# Mutation-Based Test Generation for PLC Embedded Software using Model Checking

Eduard P. Enoiu[1], Daniel Sundmark[1], Adnan Čaušević[1], Robert Feldt[2], and Paul Pettersson[1]

[1] Software Testing Laboratory, Mälardalen University, Västerås, Sweden
[2] Blekinge Institute of Technology, Karlskrona, Sweden

**Abstract.** Testing is an important activity in engineering of industrial embedded software. In certain application domains (e.g., railway industry) engineering software is certified according to safety standards that require extensive software testing procedures to be applied for the development of reliable systems. Mutation analysis is a technique for creating faulty versions of a software for the purpose of examining the fault detection ability of a test suite. Mutation analysis has been used for evaluating existing test suites, but also for generating test suites that detect injected faults (i.e., mutation testing). To support developers in software testing, we propose a technique for producing test cases using an automated test generation approach that operates using mutation testing for software written in IEC 61131-3 language, a programming standard for safety-critical embedded software, commonly used for Programmable Logic Controllers (PLCs). This approach uses the Uppaal model checker and is based on a combined model that contains all the mutants and the original program. We applied this approach in a tool for testing industrial PLC programs and evaluated it in terms of cost and fault detection. For realistic validation we collected industrial experimental evidence on how mutation testing compares with manual testing as well as automated decision-coverage adequate test generation. In the evaluation, we used manually seeded faults provided by four industrial engineers. The results show that even if mutation-based test generation achieves better fault detection than automated decision coverage-based test generation, these mutation-adequate test suites are not better at detecting faults than manual test suites. However, the mutation-based test suites are significantly less costly to create, in terms of testing time, than manually created test suites. Our results suggest that the fault detection scores could be improved by considering some new and improved mutation operators (e.g., Feedback Loop Insertion Operator (FIO)) for PLC programs as well as higher-order mutations.

## 1 Introduction

Software testing is an important verification and validation activity used to reveal software faults and make sure that actual software behavior matches its expected behavior [2]. Safety-critical and real-time software systems implemented in *Programmable Logic Controllers* (PLCs) are used in many real-world industrial application domains. One of the programming languages defined by the *International Electrotechnical Commission* (IEC) for PLCs is the *Function Block Diagram* (FBD) language. In testing IEC 61131-3 FBD programs in the railway domain, the engineering processes of software development are performed according to safety standards and regulations [5]. As an alternative to manually testing software, a few techniques for *automated test generation* have been proposed [9, 4]. While high code coverage has historically been used as a proxy for the ability of a test suite to detect faults, recent results (e.g., [17]) indicate that code coverage may not be a good measure of fault detection effectiveness. As an alternative to coverage-based test generation, mutation testing has been proposed [7, 11]. In mutation testing, test cases are generated based on the concept of mutants– small syntactic modifications in the program, intended to imitate real faults. A set of test cases that can distinguish a certain program from its mutants is sensitive to faults, and it thus hypothesized to be good at detecting real faults (a hypothesis that has strong empirical support [21]). However, for domain specific languages used in embedded software development (i.e., IEC 61131-3), there is a lack of mature approaches and tools for performing mutation test generation.

In this paper, we describe and evaluate an automated mutation-based test generation approach for IEC 61131-3 embedded software. The main contributions of the paper are:

- An approach for mutation test generation of IEC 61131-3 programs using a model checker by combining all the mutants and the original program into a single combined model that is monitored dynamically.
- An evaluation of the approach in an industrial case study. The results show that mutation-adequate test suites are worse at detecting faults than manual test suites with the cost of performing mutation testing being consistently lower than the cost of manually testing IEC 61131-3 software.
- The identification of new mutation operators for mutation testing of IEC 61131-3 software. The reduction in fault detection between manual and mutation testing was attributed based on our analysis to an incomplete list of mutation operators for IEC 61131-3 software. We propose new operators simulating this kind of faults (e.g., Feedback loop Insertion Operator (FIO)).

The rest of the paper is organized as follows. Section 2 introduces PLC embedded software, automated test generation and mutation testing. Section 3 describes the approach for mutation test generation for IEC 61131-3 programs using a model checker. Section 4 explains the experimental method, while the results are provided and discussed in Section 5. Finally, Section 6 concludes the paper.

## 2 Background and Related Work

This paper describes a method for mutation testing for PLC embedded programs implemented in the IEC 61131-3 FBD language. In this section, we provide a

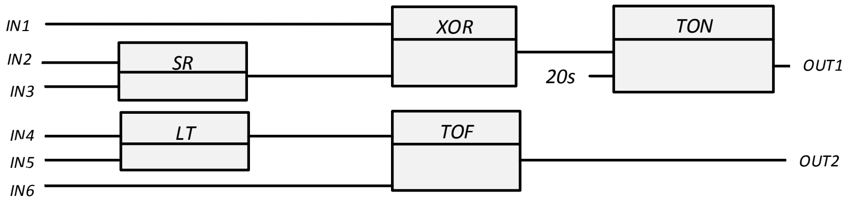


Fig. 1: An FBD program with six inputs and two outputs.

background on PLC embedded software, automated test suite generation and mutation testing.

### 2.1 PLC Embedded Software

Safety-critical embedded systems implemented using Programmable Logic Controllers (PLCs) are used in many industrial application domains such as electric, transportation, chemical, pharmaceutical, etc.. One of the programming languages defined by the *International Electrotechnical Commission* (IEC) for PLCs is the Function Block Diagram (FBD) language [16]. Programs developed in FBD are compiled into program code, which in turn is compiled into machine code by using specific engineering tools provided by PLC vendors. The motivation for using FBD as the target language in this study comes from the fact that it is the de facto standard in many industrial systems [26], such as the ones in the railway transportation domain. Programs running on a PLC execute in a loop, in which the iteration follows the *"read-execute-write"* semantics. FBD is popular because of its graphical notations and its usefulness in applications with a high degree of data flow between control components. As shown in Figure 1, predefined logical and/or stateful blocks (i.e., SR, XOR, TOF, LT and TON in Figure 1) and signals (i.e., connections) between blocks represent the behavior of an FBD program. The blocks are supplied by the hardware manufacturer or defined by a developer. PLCs contain particular types of blocks called timers (e.g., TON and TOF) that provide the same functions as timing relays in electrical circuits and are used to activate or deactivate a device after a preset interval of time. For more details on this programming language we refer the reader to the work of John et al. [20].

### 2.2 Automated Test Generation for PLC Embedded Software

In general, automated test generation has been explored in a considerable amount of work [25] in the last couple of years. Numerous techniques for automated test generation using code coverage criteria (e.g., [9, 4]) have been proposed in the last decade, since test suites can be created and executed with reduced human effort and cost. However, for domain specific languages used in embedded software development, contributions have been more sparse. For IEC 61131-3 software, a few automated test generation approaches [30, 18, 28] have been proposed in the last couple of years, but currently there is a lack of tool support. In our previous work, we developed an automated test input generation approach and tool named CompleteTest [8], which automatically produces test suites for a given coverage criterion and an IEC 61131-3 program written using the FBD language. CompleteTest supports different code coverage criteria with the default criterion being decision coverage.

### 2.3 Mutation Testing

Recent work [13, 17] suggests that coverage criteria alone can be a poor indication of fault detection in testing. To tackle this issue, researchers have proposed approaches for improving fault detection by using mutation analysis as a test criterion. Mutation analysis is the technique of automatically generating faulty implementations of a program 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. The execution of a test case on the resulting mutant may produce a different output as the original program, in which case we say that the test case **kills** that mutant. The mutation score is calculated using either an output-only oracle (i.e., strong mutation [29]) or a state change oracle (i.e., weak mutation [15]) against the set of mutants. For all programs, one needs to assess the fault-finding effectiveness of each test suite by calculating the ratio of mutants killed to total number of mutants. When this technique is used to *generate* test suites rather than evaluating existing ones, it is commonly referred to as *mutation testing* or mutation-based test generation. Despite its effectiveness [21], to the best of our knowledge, no attempt has been made to propose and evaluate mutation testing for PLC embedded software written in the IEC 61131-3 FBD programming language. This motivated us to develop an automated test generation approach based on mutation testing targeting this type of software.

## 3 Mutation Test Generation for PLC Embedded Software

Within the last decade *model-checking* has turned out to be a useful technique for generation of test cases from models [10]. In this paper, we describe an approach to automatically generate test suites using a model checker based on mutation testing for PLC embedded software. Overall, the approach is composed of the following steps, mirrored in Figure 2:

1. Mutant Generation. This first step (described in detail in Section 3.1) entails systematically making small syntactic changes (mutants) to a program based on a set of predefined operators (e.g., mimicking programming errors). The output of this step is a set of replicas of the original program, each with one inserted mutant.
2. Model Aggregation. The second step (described in detail in Section 3.2) is used for combining a program and the set of mutants into a single model.

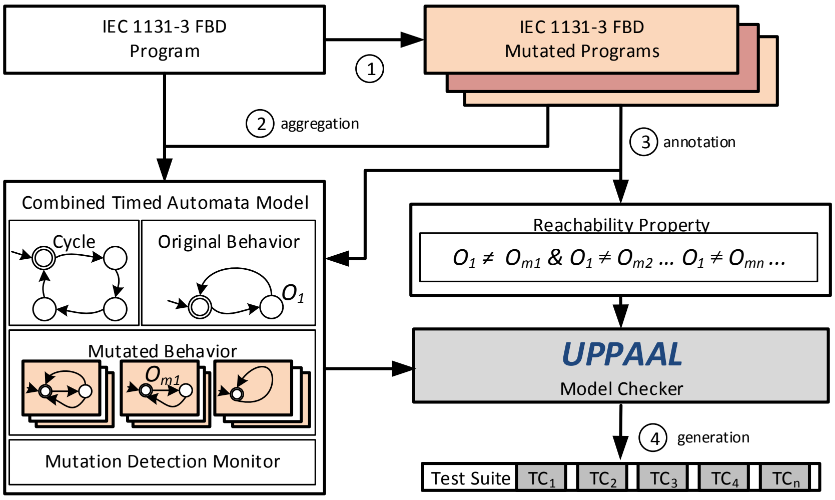


Fig. 2: Overview of mutation testing for IEC 61131-3 FBD programs.

The output of this step is a model containing the original structure and behavior of the program together with all inserted mutants.

3. Mutant Annotation. The third step (described in Section 3.3) involves the annotation of the combined model with instrumentation instructions for the detection of each mutant. This means that the mutation detection monitor is used to record the mutant execution and detection, thus for all mutants a property is created for checking the detection of mutants.
4. Test Suite Generation. The fourth step (described in Section 3.4) requires the use of the Uppaal model checker [22] to generate a set of test cases satisfying the detection of mutants by using the model checker's ability to export abstract traces witnessing a submitted property.

### 3.1 Mutation Generation

To facilitate mutation testing, we begin by generating mutated versions of the original program. The mutation generator parses a given program and processes the structural elements for performing mutations. In particular, for each mutation operator, the program is traversed invoking the corresponding mutation function at all possible locations, each mutation resulting in a separate mutant version of the program. For the creation of mutants, we rely on previous studies that looked at commonly occurring faults in IEC 61131-3 software [23, 27]. We used these common faults in this study for establishing the following mutation operators:

- *Logic Block Replacement Operator (LRO)* replaces a logical block with another block from the same function category (e.g., replacing an XOR block with an OR block),
- *Comparison Block Replacement Operator (CRO)* replaces a comparison block with another block from the same function category (e.g., replacing a Less-Than (LT) block with a Less-or-Equal (LE) block),
- *Arithmetic Block Replacement Operator (ARO)* replaces 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)* negates an input or output connection (e.g., an input variable IN1 becomes not(IN1)),
- *Value Replacement Operator (VRO)* replaces a value of a constant variable connected to a block (e.g., replacing a constant value ($const = 20s$) with its boundary values (e.g., $const = 19s$ and $const = 21s$)), and
- *Timer Block Replacement Operator (TRO)* replaces 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).

These mutation operators are systematically applied to the entire program (i.e., blocks, variables, constants, connections) and thus resulting in a set of mutants, each simulating one syntactic change.

### 3.2 Model Aggregation

We start the model aggregation step with the translation of a program and its set of mutants to a timed automata representation. We have shown in a previous study [8] how the mapping of an IEC 61131-3 program to timed automata is implemented. Timed automata, introduced by Alur and Dill [1], were chosen because there is an already existing formal semantics and tool support for simulation and model-checking using UPPAAL [22] and automated test generation using COMPLETETEST [8]. A timed automaton is a standard finite-state automaton extended with time (i.e., real-valued clocks are used for measuring time progress). A model in UPPAAL consists of a network of processes that are composed of locations. Transitions between these locations define how the model behaves. The semantics of a timed automaton $A$ is defined in terms of a state transition system, where the state of $A$ is defined as a pair $(l, u)$, where $l$ is a location (i.e. node) and $u$ is a clock assignment. A state of $A$ depends on its current location and on the current values of its clocks. A network of timed automata $B_0 \parallel ... \parallel B_{n-1}$ is a parallel composition of $n$ timed automata over synchronization functions (i.e., $a!$ is correlative with $a?$). Further information on timed automata can be found in [1]. In our previous work [8] we showed that an IEC 61131-3 FBD program can be transformed to a formal representation containing both its functional and timing behavior. In this study, the model aggregation is using this already developed translation for obtaining the model needed for running mutation-based test generation. Let $M$ be a finite set of of mutants, each of which contains one syntactic change in the original program $P$. The model aggregation step is applied as follows:

- Create a timed automaton $P$ corresponding to the original FBD program, and construct the structure of the program representing the set of blocks $b_n$, set of signals $s_m$ and set of variables $v_p$ in `P`: $b_1 \parallel ... \parallel b_n$, $s_1 \parallel ... \parallel s_m$ and $v_1 \parallel ... \parallel v_p$.
- For each mutant $m_i$ in $M$, created by changing a block, signal or variable in $P$, create a duplicate version of it (e.g., $b_{11}$ is a duplicate of $b_1$) having a different identifier and output than the original. This duplicate version has an *interface*, consisting of a name identifier. In addition, this duplicate version contains the same inputs as the original behavior, but different output variables and internal parameters in case of a mutated block. The interface is used to access both the block behavior and its duplicated version.
- Create a supervision automaton that executes each block and its mutants according to the order of execution. The execution order `N` is automatically defined according to the general rules included in the IEC 61131-3 standard [16]. This predetermined order directly dictates the data dependency in a program. Basically, each mutated entity executes in parallel with its original counterpart.

As a result of the model aggregation step we consider that the combined model is a closed network of timed automata. This model, briefly shown in Figure 2, contains four processes, two modeling the program and its mutants and the other two supervising the overall execution and monitoring the mutant detection. To show an example of an aggregated model cycle scan, different actions are executed: `read(IN)` for reading input variables, `write(OUT)` for updating the output variables, and `write(OUT(`$m_i$`))` for updating the duplicated output variables corresponding to each mutant $m_i$. When the execution order holds, the input variables are updated and the execution continues to the next block.

### 3.3 Mutant Annotation

Informally, our approach is based on the idea that in order to kill all mutants of a specific program, it would be sufficient to (i) annotate the mutants in an FBD program by adding a mutation detection monitor, (ii) formulate a reachability property for the mutation score (i.e., what portion of the existing mutants have been killed), and (iii) find a path from the initial state to some state where the mutation score is 100%. Thus, using auxiliary variables, we annotate the aggregated model such that a condition describing whether a single mutant is killed or not can be expressed.

For annotation, it should be noted that there are different interpretations of how to implement mutation analysis. The most common implementation, called strong mutation deals with the comparison of the original and mutated program outputs at the end of the execution cycle. Another way is weak mutation, which compares the state of the program immediately after the execution of the mutated part of the program. As these implementations can be useful in their different interpretation of mutation analysis, our approach employed both approaches.

**Weak Mutation.** A mutant is *weakly* killed in an FBD program if it leads to a block output change (i.e., block infection) compared to the original program behavior. For each mutation operator we define a detection monitor that precisely describes the decision that leads to a change in block output. In model checking we require a reachability property and a mutant detection monitor that guides the search towards detection. We define this weak mutation monitor for individual mutation operators. For each mutant $m_i$ in $M$, where $M$ is the entire set of mutants, there is a weak mutation monitor $wm_i(M)$ that looks at the block output change; if $wm_i(M)$ is 1 then $m_i$ is detected. Using a model checker, the aim of weak mutation testing is to achieve a state where all mutants are killed with respect to the block output change. For generating tests for weak mutation we represent the test obligations over a set of variables monitoring the original behavior and its mutants as a reachability property.

**Strong Mutation.** Weak mutation testing for an FBD program results in a test suite where an internal block is infected; however, a change in block output does not necessarily propagate to an observable program output. Using a model checker, we propose to propagate the mutated behaviors to the output of the program using additional data variables and signals and monitor the change in output using a strong mutation monitor. For each mutant $m_i$ in $M$, where $M$ is the entire set of mutants, the output of each mutant is propagated to the depended blocks until it reaches the program output. There is a strong mutation monitor $sm_i(M)$ that looks at the program output change; if $sm_i(M)$ is 1 then $m_i$ is detected. Using a model checker, the aim of strong mutation testing is to achieve a state where all mutants are killed with respect to the program output change. In our scenario, a mutant is killed if there exists a path in the model such that a test input shows that the mutated program output differs from the output of the original program.

### 3.4 Test Generation

In order to generate a test suite for mutation testing of FBD programs using Uppaal, we make use of Uppaal's ability to generate traces witnessing a submitted reachability property. A trace produced by the model checker for a given reachability property defines the set of actions executed on an FBD program which in our case is considered the system model $fbd$. An example of a diagnostic trace has the following form $(fbd_0) \xrightarrow{t_1} (fbd_1) \xrightarrow{t_2} ... \xrightarrow{t_n} (fbd_n)$, where $(fbd_k)$ are states of the combined model and $a_k$ are either internal synchronization actions, time-delays or `read!`, `execute!`, and `write!` global synchronizations. Test cases are obtained by extracting from the test path the observable actions `read!` and `write!` as these actions contain updates on input and output variables. In summary, the output of this step is a set of ordered test cases containing inputs, actual outputs and timing information (i.e., the time parameter in the test suite is expressing timing constraints within one program).

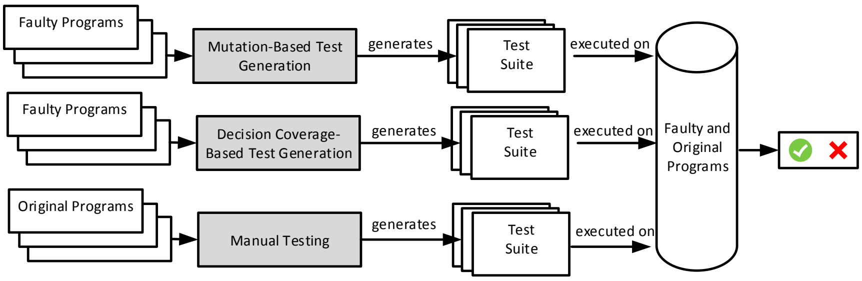


Fig. 3: Overview of the experimental setup used to perform the case study.

## 4 Experimental Evaluation

In order to evaluate the proposed mutation test generation technique, we designed an industrial case study. In particular, we aimed to answer the following research questions:

- *RQ1: Does mutation adequate test suites detect more faults than tests suites manually created by industrial engineers or automatically created test suites based on decision coverage?*
- *RQ2: Are mutation adequate test suites less costly than tests suites manually created by industrial engineers or automatically created test suites based on decision coverage?*

The case study setup is shown in Figure 3. From a high level view we started the case study by collecting: (i) a set of real industrial programs from a recently developed train control management system (TCMS), and (ii) manual test suites created for the above programs by industrial engineers. The studied programs were already thoroughly tested and are currently used in a set of operational trains. For all programs, test suites were also generated for weak mutation, strong mutation and decision coverage (as detailed below).

In order to measure fault detection, realistic faulty versions of the programs under test are required. However, the data set did not contain any information about what faults occurred during development, as Bombardier Transportation AB does not keep any such data in a format that could be directly collected post-mortem at this level of testing. To overcome this issue, several engineers from Bombardier were asked to manually create a number of faults for the programs considered in this study. We obtained faults from engineers at Bombardier Transportation manually introducing relevant faults in some of the programs considered in this study. Since mutation-based test generation is using an existing program implementation to guide the search, we automatically generate all tests suites using the seeded faults instead of the original programs. This corresponds to the realistic situation where an engineer has made a fault located in the program to be tested. In summary, we used a TCMS system containing

61 programs provided by Bombardier Transportation AB. These programs contained on average per program: 828 lines of IEC 61131-3 FBD code, 22 decisions (i.e., branches), 11 input variables and 5 output variables.

**Manually Seeding Faults.** For the TCMS programs, we provided four engineers working at Bombardier Transportation AB, who were not involved with the study with a document on doing fault seeding together with all the 61 programs. We asked each engineer to seed faults into the set of programs; we followed a specific fault seeding procedure using the IEC 61131-3 programming tools the engineers are using for developing the programs and instructed them to insert faults that were as realistic as possible. In particular, we instructed the engineers to insert any number of relevant faults, based on their experience, in the set of programs we provided as a TCMS project. We specifically instructed them to try to insert multiple faults in the same program one at the time and seed faults in at least ten programs from the total of 61. To avoid any misunderstanding, the fault seeding procedure document included information about the type of faults we were interested in: any fault that they might have encountered in their experience, as long as the interface (i.e., inputs and outputs) remained the same. This includes, but is not limited to, faults associated with variables, blocks, connections and constants. The fault seeding procedure resulted in 77 faults, versions of 33 (out of 61 in total) original programs containing a single fault (i.e., each fault contained one or more changes in the program). Each of the collected and generated test suites was executed on each of the faulty versions and its original counterpart so that a fault detection score could be calculated. Practically, each faulty variant contained one fault that had been manually seeded. A fault was considered to be detected by a test suite if the output from the faulty program differed from that of the original program.

**Test Generation** For each faulty program, we ran mutation and decision-coverage test generation ten times using a random-depth-first search (RDFS) strategy with random seed (i.e., test suites are varying from run to run), each test generation run with a stopping time limit for the search of 10 minutes. The stopping criteria for the search is three-fold: achieving 100% mutation score, reaching the time limit of 10 minutes, or getting a memory exception. We chose a time limit of 10 minutes for the sake of this experiment. In addition, we used manual test suites created by industrial engineers in Bombardier Transportation from a TCMS project delivered already to customers. Manual test suites were collected by using a post-mortem analysis of the test data available. The test suites collected in this study were based on functional specifications expressed in a natural language. Practically, we considered the original TCMS programs and for each faulty program, we executed the test suites produced by manual testing for the original program. Finally, for all test suites we collected the following measures: generation time, execution time, number of test cases and fault detection score. In order to calculate the fault detection score, each test suite was executed on both the original program and its faulty counterpart. In case the results differed between the executions, the fault was considered to be detected.

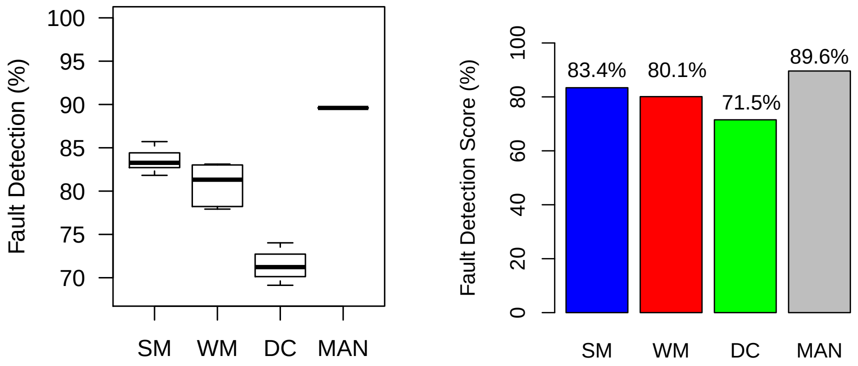
**Measuring Cost.** We measured the cost of performing testing focusing on the unit testing process as it is implemented in Bombardier Transportation for testing the programs selected in this case study. For the TCMS system, the creation and execution of test cases is performed by the implementer of the IEC 61131-3 software. In the cost measure, we use *the creation cost*, *the execution cost*, and *the result check cost*. The cost does not include the required tool preparation, the reporting and the maintenance of the test suite. We consider that all cost components related to human effort are depended 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 with a constant factor 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.

## 5 Experimental Results and Discussion

The case study presented us with a fault detection score and a cost measurements for each of the collected test suites (i.e., manually created test suites by industrial engineers (MAN), mutation-adequate test suites (i.e., weak-mutation testing (WM), strong-mutation testing (SM)) and automatically generated test suites based on decision coverage (DC)). The overall results of this study are summarized in the form of boxplots[3] in Figure 4 and 5.

**Fault Detection.** To answer RQ1 regarding the fault detection, in terms of detection of manually seeded faults, we focused on comparing all DC, WM, SM and MAN test suites. For all programs, as shown in Figure 4, the fault detection scores obtained by manual written test suites are higher in average with 9% and 6% than those achieved by weak mutation and strong mutation respectively. The difference in fault detection is slightly greater between strong-mutation testing and decision coverage-adequate testing (i.e., a difference of almost 12% on average). To understand how manual test suites achieve better fault detection than mutation-adequate test suites, we examined if the test suites are particularly weak or strong in detecting certain type of faults. We concern this analysis to what kind of faults were detected by manual testing and not by strong-mutation test generation. From a total of 77 faults, we identified eight faults (i.e., for exemplification purposes these faults are named Fault 1-8) that were not detected by any strong-mutation test suite while being detected by manual test suites. To produce meaningful results the remaining 69 faults are not included in this fault detection analysis because there is no consistent difference between manual and strong-mutation test suites. There are some broad trends for eight faults that can be used for explaining at least the difference in fault detection between manual and strong-mutation testing. Test suites written using manual testing

[3] 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.

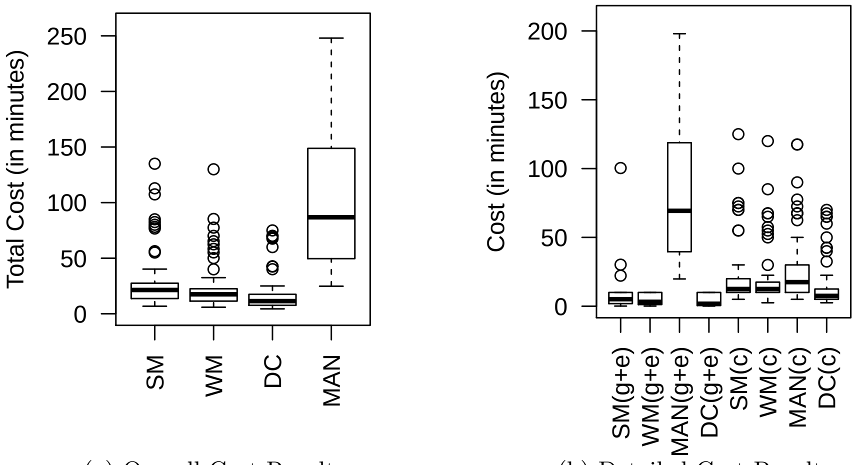


(a) Overall Fault Detection Comparison.

(b) Average Fault Detection Score.

Fig. 4: Fault detection results for manual testing (MAN), decision coverage-directed test generation (DC), weak mutation testing (WM) and strong mutation testing (SM).

are able to detect all of these eight faults. Mutation test suites are achieving a poor selection of test inputs produced for detecting certain faulty behaviors; for six faults, strong mutation testing generated test suites achieving 100% mutation score while for the remaining two faults, the model checker was unable to find a test suite detecting all mutants, given the 10 minutes time limit. It seems that manual testing has a stronger ability to detect these faults than mutation testing because of its inherent advantage of relying also on the specification of the program under test. For four of the faults, multiple changes in the program have been seeded (e.g, two or more blocks and variables have been replaced, deleted or inserted). For example, Fault 1 contains three changes combining three simpler faults corresponding to the application of CRO and VRO mutation operators. Fault 2 contains a combination of seeded changes corresponding to the creation of mutants using LRO and NIO mutation operators. In addition, Faults 3 and 4 contain multiple changes that were not captured by previously defined mutation operators. On the other hand, four of the faults are first order faults containing only one change in the program. A feedback loop signal connecting one of the outputs of the programs with one of the blocks was seeded in Faults 5 and 6. On the other hand, Fault 7 contains an extra logical block that was added to the original program while in Fault 8 a constant variable has been replaced to a non-boundary value. As a direct result, we discuss in Section 5.1 the improvement of mutation-based test generation for PLC software by considering



(a) Overall Cost Results

(b) Detailed Cost Results

Fig. 5: Cost measurement results for manual testing (MAN), decision coverage-directed test generation (DC), weak mutation testing (WM) and strong mutation testing (SM).

additional mutation operators not considered before in the literature [23, 27] to model possible faults.

**Cost.** We interviewed three engineers working on developing and manually testing TCMS software and asked them to estimate the time (in minutes) needed to create, execute and check the result of a test suite. All engineers independently provided similar cost estimations. We averaged the estimated time given by these three engineers and we calculated each individual cost using the following constants: 6.6 minutes for the creation of a test case, 3.3 minutes for the execution of a test case and 2.5 minutes for the checking of the result of a test case. Practically, for answering RQ2, we used these constants and the number of test cases in each test suites to represent the average time spend by an engineer to manually test each program. In addition, for mutation-based and decision coverage-directed test generation the total cost involves both machine and human resources. We calculated the cost of generating and executing a test suite by directly measuring the time required by the tool to run the test generation and the time required to execute each test case. For the cost of checking the test result we used the same average time as for manual testing (i.e. 2.5 minutes for the checking of the result of a test case). The resulting cost measures are reflected in Figure 5. The cost of performing testing using mutation testing either weak or strong is consistently significantly lower than for manually created test suites; automatically generated test suites have a smaller testing cost (110 and

115 minutes shorter testing time on average for WM and SM respectively) than the cost of using manual test suites. A more detailed cost measurement would be needed to obtain more confidence in the cost results obtained in this study.

### 5.1 Discussion

To explore the results of our study we consider the implications for future work and the extent to which mutation testing for PLC programs can be improved.

**Improving Mutation Testing for PLC Programs.** The results of this study indicate that fault detection scores obtained by manual test suites are better than the ones achieved by mutation testing. While comparing just strong-mutation testing with manual testing, we discovered that some of these faults are not reflected in the mutation operator list used for generating mutation adequate test suites, as described in Section 3.1. From our results, we highlight the need for improving the list of mutation operators used for mutation testing of PLC software by the addition of the following new mutation operators:

- *Feedback loop Insertion Operator (FIO)* is inserting a signal connecting an output variable to any block that is connected with the input variables.
- *Logical Block Insertion Operator (LIO)* is inserting a logical block between any other two logical blocks in the program.
- *Logical Block Deletion Operator (LDO)* is deleting a logical block and connecting the inputs of this block to the next logical block in the program.

In addition there are couple of already implemented mutation operators (shown in Section 3.1) that can be improved by considering the following operators:

- *Value Replacement Operator-Improved (VRO-I)* is replacing a value of a constant variable value connected to a block not only with its boundary values but also *with a selection of non-boundary values including 0, 1, -1.*
- *Logical Block Replacement Operator-Improved (LRO-I)* is replacing a logical block not only with logical blocks from the same category but also *with other blocks with Boolean inputs (e.g., replace an AND block with an SR block).*

By generating test suites that detect faults created based on these mutation operators, one could improve the goals of mutation testing for PLC programs. In addition, we recommend the use of higher-order mutation [19] for PLC software in order to find more complex faults.

**Mutation Testing using Model Checking.** Our study is the first to consider mutation testing using model checking for PLC programs written in IEC 61131-3 FBD language. Model checking is a formal technique based on state exploration that has been applied to mutation testing by either using a process named reflection [3], by state machine duplication [24], or by explicitly evaluating the fault coverage over multiple mutants [12, 14] thus creating test cases for manifesting fault propagation. The performance of this kind of approaches is depended not only on the model size but also the time spent on checking each and every mutated model or property against its original counterpart. This way of using the model checker for mutation testing can introduce unnecessary runs of the model checker and can considerably affect the feasibility of these approaches in practice. The method proposed in this study for the IEC 61131-3 FBD language is using a rather different approach for mutation testing, by combining all the mutants and the original model into a single combined model that is monitored dynamically using a model checking approach. By considering this way of utilizing the model checker one could potentially improve the cost of using mutation testing for other languages and models; the detection can be verified in a single run of the model checker for all mutated models rather than considering each individual case and thus removing the unnecessary model checking runs needed for detecting trivial mutants. This needs to be carefully considered in future studies and compared with other approaches on mutation testing using model checking.

## 6 Conclusions

In this paper we introduced mutation testing for PLC programs written in IEC 61131-3 programming language using a model checker. We implemented our approach in a tool and used this implementation to evaluate mutation testing on industrial programs and manually seeded faults. Our results show that mutation testing achieves lower fault detection compared to manual testing but with a significant lower cost in terms of testing time. We found out that these fault detection scores can be improved by considering some new and improved mutation operators for PLC programs as well as higher-order mutation.

## 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. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Logic in Computer Science. pp. 414–425. IEEE (1990)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
3. Black, P.E.: Modeling and marshaling: Making tests from model checker counterexamples. In: Digital Avionics Systems. vol. 1. IEEE (2000)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Symposium on Operating Systems Design and Implementation, vol. 8. USENIX (2008)
5. CENELEC: 50128: Railway Application: Communications, Signaling and Processing Systems, Software For Railway Control and Protection Systems. In: Standard. European Committee for Electrotechnical Standardization (2001)
6. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on Test Data Selection: Help for the Practicing Programmer. In: Computer, vol. 11. IEEE (1978)

7. Demillo, R.A., Offutt, J.A.: Constraint-based automatic test data generation. Transactions on Software Engineering 17(9), 900–910 (1991)
8. Enoiu, E., Čaušević, A., Ostrand, T.J., Weyuker, E.J., Sundmark, D., Pettersson, P.: Automated Test Generation using Model Checking: an Industrial Evaluation. In: Journal on Software Tools for Technology Transfer. Springer (2014)
9. Fraser, G., Arcuri, A.: Evosuite: Automatic Test Suite Generation for Object-oriented Software. In: Foundations of Software Engineering. ACM (2011)
10. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with Model Checkers: a Survey. In: Journal on Software Testing, Verification and Reliability, vol. 19. Wiley (2009)
11. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. Transactions on Software Engineering 38(2), 278–292 (2012)
12. Gargantini, A.: Using model checking to generate fault detecting tests. In: International Conference on Tests and Proofs. pp. 189–206. Springer (2007)
13. Gay, G., Staats, M., Whalen, M., Heimdahl, M.: The Risks of Coverage-Directed Test Case Generation. In: Transactions on Software Engineering. IEEE (2015)
14. Godskesen, J.C., Nielsen, B., Skou, A.: Connectivity testing through model-checking. In: FORTE. pp. 167–184. Springer (2004)
15. Howden, W.E.: Weak mutation testing and completeness of test sets. Transactions on Software Engineering (4), 371–379 (1982)
16. IEC: International Standard on 61131-3 Programming Languages. In: Programmable Controllers. IEC Library (2014)
17. Inozemtseva, L., Holmes, R.: Coverage is Not Strongly Correlated with Test Suite Effectiveness. In: International Conference on Software Engineering. ACM (2014)
18. Jamro, M.: POU-Oriented Unit Testing of IEC 61131-3 Control Software. In: Transactions on Industrial Informatics,, vol. 11. IEEE (2015)
19. Jia, Y., Harman, M.: Higher order mutation testing. Information and Software Technology 51(10), 1379–1393 (2009)
20. John, K.H., Tiegelkamp, M.: IEC 61131-3: Programming Industrial Automation Systems. Springer (2010)
21. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Foundations of Software Engineering. pp. 654–665. ACM (2014)
22. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. In: International Journal on Software Tools for Technology Transfer, vol. 1. Springer (1997)
23. Oh, Y., Yoo, J., Cha, S., Seong Son, H.: Software Safety Analysis of FBD using Fault Trees. In: Reliability Engineering & System Safety, vol. 88. Elsevier (2005)
24. Okun, V., Black, P.E., Yesha, Y.: Testing with model checker: Insuring fault visibility. In: System Science and Applied Mathematics (2003)
25. Orso, A., Rothermel, G.: Software testing: a research travelogue (2000–2014). In: Proceedings of the on Future of Software Engineering. ACM (2014)
26. Schwartz, M.D., Mulder, J., Trent, J., Atkins, W.D.: Control System Devices: Architectures and Supply Channels Overview. In: Sandia National Laboratories Sandia Report SAND2010-5183 (2010)
27. Shin, D., Jee, E., Bae, D.H.: Empirical Evaluation on FBD Model-based Test Coverage Criteria using Mutation Analysis. In: MODELS. Springer (2012)
28. Simon, H., Friedrich, N., Biallas, S., Hauck-Stattelmann: Automatic Test Case Generation for PLC Programs Using Coverage Metrics. In: ETFA. IEEE (2015)
29. Woodward, M., Halewood, K.: From weak to strong, dead or alive? an analysis of some mutation testing issues. In: STVA. IEEE (1988)
30. Wu, Y.C., Fan, C.F.: Automatic Test Case Generation for Structural Testing of FBD. In: Information and Software Technology, vol. 56. Elsevier (2014)