Wilcoxon-Mann-Whitney U-test [9], a non-parametric hypothesis test for for checking if there is any statistical difference among each measurement metric. In addition, the Vargha-Delaney test [20] was used to calculate the standardized effect size, which is a non-parametric magnitude test that shows statistical significance.



(a) Mutation score comparison (b) Decision coverage achieved (c) Number of Test Cases

Fig. 1: Mutation score, achieved code coverage and number of test cases comparison between manually created test suites (MT), automatically generated test suites (ATG), pure random test suites (Rand) of the same size as the ones created manually, and coverage-adequate tests with equal size as manual tests (ATG++); boxes spans from 1st to 3rd quartile, black middle lines mark the median and the whiskers extend up to 1.5x the inter-quartile range and the circle symbols represent outliers.

TABLE I: Results for each metric. We report several statistics relevant to the obtained results: minimum, median, mean, maximum and standard deviation values.

Metric	Test	Min	Median	Mean	Max	SD
Mutation Score(%)	MT	18.51	94.73	86.30	100.00	19.66
	ATG	25.00	89.47	82.93	100.00	18.37
Coverage (%)	MT	63.63	100.00	96.33	100.00	8.08
	ATG	54.16	100.00	97.45	100.00	8.65
# Tests	MT	2.00	8.00	12.80	47.00	10.57
	ATG	2.00	4.00	7.42	31.00	7.35

A. *Fault Detection*

How does the fault detection of manual test suites compare with that of automatically generated test suites based on decision coverage? For all programs, as shown in Figure 1a, the mutation scores obtained by manually written test suites are higher in average with 3% compared with the ones achieved by automatically generated test suites. However, there is no statistically significant difference at 0.05 as the p-value is equal to 0.087 (effect size 0.600). Consequently, a larger sample size, as well as additional studies in different contexts, would be needed to obtain more confidence to claim that automatically created test suites are actually worse than manually created test suites.

The difference in effectiveness between manual and automated testing could be due to differences in test suite size. As shown in Figure 1c, the use of automated test generation results in less number of test cases than the use of manual testing (a difference of roughly 40%). To control for size, we generated purely random test suites of equal size as the ones manually created by industrial engineers (*Rand* in Figure 1a) and coverage-adequate test suites with equal size as manual test suites (ATG++ in Figure 1a). Our results suggest that fault detection scores of manually written test suites are clearly superior to random test suites of equal size, with statistically significant differences (effect size of 0.897). In addition, even coverage-adequate test suites with equal size as manual test suites are not showing better fault detection than the ones manually created. This shows that the effect of reduced effectiveness for automated test generation is not only due to smaller test suites. This is not an entirely surprising result. Our expectation was that manual test suites would be similar or better in terms of fault detection than automatically created test suites based on decision coverage. Industrial engineers with experience in testing IEC 61131-3 programs would intuitively write good test cases at detecting common faults. Our results are not showing any statistically significant difference in mutation score between manual test suites and COMPLETETEST-based test suites.

B. *Fault Detection per Fault Type*

To understand how automatically generated test suites can be improved in their fault detection capability, we examined if these tests suites are particularly weak or strong in detecting certain fault types. We concern this analysis to what type of faults were detected by both manual testing and COMPLETETEST. For each mutation operator described in Section III-D, we examined what type of mutants were killed by tests written using manual testing and COMPLETETEST. The results of this analysis are shown in Figure 2 in the form of box plots with mutation scores broken down by the mutation operator that produced them. There are some broad trends for these mutation operators that hold across all programs considered in this study. The fault detection scores for arithmetic (ARO), value (VRO) and comparison (CRO) replacement fault types are not showing any significant difference between manually

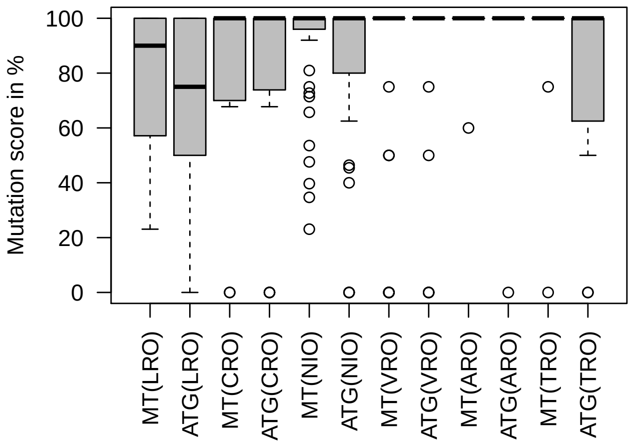


Fig. 2: Mutation analysis results per fault type.

created test suites and automatically generated test suites. On the other hand, test suites written using manual testing detect, on average, 12% more logical type of faults (LRO) than test suites generated automatically using COMPLETETEST. The increase is slightly similar for negation (NIO) and timer (TRO) replacement type of faults with manually written test suites detecting, in average, 13% more NIO and TRO fault types than automatically generated test suites. Overall, it seems that one of the reasons behind manual testing success, seems to do with its strong ability to kill mutants from certain operators.

To identify the reasons behind the differences in mutation score per fault type between manual testing and COMPLETETEST, we investigated deeper the nature of each mutation operator. For both the negation insertion and the timer replacement fault type it seems that COMPLETETEST with branch coverage as the stopping criterion, achieves a poor selection of test input conditions with too few test cases being produced; a certain input value that fails to kill the NIO and TRO type of mutant could have been made more robust with further test inputs. Logical replacement type of faults where an AND block is replaced by an OR block and vice versa tends to be relatively trivial to detect by both manual testing and COMPLETETEST. This comes from the fact that these faults are detected by any test cases where the inputs evaluate differently and the change is propagated to the output of the program. This does not mean that all logical faults can be easily detected. Consider an LRO type of mutant where OR blocks are replaced by XOR. The detection of this type of fault is harder, with manual test suites detecting 24% more LRO type of mutants where a logical block is replaced by XOR than test suites generated automatically. The detection in this case happens only with one specific test case that propagates the change in the outputs. It seems that manual testing has a stronger ability to detect these kind of logical faults than automated test generation because of its inherent limitation of only searching for inputs that are covering the branches of the program.

C. Coverage

As seen in Figure 1b, for the majority of programs considered, manually created test suites achieve 100% decision coverage. Random test suites of the same size as manually created ones achieve lower decision coverage scores (in average 61%) than manual test suites (in average 96%). The coverage achieved by manually created test suites is ranging between 63% and 100%. As shown in Table I, the use of COMPLETETEST achieves in average 97% decision coverage. Results for all programs show that differences in code coverage achieved by manual versus automatic test suites are not strong in terms of any significant statistical difference (with an effect size of 0.449 and a p-value of 0.192). Even if automatically generated test suites are created by COMPLETETEST having the purpose of covering all decisions, these test suites are not showing any significant improvement in achieved coverage over the manually written ones.

D. Cost Measurement Results

This section aims to answer RQ2 regarding the relative cost of performing manual testing versus automated test generation. The conditions under which each of the strategies is more cost effective were derived. The cost is measured in time (i.e., minutes) spent. For manual testing (MT) the total cost involves only human resources. We interviewed three engineers working on developing and manually testing TCMS software and asked them to estimate the time (in minutes) needed to create, execute, check the result and report a test suite. All engineers independently provided very similar cost estimations. We averaged the estimated time given by these three engineers and we calculated each individual cost (165 minutes in total on average). In addition, for automated test generation the total cost of performing automated test generation involves both machine and human resources. We calculated the cost of creating, executing and reporting test suites for each program, by measuring the time required to run COMPLETETEST and the time required for test execution. For the cost of checking the test result we used the average time needed by three industrial engineers to check the results using manual testing. The total cost for performing automated test generation was estimated to be 19 minutes on average. The cost of performing testing using COMPLETETEST is consistently significantly lower than for manually created tests; automatic generated tests have a shorter testing cost (146 minutes shorter testing time on average) over manual testing. Based on these results, we can clearly see that when using automatic test generation tools, the creation, execution and reporting costs are very low compared to manual testing. While these cost are low, the cost of checking the results is human intensive and is a consequence of the relative difficulty to understand each created test. A further detailed account of the cost evaluation can be found in the extended report [6].

V. THREATS TO VALIDITY

There are many tools (e.g., EVOSUITE [7]) for automatically generating tests and these may give different results. The use of these tools in this study is complicated by the transformation of IEC 61131-3 programs directly to Java or C, fact shown to be a significant problem [16] because of the difficulty to transform timing constructs and ensure the real-time nature of these programs. Hence, we went for a tool specifically tailored for testing IEC 61131-3 programs. To the best of our knowledge, COMPLETETEST is the only openly available such tool.

A possible risk on evaluating test suites based on mutation analysis is the *equivalent mutant* problem in which these faults cannot show any externally visible deviation. The mutation score in this study was calculated based on the ratio of killed mutants to mutants in total (including equivalent mutants). Unfortunately, this fact introduces a threat to the validity of this measurement.

The results are based on a case study in one company using 61 programs and manual test suites created by industrial engineers. Even if this number can be considered quite small, we argue that having access to real industrial test suites created by engineers working in the safety-critical domain can be representative. More studies are needed to generalize these results to other systems and domains.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we investigated, in an industrial context, the quality and cost of performing manual testing and automated test generation. The study is based on 61 IEC 61131-3 programs from a recently developed industrial control software and manual test suites produced by industrial professionals. Our results suggest that automated test generation can achieve similar decision coverage as manual testing performed by industrial engineers but in a fraction of the time. However, these automatically generated test suites are not showing better fault detection in terms of mutation score than manually created test suites. The fault detection rate between automated code coverage-based test generation and manual testing was found, in some of the published studies [8], [13], [21], to be relatively similar to our results. Interestingly enough, our results indicate that COMPLETETEST-based test suites might even be slightly worse in terms of fault detection compared to manual test suites. However, a larger empirical study is needed to statistically confirm this hypothesis. We found that there are more manually created test suites that are effective at detecting certain type of faults than automatically generated test suites. By considering generating test suites that are detecting these fault types one could improve the goals of automated test generation for industrial control software by using a specialized mutation testing strategy.

ACKNOWLEDGMENTS

This research was supported by The Knowledge Foundation (KKS) through the following projects: (20130085) Testing of Critical System Characteristics (TOCSYC), Automated Generation of Tests for Simulated Software Systems (AGENTS), and the ITS-EASY industrial research school.

REFERENCES

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[2] CENELEC. 50128: Railway Application: Communications, Signaling and Processing Systems, Software For Railway Control and Protection Systems. In *Standard Official Document*. European Committee for Electrotechnical Standardization, 2001.

[3] Reidar Conradi and Alf Inge Wang. *Empirical methods and studies in software engineering: experiences from ESERNET*, volume 2765. Springer, 2003.

[4] Kivanc Doganay, Markus Bohlin, and Ola Sellin. Search based Testing of Embedded Systems Implemented in IEC 61131-3: An Industrial Case Study. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013.

[5] Eduard Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. Automated Test Generation using Model Checking: an Industrial Evaluation. In *International Journal on Software Tools for Technology Transfer*. Springer, 2014.

[6] Eduard Enoiu, Adnan Čaušević, Daniel Sundmark, and Paul Pettersson. An Industrial Study on Automated Test Generation and Manual Testing of IEC 61131-3 Software. In *Technical Report, MDH-MRTC-313/2016-1-SE*. MRTC, Mälardalen University, 2016.

[7] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic Test Suite Generation for Object-oriented Software. In *Conference on Foundations of Software Engineering*. ACM, 2011.

[8] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. In *Transactions on Software Engineering and Methodology*. ACM, 2014.

[9] David Howell. *Statistical Methods for Psychology*. Cengage Learning, 2012.

[10] IEC. International Standard on 61131-3 Programming Languages. In *Programmable Controllers*. IEC Library, 2014.

[11] Laura Inozemtseva and Reid Holmes. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *International Conference on Software Engineering*. ACM, 2014.

[12] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010.

[13] Jeshua S Kracht, Jacob Z Petrovic, and Kristen R Walcott-Justice. Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites. In *International Conference on Quality Software*. IEEE, 2014.

[14] Younju Oh, Junbeom Yoo, Sungdeok Cha, and Han Seong Son. Software Safety Analysis of Function Block Diagrams using Fault Trees. In *Reliability Engineering & System Safety*, volume 88. Elsevier, 2005.

[15] Alessandro Orso and Gregg Rothermel. Software testing: a research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*. ACM, 2014.

[16] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. An Overview of Model Checking Practices on Verification of PLC Software. In *Software & Systems Modeling*. Springer, 2014.

[17] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *International Conference on Automated Software Engineering*. ACM, 2015.

[18] Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae. Empirical Evaluation on FBD Model-based Test Coverage Criteria using Mutation Analysis. In *Model Driven Engineering Languages and Systems*. Springer, 2012.

[19] Hendrik Simon, Nico Friedrich, Sebastian Biallas, Stefan Hauck-Stattelmann, Bastian Schlich, and Stefan Kowalewski. Automatic Test Case Generation for PLC Programs Using Coverage Metrics. In *Emerging Technologies and Factory Automation*. IEEE, 2015.

[20] András Vargha and Harold D Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. In *Journal of Educational and Behavioral Statistics*, volume 25. Sage Publications, 2000.

[21] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. Experience Report: How is Dynamic Symbolic Execution Different from Manual Testing? A Study on KLEE. In *International Symposium on Software Testing and Analysis*. ACM, 2015.