

TDK-dolgozat

Serban Andrada Alexia

2024.



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Mesterséges Intelligencia és Rendszertervezés Tanszék

Teszthatékonyság javítása mutációs teszteléssel PLC környezetben

Szerző:

Serban Andrada Alexia
IV. évfolyam

Témavezető:

dr. Micskei Zoltán,
egyetemi docens

2024.12.15.

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
2 Background and Related Work	3
2.1 Mutation testing	3
2.2 PLC software	4
2.3 Mutation testing in the PLC domain	5
3 Surveying challenges of PLC software testing	6
3.1 Research Methodology	6
3.2 Results	7
3.2.1 PLC-specific challenges in software testing	7
3.2.2 Identifying difficult steps in the testing process	8
3.2.3 Testing steps with automation potential	9
3.3 Threats to validity	9
3.4 Conclusions	10
4 Mutation-based solutions for improving PLC software testing	11
4.1 Presenting the industrial context	11
4.2 Adapting mutation testing for PLC software	12
4.2.1 Mutant generation and selected mutation operators	12
4.2.2 Implementation	14
4.3 Initial results	15
4.3.1 Results on industrial code	15
4.3.2 Industry feedback	16
4.4 New metrics and visualization for ST mutation testing	17
4.4.1 Visualizing results	17

4.4.2	Defining metrics for result evaluation	18
4.5	Using mutants for model checker-based test generation	19
4.5.1	Proposal	19
4.5.2	Proof of Concept	20
4.5.3	Conclusion	22
5	Evaluation	23
5.1	Qualitative evaluation: Evaluative case study	23
5.1.1	Research methodology	23
5.1.2	Results	24
5.1.3	Discussion	26
5.2	Quantitative evaluation: Experiment on industrial codebase	27
5.2.1	Evaluation on industrial code	27
5.2.2	Results on industrial code	27
5.2.3	Discussion	30
6	Conclusion	31
6.1	Research results	31
6.2	Further work	32
	Bibliography	33
	Appendix	36
A.1	Example FBD code and mutation operators	36
A.2	Example ST module	37
A.2.1	Original code	37
A.2.2	Mutants	39
A.3	Example test report	40
A.4	Examples for test generation	41
A.4.1	Simple example	41
A.4.2	Industrial example	43

Kivonat

A *programozható logikai vezérlők*, vagyis a *PLC*-k, az ipari szabályozástechnikában használt berendezések. Előfordulnak gyárakban, a járműiparban, a gyógyszeriparban és az energiaiparban is. Gyárakban gyártósorok vezérlésére vagy gyártási paraméterek monitorozására használatosak, míg a járműiparban vasúti szerelvények különböző funkcióit, pl. a motorteljestmény megfigyelését vagy a fékrendszerek vezérlését irányíthatják. Ezen alkalmazások jelentős részéről belátható, hogy biztonságkritikus funkciókat látnak el, melyeknek alapos tesztelését szabványok írják elő.

A PLC szoftver tesztelése, főleg alacsonyabb szinteken, például egy-egy szoftverkomponens vagy -modul esetén korántsem olyan egyszerű feladat, mint a hagyományos programozási nyelveken, számítógépre vagy mikrokontrollerre írt programoknál. A PLC kód tipikusan időzített, ciklikus lefutású: beolvassa az aktuális bemeneteket, az implementált logika segítségével feldolgozza azt, majd frissíti a kimenetek értékeit. A rendelkezésre álló hibakereső eszközök lényegesen kevésbé fejlettek, a tesztelés zömében a bemenetek állítására és a kimenetek időzítési kritériumok melletti megfigyelésére korlátozódik. A kódfedettség, mint a tesztek elégségességének fontos (bár nem tökéletes) indikátora ritkán mérhető a fejlesztőeszközök korlátozott funkciói miatt.

A mutációs tesztelés egy olyan módszer, melyet a tesztek elégségességének meghatározására mind a kódfedettség helyett, mind amellet is vizsgáltak hagyományos programozási nyelveken implementált szoftverek esetén. A PLC-k szoftvertesztelési folyamataiban való alkalmazásával azonban kevés kutatás foglalkozik.

A tesztelési folyamat fejlesztése érdekében először PLC-k szoftvertesztelésével foglalkozó mérnökök körében végeztem felmérést, melynek segítségével azonosítottam a teszttervezési, -fejlesztési és -futtatási folyamatok fő nehézségeit. Ezt követően kiválasztottam egy mutációs operátorkészletet több PLC programozási nyelvre, melyet egy környezetfüggetlen mutánsgenerátor implementációjára használtam fel. Ennek segítségével mutációs tesztelést végző programot implementáltam a vizsgált ipari tesztkörnyezethez, melyet a hatékonyabb eredményfeldolgozás érdekében egy adatvizualizációs modullal bővítettem ki, illetve javaslatot tettem egy mutáción alapuló tesztgenerálási algoritmus kiegészítésére.

A létrehozott eszközt egy 25 modulból álló ipari kódbázison értékeltem ki, majd átadtam a tesztmérnök csapatnak, akik egy aktív projekten, a tesztelési folyamat közben alkalmazták azt. A tapasztalatok összegyűjtését követően interjúban kérdeztem ki őket az eszköz és a módszer hatékonyságáról, alkalmazásuk létjogosultságáról az adott környezetben.

Ezen dolgozatban azonosítom a PLC-k szoftvertesztelésének fő nehézségeit, és megoldást javaslok egy kritikus, ámde kihívásokkal járó feladat támogatására. Megoldásomat valós, ipari környezetben értékelem ki szakemberek visszajelzését figyelembe véve. Az eredmények a tesztelési ráfordítás hatékonyabb beosztását teszik lehetővé, emellett pedig bővíti a mutációs tesztelés ipari alkalmazásával kapcsolatos irodalmi eredményeket PLC-k szoftverén való alkalmazással.

Abstract

Programmable logic controllers, or *PLCs* can be found in various industries, such as manufacturing, transportation, pharmaceuticals, and energy. In manufacturing, they coordinate assembly lines, regulate machine speeds, and monitor production parameters, while in the transportation sector, PLCs are extensively used in trains for various functions such as monitoring engine performance and controlling brakes. A large majority of these applications are typically safety-critical and therefore required by standards to undergo rigorous testing to ensure reliability.

Testing of PLC software, especially on a lower (i.e. component or module) level, can be a significantly more complex and challenging task than that of programs developed in conventional programming languages for microcontrollers or computers. PLC code typically runs in timed, cyclic loops, reading inputs, processing the data, and updating outputs correspondingly. The debugging tools for this kind of software are less advanced, and testing is mostly limited to providing inputs and checking the output values and their timing. Code coverage, an important (though not ultimate) indicator of test sufficiency is also less frequently measured due to development tool limitations.

Mutation testing is a technique for evaluating test efficiency that has been thoroughly studied as both a complementary and substituting method to code coverage measurement in the verification and validation process of software implemented in traditional programming languages. However, it is yet to be determined whether it is a feasible approach to improve PLC software test efficiency, as the topic has been scarcely studied.

In order to improve test efficiency, I conducted a survey among engineers working in PLC software testing, identifying the main difficulties in test design, implementation, and execution. I then selected a set of mutation operators for multiple PLC programming languages that I used to implement a generic mutant generator and an automated mutation testing tool for the test environment used at the studied company. Finally, I added a data analysis and visualization module to the created tool and proposed an addition to a mutation-based test generation algorithm.

I evaluated the created tool on an industrial code base of 25 modules and handed it over to test engineers. The tool was then used in an active industrial setting during testing, and the engineers were interviewed about the tool's and method's usability in the given context.

In this report, I identify the main difficulties in PLC software testing and present a solution that supports a critical, but difficult task in software development. I evaluate my solution in a real industrial setting and consider feedback from industry professionals. The results not only provide a more efficient distribution of test effort in PLC software testing but also add to the body of knowledge on the application of mutation testing to PLC software with results in an industrial context.

Chapter 1

Introduction

Programmable logic controllers (PLCs) are specialized computing devices designed for real-time control and automation of processes across various industries. A substantial part of these applications are safety-critical: failures could lead to significant safety, health, or financial risks. To ensure reliability, rigorous testing processes are mandated by industry standards. Meeting these standards in the PLC domain presents a unique set of challenges due to the characteristics of PLC programming and is further complicated by the limitations of available testing tools.

Motivation PLCs operate according to the *IEC61131* standard, its third part, IEC61131-3 [5] defining the programming languages and basic software architecture of such industrial control systems. The standard provides 5 programming languages – 2 textual and 3 graphical – in which PLCs can be reliably programmed and addresses the cyclical nature of code execution, where inputs are periodically processed, control logic is evaluated and outputs are updated in real time.

Verifying this specialized software presents several difficulties. First, the unique structure of PLC programs introduces challenges in simulating and testing all possible input and output scenarios. Second, there is a notable lack of metrics available for evaluating test sufficiency. Traditional software testing metrics, such as code coverage, are not as readily applied in PLC environments partially due to tool limitations. As a result, testing is often limited to providing inputs, checking outputs, and ensuring timing requirements are met.

These limitations emphasize the need for *more effective testing practices* in PLC software development, which motivates the research for potential improvements, such as *mutation testing*.

Mutation testing is a technique that can be used to determine the fault detection ability of a test suite by introducing small modifications in the tested program's code, thus creating so-called mutants. These are typically simple alterations of an operator or variable, aiming to simulate common programming errors. The mutants are checked with the original test suite in order to determine whether it is capable of detecting them (by failing at least one test).

In recent years, mutation testing has seen an increase in industrial applications, multiple case studies and surveys deeming it a feasible method of assessing test quality in conventional programming environments. The method is appropriate for complementing and potentially even substituting code coverage metrics, especially in cases where coverage can not be measured.

However, in the context of PLC software testing, mutation testing is still an underresearched area. Most studies have focused on more widely used programming languages, such as C or Java, where advanced and often open-source testing tools are prevalent. Given the lack of experience on mutation testing in PLC environments, this report aims to bridge the gap by applying the technique in order to improve test efficiency and proposing new strategies to overcome the challenges emerging in this unique domain.

Identifying problems In this report, I attempt to identify the primary challenges of PLC software testing *in an industrial setting*. To achieve this, I conducted a survey among a group of test engineers dealing with PLC software in their day-to-day work, aiming to identify the most challenging parts of the testing process. The survey results revealed that determining test sufficiency and designing adequate tests is considered the most difficult, and the limited capabilities of testing tools and lack of defined metrics pose the greatest challenges. The method and results of the survey are detailed in Chapter 3.

Proposing solutions Based on the received feedback, I implemented a mutant generation and mutation testing tool in the test environment used at the studied company, enabling engineers to receive feedback on test sufficiency. I proposed specific metrics and visualization in order to provide a more comprehensive analysis of the results and explore applicable metrics. Furthermore, I provided a proof of concept on how live mutants could be used for model-based test generation, a method that can either be automated or used to advise test engineers on test design. Additionally, it allows us to determine whether a live mutant can be killed given the controllability and observability of the software in the test setup, that can be applied to any programming language. These solutions are presented in detail in Chapter 4.

Evaluation of solutions The proposed solutions were subjected to both quantitative and qualitative evaluation. The mutation testing tool was used to evaluate an industrial code base consisting of 25 modules – varying in size, complexity and function – in order to assess the method’s ability to detect testing deficiencies in the context of PLC software development. The tool was also used in an ongoing industrial project, and the engineers using it were asked to evaluate its usability and added value in the given setting. The evaluation process and its results are presented in Chapter 5.

Contributions

- I identify the difficulties of PLC software testing by surveying industry professionals
- I adapt and evaluate mutation testing for PLCs in an industrial environment, presenting a comprehensive analysis of its potential in the PLC domain
- I propose new metrics and visualization methods making the results of mutation testing easily comprehensible
- I propose an addition to a mutation-based test generation algorithm, creating not only a feasible advisory or automated test generation method, but also providing a generic approach to detecting pseudo-equivalent mutants

Chapter 2

Background and Related Work

2.1 Mutation testing

Definition *Mutation testing* is a technique that measures the fault detection capability of a test suite by running the tests against syntactically modified variants – *mutants* – and checking whether the tests detected the mutant (failed) – these are the *killed* mutants – or the tests left the mutant undetected, resulting in a *live* mutant. The percentage of killed mutants is called *mutation score* and is the most commonly used metric in this process.

<pre>if(x<y) { return a; } else { return b; }</pre>	<pre>if(x<=y) { return a; } else { return b; }</pre>	<pre>if(x>y) { return a; } else { return b; }</pre>	<pre>if(x>=y) { return a; } else { return b; }</pre>	<pre>if(x==y) { return a; } else { return b; }</pre>	<pre>if(x!=y) { return a; } else { return b; }</pre>
(a)	(b)	(c)	(d)	(e)	(f)

Figure 2.1: Simple example of code (a) and some of its possible mutants (b-f) [10]

Operators The syntactical modifications are performed using *operators*. An operator makes a single modification to the source code, e.g. changing a relational or arithmetic operator to another. The type of applicable operators varies slightly depending on the programming language of the analyzed software, but is very similar for most textual languages. Most commonly, constants, variables, return values, relational or arithmetic operators are replaced, or conditions are removed. An example of such modifications is shown in Figure 2.1. While the set of possible operators for each language is quite large – e.g. Agrawal et al. [1] defining 80 for C in 1989 – applying all of them would be unnecessary and inefficient. In order to find an optimal set of operators Offutt et al. [14] studied the effect of the omission of certain operators and isolated 5 operators, shown in Table 2.1. This set of operators not only makes mutation testing effective but can also be directly applied to most programming languages.

Industrial applications Not many studies have been carried out on the application of mutation testing in safety-critical settings, but the results were unanimous in confirming the added value of this method to the software verification process. While integration is not always seamless and has its challenges, mutation testing has consistently demonstrated its ability to improve test quality by identifying weaknesses that conventional approaches might miss.

Operator	Description	Example
ABS	Absolute Value insertion	$a = b + c$ to $a = 0$
AOR	Arithmetic Operator Replacement	$a = b + c$ to $a = b - c$
LCR	Logical Connector Replacement	$a = b \ \&\& \ c$ to $a = b/c$
ROR	Relational Operator Replacement	$while(a < b)$ to $while(a > b)$
UOI	Unary Operator Insertion	$a = b$ to $a = -b$

Table 2.1: The 5 efficient operators isolated by Offutt et al. [14]

In 2012, Baker and Habli [2] conducted one of the first empirical studies on a relatively small sample of airborne C and Ada code, ultimately showing that mutation testing adds value to the verification process and offers a consistent measure of test quality. Case studies on larger code bases [17], conducted in other industries [8] and implementing and evaluating entire pipelines [6] followed, all presenting the positive impact of mutation testing, along with an overview from Google [15] reporting on the long-term effects on code quality and the process of making it a scalable approach. In my previous work, I also created and evaluated a mutation testing tool for C code in a safety-critical environment [20] and reported on its successful integration to the component testing workflow in a journal paper [19]. Most recently, industry professionals have also been surveyed on how they use mutation testing, what they find its benefits and limitations to be, and how different result visualization methods help them gain a better understanding of test suite quality [23, 18].

2.2 PLC software

Programming languages The IEC 61131-3 standard [5] defines 5 programming languages to be used for programming PLCs – 3 graphical and 2 textual:

- **Ladder Diagram (LD):** resembles electrical relay logic diagrams. It’s graphical and uses symbols to represent various logic operations, making it easy for electricians and engineers with little or no expertise in classical programming to understand.
- **Function Block Diagram (FBD):** is also graphical, using blocks to represent functions, operations and their interconnections. It’s suitable for complex control algorithms and provides a visual representation of the control logic.
- **Sequential Function Chart (SFC):** is a graphical language that represents the sequential control of processes. It’s particularly useful for representing state-based control systems and is often used for complex control algorithms.
- **Instruction List (IL):** is a low-level textual language resembling assembly language, consisting of a list of instructions executed sequentially by the PLC. It was declared obsolete in the 3rd edition of the standard and therefore is rarely used directly by programmers, but it is generated by higher-level languages.
- **Structured Text (ST):** is a high-level textual programming language similar to PASCAL and allowing for complex algorithms and data structures. It’s suitable for experienced programmers and for tasks requiring extensive computation.

Additionally, the **Continuous Function Chart (CFC)** language was created as an extension of FBD, allowing free placement of blocks and direct feedback. It’s specialized for

continuous control processes, focusing on mathematical operations, time-based behaviors, and integration with discrete control logic.

PLC projects are organized into *program organizational units* (POUs), which can be programs (PRG), functions (FUN) or function blocks (FB) and each written in a different language, using their variables as interfaces.

Software verification Due to the unique nature of PLC applications, software verification differs significantly from that of conventional programming languages. A common approach is to use *Software-in-the-Loop (SIL)* and *Hardware-in-the-Loop (HIL)* testing on different testing levels, which suits the cyclical nature of the software architecture well, but presents engineers with a slightly more complex task when designing and implementing tests. Unlike most programming languages, code coverage is rarely measured even in lower level software testing, limiting the exit criteria to requirement coverage.

Another feasible method is the use of formal verification techniques, such as *model checking*, that can automatically verify *liveness* (sg. that must happen) and *safety* (sg. that must not happen) properties. As these properties would be hard to prove with traditional testing alone, formal verification can play a key role in ensuring that system requirements are met. However, it requires a level of field-specific knowledge and understanding that can not be expected from all test engineers. This is what motivates the development of tools like *PLCverif* [7], that enable engineers who do not possess in-depth knowledge on formalizing requirements and creating formal models, to use model checkers like *Theta* [22] or *NuSMV* [3] to verify PLC software easily.

2.3 Mutation testing in the PLC domain

Despite its effectiveness in improving test suites in other programming languages, there has been little research on evaluating mutation testing in PLC software development. There are no open source, publicly available mutation testing tools for any PLC language, which may be due to the lack of prevalent software development tools and compilers. As the market for these tools is almost entirely industrial, vendor-specific solutions are more typical.

Among graphical languages, studies are almost exclusively concerned with the mutation of FBD programs: using mutants for model-based test generation [9, 11] and evaluation of various code coverage criteria [21]. Examples of operators used in mutation of FBD code are shown in Figure A.1.1 in the appendix. A part of these operators are similar to those used in textual programming languages, e.g. the operators modifying relational, arithmetic or logic operators, but PLC-specific mutations were also defined, e.g. the modification of timer blocks.

The mutation of ST code is not a profoundly studied topic, but due to its textual nature operators used on general-purpose programming languages are easily applicable. A proposal for operators for the ST language was made by Wang et al. [24] in their recent study, where they used generated mutants as test inputs. The suggested operator set contains the operators proposed by Offutt et al. (shown in Table 2.1), but also statement and operator deletion and insertion, and variable random assignment operators. As part of this report, I propose a model-based test generation approach for ST programs similar to that proposed by Enoiu et al. [9] for FBD programs.

Chapter 3

Surveying challenges of PLC software testing

To provide solutions that tailor well to problems faced in PLC software testing, I attempt to identify the primary challenges through a survey conducted among a group of test engineers dealing with PLC software in their day-to-day work. In this chapter, I present the research methodology of the conducted survey, its results, the threats to result validity and the conclusions drawn. To ensure reliable survey design, this research draws on the empirically evaluated checklist presented by Moller et al. [12], downscaling it to the sample size I had access to for this study.

3.1 Research Methodology

Research Questions In order to improve test efficiency, I had to gain a better understanding of the challenges engineers face in the PLC domain. To achieve this I aim to answer the following research questions:

- **RQ1** What are the field-specific challenges in PLC software testing?
- **RQ2** What are the most difficult steps in the testing process?
- **RQ3** Which parts of the testing process are repetitive and potentially could be automated?

Survey participants To answer these questions, I performed a survey on the challenges of software testing in the PLC domain. The survey participants were test engineers selected from a system development group at a company that designs and manufactures braking systems for rail vehicles. These braking systems are controlled by PLCs running software written in ST and CFC language and, due to their safety-critical nature, have to undergo rigorous testing.

Survey design Before answering survey questions, participants were required to complete an informed consent form. Answering any question was anonymous and optional and questions were designed so that participants could not be identified, despite the small population size.

Q#	Question	Type	#Resp	RQ
1a	How difficult is test design? (1-5)	SC	8	RQ2
1b	How difficult is test implementation? (1-5)	SC	8	RQ2
1c	How difficult is test execution? (1-5)	SC	8	RQ2
1d	How difficult is achieving adequate coverage? (1-5)	SC	8	RQ2
2	What poses a challenge in PLC software testing?	FT	8	RQ1
3	What parts of the testing process are repetitive and could potentially be automated?	FT	6	RQ3

Table 3.1: Survey questions, presented as in [18]

Legend: SC – single choice, FT – free text,

#Resp – number of respondents

When designing the survey, my goal was to create a questionnaire with results that can be easily analyzed, but are not biased as a result of question formulation. Therefore, participants were asked to rate the difficulty of different tasks, but when being asked to specify challenges or repetitive tasks, they were required to provide a free text answer and not e.g. choose from a list. The questions and respondent numbers are summarized in Table 3.1. The full questionnaire and received answers can be found in the attachment of this report.

3.2 Results

The survey was conducted with the participation of one team at one company, a topic that will be addressed in the next section, where I discuss the threats to result validity. The questionnaire was sent to the aforementioned system design group via email and received a total of 9 answers, one of which denied consent. This section presents the answers provided by the participants and the conclusions drawn from them.

3.2.1 PLC-specific challenges in software testing

In Question 2, participants were asked to specify challenges that emerge in the software testing process. Their responses were coded based on their content and are shown in Figure 3.1. Detailed responses can be found in the attachment of this report.

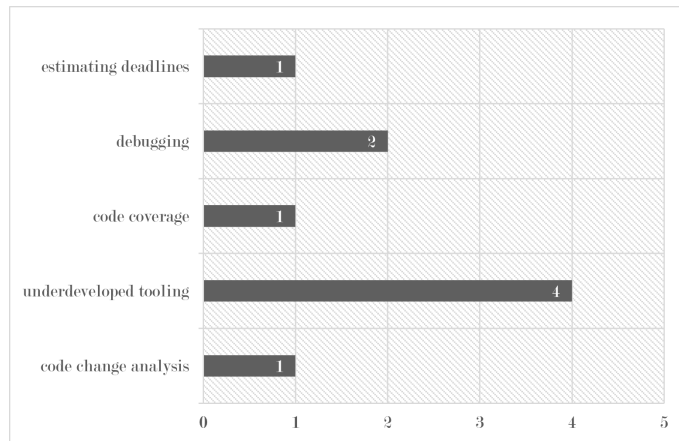


Figure 3.1: Answers to Question 2

The most commonly mentioned challenge is the lack of advanced tooling: simulating inputs is often complicated, not all data structures can be used as test inputs, existing tools are developed within the company and have limited support and reliability. The complete absence of debugging tools and coverage measurement is also mentioned separately from tooling deficiencies.

RQ1: What are the field-specific challenges in PLC software testing?

The majority of answers specified some type of deficiency in the currently used toolchain, either in test input specification, debugging or coverage measurement.

3.2.2 Identifying difficult steps in the testing process

In Question 1, participants were asked to rate the difficulty of test design, implementation, execution and achieving the adequate coverage on a scale from 1 to 5 (1 being very easy, 5 being very difficult). The scale had textual labels, answers were converted to numbers and visualized in Figure 3.2.

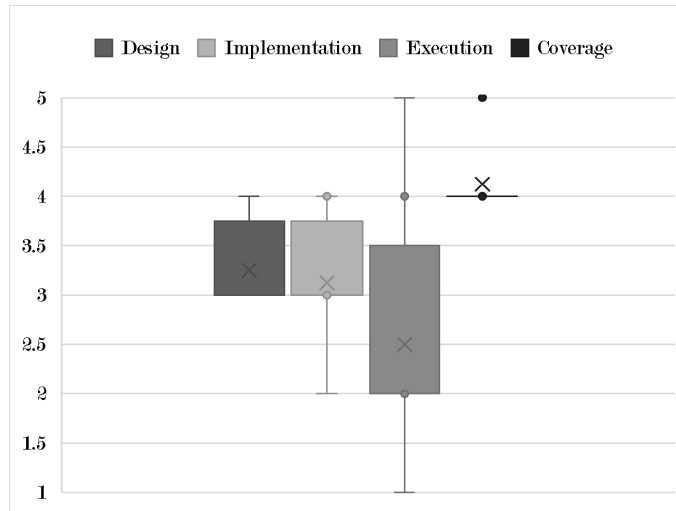


Figure 3.2: Answers to Question 1

Achieving adequate coverage was rated by far the most difficult task. It is important to note that at the studied company the exit criteria for module testing is to achieve 100% requirement coverage and that, as answers to Q2 showed, code coverage is not measured in any form. Furthermore, module-level requirements are derived manually from higher level requirements, which even when authored by the most experienced system engineers, can fail to cover the full functionality of the specified module. Additionally, covering said requirements with tests can also be a difficult task, if the requirements are not formulated clearly and also due to the limitations of the testing toolchain. In this context, even defining consistent coverage metrics is difficult.

Among the other aspects of software testing, test design was rated consistently mildly difficult, while the variance for implementation and especially execution was quite high, but on average lower than that of design.

RQ2: What are the most difficult steps in the testing process?

Achieving adequate coverage was rated by far the most difficult task in PLC software testing, followed by test design, which is considered mildly difficult. Both can be partially owed to the lack of code coverage measurement, which lead to test engineers relying solely on manually derived requirements of varying quality.

3.2.3 Testing steps with automation potential

In Question 3, participants were asked to specify tasks that they find repetitive and could potentially be automated. Their responses were labeled based on their content and are shown in Figure 3.3. Detailed responses can be found in the attachment of this report.

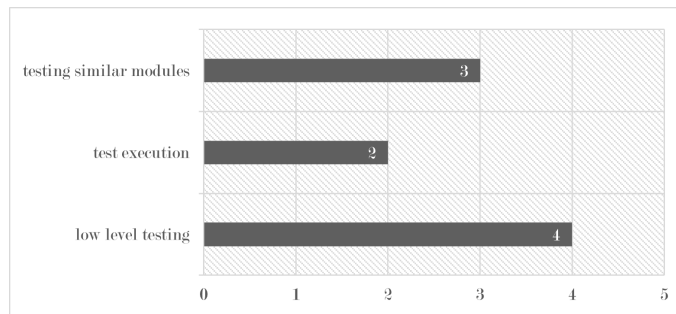


Figure 3.3: Answers to Question 3

The answers were limited to three categories: generation of lower (unit, module) level tests or for less complex modules, test generation based on previously written tests for similar modules or earlier versions of those modules, and test execution. The former two overall show a demand for automated test generation, while the latter further underlines the lack of advanced tooling.

RQ3: Which parts of the testing process are repetitive and potentially could be automated?

There seems to be a demand for automated generation for lower level tests, especially when the SUT does not implement complex logic. Furthermore, the current toolchain makes text execution a slow, time-consuming process that would benefit from automation.

3.3 Threats to validity

Internal Validity refers to the extent to which the conclusions drawn in the study can be attributed to the study design and not to external influences or bias. In survey research, controlling the conditions in which the factors are studied is difficult, and a lower internal validity is expected. Internal validity could be threatened if participants felt rushed to complete the questionnaire, limiting the completeness and accuracy of their answers and their access to clarification. However, the survey was open for over 3 weeks and was without a time limit, and participants had access to clarification as needed, as I provided my university contact information and am employed within the same company, though in a separate team. Additionally, *self-selection bias* may have occurred, as only engineers who were willing to complete the survey responded. Those who do not find PLC software

testing challenging may have not responded, despite the survey providing the possibility to express this view too.

External Validity refers to the extent to which the results of the study can be generalized to other settings. The primary threat to external validity in this study is the *small sample size* ($n = 8$) and *limited scope of participants*, as the participants are a small group of engineers working in the same team at a company specializing in a very specific domain. This could limit the applicability of the findings to other industries that also use PLC software, such as manufacturing, energy, or pharmaceuticals. The use of specific programming languages (ST and CFC) within the studied company may also limit the extent to which the results may be applied to environments where other languages are (also) used. Although many challenges in PLC software testing are likely common between industries, the specific tools, practices, and standards in different fields could lead to different findings. *Future research should aim to conduct a similar survey across multiple industries to improve both internal and external validity.*

Construct Validity refers to the accuracy with which the concepts intended to be measured are measured. A main threat to construct validity in this study is the *subjectivity of the questions*: participants were asked to rate the difficulty of specific testing tasks, but these ratings are subjective and dependent on the respondents' personal interpretations of "difficulty". Furthermore, free-text responses were used to gather challenges and repetitive tasks. While in this case, participant answers are not biased by predefined categories, the variability of how participants interpret and answer the question is introduced. To mitigate this risk, I used a labeling process to categorize responses and draw common themes, though this subjective analysis could still introduce researcher bias.

3.4 Conclusions

The survey results allow for a better understanding of the main difficulties engineers are faced with in the domain of PLC software testing, particularly in safety-critical applications. Multiple key findings surfaced from the analysis:

- **Toolchain limitations** were identified as the primary challenge in the studied environment. The *limitations of input simulation, debugging and code coverage measurement* specifically, hinder efficient software testing.
- **Lack of consistent metrics** makes it difficult for engineers to assess test sufficiency, leading them to rely solely on manually derived requirements, which may eventually result in undiscovered software bugs and errors.
- **Automation potential** was highlighted in two areas: lower-level test generation and test execution. The former is a clear demand for tools that can assist designing and even generating tests for simpler modules, while the latter further underlines the limitations of the current toolchain.

These points will be taken into account when proposing solutions to improve test efficiency in Chapter 4 of this report. The current scope of the proposed solutions is only the studied group, so certain threats to external and internal validity can be mitigated, and the extent to which the suggested solutions can be generalized will be subject to further discussion throughout this report.

Chapter 4

Mutation-based solutions for improving PLC software testing

In this chapter, I present a range of mutation-based solutions to the challenges of PLC software testing identified in Chapter 3. I begin with describing the setting in which I am to apply mutation testing and the motivation to do so in Section 4.1, followed by an overview of the selected mutation operators and initial implementation of the method in Section 4.2. The created mutation testing tool was applied in the industrial setting and presented to the previously surveyed group of test engineers. The results and their initial feedback is summarized in Section 4.3. Following this feedback, further improvements and propositions were made, which are presented in Section 4.4. Additionally, I proposed and provided a proof of concept for a method to use mutants for model checker-based test generation for ST code, demonstrated in Section 4.5.

4.1 Presenting the industrial context

Motivation The survey conducted in Chapter 3 revealed that limitations of the currently used toolchain hinder efficient software testing. Additionally, the lack of consistent metrics makes it difficult for engineers to design adequate test suites and evaluate the quality of existing ones. While modifying the toolchain would not be a feasible approach, as it would require company-wide policy and workflow changes across a multinational company, complementing it with additional, optionally applicable methods such as mutation testing may result in much-needed improvements in the software testing process.

Mutation testing can provide a consistent measure of test quality [2] while also aiding engineers in test design, as live mutants can serve as the basis for designing additional tests. Furthermore, mutants can be and have been used for automated test generation [9, 11], a demand that was also formulated by engineers in the conducted survey.

Application environment As there are no widespread software testing tools for PLCs, *mutation testing was introduced in a specific industrial setting*. In this environment, the software is responsible for the control of trains' braking systems including not only basic braking functions, but also supplementary tasks, such as brake tests or communication with the train network.

Software for these braking systems is developed in CODESYS¹, a quite prevalent and well-known commercial programming environment. The source code needs to be extracted from CODESYS projects and imported back following its mutation. Both tasks can be automated with the help of the Python API² and CLI.

Tests are defined in an internally developed tool. Modules – function blocks or programs – are tested individually, the test environment allowing the setting of input variables, timing, and the checking of output and internal variables of function blocks. Most modules have one or a few highly complex tests corresponding to them, exercising their entire functionality, rather than having several simpler tests.

Tests are executed in CODESYS on a virtual PLC. The module under test and the defined test steps are used to automatically generate PLC code, which is then compiled and run. The test results are logged and also provided to the tester in JUnit XML format. While this process is mostly meant to be manually started from the test development tool, a CLI was developed allowing for the automation of test execution.

Overall, all steps and software provided in the industrial context can be automated, allowing for the creation of a mutation testing tool. A Python script using the CODESYS Python API can be run through the CODESYS CLI to export each module separately, which then can be used for mutant generation. Each generated mutant can be imported back to the project in a similar manner, followed by test execution through the test development tool's CLI. Test results can easily be extracted from the provided XML and used to determine whether the given mutant was killed, stayed alive or caused a runtime error.

Challenges While the entire process can be automated *in theory*, integrating the aforementioned development tools posed quite a challenge. The CODESYS Python API documentation is inexact and lacks usage examples and active community or vendor support. Similarly, the CLI of the internal test design tool is poorly documented, and understanding its proper usage was not straight-forward and quite time-consuming. Additionally, opening, modifying, and saving projects in CODESYS are not particularly optimized processes, and their repeated execution results in overall slow import and export of modules and slow test execution, making both tool development and eventual usage a time-consuming process. Furthermore, both CODESYS and the internal tools are known to occasionally freeze or run into errors even when used correctly, requiring robust solutions and automating OS-level task management.

4.2 Adapting mutation testing for PLC software

This section first presents the selected mutation operators for ST and FBD (and CFC) languages and the generic mutant generation module applying them, followed by the presentation of a mutation testing tool using this module designed for the industrial environment described in Section 4.1.

4.2.1 Mutant generation and selected mutation operators

As there is a multitude of PLC software development tools and platforms, provided by different vendors and often used in parallel, the PLCOpenXML standard [16] was created

¹<https://www.codesys.com>

²https://help.codesys.com/webapp/_cds_creating_scripts;product=codesys;version=3.5.14.0

Operator	Block/connection	Mutant
LRO	AND OR XOR	OR, XOR AND, XOR AND, OR
CRO	GT LT LE GE EQ NE	GE, LE LE, GE LT, GT GT, LT NE EQ
ARO	ADD SUB MUL DIV SHL SHR ROL ROR MAX MIN	SUB ADD DIV MUL SHR SHL ROR ROL MIN MAX
TRO	TON TOF	TOF TON
NIO	[variable] NOT([variable])	NOT([variable]) [variable]
VRO	TRUE FALSE	FALSE TRUE

Table 4.1: Operators implemented for CFC/FBD POU

Operator	Code	Mutant
ABS	a := [expression]	a := 0
AOR	a + b a - b a * b a / b SHL(a, n) SHR(a, n) ROL(a, n) ROR(a, n) MAX(a, b) MIN(a, b)	a - b a + b a / b a * b SHR(a, n) SHL(a, n) ROR(a, n) ROL(a, n) MIN(a, b) MAX(a, b)
LCR	a AND b a OR b a XOR b	a OR b, a XOR b a AND b, a XOR b a AND b, a OR b
ROR	a > b a < b a <= b a >= b a = b a <> b	a >= b, a <= b a <= b, a >= b a < b, a > b a > b, a < b a <> b a = b
UOI	IF a THEN IF NOT a THEN WHILE a DO WHILE NOT a DO	IF NOT a THEN IF a THEN WHILE NOT a DO WHILE a DO

Table 4.2: Operators implemented for ST POU

to simplify and provide a unified interface for communication between them. I created a mutant generation tool that – given an export in PLCOpenXML format – is capable of generating PLCOpenXML mutants for POU

The operators implemented in the mutant generation tool are listed in Tables 4.1 and 4.2. For CFC and FBD programs, I omitted the Switched Inputs operator from the operators in Figure A.1.1 defined by Liu et al., resulting in an operator set similar to the 5 suggested by Enoiu et al. [9], keeping the names given in their study. For ST programs the 5 operators suggested by Offutt et al. [14] were adapted to the syntax of the language and implemented in the tool.

For CFC and FBD mutation, the XML document can be easily parsed and modified with built-in Python modules such as ElementTree³, whereas the mutation of ST code is a slightly more challenging task. As I had no access to compilers, or existing tools to parse ST code, I had to use *regular expressions*⁴ to find the tokens to be mutated. Due to the strict syntax of ST language, this was a far less difficult task than it would have been e.g. for C language, but not nearly without its challenges: the mutant generator was tested on open-source PLC projects found on GitHub⁵, and at the time seemed to cover all ST syntax, but during further usage it was revealed that these projects were significantly less complicated than the later studied industrial modules (e.g. the module in Listings A.2.1 and A.2.2, in the appendix) and the used regular expressions had to be updated multiple times.

4.2.2 Implementation

As concluded in the previous subsection, the primary task of the mutation testing tool was to create a connection between the existing components in order to automate the process of mutation testing. Furthermore, it had to provide a user-friendly, verbose interface to configure and execute each step.

In order to automate the entire process, importing and exporting of POU's from CODESYS and the test execution had to be automated separately, then integrated into one program. Separate scripts were created for the import and export tasks, which are then given as a parameter to the CODESYS CLI and run by it. The test executor is started through its own CLI. The automation software runs the export script for any module given in the configuration, runs the mutant generation for each exported module, then proceeds to import each mutant, execute the tests and process the XML containing the test results, creating a report from its contents. The component diagram visualizing this is shown in Figure 4.1.

To ensure a configurable, easy-to-use interface, the tool creates a skeleton for the XML file it needs to be configured with for each created project. The user is required to specify the path of each external software (CODESYS, internal test environment, scripts for import and export) and list the desired modules, specifying their name, language, related tests and operators. The tool implements a workspace-project-test session hierarchy and creates a corresponding directory structure for storing export and import XML files, logs, reports and configuration. As it was written in Python, it can be used both as an executable script or from a Python shell, depending on user preference. Each task is logged to the console, while mutant generation and test execution logs are also saved separately for each test session. While testing the tool in development, both CODESYS and the internal test execution tool occasionally froze, initially causing the test session to fail, which led to the implementation of exception handling mechanisms to increase robustness of the created tool.

³<https://docs.python.org/3/library/xml.etree.elementtree.html>

⁴<https://docs.python.org/3/library/re.html>

⁵<https://github.com/DanielMartensson/PLC-Simulations>

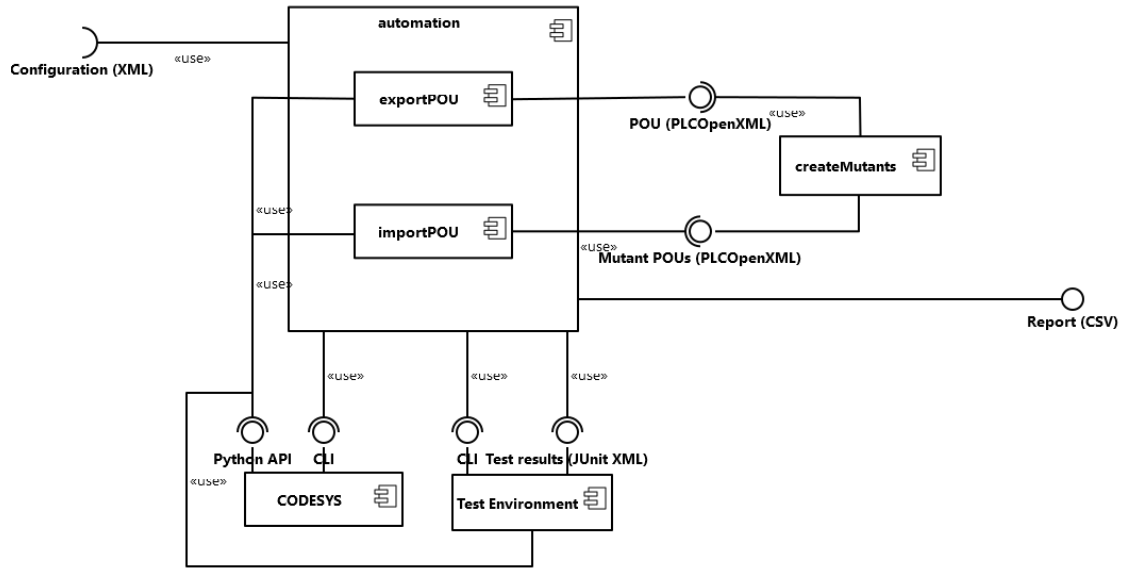


Figure 4.1: Component diagram of the mutation testing tool

4.3 Initial results

For the implementation and testing of the mutation testing tool, a software component consisting of 25 ST modules was provided to me by the previously surveyed team of engineers. Additionally, a smaller component of 5 CFC modules was also provided, 2 of which contained logic other than signal connections. While this was sufficient to test the created tool, it did not lead to any significant findings, and, as CFC modules are less frequently used at the company and usually implement simpler logic, they were not studied further and a more in-depth analysis remains the subject of further research. It is also important to note that the tests for these modules are still under development, the presented results are not their final state. This section presents the mutation testing results obtained on the ST modules and the initial feedback the tool and its results received from the surveyed engineers.

4.3.1 Results on industrial code

The created tool was used to fully test both components and their module-level tests. During testing, the regular expressions used by the mutant generator had to be updated several times to ensure the generation of syntactically correct mutants, and the error handling mechanisms of the tool were also improved to handle the occasional runtime errors of the toolchain. Additionally, the ABS operator was removed from the ST operator set, as it created by far the most mutants, the majority of which ended up causing errors during test execution and only a small portion of executable mutants were not killed. Furthermore, POUs having nested methods were not handled by the original mutant generator, as the test projects did not contain an example of this, a feature that had to be implemented later on.

Following several improvements, the results presented in Table 4.3 were acquired. The deviation among the mutation score and the total number of mutants for each module is

Module	#live	#killed	#total	Score
1	0	20	20	100.0%
2	14	53	67	79.1%
3	2	13	15	86.7%
4	0	28	28	100.0%
5	0	3	3	100.0%
6	1	16	17	94.1%
7	12	6	18	33.3%
8	0	0	0	-
9	19	47	66	71.2%
10	0	153	153	100.0%
11	0	109	109	100.0%
12	0	42	42	100.0%
13	3	53	56	94.6%
14	0	4	4	100.0%
15	1	21	22	95.5%
16	0	30	30	100.0%
17	4	104	108	96.3%
18	1	13	14	92.9%
19	5	18	23	78.3%
20	92	143	235	60.9%
21	1	76	77	98.7%
22	2	50	52	96.2%
23	0	27	27	100.0%
24	456	49	505	9.7%
25	4	13	17	76.5%

Table 4.3: Initial results on industrial code

both remarkably high, and modules with 100% mutation score are not uncommon, a trend unmatched in previous industrial case studies for other programming languages. The reason behind this could be the various quality and level of detail of requirements, the abilities of different test engineers or even the time allocated to test a certain module. Modules could also have very high, but not perfect scores because the few live mutants cannot be killed given the controllability and observability properties of the test environment.

4.3.2 Industry feedback

Mutation testing as a principle, the created tool, and the results acquired on the provided modules were presented to the previously surveyed group of engineers. The initial feedback on the usability of the tool and potential benefits of mutation testing was positive, but when faced with the results in Table 4.3, engineers found large numbers of live mutants overwhelming and hard to process. This motivated the proposition of further *metrics other than mutation score* and the creation of a *result visualization* module. They also expressed that *determining whether a mutant can be killed within the given test environment* can be a complex and time-consuming task that would decrease efficiency when improving test suites based on the results of mutation testing.

When the initial version of the mutation testing tool was created, the management of the previously surveyed group was also asked to allocate time to experiment with mutation testing in an upcoming project. Despite their enthusiasm and mostly due to strict deadlines, only one engineer was allocated to evaluate the created tool on an ongoing project. They were exhaustively interviewed on their experience with the tool and their opinion on the improvements presented in Section 4.4, and their feedback will be discussed in detail in Chapter 5.

Unsolved challenges based on initial feedback

While engineers saw and understood the benefits of mutation testing, they found *large numbers of live mutants* overwhelming and difficult to process, and expressed their concern regarding the temporal cost of *determining the equivalence of mutants*.

4.4 New metrics and visualization for ST mutation testing

When first introducing mutation testing to any environment, the initially large number of live mutants can be overwhelming and difficult to process, an issue that became exponentially more pronounced within the PLC context. This was amplified by the limitations of the development environment: engineers seeking to view mutants in context are either required to import each mutant individually, view their code in the XML export or manually recreate them using the mutation report. This motivated the visualization of results, while the need to classify and prioritize parts of the code based on the testing deficiencies present prompted the definition of metrics beyond mutation score. Both improvements are discussed in detail in this section.

4.4.1 Visualizing results

One of the main concerns expressed by test engineers when shown the results of mutation testing on a previously tested component was the overwhelming number of live mutants and their lack of experience in dealing with them. In order to organize the mutants, I started experimenting with possible methods of data visualization.

Initially, I created a module that generates HTML test reports in which lines of code are colored based on their individual mutation score. The number of killed and live mutants categorized by operator appeared as a tool-tip when hovering over each line. This provided a good overview of the location and distribution of live mutants, but information regarding the type and number of mutants was difficult to access.

In the next prototype, I moved the data from the tool-tip to a table located next to the code, which improved its accessibility, but was still difficult to overview. Taking into account the principles defined by Moody [13] for designing cognitively effective visual notations, I added another variable to visualization: brightness. In cases where the per-LOC (line of code) mutation score was not 100%, the brightness of the background color was adjusted based on the *number of live mutants* relative to the values within the module. This way, when faced with two lines of similar score, engineers can determine the complexity of those lines and thus the estimated cost of fixing the related tests based on their darkness. A line with more live mutants is darker, suggesting that killing the related mutants would be a more complex task.

This visualization summarizes multiple aspects of the results in a comprehensible way, but determining the exact results of individual mutants would still require the user to open and search the original per-mutant report. This inconvenience motivated the addition of one final feature: clicking on a line opens the per-line report, where each mutant and its result is listed. A screenshot of such a test report is shown in Figures A.3.1 and A.3.2 in the appendix.

This provides test engineers with information similar to that of a coverage report, showing which lines of code and to what extent are exercised by tests, and which are not. However,

it holds far more information than a simple coverage report, as that would not show e.g. if tests detect changing a mathematical or relational operator. In an environment where coverage measurement and reports were completely absent up to this point, this can be crucial information. Additionally, only a few, very advanced mutation testing tools, such as PIT [4] provide such mutation test reports.

Key insights on visualizing mutation testing results in PLC software

- Mutation testing reports (can) include more information than coverage reports
- Visualizing local score could ease the understanding of complex reports

4.4.2 Defining metrics for result evaluation

While useful when assessing the overall quality of tests for a module, mutation score was not particularly informative when aiming to improve the existing test suite with further test cases. However, the test report showed that addressing the mutation score of each line separately and considering other quantitative outputs of mutation testing can provide valuable additional information, especially in a setting where there is no code coverage measurement present. This motivated the proposition of further metrics based on the per-line mutation score.

To aid the definition of further metrics, the visualization module was extended to provide a CSV report containing all the data gathered on the mutated lines of code (MLOC): the number of live, killed and error-causing mutants for each operator and in total, the calculated scores, whether the lines contained relational operators and whether they were a part of an IF-condition. These attributes can be used and were used to search for correlations among data and create new metrics.

In an attempt to suggest a metric that can be used to prioritize certain areas of the code for test improvement, the information provided by the mutation score and the number of mutants generated from a line was examined. It was obvious that e.g. a lower mutation score should increase the priority to some extent, but the method to mathematically represent this was unclear to begin with. I proposed that instead of considering mutation score, further metrics ought to be derived from its complement, *the ratio of live mutants*. Additionally, the number of mutants within a line should be weighted based on the results within the given module, as declaring a concrete number of live mutants decisive when prioritizing areas for improvement would deprive the metric from its generic applicability. Ultimately, I suggested considering the product of *the relative number of mutants* within a line and its *complement mutation score* as a metric for prioritizing areas of the code for test improvement, but further work, broad feedback and longitudinal evaluation would be required to draw reliable conclusions on its applicability, even then leaving it potentially environment-specific.

In further search for a way to categorize areas of the code based on the extent to which they are exercised by tests, I recognized a much simpler, generally applicable pattern: lines can be organized into three groups based on their individual mutation score.

- **100% mutation score:** these lines are thoroughly tested and do not require the addition of further test cases
- **0% mutation score:** modifying these lines has absolutely no effect on test results. Their effect is either undetectable or the test is not at all prepared to detect its

effect. They probably require further test cases to be designed or existing ones to be extended with further checks.

- **Mutation score between 0% and 100%:** modification of these lines was *sometimes* detected by the test suite. Adding further input combinations (and corresponding checks) will most probably be sufficient for killing the live mutants.

Thus, the **line-based mutation score ratio** can be calculated for each module: the percentage of lines with 100% mutation score, that of lines with 0% score and that of lines in between the two. Higher ratio of perfect-score lines obviously means better test suite quality, while the ratio of 0 and non-zero scored lines shows what type of test improvements are needed and in what proportion.

Line-based mutation score ratio

The percentage of lines with 100% mutation score, 0% score, and scores in between.

4.5 Using mutants for model checker-based test generation

In their studies, both Enou et al. [9] and Liu et al. [11] used mutants of FBD programs for test generation. The former used model checking on an aggregated model of the original code and its mutants, while the latter used an SMT solver to find test data that distinguished mutant outputs from the original program's output. The common idea in their approaches is that tests to kill certain mutants can be generated by identifying inputs that result in different outputs from the original program and its mutant(s).

4.5.1 Proposal

In order to aid test engineers in designing tests to kill live mutants, I propose to apply this approach to only the live mutants of a given module. This is justified by the fact that the existing tests are usually designed to cover all requirements, must exist and could not always be generated automatically, while the additional tests only serve test suite improvement. To prove the feasibility of this approach for ST programs, I created a series of examples using first a simple code example, then one of the modules from the industrial context and instrumenting them in PLCverif in a way that allows the model checker to provide examples of input combinations that would result in different outputs from the original and mutant code.

Algorithm 1: Using a model checker to generate test inputs for killing live mutants

Data: original POU $\mathbf{p_o}[i[t], t]$ and its *live* mutants $\mathbf{p_1}[i[t], t], \mathbf{p_2}[i[t], t], \dots, \mathbf{p_n}[i[t], t]$

Result: input sequences $\mathbf{i_1}[t], \mathbf{i_2}[t], \dots, \mathbf{i_n}[t]$ or proof that $\nexists \mathbf{i_k}[t]$ for which

$\mathbf{p_k}[i_k[t], t] \neq \mathbf{p_o}[i_k[t], t]$, where $k \in \{1, 2, \dots, n\}$

```

1 for  $k \leftarrow 1$  to  $n$  do
2   Use a model checker to prove  $\mathbf{p_k}[i[t], t] = \mathbf{p_o}[i[t], t] \forall i[t]$ ;
3   if  $\exists i[t]$  for which  $\mathbf{p_k}[i[t], t] \neq \mathbf{p_o}[i[t], t]$  then
4     Record  $i[t]$  as  $\mathbf{i_k}[t]$ ;
5   else
6     Proof that  $\nexists \mathbf{i_k}[t]$ ;
7   end
8 end
```

4.5.2 Proof of Concept

Simple example To gain a better understanding of how PLCverif can be used for mutant-based test generation, I first created a simple example using an absolute value function and two of its mutants, shown below.

```

1  FUNCTION FuncAbs : INT
2  VAR_INPUT
3    in : INT;
4  END_VAR
5  BEGIN
6    IF in > 0 THEN
7      FuncAbs := in;
8    ELSE
9      FuncAbs := -in;
10   END_IF;
11 END_FUNCTION

```

Listing 4.1: original

```

1  FUNCTION FuncAbs : INT
2  VAR_INPUT
3    in : INT;
4  END_VAR
5  BEGIN
6    IF in >= 0 THEN
7      FuncAbs := in;
8    ELSE
9      FuncAbs := -in;
10   END_IF;
11 END_FUNCTION

```

Listing 4.2: mutant 1

```

1  FUNCTION FuncAbs : INT
2  VAR_INPUT
3    in : INT;
4  END_VAR
5  BEGIN
6    IF in < 0 THEN
7      FuncAbs := in;
8    ELSE
9      FuncAbs := -in;
10   END_IF;
11 END_FUNCTION

```

Listing 4.3: mutant 2

I then added code that fed the same input to both the original and mutated function and returned whether the two return values matched:

```

1  FUNCTION aggr_1 : BOOL
2  VAR_INPUT
3    in : INT;
4  END_VAR
5  VAR_TEMP
6    out_o : INT;
7    out_1 : INT;
8  END_VAR
9  BEGIN
10   out_o := FuncAbs(in := in);
11   out_1 := FuncAbs_1(in := in);
12   aggr_1 := (out_o = out_1);
13 END_FUNCTION

```

Listing 4.4: aggregating function for mutant 1

```

1  FUNCTION aggr_2 : BOOL
2  VAR_INPUT
3    in : INT;
4  END_VAR
5  VAR_TEMP
6    out_o : INT;
7    out_2 : INT;
8  END_VAR
9  BEGIN
10   out_o := FuncAbs(in := in);
11   out_2 := FuncAbs_2(in := in);
12   aggr_2 := (out_o = out_2);
13 END_FUNCTION

```

Listing 4.5: aggregating function for mutant 2

In order to find an input that distinguishes the mutant from the original code, I prompted the model checker to verify whether the return value of the aggregating function is always TRUE. To do this, I used the pattern requirement specification option of PLCverif to specify the requirement *aggr_{1,2}.RET_VAL is always true at the end of the PLC cycle* and both Theta and NuSMV with default configuration to verify it. For mutant 1, the requirement was satisfied, whereas for mutant 2, the requirement was violated and the counterexamples in Figures 4.2a and 4.2b were provided. Screenshots of the exact configuration and results of the verification are shown in Figures A.4.1 and A.4.2 of the appendix.

Both counterexamples show a valid test input that could be used to implement a test that kills mutant 2. Whereas for mutant 1, neither model checkers found an input that could distinguish the mutant from the original code. In this case, it is trivial that the mutant is truly equivalent, while in more complex cases, as shown below, it may not be.

Counterexample

	Variable	Beginning of Cycle 1	End of Cycle 1
signed int16	FuncAbs_2.RET_VAL	0	-16384
signed int16	FuncAbs_inlined_2.RET_VAL	0	16384
bool	aggr_2.RET_VAL	false	false
INPUT INT	aggr_2.in	-16384	-16384

(a) NuSMV

Counterexample

	Variable	Beginning of Cycle 1	End of Cycle 1
signed int16	FuncAbs_2.RET_VAL	0	-1
signed int16	FuncAbs_inlined_2.RET_VAL	0	1
bool	aggr_2.RET_VAL	false	false
INPUT INT	aggr_2.in	1	1

(b) Theta

Figure 4.2: Counterexamples for verification case of mutant 2

Industrial example To explore the scalability of this approach and the feasibility of its application in an industrial context, I applied it to the previously presented example module, shown in Listings A.2.1 and A.2.2 of the appendix.

First, I imported the module to PLCverif, resolving any external dependencies and partially rewriting the module to adhere to SCL syntax. While both CODESYS ST and STEP7 SCL conform to the IEC61131-3 standard, there were minor differences between what CODESYS and PLCverif accept when defining e.g. constants or enums.

The module had 5 live mutants – all obtained by modifying an OR operator in a complex logical expression to XOR – which I recreated by performing the same modifications on the adapted code. The mutations are shown in Listings A.2.3, A.2.4, A.2.5, A.2.6 and A.2.7. To ensure that the method also finds input combinations for already killed mutants, I also added a mutant that existing tests had already killed, shown in Listing A.2.8.

As the module is implemented as a function block, the aggregating code also had to be implemented as a function block, in order to preserve internal states and variable values between software cycles. The created function block’s inputs are the module’s inputs, its output is a single boolean value that is set to TRUE when all outputs and internal variables of the instantiated original and mutant function blocks match. An example of such an aggregating function block is shown in Listing A.4.1.

When verifying a function block, PLCverif instantiates it. Therefore, the requirement specified in this case was *instance.outputs_match is always true at the end of the PLC cycle* and the entry point was the aggregating function block. I tried using both Theta and NuSMV to verify this requirement, but Theta ran out of heap space before providing a solution. However, NuSMV found counterexamples for all provided requirements within 25 seconds, but one, mutant #36 (Listing A.2.7) for which it declared the defined requirement satisfied after approximately 4 minutes. When examining this mutant, the justification of its equivalence became trivial, and therefore will not be discussed in detail. In case of the already killed mutant, both Theta and NuSMV found that the requirement was violated and provided counterexamples to prove so.

A screenshot of the configuration of one of the verification cases can be found in the appendix (Figure A.4.3), alongside the counterexamples provided by the two model checkers for the killed mutant (Figures A.4.4 and A.4.5). The counterexamples provided for the live mutants were over 150 PLC cycles long and therefore difficult to depict within this document, but the entire PLCverif project with which this experiment can be reproduced was attached to this report.

Proposed approach for analysing live mutants

Mutants can be instrumented using the presented method and a model checker can be prompted to generate test input sequences to kill them or prove they cannot be killed. Applying this to *live* mutants allows for their automatic handling.

4.5.3 Conclusion

As demonstrated in the example above, PLCverif and a model checking backend, such as NuSMV or Theta, can be instrumented to generate input combinations that can be used to distinguish mutant code from original code, aiding test engineers in complementary test design. Additionally, it can also be used to prove how certain mutants are *pseudo-equivalent*, meaning that even though they might not be semantically equivalent with the original code, the limitations of the test environment make it impossible to design tests that would differentiate between the mutant and the original module. This can be used to aid test engineers in determining whether a mutant can be killed to begin with, addressing their initial concern on this topic.

The proposed method could be automated by extracting the interface of modules from the PLCOpenXML export, generating the aggregating code, and running PLCverif using its CLI. Furthermore, with some additions to PLCverif and automation of the test design tool, the method could be used to implement mutant-based automated test generation for ST programs in the given industrial setting, a demand formulated by the surveyed test engineers.

Chapter 5

Evaluation

In order to assess the feasibility of the solutions proposed in Chapter 4, I subjected them to both qualitative and quantitative evaluation. The former is detailed in Section 5.1, while the latter in Section 5.2.

5.1 Qualitative evaluation: Evaluative case study

The mutation testing tool was handed over to an engineer to use in an ongoing project over the span of a month, who was then exhaustively interviewed on their experiences. Additionally, the lead engineer who aided the development of the initial version of the tool was also questioned on their views on the proposed improvements. First, the exact parameters of this *evaluative case study* are presented, followed by the obtained results and the conclusions drawn from them.

5.1.1 Research methodology

I aimed to conduct an evaluative case study on the application of mutation testing in the PLC domain and its proposed improvements in the given industrial environment. To ensure reliable case study reporting, I sought to adhere to the *ACM SIGSOFT Empirical Standards for Software Engineering* on conducting a case study¹ as closely as the limitations of the studied industrial setting allowed me.

Timeline As mentioned in Section 4.3.2, mutation testing as a principle and the initial version of the created tool was presented to the group of surveyed engineers. Following this demonstration, the environment required to use the tool in production was assembled and I had a follow-up meeting with the engineer who was to use it, where we set up the project and went through the usage of the mutation testing tool again. In the following month, they used the tool while developing tests for another version of the same module the tool was tested on. Meanwhile, I developed the improvements presented in Sections 4.4 and 4.5. Following the 1-month evaluation and development period, I presented the improvements to the lead engineer and questioned them on their views, followed by an in-depth interview with the engineer who worked with the mutation testing tool.

¹<https://www2.sigsoft.org/EmpiricalStandards/docs/standards?standard=CaseStudy#>

Part 1: Questions related to the <i>learning process</i>	
1	Describe your experience of understanding mutation testing as a concept. Were there any difficulties?
2	Describe your experience of understanding the usage of the mutation testing tool. Were there any difficulties?
Part 2: Questions related to the <i>initial version of the tool</i>	
1	What useful information did the tool provide?
2	Do you miss any feature from the tool? If yes, elaborate.
3	Do you think the benefits of using the tool are worth the time and effort allocated to learn and use it?
4	Would you, personally, use the tool in your further work?
5	How would you integrate the tool into your workflow?
Part 3: Questions related to the tool complemented with <i>visualization and metrics</i>	
1	What useful information did the tool provide?
2	Do you miss any feature from the tool? If yes, elaborate.
3	Do you think the benefits of using the tool are worth the time and effort allocated to learn and use it?
4	Would you, personally, use the tool in your further work?
5	How would you integrate the tool into your workflow?
Part 4: Questions related to <i>mutant-based test generation</i>	
1	Do you find the additional information provided by this approach useful?
2	Would you use the approach to design tests to kill mutants, or prove they cannot be killed? How?
3	Would you use the approach for automated test generation? How?

Table 5.1: Interview questions

Interview structure The conversation conducted with the lead engineer did not follow a specific format. The improvements were simply presented to them and they were asked to share their thoughts and opinions, based on their prior experience in the field and at the company.

The interview conducted with the engineer using the mutation testing tool was composed of the questions shown in Table 5.1. These questions were organized around 4 topics: the learning process, the initial version of the tool, the visualization and metrics, and mutant-based test generation. The interview was conducted over a conference call, recorded, and later notes were made to summarize the received answers. There was no strict time frame for the interview, the subject could take any amount of time to answer the questions, was not required to answer every question and was free to ask for clarifications.

The interview was organized as follows: first, the subject was asked the first two parts of the interview questions, followed by a demonstration of the visualization module and the proposed metrics. Then, the third part of the questions was asked, the mutant-based test generation approach was introduced, and the interview was concluded with the fourth set of questions. Finally, the subject was asked to share any final thoughts the interview might have missed but they found important.

5.1.2 Results

The engineer managed to perform the mutation testing of 8 modules throughout the month they worked with it. During this time, they mostly focused on finding the method’s place in the workflow and developing tests, but they did report on improving two test cases directly due to their mutation testing results. The interview conducted with them was recorded, and notes were made summarizing the answers given based on the recording.

The feedback from the lead engineer was also organized into notes. Both can be found in the attachment of this report.

Learning process The lead engineer was introduced to mutation testing long before the interview was conducted and assisted the communication with the team of engineers throughout this study, therefore their experience with understanding mutation testing and its usage in the current situation would be vastly different from that of the target user's.

However, when asked to describe their experience with understanding the concept of mutation testing and tool usage, the interviewed engineer found both quite straight-forward. They reported understanding mutation testing the first time it was explained to them, and the process of its application in the given setting when the tool was first demonstrated. The usage of the tool was clear by then, but following the second demonstration session, when we set it up for a project, they required no further support.

Initial version of tool The lead engineer encouraged and followed the development of the mutation testing tool, therefore obviously saw its benefits and potential added value, and sought to find its place in the workflow with the help of this case study.

The interviewed engineer found that the added value of mutation testing lied in its capacity to either confirm the adequacy of a test suite or point out the deficiencies, also allowing them to learn from their mistakes. They explicitly phrased their need for a result visualization module and also a graphical user interface, as that is what they are familiar with from the test environment used for PLC software at the company. They also noted that the unreliability of the tools automated by the mutation testing tool caused some inconvenience, but nothing that hindered its application profoundly.

In their experience, understanding mutation testing and learning to use the tool took the 3 hours we spent in demonstrative sessions, setting up the environment to use it did not take more than 1 to 2 hours, executing tests is a process that can be left to run overnight, and evaluating results, if done regularly, is an approximately 10-minute task. Therefore, they did not find the application of mutation testing a time consuming process, while its added value, even if it was occasionally only reassurance of good test quality, was undoubted. If given the opportunity, they would continue to use the tool in their work, mainly to evaluate the quality of test suites they think to have finished developing. They would do so by running mutation testing overnight once they think to have completed the test suite for a module or a set of modules, complement their tests based on its results the upcoming day, followed by a rerun of the live mutants to verify that they were indeed killed.

Proposed improvements The visualization module and the proposed metrics were additions that had not been previously presented to the lead engineer. They found the test report highly informative and suggested adding an overview of the results to the beginning of it. When shown the initially proposed metrics, they suggested splitting the category with scores above 0% but below 100% into two parts, below and above 50%. They justified this hypothesizing that lower per-line scores are more likely to be caused by real testing deficiencies, while higher, but not perfect scores are more probably caused by mutants that cannot be killed within the given environment. During the conversation, we concluded that the result summary at the beginning of the test report could include these exact metrics.

The interviewed engineer considered the test report to be a much more ergonomic presentation of results, that could make result evaluation significantly more efficient. While

they did agree that the per-line mutation score would help them roughly categorize the type of testing deficiency present, line-based mutation score ratio, which shows the exact distribution of different scores, would be a metric they would use to show progress towards management. However, they later added how it could be used to assess the initial quality of tests when e.g. testing a change request.

As they already declared the initial version of the mutation testing tool worthwhile, their opinion did not change on the topic, but they did highlight how the improved test report spares time, making the application of mutation testing even more efficient. When asked how they would integrate the improved tool into their workflow, they added that they would also use mutation testing to evaluate test suites when receiving a change request, highlighting how it could show untested, new areas of code, similarly to a coverage report.

Test generation The mutant-based test generation method was also a new concept for the lead engineer. During the development of the tool and the course of the study they had expressed their concern regarding how difficult and time-consuming determining the equivalence of mutants would turn out to be, and were relieved by the possibility to prove that certain mutants can not be killed in the given setting.

The interviewed engineer did not experience any difficulty when determining whether certain mutants can be killed, and highlighted how e.g. certain combinations of conditions are already ruled out during testing. However, the possibility to generate tests to kill mutants or prove they cannot be killed was appealing to them. When asked how they would use this approach to gain information on live mutants, they imagined an automatically appearing status e.g. as part of a more detailed test report. When asked if and how they would use the approach for automated test generation, while not rejecting it, they expressed concern regarding the traceability of such tests to requirements and how the inability to generate mutants for a module would also mean an inability to generate tests.

5.1.3 Discussion

Summary of results While the conducted interview does not allow for generalization, the feedback received on the proposed approaches was overwhelmingly positive, and suggests that at least conceptual solutions were provided to all three difficulties identified in Chapter 3:

- **Toolchain limitations:** the proposed result visualization can serve as a coverage report, showing test engineers the extent to which different areas of the code are exercised. This can prove exceptionally useful when test suites need to be updated following code changes.
- **Lack of consistent metrics:** mutation score has previously been determined a consistent measure of test suite quality [2]. Experience now suggests that per-line mutation score could provide information on the specific type of testing deficiency, and line-based mutation score ratio a traceable and comparable metric.
- **Automation potential:** the proposed mutant-based test generation approach can also be applied to automate test generation, though questions emerge regarding its feasibility in the studied setting.

Furthermore, mutation testing was also considered an effective way to identify personal shortcomings in test design, allowing experienced and novice test engineers alike to learn

from their mistakes and improve their skills. The proposed result visualization was deemed to increase efficiency, whereas the proposed metrics helped categorize and better understand results.

Future work The result visualization, metrics, and the model checker-based approach to equivalence detection and test generation are measures applicable to mutation testing in other settings, far beyond the PLC industry. Furthermore, the designed interview can serve as the basis of a longitudinal study, or one of similar temporal length, but with significantly more participants, which would be essential in order to increase the certainty and the generalizability of the findings.

Additionally, the improvements proposed by the engineers could be added to the tool: a graphical user interface to start test sessions or even to manage the tests of entire projects, and a summary of results at the beginning of test reports. A more in-depth collaboration with the developers of the internal software testing environment would allow for the integration of mutation testing directly in the development toolchain, and could lead to more efficient execution, drastically lowering the computation time required.

5.2 Quantitative evaluation: Experiment on industrial codebase

Following the initial testing on industrial code discussed in Section 4.3, the mutation testing tool was extended with the features detailed in Section 4.4 along with other minor fixes, and used to retest the ST software component, using the results to evaluate the proposed metrics.

5.2.1 Evaluation on industrial code

The new test reports revealed that, while the generated mutants were correct, the mutant generator failed to identify operators in more complex code environments, mostly due to the syntactical complexity of the studied code. Therefore, the regular expressions were updated once again, and a retesting of the studied ST software component was performed, yielding significantly more data than the initial results.

A total of *3204 mutants* were generated from *4227 lines of code*, which, estimating with about 6 minutes of execution time per mutant, disregarding the time lost when the development tools froze, would have ran for about *13 days*. Due to the downtime caused by the unreliable development tools and hardware unavailability, the two largest modules, #20 and #24, with 496 and 967 mutants respectively, were not reexecuted yet, and their previous results are shown. Their results will be marked throughout this chapter.

5.2.2 Results on industrial code

After more than two weeks of overall execution time, the results presented in Table 5.2 were acquired. The table's columns contain, in this order, the following information: the number used to identify a module, the number of lines of code in the module, the number

Mod#	LOC	#live	#killed	#total	Score	# score/MLOC				MLOC
=0%	<50%	<100%	=100%							
1	35	0	38	38	100.0%				11	11
2	182	38	117	155	75.5%	5	3	9	25	42
3	30	4	43	47	91.5%			4	6	10
4	78	2	52	54	96.3%			2	12	14
5	11	0	6	6	100.0%				2	2
6	64	3	27	30	90.0%			3	10	13
7	56	25	15	43	37.5%	2	1	2	4	9
8	59	0	34	34	100.0%				14	14
9	145	32	78	110	70.9%	7		7	26	40
10	378	1	200	201	99.5%			1	98	99
11	243	0	226	226	100.0%				56	56
12	81	0	64	64	100.0%				22	22
13	163	9	92	101	91.1%			8	33	41
14	45	0	6	7	100.0%				3	3
15	36	5	50	55	90.0%			2	8	10
16	114	3	71	74	95.9%			3	16	19
17	208	10	145	156	93.5%	1		7	52	60
18	51	3	13	16	81.3%	1		2	9	12
19	125	19	60	79	75.9%	6	1	1	22	30
20*	440	92	143	236	60.9%	24		9	57	90
21	125	3	106	109	97.2%			3	62	65
22	162	10	67	77	87.0%			4	37	41
23	55	3	30	33	90.9%			3	9	12
24*	1278	456	49	507	9.7%	216			27	243
25	63	5	17	22	77.3%	1		2	9	12
Total*	4227	723	1749	2484	70.8%	263	5	72	630	970

Table 5.2: Summary of final results on industrial codebase

of live mutants, killed mutants and the total number of mutants², the mutation score, the number of mutated lines of code achieving per-line scores of the given category, and the total number of mutated lines of code in the module.

When categorizing lines based on their mutation score, I chose to separate the lines above and below 50% scores, as proposed by the lead engineer. Their original reasoning was that lines with higher scores are more likely to be caused by mutants that cannot be killed in the given environment. Simple examples of such lines are shown in line 9 of Listings 5.1-5.3 and in Listings 5.4-5.6. These are also examples of how lines with exactly 50% score can often be part of the category with higher scores, and the reason why lines with exactly 50% score were assigned to this group.

```

1 IF (...) THEN
2   Module_In_State_1 := TRUE;
3 ELSE IF (...) THEN
4   Module_In_State_2 := TRUE;
5 ELSE
6   Module_In_State_1 := FALSE;
7   Module_In_State_2 := FALSE;
8 END_IF; END_IF;
9 StateValid :=
  Module_In_State_1 OR
  Module_In_State_2;
```

Listing 5.1: original

```

1 IF (...) THEN
2   Module_In_State_1 := TRUE;
3 ELSE IF (...) THEN
4   Module_In_State_2 := TRUE;
5 ELSE
6   Module_In_State_1 := FALSE;
7   Module_In_State_2 := FALSE;
8 END_IF; END_IF;
9 StateValid :=
  Module_In_State_1 AND
  Module_In_State_2;
```

Listing 5.2: mutant 1

```

1 IF (...) THEN
2   Module_In_State_1 := TRUE;
3 ELSE IF (...) THEN
4   Module_In_State_2 := TRUE;
5 ELSE
6   Module_In_State_1 := FALSE;
7   Module_In_State_2 := FALSE;
8 END_IF; END_IF;
9 StateValid :=
  Module_In_State_1 XOR
  Module_In_State_2;
```

Listing 5.3: mutant 2

```

1 PressureValid := Pressure <
  REAL#0.2;
```

Listing 5.4: original

```

1 PressureValid := Pressure >=
  REAL#0.2;
```

Listing 5.5: mutant 1

```

1 PressureValid := Pressure <=
  REAL#0.2;
```

Listing 5.6: mutant 2

During the course of this test session, no further bug was found within the tool, but the delays caused by the unreliability of the development software automated by the mutation

²Error-causing mutants e.g. those that lead to test timeout, are not included in the table due to their low number. If the sum of live and killed mutants differs from the total number of mutants, the difference is the number of error-causing mutants

testing tool did bring forth a growing need for not only exception handling mechanisms, but a watchdog and automated retrying of failed tests.

Beyond testing the created tool with realistic input, the goal of this analysis was to evaluate the adequacy of the proposed metrics. To better visualize the results, I illustrated the line-based mutation score ratio of each module in Figure 5.1 and the complement of the mutation score for each module in Figure 5.2. I chose the complement of the mutation score because it visualizes the extent to which tests ought to be improved more clearly than mutation score.

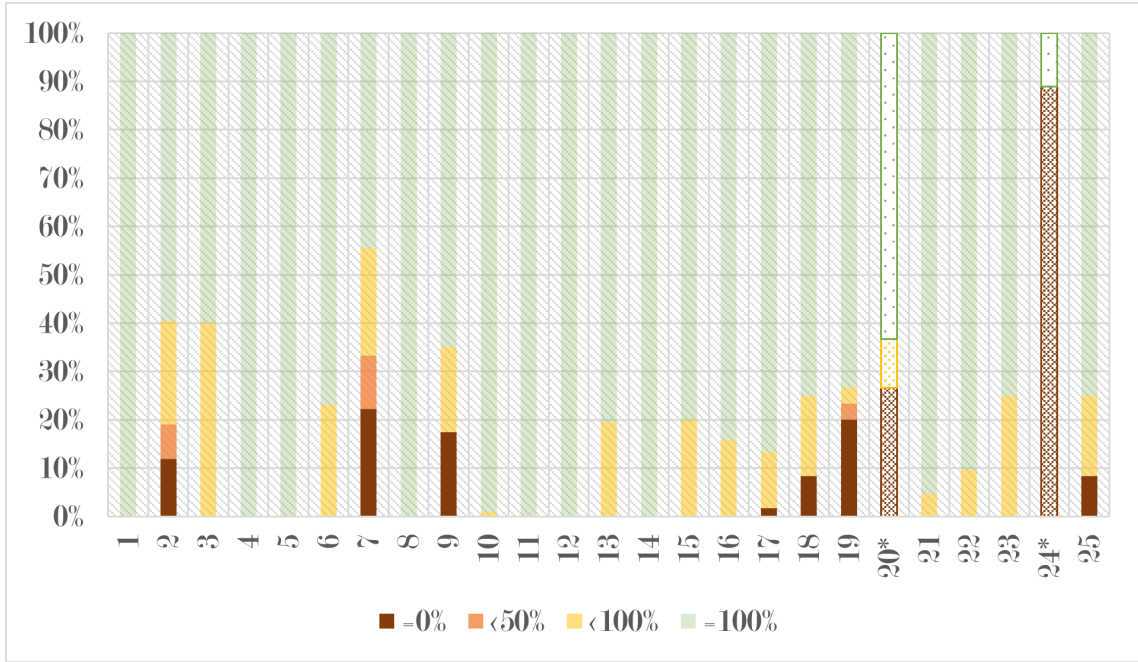


Figure 5.1: Line-based mutation score ratio of modules

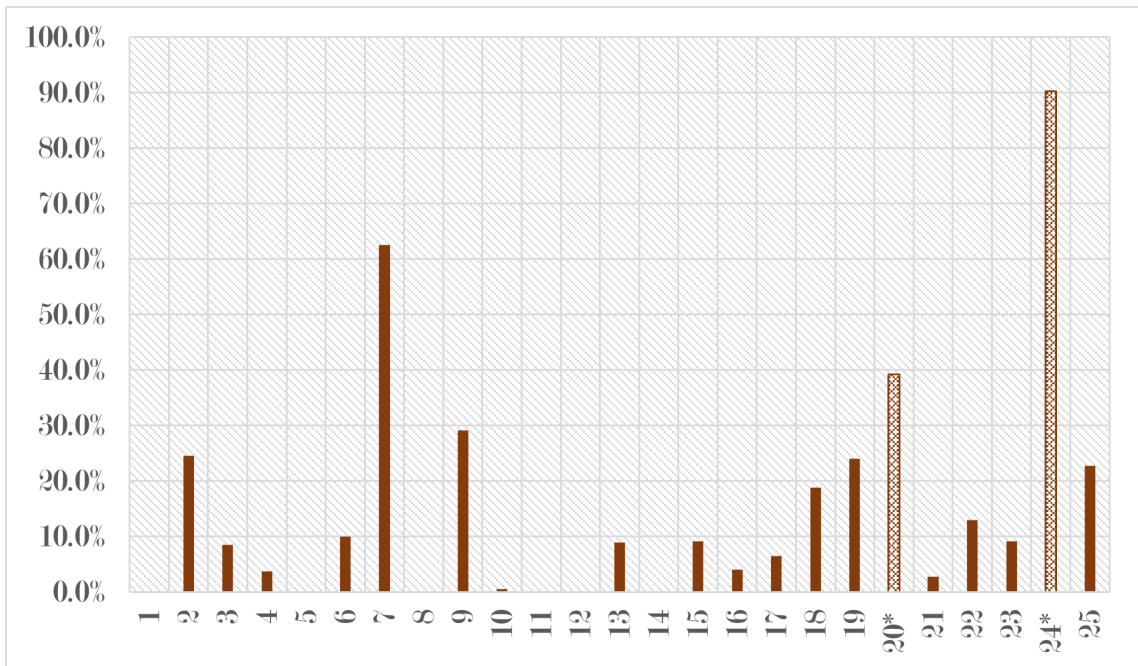


Figure 5.2: Complement mutation scores of modules

5.2.3 Discussion

Summary of results The extensive and unproblematic use of the created mutation testing tool on such a diverse and complex codebase and over a long period of time demonstrates its stability and scalability at least in the studied environment. The acquired dataset allowed for the evaluation of the proposed metrics: per-line mutation score and line-based mutation score ratio. Figures 5.1 and 5.2 show how mutation score on its own only reveals an overview of test quality for a module, while the proposed metrics reveal information about the type (and location) of testing deficiencies, mitigating the sometimes overwhelming effect of low mutation scores. The diagrams also show how high mutation scores can sometimes hide a multitude of well-distributed shortcomings within a module (e.g. Module #3), and how other times low mutation scores can originate from a small number of localized testing deficiencies (e.g. Module #19).

Future work The data acquired from the mutation testing of such a large codebase goes far beyond what is summarized in Table 5.2 and contains all the data specified in Section 4.4.2. Taking into account that, at least in this setting, acquiring such large amounts of data at this level of detail is not only resource-intensive, but also time consuming, the dataset could be published on its own, given the consent of the studied company. Even if the code itself is filtered out, the information can be used to further study applicable metrics and emerging phenomena in the field of PLC mutation testing.

Additionally, algorithmic improvements to mitigate the unreliability of the automated software could be added, in order to make the created tool more robust, and increase the efficiency of its application.

Chapter 6

Conclusion

In this report, I conducted a *mixed method study* to identify problems in PLC software testing, provide solutions to the emerged challenges, and evaluate them qualitatively and quantitatively with the help of industry professionals. The proposed solutions revolve around the topic of mutation testing, a test evaluation method that has been thoroughly studied and improved over the past decades, and numerous case studies proved its rightful place in industrial software development. Despite all evidence, mutation testing is yet to become a prevalent technique of test quality assessment. One of the underlying causes is how its application and benefits can seem trivial to academics familiar with it, but its target users, engineers, may struggle to comprehend and efficiently use the provided results.

This report focuses on proposing and evaluating methods that *improve the applicability* of mutation testing, and measures the *industrial impact* of the provided solutions. It is among the first to study mutation testing in the PLC industry and lays the groundwork for multiple areas of further research, within and outside the field of PLC software testing.

6.1 Research results

Identified challenges Chapter 3 presented the research methodology and results of the *questionnaire survey* conducted at a company that develops PLC software for railway applications. The results of the survey were analyzed and used to identify 3 problematic areas in PLC software testing. This established the main focus of the following research, as I proposed methods to improve test efficiency targeting these exact areas.

Proposed solutions In Chapter 4, I performed *design science research*, proposing the application of mutation testing in the studied environment and reporting on its adaptation for PLC software, the initial results and industry feedback received. Based on the problems formulated by potential users, I suggested a visualization method, and new metrics derived from the results of mutation testing, *line-based mutation score ratios*. Both approaches are applicable outside the PLC industry. Furthermore, I showed how model checkers and mutants can be used for test generation and *pseudo-equivalent mutant detection* in textual PLC software, implementing a theory that can be applied to virtually any programming language.

Industrial impact research Finally, in Chapter 5, I evaluated the proposed solutions in an *industrial case study*. Engineers were interviewed on their experience with the created mutation testing tool and their views on the proposed further improvements, and provided overall positive feedback. The mutation testing tool was also used to evaluate an industrial software component in development, proving its scalability. Analysis of test results also showed how the newly proposed metric, *line-based mutation score ratio*, *provide additional information* and a *better overview* of mutation testing results. Overall, it can be concluded that at least conceptual solutions were provided to all 3 challenges identified in Chapter 3 and significant advancements were made on the application of mutation testing in the PLC industry.

6.2 Further work

To gain better understanding of the challenges of PLC software testing, the questionnaire survey conducted in Chapter 3 could be extended, then executed with a significantly larger number of participants, from several different companies, serving different industries. This would allow for the generalization of results and aid academics in identifying areas demanding improvement.

The methods and tools presented in Chapter 4 could be studied in greater detail or applied to new areas altogether. The mutant generation engine could be used to create similar mutation testing tools for different environments, allowing for the application and further study of mutation testing in the PLC industry. Additionally, the proposed visualization method and metrics could be studied in other settings, even vastly different from PLC software, and users could be surveyed or interviewed on their feasibility. As previously stated, using mutants for test generation with model checkers, and using model checkers to detect equivalence *in theory* can be applied to any language, and could result in advancements on both topics.

Lastly, the evaluative case study presented in Section 5.1 could be extended to a larger number of participants within the studied company, and the effects of mutation testing on overall test and software quality could be examined over a longer period of time. The dataset acquired from the industrial codebase as part of the quantitative evaluation Section 5.2 reports on could be published after full execution is completed and sensitive data is filtered out. It could then be used to identify further applicable metrics and potential correlations within the data.

Bibliography

- [1] Hiralal Agrawal, Richard A DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, and Eugene Spafford. Design of mutant operators for the C programming language. Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue ..., 1989.
- [2] Richard Baker and Ibrahim Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, 2012. DOI: 10.1109/TSE.2012.56.
- [3] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification: 14th International Conference*, pages 359–364. Springer, 2002. DOI: 10.1007/3-540-45657-0_29.
- [4] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, page 449–452. ACM, 2016. DOI: 10.1145/2931037.2948707.
- [5] International Electrotechnical Commission. IEC 61131-3: Industrial automation systems - Programmable controllers - Part 3: Programming languages. *International Electrotechnical Commission Standard*, 61131-3, 2003.
- [6] Oscar Cornejo, Fabrizio Pastore, and Lionel C Briand. Mutation analysis for cyber-physical systems: Scalable solutions and results in the space domain. *IEEE Transactions on Software Engineering*, 48(10):3913–3939, 2021. DOI: 10.1109/TSE.2021.3107680.
- [7] Dániel Darvas, Enrique Blanco Vinuela, and Borja Fernández Adiego. PLCverif: A tool to verify PLC programs based on model checking techniques. In *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 911–914, 2015. DOI: 10.18429/JACoW-ICALEPCS2015-WEPGF092.
- [8] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. Evaluation of mutation testing in a nuclear industry case study. *IEEE Transactions on Reliability*, 67(4):1406–1419, 2018. DOI: 10.1109/TR.2018.2864678.
- [9] Eduard P Enoiu, Daniel Sundmark, Adnan Čaušević, Robert Feldt, and Paul Pettersson. Mutation-based test generation for PLC embedded software using model checking. In *Testing Software and Systems: 28th IFIP WG 6.1 International Conference, ICTSS 2016*, pages 155–171. Springer, 2016. DOI: 10.1007/978-3-319-47443-4_10.

- [10] Zhan Huang and Rob Alexander. Semantic mutation testing for multi-agent systems. In *Engineering Multi-Agent Systems: Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers 3*, pages 131–152. Springer, 2015. DOI: 10.1007/978-3-319-26184-3_8.
- [11] Lingjun Liu, Eunkyong Jee, and Doo-Hwan Bae. MuFBDTester: A mutation-based test sequence generator for FBD programs implementing nuclear power plant software. *Software Testing, Verification and Reliability*, 32(8):e1815, 2022. DOI: 10.1002/stvr.1815.
- [12] Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. An empirically evaluated checklist for surveys in software engineering. *Information and Software Technology*, 119:106240, 2020. ISSN 0950-5849. DOI: 10.1016/j.infsof.2019.106240.
- [13] Daniel Moody. The “physics” of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 35(6):756–779, 2009. DOI: 10.1109/TSE.2009.67.
- [14] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996. DOI: 10.1145/227607.227610.
- [15] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from Google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2021. DOI: 10.1109/TSE.2021.310763.
- [16] PLCOpen Technical Committee 6. XML Formats for IEC 61131-3. Technical Report Version 2.01 - Official Release, PLCOpen, 2009. URL https://plcopen.org/system/files/downloads/tc6_xml_v201_technical_doc.pdf.
- [17] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. An empirical study on the application of mutation testing for a safety-critical industrial software system. In *Proceedings of the Symposium on Applied Computing*, pages 1401–1408, 2017. DOI: 10.1145/3019612.3019830.
- [18] Ana B Sánchez, José A Parejo, Sergio Segura, Amador Durán, and Mike Papadakis. Mutation testing in practice: Insights from open-source software developers. *IEEE Transactions on Software Engineering*, 2024. DOI: 10.1109/TSE.2024.3377378.
- [19] Andrada A Serban and Zoltán Micskei. Application of mutation testing in safety-critical embedded systems: A case study. *Acta Polytechnica Hungarica*, 21(8), 2024. DOI: 10.12700/APH.21.8.2024.8.5.
- [20] Andrada A. Serban, Zoltán Micskei (konzulens), and Péter Rozgonyi (konzulens). Mutációs tesztelés alkalmazása biztonságkritikus beágyazott rendszerek tesztelésében – BME-VIK TDK, november 2022.
- [21] Donghwan Shin, Eunkyong Jee, and Doo-Hwan Bae. Empirical evaluation on FBD model-based test coverage criteria using mutation analysis. In *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012*, pages 465–479. Springer, 2012. DOI: 10.1007/978-3-642-33666-9_30.
- [22] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In *Proceedings of the*

- 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. DOI: 10.23919/FMCAD.2017.8102257.
- [23] Stefan Alexander van Heijningen, Theo Wiik, Francisco Gomes de Oliveira Neto, Gregory Gay, Kim Viggedal, and David Friberg. Integrating mutation testing into developer workflow: An industrial case study. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2110–2120, 2024.
- [24] Kun Wang, Jingyi Wang, Christopher M Poskitt, Xiangxiang Chen, Jun Sun, and Peng Cheng. K-ST: A formal executable semantics of the structured text language for PLCs. *IEEE Transactions on Software Engineering*, 2023. DOI: 10.1109/TSE.2023.3315292.

Appendix

A.1 Example FBD code and mutation operators

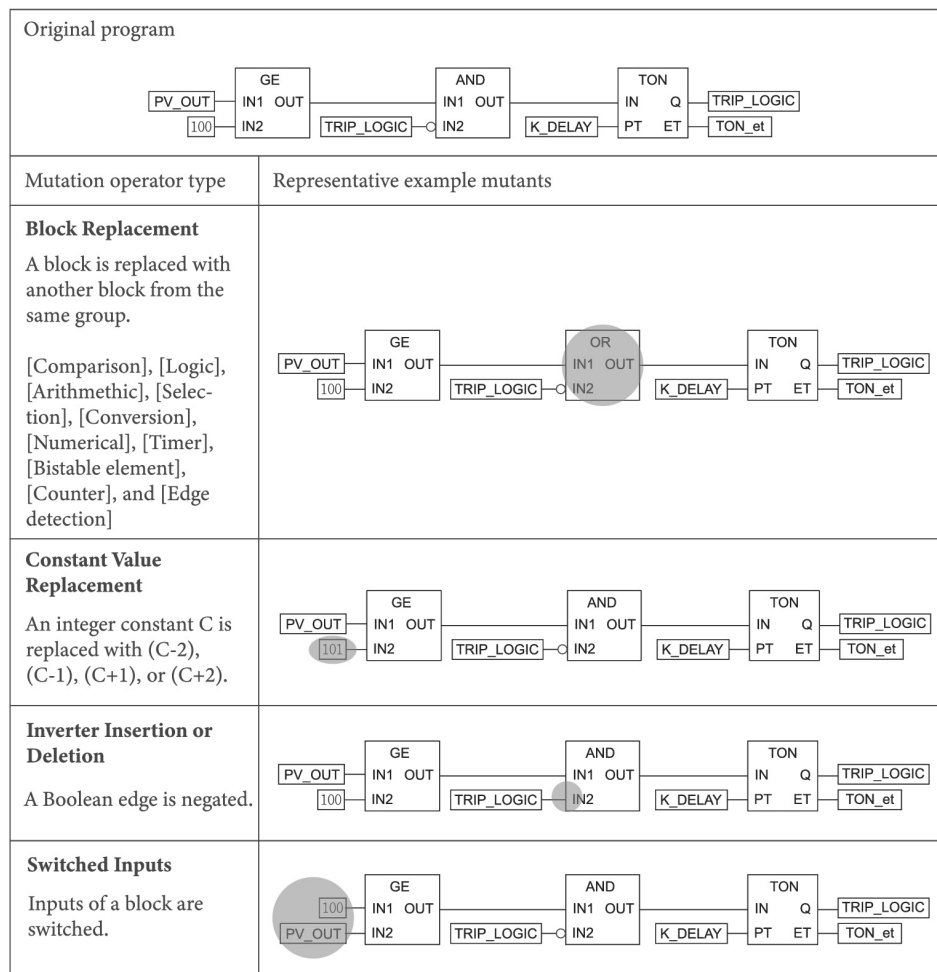


Figure A.1.1: Examples of FBD code and mutation operators [11]

A.2 Example ST module

A.2.1 Original code

```
1  VAR_INPUT
2    DemandAvailability      : BOOL;
3    AvailabilityChanged     : BOOL;
4    // Signals from Local
5    ComponentSetup_Done    : BOOL;
6    // Signals from application
7    CommandIn              : BYTE;
8    Unit_is_Type1          : BOOL;
9    Sig1_from_Type2        : BOOL;
10   Sig2_from_Type2        : BOOL;
11   Orientation_Valid       : BOOL;
12 END_VAR
13 VAR_OUTPUT
14   State_Type1             : INT;
15   // 0 - zero state,
16   //10 - setup state,
17   //15 - lock state,
18   //20 - basic state,
19   //30 - avilability check
20   Type1_SetupDone         : BOOL;
21 END_VAR
22 VAR
23   Setup_State_Type1_T_ON   : TOnDelay;
24   LockCond                 : BOOL;
25   Communication_Test_Command : BOOL;
26   Communication_Test_Command_Old : BOOL;
27   State_Type1_Old          : INT;
28   // 0 - zero state,
29   //10 - setup state,
30   //15 - lock state,
31   //20 - basic state,
32   //30 - avilability check
33 END_VAR
34 VAR CONSTANT
35   CMD_BASIC := BYTE#251;
36   CMD_FINISHED := BYTE#252;
37   CMD_ABORTED := BYTE#254;
38   CMD_UNAVAILABLE := BYTE#255;
39 END_VAR
```

Listing A.2.1: Example module – variable declarations

```

1  //R316, RI317
2  IF (NOT Unit_is_Type1) THEN State_Type1:=eState_Type1.ZERO_STATE; END_IF
3  IF (Unit_is_Type1 AND (State_Type1=eState_Type1.ZERO_STATE)) THEN State_Type1:=eState_Type1.
   SETUP_STATE; END_IF
4
5  //R428, RI429
6  Type1_SetupDone:=((NOT(State_Type1=eState_Type1.ZERO_STATE)) AND (NOT(State_Type1=eState_Type1.
   SETUP_STATE)));
7
8  Setup_State_Type1_T_ON(In:=((State_Type1=eState_Type1.SETUP_STATE) AND ComponentSetup_Done),
   TimeOnDelay:=T_TYPE1_Unit_SETUP, InitState:=FALSE);
9
10 LockCond:=FALSE;
11 IF (Setup_State_Type1_T_ON.Out
12 OR
13 ((NOT(State_Type1=eState_Type1.ZERO_STATE)) AND (NOT(State_Type1=eState_Type1.SETUP_STATE)) AND
14 ((CommandIn=BYTE#0) OR Sig1_from_Type2 OR Sig2_from_Type2 OR (NOT Orientation_Valid)))
15 ) THEN
16 LockCond:=TRUE;
17 END_IF
18
19 IF (LockCond) THEN State_Type1:=eState_Type1.LOCK_STATE; END_IF
20
21 //R430, RI431
22 IF ((CommandIn=CMD_BASIC) AND (State_Type1=eState_Type1.LOCK_STATE) AND (NOT LockCond)) THEN
23 State_Type1:=eState_Type1.BASIC_STATE;
24 END_IF
25
26 //R003, RI004
27 Communication_Test_Command:=RangeOfCommTest(CommandIn);
28
29 IF ((State_Type1=eState_Type1.BASIC_STATE) AND (NOT Communication_Test_Command_Old) AND
   Communication_Test_Command AND (NOT LockCond)) THEN
30 State_Type1:=eState_Type1.AVAILABILITY_CHECK;
31 END_IF
32 IF ((State_Type1_Old=eState_Type1.AVAILABILITY_CHECK) AND (State_Type1=eState_Type1.
   AVAILABILITY_CHECK) AND (NOT LockCond) AND ((NOT AvailabilityChanged) AND (NOT
   DemandAvailability)) OR (CommandIn=CMD_BASIC) OR (CommandIn=CMD_ABORTED))) THEN
33 State_Type1:=eState_Type1.BASIC_STATE;
34 END_IF
35 Communication_Test_Command_Old:=Communication_Test_Command;
36 State_Type1_Old:=State_Type1;

```

Listing A.2.2: Example module – implementation

A.2.2 Mutants

```
11 IF (Setup_State_Type1_T_ON.Out
12 OR
13 ((NOT(State_Type1=eState_Type1.ZERO_STATE)) AND (NOT(State_Type1=eState_Type1.SETUP_STATE)) AND
14 ((CommandIn=BYTE#0) XOR Sig1_from_Type2 OR Sig2_from_Type2 OR (NOT Orientation_Valid))) //MUTATED
15 ) THEN
16 LockCond:=TRUE;
17 END_IF
```

Listing A.2.3: mutant #10

```
11 IF (Setup_State_Type1_T_ON.Out
12 OR
13 ((NOT(State_Type1=eState_Type1.ZERO_STATE)) AND (NOT(State_Type1=eState_Type1.SETUP_STATE)) AND
14 ((CommandIn=BYTE#0) OR Sig1_from_Type2 XOR Sig2_from_Type2 OR (NOT Orientation_Valid))) //MUTATED
15 ) THEN
16 LockCond:=TRUE;
17 END_IF
```

Listing A.2.4: mutant #12

```
11 IF (Setup_State_Type1_T_ON.Out
12 OR
13 ((NOT(State_Type1=eState_Type1.ZERO_STATE)) AND (NOT(State_Type1=eState_Type1.SETUP_STATE)) AND
14 ((CommandIn=BYTE#0) OR Sig1_from_Type2 OR Sig2_from_Type2 XOR (NOT Orientation_Valid))) //MUTATED
15 ) THEN
16 LockCond:=TRUE;
17 END_IF
```

Listing A.2.5: mutant #14

```
32 IF ((State_Type1_Old=eState_Type1.AVAILABILITY_CHECK) AND (State_Type1=eState_Type1.
    AVAILABILITY_CHECK) AND (NOT LockCond) AND (((NOT AvailabilityChanged) AND (NOT
    DemandAvailability)) XOR (CommandIn=CMD_BASIC) OR (CommandIn=CMD_ABORTED))) THEN //MUTATED
33 State_Type1:=eState_Type1.BASIC_STATE;
34 END_IF
35 Communication_Test_Command_Old:=Communication_Test_Command;
36 State_Type1_Old:=State_Type1;
```

Listing A.2.6: mutant #34

```
32 IF ((State_Type1_Old=eState_Type1.AVAILABILITY_CHECK) AND (State_Type1=eState_Type1.
    AVAILABILITY_CHECK) AND (NOT LockCond) AND (((NOT AvailabilityChanged) AND (NOT
    DemandAvailability)) OR (CommandIn=CMD_BASIC) XOR (CommandIn=CMD_ABORTED))) THEN //MUTATED
33 State_Type1:=eState_Type1.BASIC_STATE;
34 END_IF
35 Communication_Test_Command_Old:=Communication_Test_Command;
36 State_Type1_Old:=State_Type1;
```

Listing A.2.7: mutant #36

```
21 //R430,RI431
22 IF ((CommandIn=CMD_BASIC) AND (State_Type1=eState_Type1.LOCK_STATE) OR (NOT LockCond)) THEN //MUTATED
23 State_Type1:=eState_Type1.BASIC_STATE;
24 END_IF
```

Listing A.2.8: mutant #17

A.3 Example test report

line#code	Score	ABS	ADR	LCR	ROR	JOI	ALL
1_1 //SR316,\$RI317							
1_2 IF (NOT Unit_is_Type1) THEN State_Type1:=eState_Type1.ZERO_STATE; END_IF	100.0%	[0,0]	[0,0]	[0,0]	[1,0]	[1,0]	
1_3 IF (Unit_is_Type1 AND (State_Type1=eState_Type1.ZERO_STATE)) THEN State_Type1:=eState_Type1.SETUP_STATE; END_IF	100.0%	[0,0]	[0,0]	[2,0]	[1,0]	[1,4]	
1_4							
1_5 //SR428,\$RI429							
1_6 Type1_SetupDone:=(NOT(State_Type1=eState_Type1.ZERO_STATE)) AND (NOT(State_Type1=eState_Type1.SETUP_STATE));	100.0%	[0,0]	[0,0]	[2,0]	[2,0]	[0,4]	
1_7							
1_8 Setup_State_Type1_T_ON(In:=((State_Type1=eState_Type1.SETUP_STATE) AND ComponentSetup_Done),TimeOnDelay:=T_BT_Type1_Unit_SETUP,InitState:=FALSE);	100.0%	[0,0]	[0,0]	[2,0]	[1,0]	[0,3]	
1_9							
1_10 LockCond:=FALSE;							
1_11 IF (Setup_State_Type1_T_ON.Out							
1_12 OR							
1_13 ((NOT(State_Type1=eState_Type1.ZERO_STATE)) AND (NOT(State_Type1=eState_Type1.SETUP_STATE)) AND	100.0%	[0,0]	[0,0]	[2,0]	[2,0]	[0,4]	
1_14 ((CommandIn=BYTE#0) OR Sig1_from_Type2 OR Sig2_from_Type2 OR (NOT Orientation_Valid)))	87.1%	[0,0]	[0,0]	[3,0]	[1,0]	[0,4]	
1_15) THEN							
1_16 LockCond:=TRUE;							
1_17 END_IF							
1_18							
1_19 IF (LockCond) THEN State_Type1:=eState_Type1.LOCK_STATE; END_IF	100.0%	[0,0]	[0,0]	[0,0]	[0,1]	[1,1]	
1_20							
1_21 //SR430,\$RI431							
1_22 IF ((CommandIn=CMD_BASIC) AND (State_Type1=eState_Type1.LOCK_STATE) AND (NOT LockCond)) THEN	100.0%	[0,0]	[0,0]	[4,0]	[2,0]	[1,7]	
1_23 State_Type1:=eState_Type1.BASIC_STATE;							
1_24 END_IF							
1_25							
1_26 //SR003,\$RI004							
1_27 Communication_Test_Command:=RangeOfCommTest(CommandIn);							
1_28							
1_29 IF ((State_Type1=eState_Type1.BASIC_STATE) AND (NOT Communication_Test_Command_Old) AND Communication_Test_Command	100.0%	[0,0]	[0,0]	[6,0]	[1,0]	[1,8]	
1_30 AND (NOT LockCond)) THEN							
1_31 END_IF							
1_32 IF ((State_Type1_Old=eState_Type1.AVAILABILITY_CHECK) AND (State_Type1=eState_Type1.AVAILABILITY_CHECK) AND (NOT	87.5%	[0,0]	[0,0]	[10,0]	[4,0]	[0,14]	
1_33 LockCond) AND (((NOT AvailabilityChanged) AND (NOT DemandAvailability)) OR (CommandIn=CMD_BASIC) OR							
1_34 (CommandIn=CMD_ABORTED))) THEN							
1_35 State_Type1:=eState_Type1.BASIC_STATE;							
1_36 END_IF							
1_37 Communication_Test_Command_Old:=Communication_Test_Command;							
1_38 State_Type1_Old:=State_Type1;							
1_39 EOF							

Figure A.3.1: Mutation test report for the example module

Line	Operator	Mutant	Result
14	XOR		LIVE
14	AND		KILLED
14	XOR		LIVE
14	AND		KILLED
14	XOR		LIVE
14	<>		KILLED
14	AND		KILLED

Figure A.3.2: Mutation test report for line 14 of the example module

A.4 Examples for test generation

A.4.1 Simple example

Verification case

Metadata

Source files
Here the scope of the verification (i.e., the included source files) needs to be selected.

Source files:

☐ abs_2.scl
☐ abs_o.scl
☐ test.scl
☒ *.scl (all scl files in this project's root)

Reload source files

Language fronte

STEP 7

Entry block:

aggr_1

Verification backend
Selection and configuration of the external verification tool to be used.

Backend:

NuSMV

Algorithm:

(default)

Advanced settings

Requirement
Description of the requirement to be verified.

Requirement type

Pattern requirement

Pattern:

{1} is always true at the end of the PLC cycle.

?

1:

aggr_1.RET_VAL

aggr_1.RET_VAL is always true at the end of the PLC cycle.

Requirement – advanced

Reporters

Misc. Tools

Advanced settings (0)

Verify
Everything is ready? Buckle up and hit the 'Verify' button!

Verify!

Last result:

Satisfied

Figure A.4.1: Verification case for mutant 1 of the absolute value function

Verification case

Metadata

Source files
Here the scope of the verification (i.e., the included source files) needs to be selected.
Source files:

☐ abs_1.scl
☐ abs_2.scl
☐ abs_o.scl
☐ test.scl
☒ *.scl (all scl files in this project's root)

Reload source files

Language fronte: STEP 7
Entry block: aggr_2

Verification backend
Selection and configuration of the external verification tool to be used.
Backend: NuSMV
Algorithm: (default)
Advanced settings

Requirement
Description of the requirement to be verified.
Requirement type: Pattern requirement
Pattern: {1} is always true at the end of the PLC cycle.
1: aggr_2.RET_VAL
aggr_2.RET_VAL is always true at the end of the PLC cycle.

Requirement – advanced

Reporters

Misc. Tools

Advanced settings (0)

Verify
Everything is ready? Buckle up and hit the 'Verify' button!

Verify!

Last result: **Violated**

Figure A.4.2: Verification case for mutant 2 of the absolute value function

A.4.2 Industrial example

```
1 FUNCTION_BLOCK aggr_10
2   VAR_INPUT
3     DemandAvailability : BOOL;
4     AvailabilityChanged : BOOL;
5     ComponentSetup_Done : BOOL;
6     CommandIn : BYTE;
7     Unit_is_Type1 : BOOL;
8     Sig1_from_Type2 : BOOL;
9     Sig2_from_Type2 : BOOL;
10    Orientation_Valid : BOOL;
11  END_VAR
12  VAR_OUTPUT
13    outputs_match : BOOL;
14  END_VAR
15  VAR
16    fb_o : TYPE1_State; //original function block
17    fb_10 : TYPE1_State_10; //mutant function block
18    State_Type1_o : INT;
19    State_Type1_10 : INT;
20    Type1_SetupDone_o : BOOL;
21    Type1_SetupDone_10 : BOOL;
22    Communication_Test_Command_o : BOOL;
23    Communication_Test_Command_10 : BOOL;
24  END_VAR
25  BEGIN
26    fb_o(
27      Orientation_Valid := Orientation_Valid,
28      ComponentSetup_Done := ComponentSetup_Done,
29      AvailabilityChanged := AvailabilityChanged,
30      DemandAvailability := DemandAvailability,
31      CommandIn := CommandIn,
32      Sig2_from_Type2 := Sig2_from_Type2,
33      Sig1_from_Type2 := Sig1_from_Type2,
34      Unit_is_Type1 := Unit_is_Type1
35    );
36    Type1_SetupDone_o := fb_o.Type1_SetupDone;
37    State_Type1_o := fb_o.State_Type1;
38    Communication_Test_Command_o := fb_o.Communication_Test_Command;
39
40    fb_10(
41      Orientation_Valid := Orientation_Valid,
42      ComponentSetup_Done := ComponentSetup_Done,
43      AvailabilityChanged := AvailabilityChanged,
44      DemandAvailability := DemandAvailability,
45      CommandIn := CommandIn,
46      Sig2_from_Type2 := Sig2_from_Type2,
47      Sig1_from_Type2 := Sig1_from_Type2,
48      Unit_is_Type1 := Unit_is_Type1
49    );
50
51    Type1_SetupDone_10 := fb_10.Type1_SetupDone;
52    State_Type1_10 := fb_10.State_Type1;
53    Communication_Test_Command_10 := fb_10.Communication_Test_Command;
54
55    outputs_match := (Type1_SetupDone_o = Type1_SetupDone_10) AND (State_Type1_o = State_Type1_10) AND
56      (Communication_Test_Command_o = Communication_Test_Command_10);
57  END_FUNCTION_BLOCK
```

Listing A.4.1: Example of an aggregating function block

Verification case

Metadata

Source files
Here the scope of the verification (i.e., the included source files) needs to be selected.

Source files:

☐ example_17.scl
☐ example_34.scl
☐ example_36.scl
☐ test.scl
☒ *.scl (all scl files in this project's root)

Reload source files

Language frontend: STEP 7

Entry block: aggr_17

Verification backend
Selection and configuration of the external verification tool to be used.

Backend: NuSMV

Algorithm: (default)

Advanced settings

Requirement
Description of the requirement to be verified.

Requirement type: Pattern requirement

Pattern: {1} is always true at the end of the PLC cycle.

1: instance.outputs_match

instance.outputs_match is always true at the end of the PLC cycle.

Requirement – advanced

Reporters

Misc. Tools

Advanced settings

Verify
Everything is ready? Buckle up and hit the 'Verify' button!

Verify!

Last result: Violated

Figure A.4.3: Verification case for the already killed mutant of the example module

Counterexample

	Variable	Beginning of Cycle 1	End of Cycle 1
INPUT BOOL	instance.AvailabilityChanged	false	false
INPUT BYTE	instance.CommandIn	255	255
INPUT BOOL	instance.DemandAvailability	false	false
INPUT BOOL	instance.Orientation_Valid	false	false
INPUT BOOL	instance.Sig1_from_Type2	false	false
INPUT BOOL	instance.Sig2_from_Type2	false	false
INPUT BOOL	instance.Unit_is_Type1	true	true
LOCAL BOOL	instance.fb_17.Communication_Test_Command_Old	false	false
LOCAL BOOL	instance.fb_17.LockCond	false	false
OUTPUT BOOL	instance.fb_17.Setup_State_Type1_T_ON.Out	false	false
OUTPUT INT	instance.fb_17.State_Type1	0	20
LOCAL INT	instance.fb_17.State_Type1_Old	0	20
OUTPUT BOOL	instance.fb_17.Type1_SetupDone	false	false
INPUT BOOL	instance.fb_17.Unit_is_Type1	false	true
INPUT BYTE	instance.fb_o.CommandIn	0	255
LOCAL BOOL	instance.fb_o.Communication_Test_Command_Old	false	false
LOCAL BOOL	instance.fb_o.LockCond	false	false
OUTPUT BOOL	instance.fb_o.Setup_State_Type1_T_ON.Out	false	false
OUTPUT INT	instance.fb_o.State_Type1	0	10
LOCAL INT	instance.fb_o.State_Type1_Old	0	10
OUTPUT BOOL	instance.fb_o.Type1_SetupDone	false	false
INPUT BOOL	instance.fb_o.Unit_is_Type1	false	true
OUTPUT BOOL	instance.outputs_match	false	false

Figure A.4.4: NuSMV counterexample for the already killed mutant

Counterexample

	Variable	Beginning of Cycle 1	End of Cycle 1
INPUT BYTE	instance.CommandIn	248	248
INPUT BOOL	instance.Unit_is_Type1	false	false
LOCAL BOOL	instance.fb_17.Communication_Test_Command_Old	false	false
LOCAL BOOL	instance.fb_17.LockCond	false	false
OUTPUT BOOL	instance.fb_17.Setup_State_Type1_T_ON.Out	false	false
OUTPUT INT	instance.fb_17.State_Type1	0	20
LOCAL INT	instance.fb_17.State_Type1_Old	0	20
OUTPUT BOOL	instance.fb_17.Type1_SetupDone	false	false
INPUT BOOL	instance.fb_17.Unit_is_Type1	false	false
INPUT BYTE	instance.fb_o.CommandIn	0	248
LOCAL BOOL	instance.fb_o.Communication_Test_Command_Old	false	false
LOCAL BOOL	instance.fb_o.LockCond	false	false
OUTPUT BOOL	instance.fb_o.Setup_State_Type1_T_ON.Out	false	false
OUTPUT INT	instance.fb_o.State_Type1	0	0
LOCAL INT	instance.fb_o.State_Type1_Old	0	0
OUTPUT BOOL	instance.fb_o.Type1_SetupDone	false	false
INPUT BOOL	instance.fb_o.Unit_is_Type1	false	false
OUTPUT BOOL	instance.outputs_match	false	false

Figure A.4.5: Theta counterexample for the already killed mutant