A Controlled Experiment in Testing of Safety-Critical Embedded Software

Eduard P. Enoiu*, Adnan Čaušević*, Daniel Sundmark*†, Paul Pettersson*
*Software Testing Laboratory, Mälardalen University, Västerås, Sweden
†Swedish Institute of Computer Science, Kista, Sweden

***Abstract*—In engineering of safety critical systems, regulatory standards often put requirements on both traceable requirements-based testing, and structural coverage on system software units. Automated test input generation techniques can be used to generate test data to cover the structural aspects of a program. However, there is no conclusive evidence on how automated test input generation compares to manual test generation, or how test case generation based on the program structure compares to specification-based test case generation.**

This paper aims at investigating automated and manual specification- and implementation-based testing of IEC 61131-3 **Function Block Diagram Code, a programming standard in many embedded safety critical software systems, measuring the efficiency and effectiveness in terms of fault detection. For this purpose, a controlled experiment was conducted, comparing both test suites created by a total of 23 industrial software engineering master students. The experiment participants worked individually on manually designing and automatically generating tests for two industrial programs. All tests created by each participant during the experiment were collected and analyzed by means of mutation score, decision coverage, test length and time required for creating, executing, and checking the test. We found that, when compared to implementation-based testing, specification-based testing yields significantly more effective test cases in terms of the number of faults detected. Specifically, specification-based tests more effectively detect comparison and value replacement type of faults, compared to implementation-based tests. On the other hand, implementation-based automated testing leads to shorter tests (up to 85% improvement) created in less time than the ones manually created based on the specification.**

***Index Terms*—automated testing, controlled experiment**

I. Introduction

The IEC 61131-3 language [4] is a programming standard for process control software, commonly used in the engineering of embedded safety-critical systems (e.g., in the railway and power control domains). Engineering of this type of systems typically requires a certain degree of certification according to safety standards. These standards pose specific requirements on testing (e.g., the demonstration of some level of structural coverage on the developed software). In our previous work, we have shown how to generate test input data achieving high coverage for a domain-specific language like IEC 61131-3 [7]. Generally, implementation-based testing techniques automatically generate a test suite (i.e., a set of tests) that, when fed to the system under test, systematically exercises the structure of a program (e.g, covering all decisions). However, there is little evidence on the extent to which such techniques effectively contribute to the development of reliable systems. More recent work [11] suggests that coverage criteria alone can be a poor indication of effectiveness of the testing process. Given that most of the safety standards in the safety-critical domain require some type of structural coverage, we seek to investigate the implications of using both specification-based testing and automated implementation-based testing.

In this study, we seek to compare the efficiency and effectiveness of testing programs written in IEC 61131-3 by comparing tests manually written by human subjects based on a specification, tests manually written based on the implementation, and tests produced with the help of an automated test input generation tool named CompleteTest[1]. Our research objective can be stated as follows:

To compare the efficiency and effectiveness of tests manually written based on a specification with implementation-based tests written manually or generated automatically.

To address this objective, an experiment was organized with master students enrolled in a software verification and validation course at Mälardalen University. Twenty-three master level students in industrial software engineering took part as subjects in a controlled experiment. The subjects were given two industrial IEC 61131-3 programs and were asked to construct tests manually based on a specification, and with the help of an automated implementation-based test generation tool. In addition, students were asked to manually create tests for covering the implementation. All tests created during the experiment were analyzed using the the following metrics: mutation score, decision coverage, test length, and time required for creating and executing the test.

Our study shows that manually created tests based on the specification are more effective, in terms of fault detection, than tests created based on the implementation either manually or with the help of an automated test generation tool. Specifically, compared to the implementation-based tests, specification-based tests more effectively detect comparison and value replacement type of faults. We also found that implementation-based automated test generation leads to shorter and faster tests than either manual specification-based testing or manual implementation-based testing.

[1]The tool is available for download at http://www.completetest.org/.

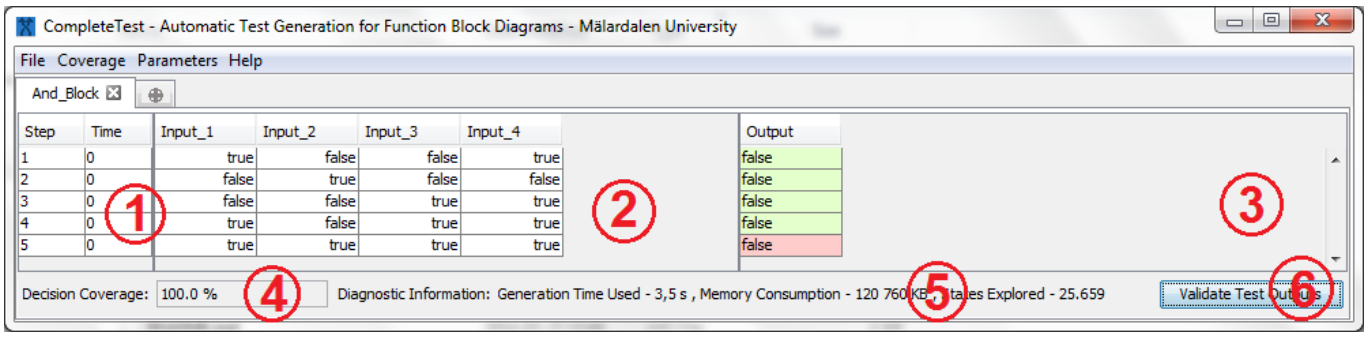


Fig. 1: Graphical Interface of CompleteTest

II. Preliminaries

A. Programmable Logic Controllers

Programmable Logic Controllers (PLC) are real-time software systems used in numerous industrial domains, i.e., nuclear plants and train systems. A program running on a PLC [15] executes in a cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs).

Function Block Diagram (FBD) [4] is an IEC 61131-3 PLC programming language, that is very popular in automation industry. A programmer uses graphical notations and describes the program in a data flow manner. Blocks and connections between blocks are the basis for creating an FBD program. These blocks are supplied by the PLC manufacturer, defined by the user, or predefined in a library. An application generator is utilized to automatically transform each program to a compliant program with its own thread of execution.

The motivation for using FBD as the target language in this study comes from the fact that it is the standard in many embedded systems, such as the ones in the railway and power domain. According to a Sandia National Laboratories study [21] from 2007, PLCs are widely used in a large number of industries with a global market of approx. $ 8.99 billion.

B. An Automated Testing Tool for IEC 61131-3

The automated test input generation tool used in this study is CompleteTest [7], which automatically produces tests for a given coverage criteria and an IEC 61131-3 program written using the FBD language. As input for the test case generation, the tool requires a PLCOpen[2] XML implementation of the program under test. Currently CompleteTest supports different logic 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 [17] model-checker as the underlying search engine. A generated test suite consists of a timed and ordered sequence of test inputs. As the main purpose of CompleteTest at present is to generate tests that satisfy a certain coverage criteria, the tool does not generate expected outputs. Expected outputs are provided manually in the user interface, shown in Figure 1, by a human tester.

CompleteTest can be used both in a command-line and graphical interface. For the purpose of this controlled experiment, the user interface shown in Figure 1 was used, since this is the exact interface used by an industrial end user. The interface shows several types of information presented to the user. The numbered points in Figure 1 represent:

1) steps and timing information regarding when the specific test input is provided to the program,
2) generated test inputs needed to achieve a maximum coverage for the given program,
3) editable area of the test outputs where the user can provide expected outputs for a specific set of test inputs based on a defined behavior in the specification,
4) percentage of the code coverage achieved by the generated test inputs,
5) diagnostic information with respect to the time spent on generating test inputs, memory usage and size of the state space, and
6) an action to compare expected outputs with the actual ones, computed by the program under test.

It should be noted that the generated test suite, containing test inputs, actual test outputs and, expected test outputs, is not intended to replace specification-based written tests, but to complement it with a structural perspective.

III. Experimental Design

The reporting of the controlled experiment is described in this section. Additional details on the study (e.g, instruction material and programs used) can be found at the experiment website created for storing the information needed for replication and reviewing[3].

A. Research Questions

We defined the following research questions as a starting goal to the experiment design:

[2]For more information on the PLCOpen standard we refer the reader to http://www.plcopen.org

[3]We provide all experimental material of this study at our website http://www.testinghabits.org/completetest/

RQ1: Does manually-written specification-based tests detect more faults than tests manually or automatically created based on the structure of the program under test?

RQ2: Are manually-written specification-based tests more costly to exercise than test cases manually or automatically created based on the structure of the program under test?

In addition to these questions we are interested in identifying improvement potentials for automated test input generation, such that it becomes a more efficient and effective technique.

Based on these research questions, our experiment handles two independent variables: the testing method used to solve the tasks (e.g., specification-based manual testing) and the object of study (i.e., program under test). The dependent variables of our experiment are: *effectiveness score* (i.e., measure of effectiveness in terms of faults detected), *testing duration* and *number of tests* (i.e., measures of efficiency).

B. Experimental Setup Overview

As part of the laboratory session, within the CDT414 software verification & validation course at Mälardalen University, the subjects were given the task of writing tests manually and writing tests with the aid of an automated test input generation tool. We present the design of this experiment around the subjects and the selected objects.

1) Study Subjects: As the study setting available to use was limited to a non-industrial environment and a physical space at Mälardalen University in Västerås, we restricted the experiment as part of a final-year master level course on software verification & validation. The subjects earned credits for participation but were informed that the final grade for the course would be influenced only by their written exam, and not by their performance in the experiment.

TABLE I: Study Objects: "LOC" refers to the number of XML code lines contained on each of the programs, "NOD" refers to the number of decision outcomes reported by COMPLETETEST, and "Mutants" is the number of faulty versions created with the mutation extension of the COMPLETETEST tool

Program	LOC	NOD	Inputs	Outputs	Mutants
X Trip	297	14	4	1	38
Fan Control	755	28	1	6	100

2) Object Selection: The objects of study were chosen manually, based on the following criteria:

- The programs should have a specification that is understandable and sufficiently rich in details for a tester to write executable tests.
- The programs should not be trivial, yet fully manageable to test within three hours of laboratory time. No domain-specific knowledge should be needed to understand the program.
- The programs should represent different types of real testing scenarios in different areas where the IEC 61131-3 standard is used.
- The programs should be developed by industrial engineers using the IEC 61131-3 FBD language.
- The COMPLETETEST tool should be able to automatically generate tests. This excludes programs for which the underlying search engine does not support the data types used (i.e., strings).

We investigated the industrial libraries provided by Bombardier Transportation AB, a leading, large-scale company focusing on development and manufacturing of trains and railway equipment, used in our earlier studies [7]. In addition, we searched through previous research studies on testing IEC 61131-3 software. We identified several candidate programs matching our criteria. We then assessed the relative difficulty of the identified programs by manually writing and automatically generating tests using COMPLETETEST. This process resulted in the identification of two suitable programs written in IEC 61131-3 FBD. Details on the programs used in the experiment can be found in Table I. We note here that an FBD program is written in a graphical environment that can be saved in an PLCOpen XML format[4]. The first program, is a function used in a nuclear power plant controlling the shutdown system. We used the program for calculating *th_X_Trip*, as taken from the paper by Jee et al. [14] (Figure 1 in [14]). In the rest of the paper, this program is named *X Trip*. The second program, named *Fan Control*, was selected from a train control management system developed by Bombardier Transportation AB in Sweden. The system is in development and uses processes influenced by safety-critical requirements and regulations including the EN 50128 standard [1] which requires different logic coverage levels (e.g., decision coverage). In addition, engineers developing IEC 61131-3 FBD software from Bombardier Transportation AB indicated that their certification process for programs like Fan Control involves achieving high decision coverage. In this paper we use decision coverage as the criterion for which tests are automatically generated.

C. Operationalization of Constructs

In this experiment, we compare the effect of using different test techniques on the code coverage, *efficiency* and *effectiveness* of the resulting tests. Code coverage and effectiveness can be operationalized using the following metrics:

- *Decision Coverage Score.* A coverage score indicator of the created tests is obtained for each individual solution. Using COMPLETETEST a decision coverage score indicator of the written tests can be obtained for each individual solution.
- *Mutation Score.* A mutation score is calculated by automatically seeding faults to measure the fault detecting

[4]While XML has no procedural statements and contains just structural declarations, it can be argued that FBD programs in XML require significant effort in software development and lines of code in an XML file should be counted and considered in the details of the selected objects.

capability of the written tests. Using the mutation extension of the CompleteTest tool we obtain a mutation score indicator of the created test suite for each individual solution.

FBD Code Coverage. Code coverage criteria are used in software testing to assess the thoroughness or adequacy of tests [2]. These criteria are normally used at the code level to assess the extent to which the program structure has been exercised by the tests. Out of the many criteria that have been defined, logic coverage [7] can be used to measure the thoroughness of test coverage for the structure of FBD programs. The flow in an FBD program is largely controlled by atomic Boolean connections called *conditions*, and by blocks called *decisions* made up of conditions combined with Boolean operators (not, and, or, xor). A condition can be a single Boolean variable, an arithmetic comparison with a Boolean value (e.g., $out1 > in2$), or a call to a function with a Boolean value, but does not contain any Boolean operators.

A test suite satisfies decision coverage if running the tests causes each decision in the FBD program to have the value *true* at least once and the value *false* at least once. In the context of traditional sequential programming languages, decision coverage is usually referred to as *branch coverage*.

Mutant Generation. 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 [5]. During the process of generating mutants, the mutation tool typically creates syntactically and semantically valid versions of the original program by introducing a single fault into the program. As exhaustive categorization of all possible faults that may occur when using the FBD language is impractical, we rely on previous studies that looked at commonly occurring FBD faults [18], [22]. By considering these specific faults we used the following mutation operators:

- *Logic Block Replacement (LRO)*: replacing a logical block with another block from the same function category (e.g., replacing an OR block with an XOR block).
- *Comparison Block Replacement (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 (ARO)*: replacing an arithmetic block with another block from the same function category (e.g., replacing an adder (ADD) block with a subtraction (SUB) block).
- *Negation Insertion (NIO)*: Negating a boolean input or output connection (e.g., an input variable in becomes NOT(in)).
- *Value Replacement (VRO)*: Replacing the value of a constant variable connected to a block (e.g., replacing a constant value ($const = 0$) with its boundary values ($const = -1$ and $const = 1$))).

To generate mutants, each of the mutation operators was applied to each program element whenever possible. In total, 138 mutants (faulty programs based on LRO, CRO, NIO and VRO operators) were created for both programs by automatically introducing a single fault into the correct implementation. We computed the mutation score using an output-only oracle against the set of mutants. For both programs, we assessed the fault-finding effectiveness of each test suite by calculating the ratio of mutants killed to total number of mutants.

Efficiency Metrics. In addition to fault finding effectiveness, we determined estimates of efficiency when writing tests. This is an important aspect to consider as it emphasize the practical usage of a specific test approach. We measured efficiency using the following indicators:

- *Duration:* Number of minutes spent on preparing and execution the tests. This surrogate measure of cost includes the following actions: preparing, writing, executing the tests, and checking the expect versus actual outputs.
- *Number of tests:* This metric is defined by the size of the test suite. Recall that each FBD program operates as a large loop receiving input and producing output. In this way, a generated test suite is thus a finite number of steps (tests), with each step corresponding to a set of test inputs.

D. Instrumentation

Two laboratory sessions were organized for the sake of the experiment: the first one for writing tests manually based on the specification (SMT) and the other one for implementation-based manual and automated testing (IMT and IAT respectively):

- *Session 1*. The subjects were given the task to completely test (to the extent they consider sufficient based on the specification) two industrial programs already implemented. The subjects were not grouped and the specification needed for testing the program was provided digitally and in written form.
- *Session 2*. The subjects were given the task to test (to achieve full decision coverage) the same two programs tested in session 1 by (i) manually creating tests to achieve full decision coverage and (ii) by automatically creating tests to achieve full decision coverage. The CompleteTest tool was used to automatically create tests. Before commencing session 2, a short tutorial of approximately 10 minutes on IEC 61131-3 and FBD syntax was provided to the subjects in order to avoid further problems with subjects' unfamiliarity with the concepts used. The tutorial included screencasts demonstrating programming and testing of FBD programs both manually and automatically using CompleteTest.

Detailed information about the problem and instructions were provided in each experiment session.

E. Data Collection Procedure

As part of the instructions, subjects uploaded their solutions using the Blackboard online learning platform at the end of each assignment. This way we had a complete log of subjects' activities. Data from both experiment sessions were then exported in a comma separated values (.csv) file format.

IV. Experiment Conduct

Once the experiment design was defined, the requirements for executing the experiment were in place. Session 1 and 2 were held two days after a theoretical lecture on specification-based testing and implementation-based testing respectively [5]. These practical sessions were performed one week apart from each other.

A. Sample

In total, we had twenty-three participants in the experiments. Initially, thirty participants showed up during each of the two sessions of the experiment. Before starting the experiment the participants were informed that their work would be used for experimental purposes. The participants had the option of not participating to the experiment and not allowing their data to be used in this way. The data provided by seven of the subjects had to be considered separately, as these participants produced the tests a long time after the experiment had finished. As these tests were produced outside the frame of the experiment we decided to discard this data from our experimental analysis.

B. Preparation

Prior to starting with the experiment, the subjects answered an initial set of questions using an online survey system in the Blackboard learning platform. In addition, before starting *Session 2* the first author of this paper initiated a tutorial demonstrating the use of CompleteTest and FBD language. A video of the presentation was made available on Blackboard for reference during the experiment.

To start the experiment, each subject entered the assignment page on Blackboard online platform, where the assignment information was provided. The subjects worked individually and during the experiment; the first two authors of this paper interacted with the participants to ensure that everybody had sufficient understanding of the involved tools without getting involved in the writing of the solution. All subjects used machines provided in the university premises of the same hardware configuration, booting Windows 7.

The experiment was fixed to three hours per lab session. To complete the assignments in both sessions, the subjects were given the same time to work on testing the programs according to the given instructions. For measuring the mutation score, the achieved decision coverage, the test length and the time required for creating and executing the tests produced manually and automatically, we provided a template to enforce the usage of the same reporting interface. By having a common template for test reporting we eased the process of performing the data collection and analysis.

To finish the assignment, we required the participants to send the produced tests through Blackboard online system as soon as they finished writing the tests. During the experiment the subjects were not allowed to directly communicate with others in order to avoid introducing any bias.

[5]The material taught in these lectures can be viewed in a video format at http://www.testinghabits.org/completetest/

TABLE II: Results of the experiment. For each metric we report several statistics relevant to the obtained results: minimum, median, mean, maximum and standard deviation values.

(a) X Trip

Metric	Method	Min	$Median$	$Avg.$	Max	SD
Mutation score (%)	SMT	68,42	97,37	93,94	100,00	8,99
	IMT	57,89	73,68	72,31	92,11	9,01
	IAT	63,16	71,05	72,54	84,21	5,66
Decision coverage (%)	SMT	92,86	100,00	99,15	100,00	2,27
	IMT	85,71	100,00	97,21	100,00	4,16
	IAT	100,00	100,00	100,00	100,00	0,00
Length	SMT	6,00	32,00	33,08	95,00	21,48
	IMT	3,00	3,0	5,13	16,00	3,32
	IAT	3,00	5,0	4,82	6,00	1,11
Duration (min.)	SMT	17,40	59,78	58,43	120,90	25,11
	IMT	10,07	27,57	30,05	54,08	12,18
	IAT	0,78	3,87	4,49	9,58	1,94

(b) Fan Control

Metric	Method	Min	$Median$	$Avg.$	Max	SD
Mutation score (%)	SMT	97,00	98,00	98,57	100,00	1,34
	IMT	80,00	84,00	88,76	100,00	7,31
	IAT	85,00	92,00	90,78	98,00	4,08
Decision coverage (%)	SMT	92,86	100,00	97,83	100,00	3,36
	IMT	78,00	100,00	96,73	100,00	6,18
	IAT	100,00	100,00	100,00	100,00	0,00
Length	SMT	8,00	10,00	10,04	17,00	1,94
	IMT	3,00	4,00	5,96	15,00	2,99
	IAT	5,00	6,00	5,83	7,00	0,78
Duration (min.)	SMT	11,35	29,42	31,85	61,85	12,75
	IMT	12,33	27,58	26,43	45,25	7,20
	IAT	2,05	3,67	4,10	9,30	1,93

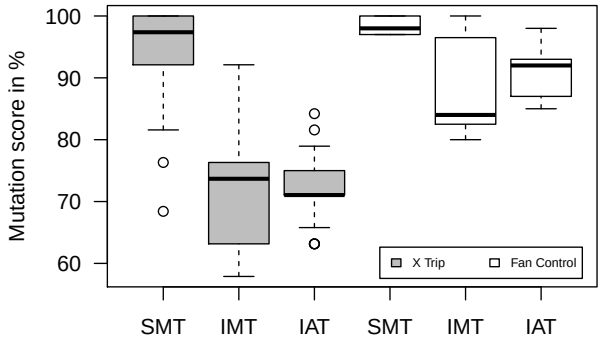
C. Data Collection

By using the Blackboard online system, we had a complete log of activities during the experiment with the ability to obtain the test suites. After each student finished their assignment, a complete solution was saved containing the tests and the timing information for each student solution. In addition, we separated the data provided by the twenty-three participants from their names.

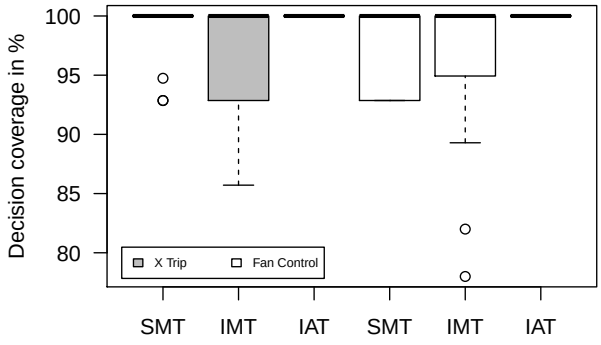
V. Experiment Analysis

This section provides an analysis of the data collected in the experiment. In the analysis, we follow the guidelines on statistical procedures for assessing randomized algorithms in software engineering provided by Arcuri and Briand [3].

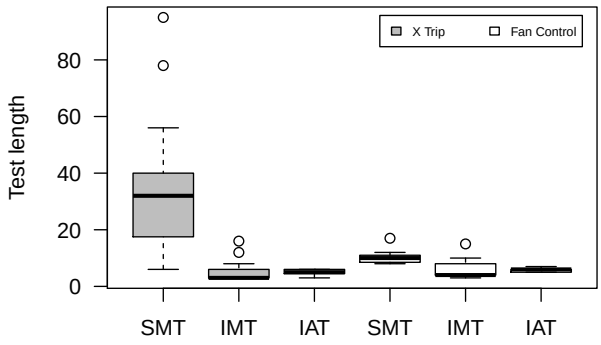
For each program under test and each testing technique (SMT, IMT, IAT), each subject in our study provided a test suite. These sets of tests were used to conduct the experimental analysis. For each test suite produced, we derived four distinct



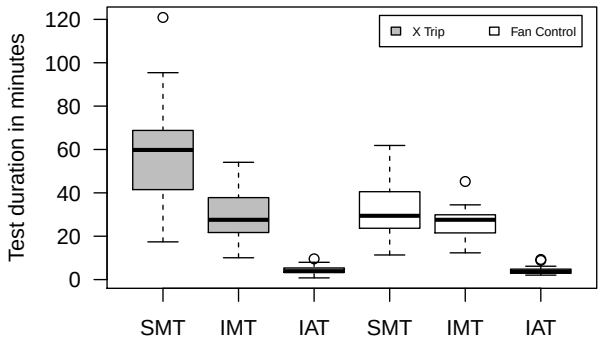
(a) Generated Mutants Killed



(b) Decision Coverage



(c) Test Length



(d) Testing Duration

Fig. 2: Test metrics comparing specification-based manual testing (SMT) against implementation-based manual testing (IMT) and implementation-based automated testing (IAT); 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.

metrics: mutation score, decision coverage, test length, and testing duration. These metrics form the basis for our statistical analysis towards the goal of answering the research questions. Statistical analysis was performed using the R statistical tool.

Table II and Table III list the detailed statistics on the obtained results, like minimum values, median, mean and standard deviation. The results of this study are also summarized in the form of boxplots in Figure 2.

Our observations are drawn from an unknown distribution. To evaluate if there is any statistical difference between each metric without any assumption on the distribution of the collected data, we use a Wilcoxon-Mann-Whitney U-test [13], a non-parametric hypothesis test for determining if two populations of samples are drawn at random from identical populations. This test is used for checking if there is any statistical difference among the three groups for each metric. In addition, the Vargha-Delaney test [25] was used to calculate the standardized effect size, which is a non-parametric effect magnitude test that shows significance by comparing two populations of samples and returning the probability that a random sample from one population will be larger than a randomly selected sample from the other. According to Vargha and Delaney [25] statistical significance is determined when the effect size measure is above 0,71 or below 0,29.

For each metric, we calculate the effect size of specification-based manual testing (SMT), implementation-based manual testing (IMT) and implementation-based automated testing (IAT). To this end, we report the p-values of these Wilcoxon-Mann-Whitney U-tests with statistical significant effect sizes shown in bold.

TABLE III: Results of the experiment. For each metric we calculated the effect size of each method compared to each other. We also report the p-values of a Wilcoxon-Mann-Whitney U-tests with significant effect sizes shown in bold.

(a) X Trip

Metric	Method	Effect Size	p-value
Mutation score (%)	SMT IMT	**0,900**	< 0,001
	IMT IAT	0,507	0,920
	SMT IAT	**0,911**	< 0,001
Decision coverage (%)	SMT IMT	0,607	0,066
	IMT IAT	0,339	< 0,001
	SMT IAT	0,439	0,040
Test Length	SMT IMT	**0,928**	< 0,001
	IMT IAT	0,426	0,341
	SMT IAT	**0,946**	< 0,001
Duration	SMT IMT	0,819	< 0,001
	IMT IAT	**0,958**	< 0,001
	SMT IAT	**0,958**	< 0,001

(b) Fan Control

Metric	Method	Effect Size	p-value
Mutation score (%)	SMT IMT	**0,848**	< 0,001
	IMT IAT	0,398	0,205
	SMT IAT	**0,923**	< 0,001
Decision coverage (%)	SMT IMT	0,511	0,859
	IMT IAT	0,359	0,004
	SMT IAT	0,359	0,004
Test Length	SMT IAT	**0,844**	< 0,001
	IMT IAT	0,366	0,087
	SMT IAT	**0,958**	< 0,001
Duration	SMT IMT	0,614	0,147
	IMT IAT	**0,958**	< 0,001
	SMT IAT	**0,958**	< 0,001

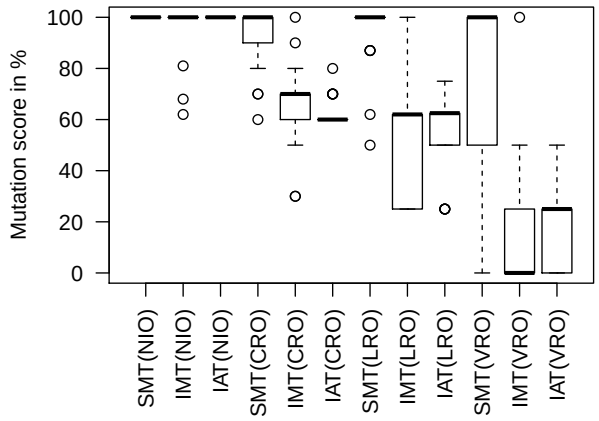
A. Fault Detection

For both programs, the fault detection scores of tests manually written based on the specification (SMT) were superior to tests written based on the implementation with statistically significant differences between SMT and IMT or IAT (effect size of over 0,900). For example, from Figure 2a we see that tests written for *X Trip* using SMT show an average fault detection of 93,94% compared to 72,31% for IMT and 72,54% for IAT. For *Fan Control*, SMT tests detect in average 98,57% of the faults versus 88,76% for IMT and 90,78% for IAT. with statistically significant differences between SMT and IMT or IAT (effect size of over 0,848). For both programs, the difference in fault detection between the SMT and IMT or IAT is statistically significant. None of the cases show any statistically significant differences in fault detection between IMT and IAT (at 0,05), as the lowest p-value is equal to 0,205 for *Fan Control*. A question emerging from these results concerns why tests written using specification-based manual testing are far better than the ones written using implementation-based testing.

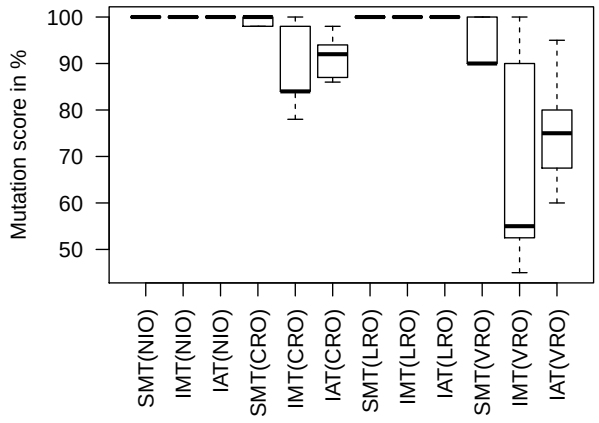
> *Answer RQ1: Specification-based manual testing yields significantly more effective test suites in terms of the number of faults detected than implementation-based manual or automated testing.*

For the purpose of shedding some light on this matter, we set out to investigate if these results could be explained by the fact that tests generated based on the implementation are particularly weak in detecting certain type of faults. More precisely, we examined what type of mutants were killed by tests written using SMT to tests written using IMT and IAT. For each of the mutation operators described in Section III, we examined the faults detected by each technique for both programs. The results of this analysis are shown in Figure 3 in the form of box plots. For the Fan Control program, both negation type of faults (NIO) and logical type of errors (LRO) are 100% detected by all three testing techniques. This shows that, for this program, all LRO and NIO injected faults are easily detected by every participant' test. On the other hand, tests written using SMT detect, on average, 7,9% more comparison type of faults (CRO) than tests produced using IAT. The increase is bigger for value replacement type of faults (VRO) with tests produced using SMT detecting, in average, 19,1% more faults than IAT. For the X Trip program, the situation is different, with SMT detecting more comparison (with 30% more faults in average), logical (with 38% more faults in average) and value replacement faults (with 51% more faults in average) than IAT. The exception, for X Trip program, is that NIO type of faults are detected by the majority of tests produced using all three testing techniques.

To further investigate the differences in fault detection for different mutation operators, we looked at one particular test suite automatically generated using IAT by one of the participants with CompleteTest tool. The generated test suite contains five tests achieving 81,58% mutation score with seven mutants not being detected. This test suite achieves 100% decision coverage on the non-mutated version of the X Trip program. The mutants that are not killed by this test suite are shown in Table IV. Interestingly enough the test suite achieves 100% decision coverage also on the mutated program except for M4 on which the test suite achieves just 92,85%



(a) X Trip



(b) Fan Control

Fig. 3: Mutation score comparing specification-based manual testing (SMT) against implementation-based manual testing (IMT) and implementation-based automated testing (IAT); NIO is the negation insertion operator, CRO is the comparison block replacement operator, LRO is the logical block replacement operator, and VRO is the value replacement operator.

decision coverage. There is an obvious reduction in achieved coverage of the generated test suite for M4 but not for the other mutants. To determine if this behavior stems from the generation of poor tests and what tests would improve the mutation score, we observed that one extra test targeting the detection of the value replacement fault $k\_xmin = -54$ in the X Trip program would detect M4 and, as a byproduct, all CRO faults (M0 to M1 in Table IV). In addition, three extra tests were created targeting the detection of M4 to M6 faults. With a final test suite of nine tests all faults were detected. In this case, the addition of four tests targeting the detection of the remaining faults has improved the fault-finding effectiveness. As a secondary result this particular example shows that for achieving better tests one should not solely rely on a decision coverage criterion alone.

TABLE IV: Mutants not killed by a test generated automatically using COMPLETETEST: the type of fault is represented by the mutant operator used to create it and the achieved decision coverage (DC) is obtained from running the generated test on each mutant.

ID	Mutant	Type	DC (%)
M0	$GE \rightarrow GT$	CRO	100
M1	$LE \rightarrow GE$	CRO	100
M2	$LE \rightarrow EQ$	CRO	100
M3	$k\_xmin = -54$	VRO	100
M4	$k\_xmin = 126$	VRO	92,85
M5	$OR \rightarrow XOR$	LRO	100
M6	$OR \rightarrow XOR$	LRO	100

B. Decision Coverage

As seen in Figure 2b, for both *X Trip* and *Fan Control* programs, the use of COMPLETETEST (IAT) entails 100% decision coverage (which is natural, as covering all decisions is the search objective for the test generation). Considering the effect sizes and the corresponding p-values in Table III, results for both programs are not strong in terms of effect size and we did not obtain any significant statistical difference for decision coverage. The results for both programs matched our expectations: even if IAT achieves tests for both programs satisfying 100% decision coverage, tests written using SMT achieved relatively high coverage (in average 99,15% for *X Trip* and 97,83% for *Fan Control*). This shows that, for the two programs studied in this experiment, SMT achieves high structural coverage for both programs. This is likely due to the relatively limited complexity of the studied programs. It is possible that a more complex program would yield greater coverage differences between tests written using SMT and IAT or IMT.

C. Test Length

As seen in Figure 2c, the use of IAT and IMT consistently results in shorter tests for both programs. This is perhaps most pronounced for IAT, for which we can see in average shorter test lengths with 42% to 85,5% when using COMPLETETEST tool (IAT) than SMT. Examining Table III, we see the same pattern in the statistical analysis: standardized effect sizes being higher than 0,844, with p-values below the traditional statistical significance limit of 0,05. The effect is the strongest for the Fan Control program with a standardized effect size of 0,958. It seems that a human tester, given sufficient time will create much more tests using SMT than IMT or IAT. This can be explained, for IAT, by considering that COMPLETETEST tool optimizes first for decision coverage, and secondary for test length. It is likely that specification-based manual testing (SMT) will in practice achieve more tests for a similar level of coverage.

Answer RQ2: Implementation-based automatically generated tests are shorter than specification-based manually created tests.

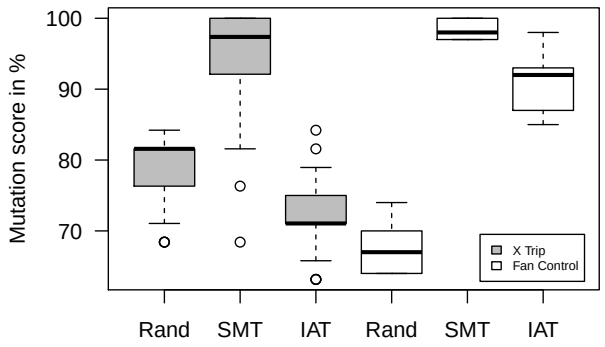
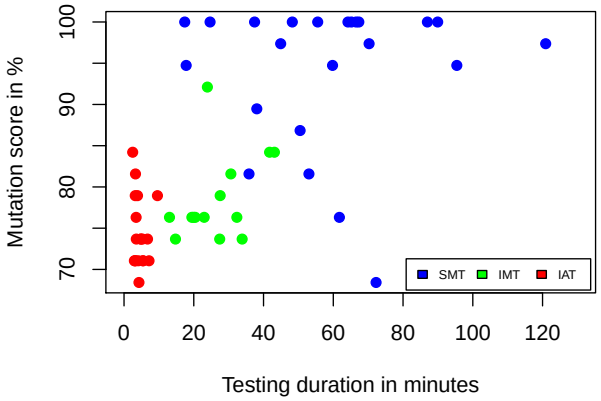


Fig. 4: Mutation score (each box spans from 1st to 3rd quartile, middle black lines mark the median and whiskers extend up to 1.5x the inter-quartile range) comparison between pure random tests (Rand), manual tests based on specification (SMT) and automatically generated tests (IAT).

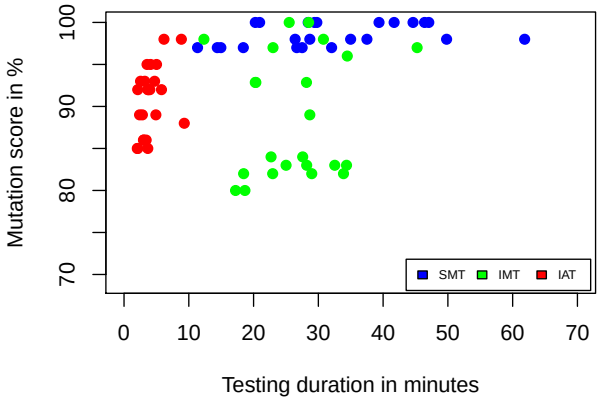
To investigate the effect of the test length on fault-finding effectiveness we produced purely random test suites of equal size as the ones created by the participants using SMT. In this way we controlled random tests for their length. The results are shown in Figure 4 as box plots. For all programs, random generated tests are less effective in terms of mutation score than tests written using SMT, in average by 15% to 31%. This indicates that tests produced using SMT are good indicators of test effectiveness. When comparing these random test suites with implementation-based automated tests (IAT), we can observe from Figure 4 that, for the Fan Control program, decision coverage alone is a better indicator of test suite effectiveness than random tests. For X Trip, random tests are more effective than tests generated using IAT. In this case decision coverage is not necessarily a good indicator of test suite effectiveness with factors other than coverage impacting the testing process.

D. Testing Duration

Analyzing testing duration is partially related to the test length analysis, but this metric gives a slightly different picture as the effort per created test suite is not necessarily constant over the different techniques under investigation. As seen in Figure 2d, the duration of writing tests using CompleteTest (IAT) is consistently significantly lower than for manually derived tests based on the specification (SMT). First, consider the data related to both programs (Figure 2d); CompleteTest assisted subjects have a shorter completion time (from 85,5% to 15,5% shorter in average) over specification-based manual testing (SMT) and implementation-based manual testing (IMT). Examining Table III, we observe that there is enough evidence to claim that these results are statistically significant with p-values below the traditional statistical significance limit of 0,05 and



(a) X Trip



(b) Fan Control

Fig. 5: The relation between cost and effectiveness for tests manually written based on the specification (SMT), tests manually written based on the implementation (IMT) and implementation-based tests generated automatically using CompleteTest (IAT).

a standardized effect size of 0,958.

> *Answer RQ2: Implementation-based automated tests are created and executed in less time than manually created specification-based tests or implementation-based tests.*

E. Cost-effectiveness Tradeoff

One important question in software testing is how the use of the investigated testing techniques affect the cost-effectiveness relation. In Figure 5 we show the relation between cost and effectiveness for tests written using SMT, tests written using IMT and tests generated with IAT. We use a proxy measure

for cost, duration time (preparation and execution time) and a surrogate measure for effectiveness, namely mutation score. Obviously for both programs the ideal scenario would be to have low values for duration time while achieving high mutation scores. As shown in Figure 5a, for the X Trip program, test suites derived using SMT provided a good mutation score (93,94% in average) and an inconsistent testing duration that spans from 17,40 minutes to 120,90 minutes. Test suites derived using IAT are significantly consistent in terms of testing duration (between 0,78 minutes to 4,49 minutes) while achieving lower mutation scores than SMT (72,31% in average) but similar to the effectiveness shown by test written using IMT. On the other hand, Figure 5b shows that the achieved mutation scores for SMT are very consistent even if this comes at the price of having expensive tests in terms of testing duration. As for X Trip, tests generated using IAT are cheap (completion time between 2,05 to 9,30 minutes) with fairly good fault-detection capability between 85,00% to 98,00% mutation score.

F. Limitations of the Study and Threats to Validity

External Validity. All of our subjects are master students and have limited professional development experience. This fact has been shown to be of somehow minor importance in certain conditions in a study by Höst et al. [12] with software engineering students being good substitutes in experiments for software professionals. Furthermore, in the light of our results regarding specification-based manual testing being better at fault detection than implementation-based manual or automated testing, we see no reason why the use of professionals in our study would yield a completely different result. Testing professionals with experience in FBD software would intuitively write better tests at detecting common FBD faults than tests written by student subjects.

We have used CompleteTest tool for automated test input generation. There are many tools for generating test inputs and these may give different results. Nevertheless CompleteTest is a tool based on a well know search technique, named model-checking, and its output in terms of tests is similar to the output produced by other test input generation tools for other programming languages, such as *EvoSuite* [8], *Java PathFinder* [26], and *Pex* [23].

Internal Validity. All subjects were assigned to perform specification-based testing in the first experiment session and after one week the same subjects were asked to perform implementation-based testing. This was dictated by the way the software verification and validation course was organized with lectures being followed by practical work. A potential bias is that participants can be expected to generate better tests in the second session. We controlled for that by putting the most mechanical process (i.e., implementation-based automated and manual testing) last, that is, the process that uses the least knowledge from the participant.

Construct Validity. In our study we automatically seeded faults to measure the fault detecting capability of the written tests. While it is possible that faults created by industrial developers would give different results, there is scientific evidence [16] to support the use of injected faults as substitutes for real faults.

Conclusion Validity. The results of the study were based on an experiment using 23 participants and two FBD programs. For each program all participants performed the study which is a relatively small number of subjects. Nevertheless, this was sufficient to obtain a statistical power showing an effect between manual testing and automated testing.

VI. Related Work

Among the various fields of research in software engineering, automated test input generation has gain a considerable amount of work [19] in the last couple of years. Implementation-based automated testing techniques are used for generating a set of input values for a program, typically with the final aim of fulfilling a certain coverage criteria or reachability property. When using this technique in practice, a human tester just needs to check that actual outputs for the test inputs are matching the expected outputs.

According to a survey [6] of testing techniques studies, published between 1994 and 2003 in top software engineering publication venues, only 16% of the techniques were evaluated using controlled experiments. Although controlled experiments using humans are not common in software testing, in recent years researchers have evaluated different techniques with users. Tillmann et al. [24], in their experience report, looked at transferring an automated test generation tool, named *Pex*, in practice and considered human factors as a central part of generating user-friendly tests by seeking feedback from users as well as setting realistic expectations on how the test generation works.

Ramler et al. [20] conducted a study, carried out with 48 master students, addressing the question of how automated testing tools compare to manual testing. In this specific experiment, they found that the number of faults detected by the automated testing tool was similar to manual testing. Recently Fraser et al. [10], [9] performed a controlled experiment and a follow-up replication experiment on a total of 97 subjects. They found that automated test input 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. Fault detection rate between automated testing and manual testing was found to be different from our study.

VII. Conclusions and Future Work

In this paper we investigated and compared the efficiency and effectiveness of specification-based manual testing, implementation-based manual testing, and implementation-based automated testing for embedded safety-critical software written using IEC 61131-3 FBD language.

The results of this experiment indicate that while the use of implementation-based automated testing yields high structural coverage and improves the length of the tests and the testing time over specification-based manual testing, this is not reflected in the ability of the written tests to detect more faults.

These results shows the need to take caution in selecting test suite generation objectives when using tools for automated test input generation, as well as continued research in establishing more effective test adequacy criteria. To achieve this, we need a better view of what aspects of software testing affect test effectiveness.

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. The authors would like to thank Jeff Offutt, Birgitta Lindström, Emil Alégroth, Jan Schroeder, and Shirin Tavara for their valuable comments on an early manuscript of this work.

REFERENCES

[1] 50128: Railway Application–Communications, Signaling and Processing Systems–Software for Railway Control and Protection Systems. 2001.

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

[3] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24, 2014.

[4] International Electrotechnical Commission. IEC International Standard 1131-3. *Programmable Controllers*, 2014.

[5] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11, 1978.

[6] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10, 2005.

[7] 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. *International Journal on Software Tools for Technology Transfer*, 2014.

[8] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Conference on Foundations of software Engineering*. ACM, 2011.

[9] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *International Symposium on Software Testing and Analysis*. ACM, 2013.

[10] 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.

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

[12] Martin Höst, Björn Regnell, and Claes Wohlin. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5, 2000.

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

[14] 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. *Software Testing, Verification and Reliability*, 24, 2014.

[15] K.H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010.

[16] 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 the Foundations of Software Engineering*, 2014.

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

[18] Younju Oh, Junbeom Yoo, Sungdeok Cha, and Han Seong Son. Software safety analysis of function block diagrams using fault trees. *Reliability Engineering & System Safety*, 88, 2005.

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

[20] Rudolf Ramler, Dietmar Winkler, and Martina Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2012.

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

[22] 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.

[23] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*. Springer, 2008.

[24] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *International Conference on Automated Software Engineering*. ACM, 2014.

[25] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[26] Willem Visser, Corina S Pasareanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29, 2004.