An Industrial Study on Automated Test Generation and Manual Testing of IEC 61131-3 Software

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

Abstract

The recent maturation of techniques and tools for automated test generation offers the opportunity to produce test suites with high code coverage in a matter of seconds. Nonetheless, it is not very well studied how such test suites compare to manually written ones in terms of cost and effectiveness. To a certain extent such comparisons have been made using open-source programs. However, evidence on how automated coverage-directed test generation could be used for industrial software is sparse. 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 the cost and effectiveness between manually and automatically created test suites. In particular, we measured the cost and effectiveness in terms of fault detection of test suites created using a coverage-directed automated test generation tool and test suites manually created by industrial engineers for an existing train control system. We used recently developed real-world industrial programs written in the IEC 61131-3 FBD, a popular programming language for developing safety-critical systems using programmable logic controllers. The results show that automatically generated test suites achieve similar code coverage as manually created test suites, 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 test suites more effectively detect logical, timer and negation type of faults, compared to automatically generated test suites. The results underscore the need to further study how manual testing is performed in industrial practice. We suggest some improvement opportunities for supporting the use of automated test generation tools in testing of industrial control software.

I. Introduction

Testing is an important activity in engineering of safety-critical control software. In certain application domains (e.g., the railway industry) engineering software requires certification according to safety standards [3]. These standards mandate the use of specification-based testing and recommends the demonstration of some level of code coverage on the developed software. In this way, a developer needs to check software conformance with the specification: each test case should contribute to the demonstration that a specified requirement has indeed been satisfied. To achieve a certain level of code coverage, the designed test cases, when fed to the system under test, should systematically exercise the implementation (e.g., covering all branches). Naturally, developers want their test cases to be of high quality: test cases should be cost-effective and good at detecting faults. To support developers in testing, 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 [10], [36], [28], [34] 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 [11], 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 [15], a popular language in safety-critical industry for programming control software [19], [30]. We have applied a state-of-the-art test generation tool for IEC 61131-3 software, CompleteTest [9], 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 the following main results:

1) 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.
2) 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.
3) Automatically generated test suites are significantly less costly in terms of testing time than manually created test suites. The use of automated test generation in IEC 61131-3 software development can potentially save around 90% of testing time.

These results point out important issues that need to be addressed in order to use automated test generation tools in testing safety critical control software. Investigating the quality of the test suites revealed the need for improving the detection of faults. We found that there are more manually created test suites that are effective at detecting certain fault types than the automatically generated test suites. By using these fault types in addition to structural properties as the coverage criterion used by an automated test generation tool, we could generate more effective test suites. This can be achieved by considering an automated approach to generate test suites that can detect these specific faults. We argue that based on the results of this study, automated test generation could potentially be improved and be used in industrial practice for IEC 61131-3 software development to aid manual testing.

II. Related Work

Software testing is an important verification and validation activity used to reveal software faults and make sure that the expected behavior matches the actual software execution. In the software testing literature [1] a *test case* is a test executing the program, corresponding to a set of test inputs and expected outputs. A set of test cases is called a *test suite*. In this way, a test suite is thus a finite number of test cases executing one after the other.

Implementation-based testing is usually performed at unit level to manually or automatically create tests that exercise different aspects of the program structure. To support software developers in testing of their programs, automated test generation has been explored in a considerable amount of work [26] in the last couple of years. Numerous techniques for automated test generation [10], [2], [36], [34], [39], [21] have been proposed in the last decade to complement manual testing. Many of these techniques mainly target object-oriented programs. Specifically, EVOSUITE [10] is a tool based on genetic algorithms, for search-based testing of Java programs. Another automated test generation tool is KLEE [2] which is based on dynamic symbolic execution and uses constraint solving optimization as well as search heuristics to obtain high code coverage.

In the context of developing safety-critical control software, IEC 61131-3 [19] 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 [18], [38], [17], [33], [9], [7] 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. [16] and Gay et al. [12] 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. For example, in the safety-critical control software domain, software testing processes are influenced by different safety standards mandating and recommending different test design techniques, and one might speculate that other factors may affect the test suite quality. This motivated us to investigate a thorough comparison between manual testing and implementation-based automated test generation.

There are studies investigating the use of both manual testing and automated implementation-based testing of real-world programs. Several researchers have performed case studies [37], [22], [31] and focused on already created manual test suites while others performed controlled experiments [11] with human participants manually creating and automatically generating test suites. In 2015, Wang et al. [37] compared KLEE-based test suites with already created manual test suites on several open source programs. They found that automatically generated test suites are able to achieve higher code coverage but lower fault detection scores with manual test suites being also better at discovering hard-to-cover code and hard-to-kill type of faults. Another closely related study done by Kracht el al. [22] used EVOSUITE on a number of open-source Java projects and compared those test suites with the ones already manually created by developers. EVOSUITE-based test suites achieved similar code coverage and fault detection scores to manually created test suites. In addition, Fraser et al. [11] performed a controlled experiment and a follow-up replication experiment on a total of 97 subjects. They found that automated test generation, and specifically the EVOSUITE tool, leads to high code coverage but no measurable improvement over manual testing in terms of number of faults found by developers. EVOSUITE was used in another study by Shamshiri et al. [31] which found that test suites automatically generated achieved higher code coverage than developer-written test suites and detected 40% out of 357 real faults from different open-source projects.

These results kindled our interest in studying how manual testing compares to automated implementation-based

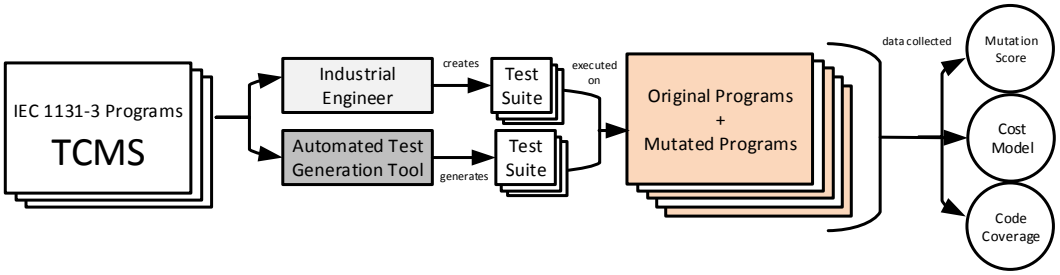


Fig. 1: Overview of the experimental method. For each program in the Train Control Management System (TCMS), test suites are collected, generated and executed on both the original and the mutated programs.

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

We designed a case study according to the method shown in Figure 1. 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 real industrial programs from a recently developed train control management system (TCMS), (iii) a cost model and (iv) a fault detection and code coverage metric. Consequently, 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?*
- *RQ2: Are automatically generated test suites less costly than tests suites manually created by industrial engineers?*

We considered an industrial project containing IEC 61131-3 programs and 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 decision coverage, the cost of performing testing and the mutation score as a proxy for fault detection. In order to calculate the mutation score, each test suite was executed on the mutated version of the original TCMS program to determine whether it detects the injected fault or not. This section describes in detail the case study procedure.

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

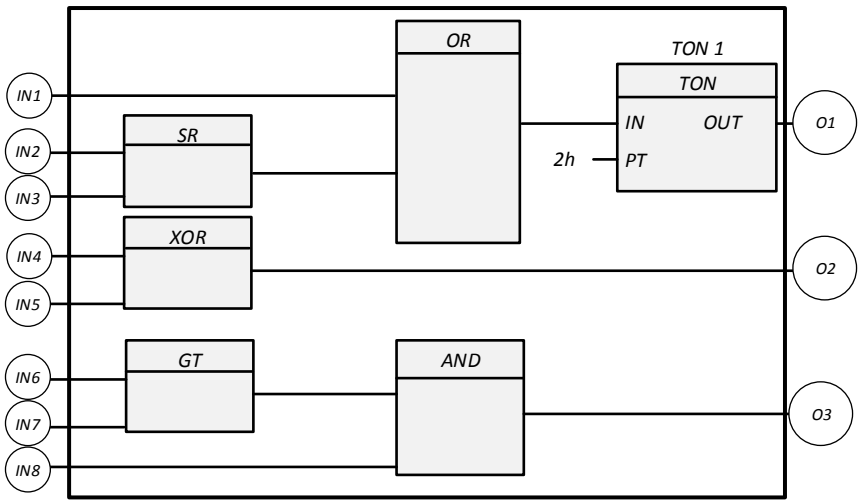


Fig. 2: A program with eight inputs and three outputs written using the IEC 61131-3 FBD programming language.

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. The system allows for integration of control and communication functions for high speed trains and contains all functions controlling the train. These functions are developed as software programs using an IEC 61131-3 graphical programming language named Function Block Diagram (FBD) [15].

A program running on a PLC [19] 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. As shown in Figure 2, predefined logical and/or stateful blocks (e.g., bistable latch SR, OR, XOR, AND, greater-than GT and timer TON) and connections between blocks represent the behavior of an FBD program. These blocks are supplied by the hardware manufacturer or defined by a developer. PLCs contain particular types of blocks, such as timers (e.g., TON) that provide the same functions as timing relays and are used to activate or deactivate a device after a preset interval of time. An FBD program is translated to a compliant executable PLC code. For more details on this programming language we refer the reader to the work of John et al. [19].

B. Test Suite Creation

As a baseline, we used manual test suites created by industrial engineers in Bombardier Transportation from a TCMS project delivered already to customers. A test suite created for an FBD program contains a set of test cases containing inputs, expected and actual outputs and timing information (i.e., Time parameter in the test suite is expressing timing constraints within one program).

Manual test suites were collected by using a post-mortem analysis [5] of the test data available. In testing IEC 61131-3 FBD 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 [3] 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 IEC 61131-3 FBD software (e.g., decision coverage which is also known as branch coverage). To the best of our knowledge, COMPLETETEST [9] is the only available automated test generation tool for IEC 61131-3 FBD software that produces tests for a given coverage criterion. As input for the test case generation, the tool requires a standard PLCopen XML implementation of the program under test. COMPLETETEST supports different coverage criteria with the default criterion being decision coverage. The tool stops searching for test inputs when it achieves 100% coverage or when a stopping condition is achieved (i.e, timeout or out of memory). COMPLETETEST uses the UPPAAL [23] model-checker as the underlying search engine and can be used both in a command-line and a graphical interface. A developer using COMPLETETEST can automatically generate test suites needed to achieve a maximum achievable code coverage for a given FBD program and can manually provide the expected outputs for a specific test case based on the defined behavior written in the specification. The tool will automatically run the tests and compare expected outputs with the actual ones, computed using the program under test.

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: 825 lines of IEC 61131-3 FBD code, 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 Code Coverage

Code coverage criteria are used in software testing to assess the thoroughness of test cases [1]. These criteria are normally used to assess the extent to which the program structure has been exercised by the test cases. In this study, code coverage is operationalized using the decision coverage criterion. For the TCMS system the EN 50128 safety standard [3] requires different code coverage levels (e.g., statement coverage). For the programs selected in this study and developed by Bombardier Transportation AB, engineers developing IEC 61131-3 software indicated that their certification process involves achieving high decision coverage. In the context of traditional programming languages (e.g., Java and C#), decision coverage is usually referred to as *branch coverage*. A decision coverage score is obtained for each individual test suite. A test suite satisfies decision coverage if running the test cases causes each decision in the IEC 61131-3 program to have the value *true* at least once and the value *false* at least once. Decision coverage was previously used by Enoiu et al. [9] to measure the thoroughness of code coverage for the specific structure of IEC 61131-3 programs. In this study we used our own tool implementation to collect the decision coverage achieved by each test suite.

E. 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 [6].

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. Just et al. [20] showed that if a test suite can detect or kill most mutants, it can also detect real software faults, thus providing evidence that the mutation score is a fairly good proxy for real fault detection ability.

In the creation of mutants we rely on previous studies that looked at commonly occurring faults in IEC 61131-3 software [25], [32]. We used these common faults in this study for establishing the following mutation operators:

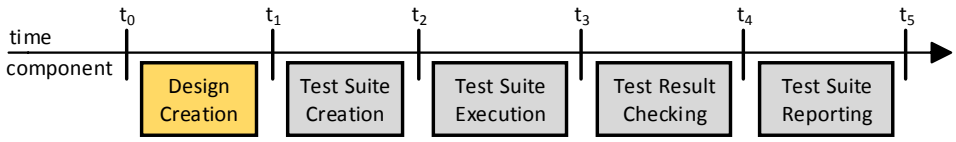


Fig. 3: Program Testing Cycle for IEC 61131-3 FBD software in TCMS.

- *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).

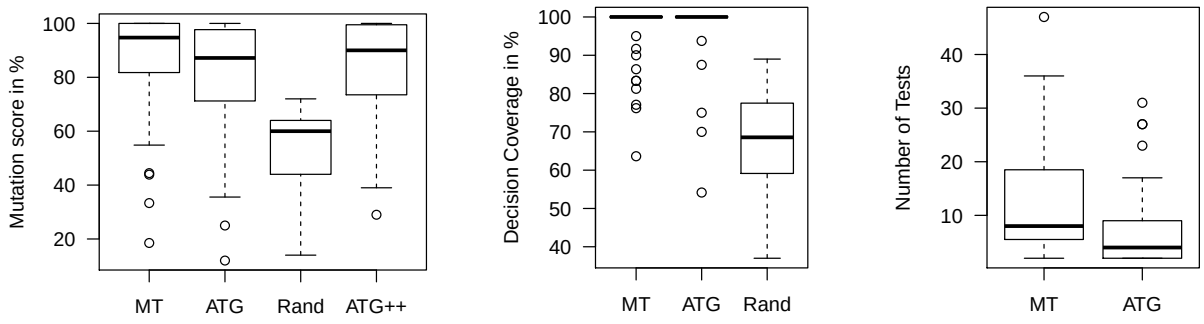
To generate mutants, each of the mutation operators was systematically applied to each program element wherever possible. In total, for all of the selected programs, 5161 mutants (faulty programs based on ARO, LRO, CRO, NIO, VRO and TRO operators) were generated by automatically introducing a single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

F. Measuring Efficiency

Many factors affect the cost of testing IEC 61131-3 FBD programs. According to Leung and White [24], testing involves two cost types: direct and indirect costs. A direct cost includes the implementer time for performing all activities related to unit testing and the machine resources such as the test infrastructure. Indirect costs include: the management of the testing process, test tool development, and execution history. Ideally, the 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. Figure 3 presents a timeline depicting the unit testing process for a single program. The EN 50128 safety standard requires that a software design is produced for each program and that each program is then tested using the design as the test oracle. For the TCMS system used in this case study, the test specification, execution and reporting are performed by the implementer of the IEC 61131-3 software. In

TABLE I: Constituent cost components and factors that cause the cost to vary when using both manual testing (MT) and automated test generation (ATG). T represents the number of test cases created in a single test suite, δ, ε, α and τ represent the average estimated time an engineer spends to manually create, execute, check and report a test case, respectively.

Component	MT	ATG
TEST CREATION (C_δ)	$\delta \cdot T$	t_e
TEST EXECUTION (C_ε)	$\varepsilon \cdot T$	
TEST RESULT CHECK (C_α)	$\alpha \cdot T$	$\alpha \cdot T$
TEST REPORTING (C_τ)	$\tau \cdot T$	t_r



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

Fig. 4: 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 by industrial engineers, 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.

the rest of the paper we will not use the cost of creating the design of the software because even if it varies with the size and characteristics of each program and requirement, the cost is constant between manual and automated test generation. A direct cost includes the implementer time for performing all activities related to unit testing and the machine resources such as the test infrastructure required for executing the unit tests. An indirect costs includes: the management of the unit testing process, the infrastructure costs for storing the tests, test tool development, and execution history.

In the cost model, we concentrated on the following components (mirrored in Table I): the TEST CREATION COST (C_δ) represents the cost of writing the necessary test suite, the TEST EXECUTION COST (C_ε) represents the cost of executing a test suite, TEST RESULT CHECK COST (C_α) represents the cost of checking the result of the test suite, and TEST REPORTING COST (C_τ) represents the cost of reporting a test suite. The cost model does not

TABLE II: 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

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 the cost components shown in Table I, 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. For automated test generation the cost of checking the test result (i.e., C_α in Table I) is related to human effort with the other costs (i.e., the combined generation and execution time t_e and the reporting time t_r when using COMPLETETEST) measured in machine time needed to compute the results. In this case study we consider that all cost components are related to the number of test cases. The higher the number of tests cases, the higher are the respective costs. We assume this relationship to be linear (δ, ε ,α and τ in Table I are factors representing the average time spend by an engineer in each cost component for a test case). 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.

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 4. Statistical analysis was performed using the R software [29]. In Table II 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 [14], a non-parametric hypothesis test for determining if two populations of data samples are drawn at random from identical populations. This statistical test was used in this case study for checking if there is any statistical difference among each measurement metric. In addition, the Vargha-Delaney test [35] was used to calculate the standardized effect size, which is a non-parametric magnitude test that shows significance by comparing two populations of data samples and returning the probability that a random sample from one population

TABLE III: For the mutation score we calculated the effect size representing the magnitude of the difference between manual testing (MT), automated test generation (ATG) and random testing (Rand). We also report the p-values of a Wilcoxon-Mann-Whitney U-tests with significant effect sizes shown in bold.

Measure	Method	Effect Size	p-value
Mutation Score	MT ATG	0.600	0.087
	MT Rand	**0.897**	< 0.001

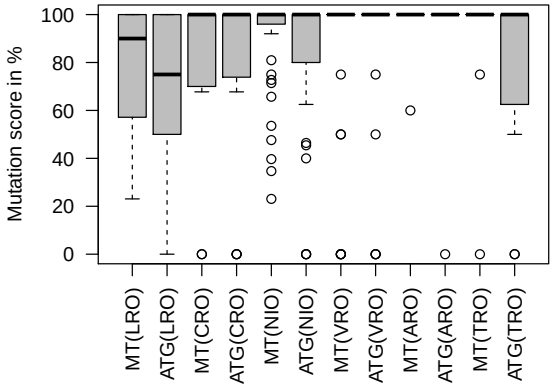
will be larger than a randomly selected sample from the other. According to Vargha and Delaney [35] statistical significance is determined when the effect size is above 0.71 or below 0.29.

A. Fault Detection

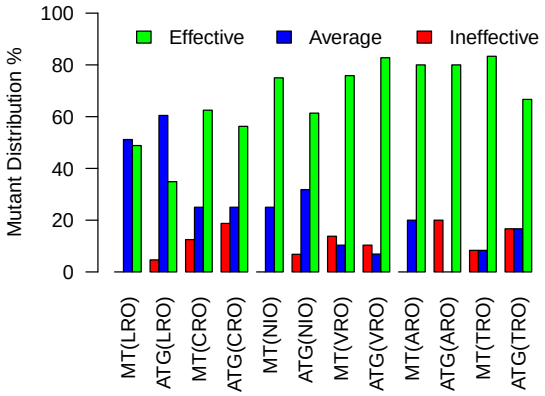
How does the fault detection of manual test suites compare with that of automatically generated test suites based on decision coverage? For all programs, as shown in Figure 4a, the mutation scores obtained by manually written test suites are higher in average with 3% compared with the ones achieved by 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 in Table III). 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.

> *Answer RQ1: Using automatically generated test suites does not yield better fault detection than using 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 4c, 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 4a) and coverage-adequate test suites with equal size as manual test suites (ATG++ in Figure 4a). 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 Table III). In addition, even coverage-adequate test suites with equal size as manual test suites (ATG++ in Figure 4a) 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 FBD 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.



(a) Mutation scores comparing manual testing (MT) against implementation-based automated test generation (ATG).



(b) Distribution of different types of mutants killed by manual versus automated test generation and categorized according to their effectiveness level.

Fig. 5: Mutation analysis results per fault type: LRO is the logical block replacement fault type, CRO is the comparison block replacement fault type, NIO is the negation insertion fault type, VRO is the value replacement fault type, ARO is the arithmetic block replacement fault type and TRO is the timer block replacement fault type.

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-E, 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 5a 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 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.

> *Manual test suites are detecting more logical, timer and negation type of faults than automatically generated test suites.*

We further classified all manual and automated test suites into three types as follows:(i) Effective: Test suites

that score above 95% on a specific fault type. Test suites achieving this kind of mutation score have been shown [13] to be very effective at finding faults. (ii) AVERAGE: Test suites that score between 5 and 95% on a specific fault type. (iii) INEFFECTIVE: Test suites that score lower than 5% on a specific fault type. Each test suite type has meaningful implications. For example, effective tests indicate strong fault detection effectiveness per fault type while ineffective tests indicate weak fault detection for a certain type of fault. The results of this analysis are shown in Figure 5b in the form of bar plots. For logical (LRO) faults, 48% of the manual test suites are effective at detecting this type of faults, an improvement of 13% over automatically generated test suites. We found more manual test suites being effective at detecting comparison (CRO) replacement and negation insertion (NIO) faults (an improvement of 6% and 13%, respectively) over automatically generated test suites. The increase is bigger for timer replacement (TRO) faults with manual testing having 16% more effective test suites than automated test generation. On the other hand, for value replacement (VRO) fault types, we found more effective automatically generated test suites than manually created test suites (an improvement of 6%). For arithmetic (ARO) faults, there is no difference between manual and automatically-generated effective test suites. In addition, in Figure 5b, we can observe that automated test generation results in more ineffective tests compared to manual testing when considering logical replacement (LRO), negation insertion (NIO) and timer replacement (TRO) fault types.

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 4b, 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 II, the use of COMPLETETEST achieves in average 97% decision coverage. Results for all programs (in Table IV) 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

TABLE IV: For code coverage we calculated the effect size representing the difference between manual testing (MT), automated test generation (ATG) and random testing (Rand). We also report the p-values of a Wilcoxon-Mann-Whitney U-tests with significant effect sizes shown in bold.

Measure	Method	Effect Size	p-value
Coverage	MT ATG	0.449	0.192
	MT Rand	**0.971**	< 0.001

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.

Overall, we confirm that the code coverage scores achieved by COMPLETETEST-based test suites are similar to the ones created manually by industrial engineers. While developing TCMS programs, engineers manually writing test suites have to demonstrate the use of specification-based testing while maintaining a certain degree of decision coverage. After discussions with three engineers developing IEC 61131-3 FBD software at Bombardier Transportation, functional specifications seem to be the main source of information for performing manual testing in our case study. From our results one question arises: Is high decision coverage achieved by test suites just a byproduct of performing specification-based manual testing? This underscores the need to study further how manual testing is actually performed in practice and what makes it so good at achieving high code coverage and fault detection.

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 variables presented in Section III-F are measured in time (i.e., minutes) spent, and their calculation depends on several cost components.

For manual testing (MT) the total cost C_{total} 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 (δ), execute (ε), check the result (α) and report a test suite (τ). All engineers independently provided very similar cost estimations. We averaged the estimated time given by these three engineers and based on the formulae in Table I we calculated each individual cost using the following average estimations: $\delta = 6.6$, $\varepsilon = 3.3$, $\alpha = 2.5$ and $\tau = 0.5$. The overall cost measures are reflected in Table V. 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 to execute each test case (i.e., t_e in Table I). For the cost

TABLE V: Cost measurement results for both manual testing and automated test generation using COMPLETETEST.

(a) Manual Testing

Cost (min.)	Min	Median	Mean	Max	SD
C_δ	13.2	52.8	84.5	310.2	70.5
C_ε	6.6	26.4	42.2	155.1	35.2
C_α	5.0	20.0	32.0	117.5	26.7
C_τ	1.0	4.0	6.4	23.5	5.3
C_{total}	25.8	103.2	165.2	606.3	137.8

(b) Automated Test Generation using COMPLETETEST

Cost(min.)	Min	Median	Mean	Max	SD
$C_\delta + C_\varepsilon$	0.003	0.012	1.120	10.900	3.185
C_α	5.000	10.000	18.563	77.500	18.597
C_τ	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
C_{total}	5.003	15.007	19.684	77.683	18.433

of checking the test result we used the average time needed by three industrial engineers to check the results using manual testing ($\alpha = 2.5$). The resulting cost measures are reflected in Table V.

Analyzing the cost measurement results is directly related to the number of test cases giving a picture of the effort per created test case. As seen in Table V, the cost of performing testing using COMPLETETEST is consistently significantly lower than for manually created tests; automatic generated tests have a shorter testing cost (145.5 minutes shorter testing time in average) over manual testing.

Answer RQ2: Automatically generated tests are significantly less costly in terms of testing time than manually created tests.

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.

V. DISCUSSIONS AND FUTURE WORK

Developers of safety critical control software use different test design techniques for testing their programs. We showed in Section IV the results obtained from a case study performed at Bombardier Transportation, a large-scale company focusing on development of trains. The programs considered in this study have been in development and are used in different train products all over the world. The obtained results could prove useful for both practitioners, tool developers and software testing researchers. To further explore the results of our case study we considered the implications for future work and the extent to which automated test generation can be used in the development of reliable systems.

To the best of our knowledge, our case study is the first to undertake experimental comparison between automated test generation and manual testing performed by industrial engineers on a safety-critical control software in terms of

fault detection, code coverage and cost. Our results indicate that, in IEC 61131-3 software development, automated test generation can achieve similar decision coverage to manual testing performed by industrial engineers. However, these automatically generated test suites are not yielding better fault detection in terms of mutation score than manually created test suites. The fault detection rate between automated implementation-based test generation and manual testing was found, in some of the published studies [11], [22], [37], 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.

Our study is the first to consider fault detection per fault type in an industrial context to understand how automatically generated test suites can be enhanced. From our results we highlight the need for improving the goals used by automated test generation tools for creating test suites. Code coverage-based test generation needs to be carefully complemented with other techniques such as mutation testing. 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 by using a specialized mutation testing strategy. This needs to be carefully considered in future studies.

As part of our study, we used cost measurements to estimate the efficiency of performing automated test generation. Our study suggests that automatically generated test suites are significantly less costly in terms of testing time than manually created test suites. The use of COMPLETETEST in IEC 61131-3 FBD software development can potentially save around 90% of testing time. The fact that automated test generation is faster, cheaper and possibly as good as manual testing, stands as a significant progress in aiding developers performing unit testing.

It is not commonly thought that automated test generation could eventually replace manual testing in industrial practice. This idea implies that automated test generation should be at least as effective and more efficient than manual testing for it to be considered ready to replace the manual effort. The overall suggestion, from this case study, is that we are not yet fully ready to solely use automated test generation as a replacement for manual testing of safety-critical embedded software. However, COMPLETETEST and similar automated test generation tools are capable of aiding in testing of IEC 61131-3 embedded software.

VI. THREATS TO VALIDITY

In our study we automatically seeded mutants to measure the fault detection capability of the written tests. While it is possible that faults created by industrial developers would yield different results, there is some scientific evidence [20] to support the use of injected faults as substitutes for real faults.

Another 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, as we do not know which mutants are equivalent). Unfortunately, this fact introduces a threat to the validity of this measurement.

There are many tools (e.g., KLEE [2], EVOSUITE [10], JAVA PATHFINDER [36], and PEX [34]) 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 [27] 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.

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.

We note here that the cost of automatically testing IEC 61131-3 FBD programs is heavily influenced by the human cost of checking the test result. We assumed that the average time of checking the results per test case for automated testing is the same as in the case of manual testing. In practice, this might not be the real situation. A test strategy, that requires every decision in the program to be executed, could contain test cases that are not specified. This might increase the cost of checking the test case result. If we assume that the cost of checking the result for automatic tests is different by one order of magnitude, α (in Table I) could be about ten times different in quantity. Hypothetically, the cost of automatically generating test suites could be slightly higher (with 20 minutes in average) than manually testing. A more accurate cost model would be needed to obtain more confidence to claim that CompleteTest actually is worse in terms of testing time than manual testing.

A final threat is that we did not use other stronger criteria than decision coverage for automatically generating test suites, such as MCDC and its variants [4], [1]. Intuitively, MCDC-based test suites would show better fault detection than decision coverage-based test suites. We did not consider these criteria in our study for two reasons. First, engineers from Bombardier Transportation AB testing the programs considered in this study suggested that their certification process recommends the use of decision coverage for assessing the thoroughness of their test suites. Second, a recent study [8] on the complexity of both safety-critical and non-critical Java programs has shown that MCDC and similar criteria are only needed on a small fraction of programs containing more complex branches or decisions. We therefore argue that using decision coverage for automatically generating test suites is a suitable and realistic scenario for many programming languages.

VII. Conclusions

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 real-world programs from a recently developed safety-critical control software and manual test suites produced by industrial professionals. We posed our first research question RQ1 (i.e., *Are automatically generated test suites able to detect more faults than test suites manually created by industrial engineers?*) in order to understand how good in terms of fault detection are automatically generated tests by comparing them with manual test suites. Our results do not confirm the hypothesis that automatically generated test suites are better at finding faults in terms of mutation score than manually created test suites. In addition, we showed that the effect of reduced fault detection for automated test generation is not only due to smaller test suites. Overall, it seems that manual testing shows a stronger ability to detect faults of certain type than automated

test generation. With regard to the second research question RQ2 (i.e., *Are automatically generated test suites less costly than test suites manually created by industrial engineers?*) we aimed to bring some industrial experimental evidence to the basic understanding of how automated test generation compare in terms of testing cost with manual testing. 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.

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] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation*, volume 8. USENIX, 2008.

[3] 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.

[4] John Joseph Chilenski and Steven P Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. In *Software Engineering Journal*, volume 9. IET, 1994.

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

[6] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. In *Computer*, volume 11. IEEE, 1978.

[7] 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.

[8] Vinicius H.S. Durelli, Jeff Offutt, Nan Li, Marcio E. Delamaro, Jin Guo, Zengshu Shi, and Xinge Ai. What to Expect of Predicates: An Empirical Analysis of Predicates in Real World Programs . In *Journal of Systems and Software*, volume 113. Elsevier, 2016.

[9] Eduard P 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.

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

[11] 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.

[12] Gregory Gay, Matt Staats, Michael Whalen, and Mats Heimdahl. The Risks of Coverage-Directed Test Case Generation. In *Transactions on Software Engineering*. IEEE, 2015.

[13] Moheb Ramzy Girgis and Martin R Woodward. An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria. In *Proceedings of the Workshop on Software Testing*. IEEE, 1986.

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

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

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

[17] Marcin Jamro. POU-Oriented Unit Testing of IEC 61131-3 Control Software. In *Transactions on Industrial Informatics,*, volume 11. IEEE, 2015.

[18] Eunkyoung Jee, Donghwan Shin, Sungdeok Cha, Jang-Soo Lee, and Doo-Hwan Bae. Automated Test Case Generation for FBD Programs Implementing Reactor Protection System Software. In *Software Testing, Verification and Reliability*, volume 24. Wiley, 2014.

[19] 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.

[20] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are Mutants a Valid Substitute for Real Faults in Software Testing? In *International Symposium on Foundations of Software Engineering*. ACM, 2014.

[21] Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing. In *International Conference on Automated Software Engineering*. IEEE, 2013.

[22] 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.

[23] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. In *International Journal on Software Tools for Technology Transfer*, volume 1. Springer, 1997.

[24] Hareton KN Leung and Lee White. A Cost Model to Compare Regression Test Strategies. In *Software Maintenance*. IEEE, 1991.

[25] 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.

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

[27] 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.

[28] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering*. IEEE, 2007.

[29] R-Project. R: A language and environment for statistical computing. http://www. R-project. org. The R Foundation for Statistical Computing, 2005.

[30] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. Control System Devices: Architectures and Supply Channels Overview. In *Sandia National Laboratories Sandia Report SAND2010-5183*. 2010.

[31] 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.

[32] 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.

[33] 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.

[34] Nikolai Tillmann and Jonathan De Halleux. Pex–White Box Test Generation for. net. In *Tests and Proofs*. Springer, 2008.

[35] 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.

[36] Willem Visser, Corina S Pasareanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. In *Software Engineering Notes*, volume 29. ACM, 2004.

[37] 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.

[38] Yi-Chen Wu and Chin-Feng Fan. Automatic Test Case Generation for Structural Testing of Function Block Diagrams. In *Information and Software Technology*, volume 56. Elsevier, 2014.

[39] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. Combined Static and Dynamic Automated Test Generation. In *International Symposium on Software Testing and Analysis*. ACM, 2011.