Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Bachelor's Degree in Computer Science, 15 credits

AN INTEGRATED TOOL CHAIN FOR COMBINATORIAL TESTING OF INDUSTRIAL CONTROL SOFTWARE

Liene Andersone
lae18001@student.mdh.se

Anne Christine Carlsson
acn18003@student.mdh.se

Examiner: Wasif Afzal
Mälardalen University, Västerås, Sweden

Supervisor: Eduard Paul Enoiu
Mälardalen University, Västerås, Sweden

June 8, 2021

Abstract

Testing is an important activity in software development used to ensure the quality of a product. In industrial practice when developing control software, such as Programmable Logic Controllers (PLCs), software testing is, in some cases, neglected during the development process. There is little support for automated testing on PLC software and the creation of test cases is hence mainly a manual activity that can be both costly and time-consuming.

In this thesis, we propose a solution for an integrated tool chain for automating the testing process for IEC 61131-3 PLC software in the integrated development environment (IDE) of CODESYS (Controller Development System), using a combinatorial testing tool (i.e., SEAFOX) for test case generation. Further, we measure the applicability and usefulness of the integrated tool chain in terms of the decision coverage (i.e, Observable Decision Coverage) achieved by the generated test cases. For this purpose, several available tools for testing in CODESYS IDE have been examined as well as solutions on how to integrate the chosen tool (i.e., CfUnit) with SEAFOX. After a solution for the tool chain was implemented and integrated, an experiment was conducted to measure decision coverage for the generated test cases on nine industrial programs using Pairwise, Base choice and Random combinatorial testing techniques.

The result of our thesis is a fully integrated tool chain consisting of CODESYS, CfUnit, SEAFOX, and a python script used to combine these, and where the SEAFOX tool was further extended to support additional standard data types. This tool chain can be used to create test cases, generate new input values for these using SEAFOX, automate the additional test case creation and integration with CfUnit in CODESYS IDE, as well as for test execution. The results of the experiment show that test cases generated by this tool chain achieved on average 90% decision coverage or higher regardless of the combinatorial technique used, with the overall average level being 94%. Interestingly enough, Random testing scored higher than both Pairwise and Base choice testing.

We present our integrated tool chain as a contribution to the automation of test creation and execution for industrial control software in CODESYS IDE. However, we identified several limitations with testing CODESYS timer blocks and when executing larger test suites that can make it problematic to fully test PLC programs in a real-time simulation environment. These challenges should be investigated further by both researchers and practitioners.

Table of Contents

List of Figures

List of Tables

1. Introduction

Safety-critical machines like trains and cranes are built and include complex control and monitoring systems used to guarantee precise movements and high safety while operating them. In the same way, large industrial manufacturer facilities and nuclear power plants rely on safety-critical control systems that are usually built from countless programmable logic controller (PLC) devices connected with each other. Each of these PLCs is responsible for its own task in the system by holding a software program with the relevant instructions in its memory [1, p. 252]. In many cases, software for safety-critical PLC systems is written using IEC 61131-3 compliant languages. One of the defined graphical languages widely used for developing industrial controllers is called Function Block Diagram (FBD) [1, p. 134]. An FBD program operates based on input signals that regulate and activate different parts of the control system. Testing such powerful systems is a crucial part of the industrial software development process and special attention needs to be given to what approaches can be used for designing efficient and effective test cases.

There are different testing techniques like structural testing, model-based testing, and combinatorial testing etc., that have been proposed for automatically creating test cases during the software engineering process [2]. Even if combinatorial testing techniques have been used since 1985, for PLC software these have received more attention only recently [3], [4]. This technique is used to automatically generate combinations of input values to effectively find faults in a program under test [2].

There are several combinatorial testing tools on the market (e.g. ACTS [3], PICT [5], etc.) that implement a combinatorial testing approach to generate relevant test cases. Another combinatorial testing tool called SEAFOX has been developed by students at Mälardalen University a few years ago and has already been used in a research study [4]. Compared to other tools, SEAFOX has been developed specifically for PLC software, but like most combinatorial testing tools, lacks the integration with domain specific development environments and test execution frameworks. The SEAFOX tool, in its current form, can only load programs in the PLCopen XML file format (i.e., specifically structured xml file that can be exchanged between PLC software development tools) to generate test case input values.

The first part of this thesis is based on tailoring the combinatorial testing tool SEAFOX with a control software development tool named CODESYS, that follows the IEC 61131-3 standard, to make the automated testing process of PLC programs more applicable and generalizable when compared with the current state of the practice in these kinds of development environments. CODESYS IDE has two versions that currently are used in the field. To create the tool chain we explored both versions. For the SEAFOX tool, to be able to integrate with CODESYS v2.3, we extended SEAFOX functionality by implementing a parser for .exp file (i.e., a structured text file that is exported from CODESYS v2.3, containing the PLC program units). Another extension for the SEAFOX tool was the implementation of additional standard data types (e.g. WORD and TIME) that are frequently used in PLC programs.

Early discovered limitations for test execution in CODESYS v2.3, led us to change our focus to version 3.5 of CODESYS as our targeted IDE in the tool chain. CODESYS v3.5 is able to export a PLCopen XML file, so that SEAFOX can load and process it accordingly. To integrate the tool chain further, we selected a test execution tool (i.e., CfUnit) compatible with CODESYS v3.5. Lastly, we developed an additional python script (i.e., TEST CONCRETIZER) that would act like a bridge between the tools enabling the merge of data between them for test case generation and concretization.

The second part of the thesis is devoted to the conduct of an experiment where the integrated tool chain is evaluated in terms of applicability and usefulness by measuring the observable decision coverage achieved by the generated test cases on several industrial programs. The combinatorial testing tool SEAFOX implements three test case input generation methods (i.e., Base choice, Pairwise and Random) that we applied on nine function blocks selected from real industrial FBD programs provided to us by an automation company. The results show that the generated test cases achieved 94% decision coverage on average. The results spread from 90.74% achieved with Pairwise and Base choice methods, up to 98.15% achieved with a random test suite of 90 test cases. Judgments regarding e.g. input ranges in the experiment and limitations in the testing tool can influence the results and can explain some of the differences in the decision coverage achieved for

test cases between these different methods.

Based on the increasing demand for more efficient automated testing during the control software development process, it is important to decide which methods would improve the efficiency of PLC software testing. The integrated tool chain as well as the overall results of this thesis can help PLC software program developers and testers to choose the most efficient and effective combinatorial testing method that meets their needs in their specific control software engineering process.

The thesis report is organised in following sections: in Section 2 we introduce the background information needed to understand PLC programming basics and software testing techniques used in the thesis. Section 3 presents previous studies and related work done in the field of PLC software testing. Section 4 describes the main problem and Section 5 - the way how to solve the presented problem. In Section 6 we present the process of the tool chain creation. In Section 7 we describe the conduct of the experiment in more detail and Section 8 follows with the description of the experiment results. Section 9 discusses the overall implications of using the created tool chain, its evaluation in terms of decision coverage levels as well as the threats to validity of our research work. Finally, Section 10 summarises the accomplished results and draws conclusions for the overall research process. Suggestions for future work are outlined in Section 11.

2. Background

This section gives an overview of the relevant knowledge needed to understand PLC programming and software testing techniques used in this thesis. Here we are explaining a standard that industries developing control systems follow, as well as specifics of the software development tool and combinatorial testing tool, that will be used during the research process in our thesis.

2.1. Programmable Logic Controller (PLC)

A PLC is a microprocessor-based device that can store, and process instructions based on input signals by using programmable memory [6, pp. 3-4]. By combining several PLC units, a complex system can be built to control machines and industrial processes. Control programs for PLC systems are pre-programmed using boolean logic and switching operations. PLCs are commonly used to automate and control complex processes in production [6, p. 9], as well as in safety-critical systems, such as nuclear power plants, trains, cranes, etc. Engineers that operate in an industrial environment can enter a program into the PLC memory where the controller processes the inputs by applying the instructions from program to generate the outputs. In this way, the PLCs can control the tasks of any industrial machine that are connected to them as an output device.

2.2. The IEC 61131 Standard and Function Block Diagram (FBD)

The International Electrotechnical Commission (IEC) 61131 standard [1, p. 12] was created as a basis for uniform PLC programming by a working group of the international standardisation organisation IEC[1] that represented different PLC manufacturers, software developers and users of both kinds of PLCs. Using the concepts of the modern software technologies the IEC established this standard to ensure that the common guidelines were followed amongst the PLC hardware manufacturers and software developers for PLCs. The first publications of the assembled documentation of the standard appeared in 1993. Since then, it has been updated continuously and now consists of a total of ten parts, where part three (i.e., IEC 61131-3) is dedicated to PLC programming languages. There are five standard defined PLC programming languages [1, p. 24], that can be used to develop various PLC programs:

- Instruction List (IL)
- Structured Text (ST)
- Ladder Diagram (LD)
- Function Block Diagram (FBD)
- Sequential Function Chart (SFC)

The first two are textual languages and the remaining three are graphical programming languages [7, p. 200]. These languages are used depending on the preferences of the PLC software developers writing the actual PLC programs.

The Function Block Diagram, one of the graphical languages, has been used the most in industries developing control systems [1, p. 134]. Graphical objects are used to represent the logic and functionality of the FBD program. A simple example of the FBD flow is illustrated in Figure 1. Graphical objects consist of three units in their graphical representation: *connections, graphical elements*, and *connectors*. Data flows into *graphical elements* as input variables where data is processed according to instructions and the result is stored in output variables, that can be sent further to the next function or function block (FB) as input. FBs are represented as boxes with specified names and with input and output variables. Boxes are linked with horizontal or vertical lines – *connections*. *Connectors*, on the other hand, are named lines that represent a *new line* to divide a larger flow and make programs more readable [1, pp. 135-139].

[1]Link to IEC organisations web site: https://www.iec.ch

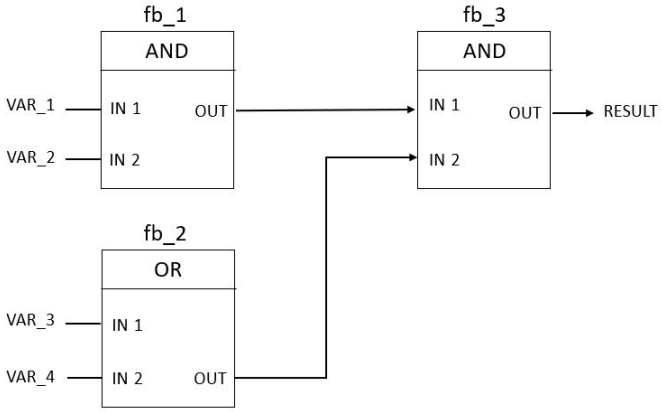

Figure 1: An example of an FBD with three connected blocks.

All the programs written in standard defined languages can be exchanged amongst the software tools that follows the IEC 61131-3 standard. In that way, they can be reused or adapted for further development. The file format of the exported programs can differ from tool to tool but the most used file format in the field today is PLCopen XML exchangeable format [8], that recently became known also as a part ten (i.e., IEC 61131-10) of the standard. The structure of the file tree with one FB in it is shown in Figure 2.

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <project xmlns="http://www.plcopen.org/xml/tc6_0200">
3     <fileHeader companyName="" productName="CODESYS" productVersion="CODESYS V3.5 SP16 Patch 4"
4     <contentHeader name="" modificationDateTime="2021-05-03T17:29:57.4248795">
28    <types>
29      <dataTypes />
30      <pous>
31        <pou name="FB_5"pouType="functionBlock">
32          <interface>
33            <inputVars>
34              <variable name="IN_var1">
35                <type>
36                  <BOOL />
37                </type>
38              </variable>
39            </inputVars>
40            <outputVars>
41              <variable name="OUT_var1">
42                <type>
43                  <BOOL />
44                </type>
45              </variable>
46            </outputVars>
47          </interface>
48          <body>
49            <FBD>
257         </body>
258         <addData>
263       </pou>
264     </pous>
265   </types>
266   <instances>
269   <addData>
270     <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
279   </addData>
280 </project>
```

Figure 2: A snippet of PLCopen XML file structure overview. A file containing one FB with one input and one output variable. Content Header part (lines 4 to 28) and FB body part with its structure (from line 49 to 257) are hidden.

2.3. CODESYS

CODESYS stands for controller development system [7, p. 486] and is a software development tool for programming different PLC systems. It is developed by Smart Software Solutions GmbH based in Germany. A several hundreds of PLC hardware manufacturers worldwide prefer to use CODESYS development tool as Integrated Development Environment (IDE) for their device set-ups. All five of the previously mentioned PLC programming languages in Section 2.2 are possible to use in CODESYS IDE as it follows the IEC 61131-3 standard almost completely.

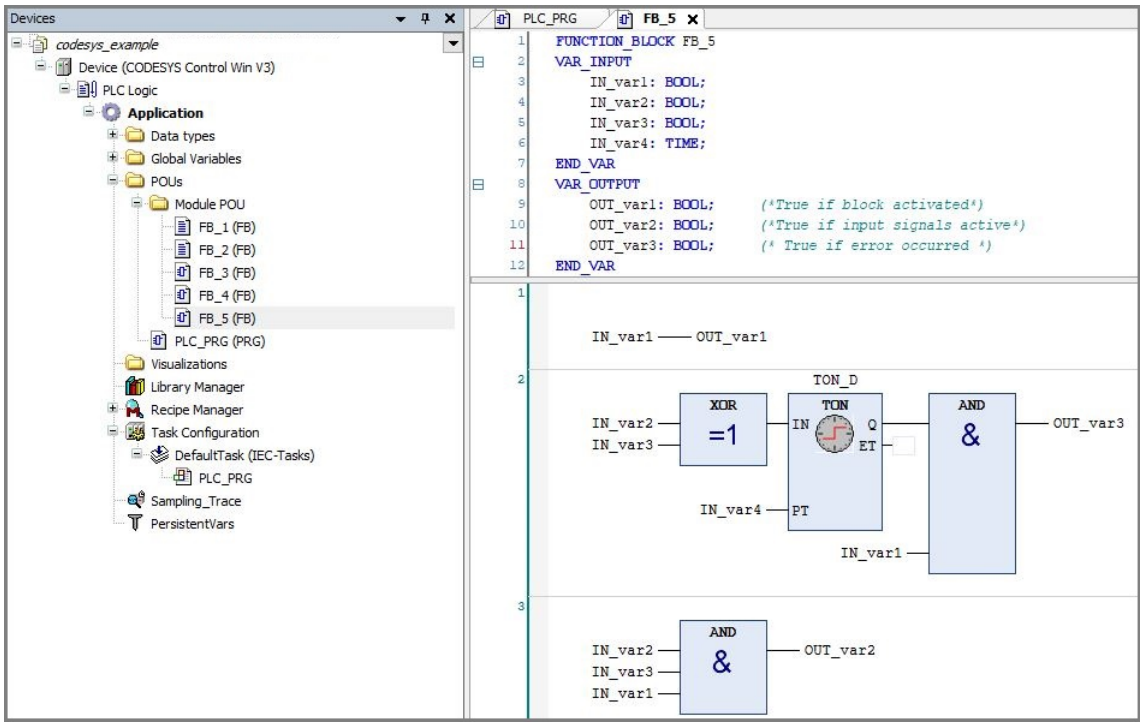

Figure 3: A screenshot of CODESYS IDE v3.5. The functionality and logic of the Project Application is built with several different POUs located in the POU folder.

Using CODESYS development tool, it is possible to execute the code by simulating various PLC hardware set-ups without the need of a real hardware device involved, hence it is a hardware-independent programming environment and contains the simulator. Today there are two CODESYS IDE versions accessible – version 2.3, and the latest version 3.5. Both versions are available to download from CODESYS official store[2] and are possible to use without a fee.

The projects in CODESYS IDE are built using program organization units (POUs) [7, p. 489] and, as the name states, it is a way to organize your program by choosing the appropriate code structure and PLC programming language. A small project in CODESYS programming environment, as shown in Figure 3, is located in one POU folder where the main program (i.e., PLC_PRG) and a folder holding five separate function blocks is located in it. An example of the whole project structure in CODESYS on the right and a function block FB_5 written in FBD language on the left side is demonstrated in Figure 3.

POU can be represented as a program, a single function block, or a simple function. A POU as a program can hold several other POUs (e.g. function blocks) itself. It is even possible that one program can contain several FBs that are written in two different languages, for example one in FBD and the other one in ST language. Every standard defined language can be combined and used in a single PLC program. Later, the project can be exported as a whole program or as separate POUs, depending on the needs for a further software development process. CODESYS IDE v2.3 allows you to export and import projects as exp-files. Latest version of CODESYS IDE supports PLCopen XML file format as import and export possibilities. This file then contains a

[2]Link to CODESYS store: https://store.codesys.com/all-products.html

key information about the exported program or FB, such as, specified FB names, input and output variable names as well as their data types.

2.4. Combinatorial Testing

Software testing is an important step in software development to show that a system meets its requirements, and to discover defects in the system [9, Ch. 8]. During the testing activity, test cases are designed to reflect the systems expected use, and to expose defects. A test case is a set of input values for the item under test, conditions for executing the item, and the expected result of the test [10]. Several test cases are further bundled to create a test suite. A difficulty with testing is that the number of possible inputs and combinations between these can be extremely large and it is not practical to test them all [11, p. xiii]. However, it has been shown that many failures are caused by interactions of only a few parameters. Combinatorial testing focus on finding the combinations that produce those failures using different combinatorial techniques, t-way (e.g. pairwise) and random being some [11, Ch. 1]. In t-way testing, t levels of interactions (also referred as the strength) are covered by a test. For example, pairwise testing is a form of t-way testing with two interactions. With pairwise (2-way) testing, all possible pair of parameter values are covered in at least one of the test cases in a test suite. An example is shown in Table 1, where we can see that all combinations of input values are represented between P1 and P2, P1 and P3 as well as between P2 and P3.

Table 1: Test suite for pairwise testing of an FB with three parameters and three input values.

Test case	P1	P2	P3
TC 1	0	0	0
TC 2	0	1	1
TC 3	0	2	2
TC 4	1	0	1
TC 5	1	1	2
TC 6	1	2	0
TC 7	2	0	2
TC 8	2	1	0
TC 9	2	2	1

The power of combinatorial testing is that the size of the test suite is reduced by not testing all possible parameter combinations [12]. For example, a function block with three parameters and tested with three different input values would need 3x3x3 = 27 test cases to test all possible combinations. With pairwise testing only 9 test cases are produced as shown in Table 1. Pairwise testing is considered effective [11, Ch. 1], [4], but it has also been discussed that it is not sufficient enough and that a 3-way strength is needed to achieve a satisfactory coverage for fault detection [11, Ch. 1].

2.5. SEAFOX

SEAFOX[3] is a combinatorial testing tool that was developed a few years ago by students at Mälardalen University and later improved and extended in a bachelor thesis [13]. This tool can take a PLCopen XML file as an input, that contains information about the program to be tested. Then the tool parses the file by extracting the needed information that will be used to generate test cases, such as, the name of the input parameters and their respective data types. Another option is to manually provide the parameters and data types in the tool. Further, SEAFOX tool creates test cases by generating the inputs using one of the available algorithms implemented in the tool: Random, Base choice or Pairwise algorithm. These input values can then be exported as comma-separated values in a csv-file where each line is a new set of input parameters representing a certain test case.

[3] https://github.com/CharByte/SEAFOX

2.6. Logic Coverage

Logic coverage is a coverage criterion used when testing safety critical software and PLCs [14]. It is normally applied during unit testing to check that various aspects of the code structure have been invoked during the test. By measuring if different program artifacts, such as decisions and conditions, were exercised during the test, a certain level of logic coverage is achieved.

One form of logic coverage is decision coverage. This criterion measures how many decisions, i.e., outcomes of boolean expressions, that are exercised in the system under test [15]. For a test case to achieve full decision coverage, every point of entry and exit in the system under test must have been invoked at least once. Also, every decision must have taken all possible outcomes at least once, i.e., the outcome must have the value True at least once and False at least once.

For example, consider an FB that contains a boolean OR expression with the inputs A and B. The input values (A = True, B = False) and (A = False, B = False) would test both decisions, having the outcome of the FB being first True and then False. A limitation of this coverage criterion is that changing the value for B in this case is not needed to achieve full decision coverage. We thereby do not know if changing this input value would cause any defects in the system, which implies that a high decision coverage does not guarantee that the system behaves correctly in all situations. Enoiu et. al [16] provided experimental evidence suggesting that although most test suites achieved high decision coverage, these automatically generated suites were not as good at discovering faults injected in the code as manually created test cases.

3. Related Work

Testing of industrial control software such as PLCs is important to ensure that the software works as expected. Despite that, in some cases, testing is a neglected activity in the industry and there is little support for automated testing on PLC programs [17], [18]. The creation of test cases for these programs during software testing is largely a manual activity and industry mainly relies on manual testing. Research and development contributions have been made to make testing of control software easier to conduct. Jamro [18] presents an approach to POU-testing for IEC 61131-3 languages and introduced a test definition language called CPTest+. This language relies on the CPDev engineering environment and is not compatible with CODESYS. Hofer and Russo [17] focuses on CODESYS and acknowledged that there are limited resources for testing in CODESYS IDE (especially in v2.3). The authors developed a unit-testing framework, APTest, based on native IEC 61131-3 languages, that focuses on software testing. The framework is available on GitHub and can be integrated in the CODESYS platform. The development of this framework is based on a master thesis authored by one of the authors and is still a prototype and limited in its use.

An important part of the testing activity is to create effective test suites, i.e., sets of test cases. Methods such as combinatorial testing, Model-Based Testing (MBT), and Search-Based Software Testing (SBST) can be used for this purpose [2]. Combinatorial testing is widely researched, and several studies have been conducted on using and comparing different combinatorial techniques [3], [4], [5]. Closely related to our work, Charbachi et al. [4] performed a case study where test cases generated with the pairwise algorithm in the SEAFOX tool were compared with test cases manually generated by industrial engineers. Similar to our thesis, this study also used PLC industrial programs that were implemented using the IEC 61131-3 FBD language. The results show that pairwise testing achieved a branch coverage score that was equal to manual testing in 62% of the programs used and in 9%, pairwise testing scored better than manual. On average, test suites created with the pairwise technique achieved a relatively high branch coverage of 94%.

In addition, Enoiu et al. [16] did an experiment that also compares test cases manually written and automatically generated, executed on software programs developed using the IEC 61131-3 FBD language. The manual test cases were written by a group of master students and were based on two different input artifacts – specifications, and the implementation of a program written in the FBD language. An automated test generation tool called CompleteTest was used to automatically generate test cases based on the implementation. CompleteTest generates test suites that focus on satisfying a certain coverage criterion, e.g., decision coverage. The effectiveness and efficiency of the test execution was analysed by measuring fault detection, decision coverage, number of tests, and testing duration. The results show that tests generated with CompleteTest were more efficient than the manually created tests in terms of decision coverage and testing duration. This was considered as a natural result since coverage is the main focus when creating a test suite with this tool. For fault detection however, manually created tests based on a specification were more effective than both the implementation-based manual written tests and the implementation-based automatic generated tests.

Different from these studies [4], [16] we will create test cases using several different combinatorial techniques with the SEAFOX tool, and extend the studies by analysing the achieved decision coverage between the techniques. SEAFOX, compared to CompleteTest, does not base the test cases on a code coverage criterion. Therefore, using the SEAFOX tool for automatically generating test suites can give a different perspective on how well decision coverage is achieved by automated test cases when testing a program developed using the IEC 61131-3 FBD language in CODESYS.

Another combinatorial testing tool, i.e., ACTS, was evaluated by Ericsson et al. [3] for test generation on industrial programs developed in the IEC 61131-3 programming language standard. The authors generate test cases using Base Choice and t-way testing techniques with a variety of strengths from one to six. The test suites were then evaluated in terms of efficiency, i.e., the generation time, the number of tests, and applicability. The results show that ACTS is an efficient tool for generating test cases using the Base Choice technique, while there are some limitations when generating t-way test cases. During the generation of t-way test cases, some algorithms ran out of memory or were too time consuming while generating the test suite. The cut-off time for the test generation was set to one hour. Similarly to [3], we will use a combinatorial testing tool for generating test cases. Our focus on the other hand will be on measuring the coverage of the

generated test suites, which can complement the study of Ericsson et al. [3] to provide a broader spectrum on the efficiency of test case generation for industrial programs.

Ghandehari et al. [5] points out that previous research comparing t-way combinatorial testing and random testing has resulted in conflicting results with regards to both their efficiency and effectiveness. They found that some studies showed that t-way testing was more efficient, while other studies on the other hand showed no significant difference between t-way and random testing. They suggest that the lack of consensus calls for more studies within the field. They performed an experiment comparing t-way testing of strengths 2-5 with random testing. The result is measured in achieved code coverage (line and branch coverage) and fault detection. Their results suggest that t-way testing performs as good as or better than random testing. However, if focusing only on the result of 2-way testing, random testing did performed better in one of the seven test programs. A tool called PICT was selected for the test generation. This tool's random test generation function works in a similar way as SEAFOX, but it differs when generating t-way test cases in the way that it provides an option for choosing the number of t-way test cases to generate. In the experiment, they created 100 test cases with each technique. This study was conducted using the Siemens test suite and our study will add on to this with the comparison between pairwise and random testing using industrial programs written in an IEC 61131-3 language.

By studying the previously mentioned researched tools and different testing techniques we see that there still is a need for efficient testing techniques to be established in the field, and a room for improvement in terms of finding a way of automate the test case creation possibilities for PLC programs. Like the majority of studies previously written by researchers, we are also focusing on a unit-based testing approach for FBD programs or more specifically for the function blocks in those programs. Our main focus is to establish a reliable and fully integrated test automation process working with CODESYS IDE and generating inputs for test cases using combinatorial testing tool SEAFOX.

4. Problem Formulation

During the software development process, testing is one of the most important parts to ensure high quality of the final developed product. Software testing can be a time consuming and expensive activity [9, Ch. 8], so it is of interest to make it as efficient as possible. One method for effective fault detection in the program under test is a Combinatorial testing [2]. There are many Combinatorial testing tools that are used to automate test case generation (e.g. ACTS [3], PICT [5], and SEAFOX, etc.). However, some of them, for example, SEAFOX is not directly connected to any of the PLC software development IDEs (e.g. CODESYS IDE) used in the field due to lack of integration possibilities between them.

In this thesis, we use CODESYS as the development environment and SEAFOX as the combinatorial testing tool. When using SEAFOX, test cases can be automatically generated by importing a PLCopen XML file containing the unit to test. Preferably, the generated test cases could also be imported into the CODESYS IDE to run the tests in the programming environment. However, only the newer version of CODESYS (version 3.5) has the functionality of exporting and importing PLCopen XML files. The earlier version (version 2.3) can only export and import files with an exp-format. This means that CODESYS version 2.3 cannot easily be used with SEAFOX for test case generation. Further, the CODESYS IDE does not include a library or program that can execute such tests.

The main problem of this thesis is how the SEAFOX tool and CODESYS IDE can be integrated to create and run test cases so that the generalizability of automated combinatorial testing of programs, written according to 61131-3 standard, can be enhanced. To be able to accomplish that, we propose our first research question with following sub-questions:

- RQ1: How can combinatorial testing be integrated in a tool chain together with a development environment for PLCs and a test execution framework?
  - RQ1.1: How can the parser in SEAFOX tool be modified and extended to take a CODESYS v2.3 .EXP file format as input?
  - RQ1.2: What existing tools are available to execute test cases in CODESYS IDE?
  - RQ1.3: How can the generated output from SEAFOX be integrated with an existing open-source tool for test execution in CODESYS IDE?

Another aspect of the presented problem is the further evaluation of the created tool chain in terms of decision coverage on provided industrial PLC programs. To measure the performance of resulting tool chain, we are going to answer another research question:

- RQ2: What is the level of decision coverage that combinatorial testing can achieve using the integrated tool chain?

The goal of this thesis is to increase the generalizability of combinatorial testing of industrial PLC software developed according to IEC 61131-3 standard by creating a tool chain that enables the automation of test execution in CODESYS IDE. The tool chain includes combinatorial test cases to be created using SEAFOX and integrated in CODESYS IDE to run test cases using a testing tool or library. When the goal for testing method has been accomplished, the next aim is to research the level of the code coverage that this testing method can achieve in order to evaluate the resulting tool chain's usefulness.

5. Method

To solve the previously presented problems, we divide the thesis into two parts. The first part includes technical contributions in the form of an integration of a tool chain (answering RQ1), and in the second part we focus on evaluating the performance of this tool chain through measuring the decision coverage (answering RQ2). By using the method described in this section, we expect to connect the combinatorial testing tool SEAFOX with CODESYS and a chosen tool for executing test cases in CODESYS IDE, as shown in Figure 4. We then expect to run test cases using this tool chain and analyse the results.

5.1. Tool Chain Integration

This part of the thesis focus on finding a solution on how to design the tool chain, which testing program to use and how to integrate the needed parts. We approach this by studying available resources and previous research within the field to learn more about how this can be achieved.

Early in the process, we selected CODESYS as the development environment and SEAFOX as the combinatorial testing tool to use in this thesis. CODESYS IDE is the leading manufacturer-independent IEC 61131-3 programming tool available today [19]. The tool is used to develop a PLC software in several different industries as well as at educational institutions. It can be used free of charge and the programs provided to us were developed in this environment. As for SEAFOX, this tool is open-source and a copy of the source code can be downloaded and modified, if needed, to integrate it in the tool chain. It also generates test cases using several different combinatorial techniques, which was an interesting factor for the evaluation of the tool chain.

We then started by investigating the background information needed to learn about PLC programming, CODESYS IDE and how the SEAFOX testing tool works. By using the gathered information, and by studying the SEAFOX tool's implementation, we analysed how the tool could be modified to load an exp-file as input. To provide the modifications, an additional parser was implemented in SEAFOX as the answer of RQ1.1. Further, by studying previous research in the area and searching through other sources (e.g. internet) we investigated if there are any available open-source tools that could be integrated with CODESYS IDE to run the tests in programming environment directly, leading to the answer of RQ1.2.

After choosing the suitable testing tool, we examined how test cases are created and the options available for integrating them with the exported test case inputs from SEAFOX to automate the test execution in CODESYS IDE. To reach the goal of RQ1.3, the selected testing tool and CODESYS were studied, as well as their web pages and manuals. This led to the implementation of an additional script that acts like a bridge between the tools, enabling the merge of data between them.

5.2. Experiment

To evaluate the tool chain's applicability and usefulness and answering RQ2, we measure the decision coverage by executing test suites on FBs from the provided industrial programs using the above created tool chain. Since the SEAFOX tool supports three different methods to generate input values, we used all three of those methods in our research process. The data from the executed tests needed to be collected and analysed to draw conclusions about the result. We decided to perform this part of the thesis as an experiment to enable the possibility of generating the desired test cases and collect the data in a controlled environment.

5.2.1. Experimental setup

We set up the experiment by following the recommendations described in Säfsten and Gustavsson [20, ch. 4]. The experiment is used to study the decisions taken by each FB under test. As a part of the setup, we identified one dependent variable - the decision coverage - and two independent variables - the algorithm used in SEAFOX (i.e., Random, Pairwise, and Base choice), and the FB under test. Based on the reported code coverage when using the SEAFOX tool on PLC

programs in previous research by Charbachi et al. [4], and on the results reported by Ghandehari et al. [5], we have set up a hypothesis to help us answer RQ2. Charbachi et al. [4] reported 93.74% branch coverage on PLC programs when using pairwise testing with the SEAFOX tool. Ghandehari et al. [5], suggest that pairwise testing in most cases achieve a higher or equal code coverage than random testing. We measure decision coverage on test cases generated with both Pairwise and Random techniques, and also with a Base Choice technique. The previous research does not include Base Choice testing, so we will assume that it can achieve a similar code coverage as Pairwise testing when we set our hypothesis. Based on the results of these studies [4], [5] and our assumption on the achievable decision coverage for Base choice, we can assume that an average decision coverage when using these three combinatorial techniques will score a similar coverage or slightly less than the levels reported by Charbachi et al. [4] We therefore hypothesize that if the tool chain is used to create and run test cases, then a level of at least 93% average decision coverage can be achieved by these test cases.

The experimental setup consists of first selecting the FBs to test out of the provided programs (step 1 in Figure 4). Next, the input ranges for every input variable in these FBs were specified using the analysis of the program source code (step 2 in Figure 4). How this was performed is explained in detail in Sections 7.1 and 7.3. Inputs for different test suites in the experiment were generated once using Base Choice and Pairwise methods, and three times for each Random method. We chose to generate three different sets with input values and calculate an average value of the results when using Random to get more reliable test results for the method itself. We argue that three sets are sufficient enough to provide this reliability since we could see in the results that the three test suites for each Random method most often generated similar results in the number of passed and failed test cases.

The test suite sizes for Pairwise and Base choice methods are dependent on the number of inputs and the size of the input space. For Random method however, the number of test cases that are created for a test suite is chosen manually. We decided to create six different sizes of Random test suites for each FB. One random test suite of the same size as the Pairwise test suite, and one of the same size as the Base choice test suite to enable a comparison between Random and these methods. Further, we create test suites of sizes 10, 30, 60, and 90 to analyse how different test suite sizes affect the decision coverage. We chose these specific test suite sizes to have a fairly distributed wide range from small suite up to a large suite that is still smaller than the limitation of 100 test cases that we encountered with the later chosen CfUnit tool for testing in CODESYS.

5.2.2. Measuring Decision Coverage

The decision coverage measured in this thesis is based only on the Boolean outputs of the FBs. This technique can partly be compared to Observable Modified Condition/Decision Coverage (OMC/DC), which was defined by Whalen et al. in [21]. They state that an expression in a program is observable if the inputs can be modified and the changes in the output can be observed, while the rest of the program is left intact. They then combine this with Modified Condition/Decision Coverage (MC/DC), which is a different criterion for measuring coverage. We will use a variant of this technique by measuring the observable decision coverage. By this, we mean that we will modify the inputs and observe the Boolean outputs of the FB under test. The decisions taken inside the FB are not considered observable and will hence not be measured.

We take this approach to decision coverage due to limitations of measuring other parameters than outputs in the later chosen CfUnit tool for testing, and also because we write the test cases ourselves without expert advice on the internal functionality of the code. Writing test code that will measure all internal decisions in the provided programs would need a great deal of knowledge and experience with CODESYS and code coverage in software testing, and as mentioned, a different tool for executing the test cases.

5.2.3. Experimental Steps

The experiment was performed using the following steps, which were repeated for each FB under test. The overall process and the order of the steps are illustrated in Figure 4, where the objects highlighted in orange are the parts that we modified or provided as the additional implementation during our thesis. Section 7 describes the execution of the experiment in more detail.

The first step after selecting programs and specifying input ranges was to load these programs (written in FBD and ST programming languages) into CODESYS IDE and then create the necessary test suites using CfUnit for the FB that were to be tested (step 3). Each test suite is able to hold several test cases, where the input parameters for those test cases were generated through the SEAFOX tool (step 4) and the expected outputs were provided manually. In the next step, the exported csv-file from SEAFOX and the file with the test suite from CODESYS were processed within the script to create a new test suite containing test cases with all test case inputs from the csv-file as an PLCopen XML file (step 5). This file was then imported back to the CODESYS programming environment (step 6) where tests were executed with the selected testing tool CfUnit (step 7).

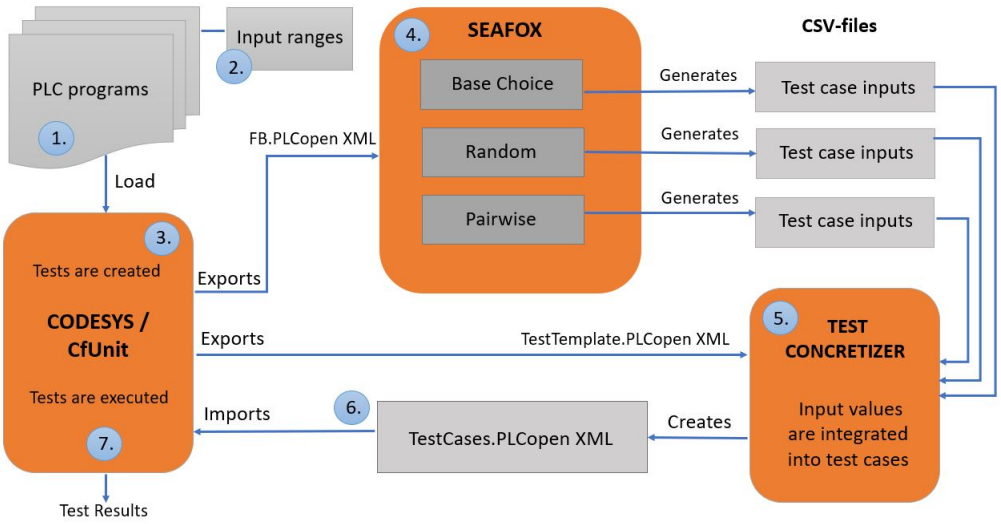

Figure 4: An overview of our integrated tool chain for combinatorial testing in CODESYS for IEC 61131-3.

5.2.4. Data collection and analysis

During the process of test execution an xml document was generated, that showed the results of how many tests were run in total and how many of those passed or failed. These results were manually inserted in an excel file for further analysis.

The collected data was analysed by comparing mean, minimum, maximum, and median values of the decision coverage for each combinatorial test case generation method. By generating a mean value for each method used, we expect to evaluate if the hypothesis was correct and also give an answer to RQ2. To further provide insight on the achieved decision coverage, we compared the mean value between both the FBs and the methods used to draw conclusions on the result with regards to any possible relations between the structure of the FB and the decision coverage achieved for different methods.

6. Integration of SEAFOX and CODESYS IDE

In this chapter, we describe in detail all the tool development and improvements that were made in order to create a working tool chain between CODESYS IDE, SEAFOX and the selected testing tool CfUnit to automate the test execution process. Figure 5 illustrates the order of tools in which data passing through specified files would generate the relevant test cases. The connection between the tools starts and ends with CODESYS IDE where the testing tool also is used.

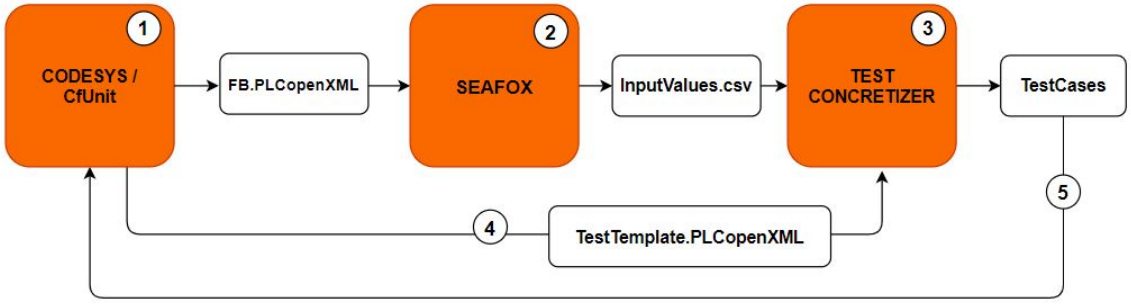

Figure 5: The expected integration between the tools in our Tool Chain.

6.1. Exploring CODESYS IDE

Based on the provided programs, some of which were written in CODESYS IDE v2.3 (V2), we started to explore the possible tool chain creation by studying this CODESYS version first. We experimented with the example programs that were included when downloading the CODESYS development platform and tried to modify and run them. To familiarise us with the way the PLC programs were written, we needed to grasp the concept of programming in FBD language, because for us it was a new way of programming. There were very limited sources available on how to debug or run graphical language programs in CODESYS V2, or how to handle CODESYS programming environment in general as a beginner.

By further examining the collaboration possibilities between CODESYS V2 and SEAFOX, we soon discovered limitations in CODESYS V2 with the tool's export and import possibilities. This version of CODESYS can only export a program as .exp file, but the SEAFOX could only load one type of file format that was PLCopen XML. The solution to this problem is described in Section 6.2.2. When further examining the options for testing tools that could be used with CODESYS V2, we found that there are limited possibilities to do that, which made us consider to investigate CODESYS v3.5 (V3) instead. More information regarding available testing tools for both versions are described in Section 6.3.

CODESYS V3 comes with online help and manuals on how to set up the simulation environment and run the code. This turned out to be very helpful when learning to use the IDE and when creating the tool chain. This version also supports importing and exporting PLCopen XML files, which we see as a great advantage since this format is a part of the IEC 61131 standard. Seeing that most of the programs provided to us were created in CODESYS V2, they needed to be adjusted so that we could use them in version 3.5 as well. The programs created in older version are not compatible to use directly in the latest version. The only way to import or open the programs written in version 2.3 is by using a CODESYS v2.3 Converter. The Converter is available to download for free as an extension for CODESYS V3[4]. This conversion possibility gave us a great opportunity to proceed working with version 3.5 while using the programs written in the 2.3 version.

[4]Link to CODESYS store: https://store.codesys.com/codesys-v23-converter.html

6.2. SEAFOX

Function blocks from the PLC programs are exported from CODESYS and imported to SEAFOX for test case generation (Step 1 and 2 in Figure 5). This section explains how to work with the SEAFOX tool and the modifications we did in the tool to enable the integration.

6.2.1. Creating test cases with SEAFOX

SEAFOX has a graphical user interface (GUI), as shown in Figure 6, where the user starts by loading a folder with the desired program or function block stored as an xml-file. Alternatively, the parameter names and data types can be added manually. A limitation in the tool when importing an xml-file is that the file only can contain one function block with input parameters. If it contains two or more function blocks with input parameters, the tool will assume that all parameters can be combined in a test case. After choosing the file to import, input variables from the file are transferred to the tool and visualised in the middle container of Figure 6. Below this container, the user choose which combinatorial algorithm to use – Random choice, Base choice, or Pairwise. Then, a range for each parameter as well as other additional information depending on the chosen algorithm needs to be set before the test cases can be generated. The ranges specify the values each parameter can take and is either a single value, a closed interval, or a combination of both. Examples of ranges and the notations used in the tool:

- 7 = a single value of only 7
- 1_3 = a closed interval (1 and 3 are included)
- 1_3;7 = a combination of both

Once all this is set, test cases can be generated and displayed in the right container. Lastly, the user can export the test cases as a csv-file by clicking the Save test button. An example of test cases generated using Base choice is depicted in Figure 6.

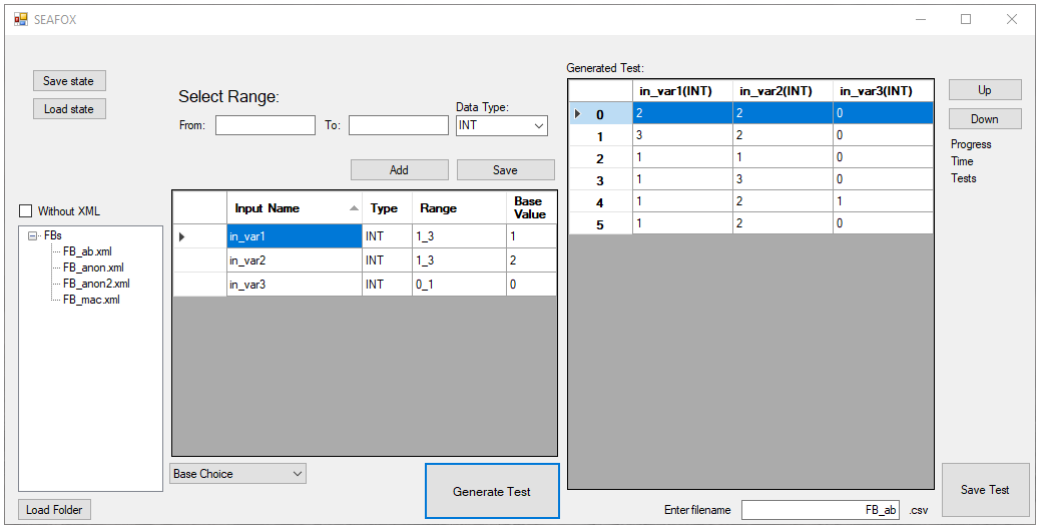

Figure 6: Base choice test suite generated in SEAFOX. The test cases have three parameters of integers of which each has a range and base choice value set as shown in the middle container in the image. The generated test cases are located in the box to the right.

The available algorithms use different techniques when generating test cases. Pairwise testing is previously described in Section 2.4. The Random choice algorithm for generating test cases is, as the name reveals, an ad hoc approach that randomly selects test cases out of a complete set [12]. SEAFOX handles this by generating a random value one by one for each parameter within the chosen range. The user chooses how many random test cases to produce.

In the Base choice algorithm, a value for each parameter is chosen as a base choice. A software system usually has one or more normal, or default, modes of the operations and the base choice

should be corresponding to one of these modes [22]. The base choice could also be the most likely used value from a user perspective. The first test case is generated by setting all parameters to their base choice [23]. The remaining test cases are then created by varying one parameter at the time with another possible input value, keeping the remaining parameters fixed at their base choice value. In SEAFOX, this is done by the user choosing a base value within the range of inputs for each parameter. The tool will then generate test cases as described above. An example is shown in Figure 6 where a test suite of test cases with three parameters of integers is generated. Their parameter ranges are [1,3], [1,3] and [0,1], and their base choices are 1, 2 and 0 respectively.

6.2.2. SEAFOX Extension

The version of SEAFOX that we started from only accepted input files in the PLCopen XML format. Since CODESYS v2.3 only exports the program and function blocks as an .exp file, the SEAFOX tool could not be used to generate test cases by importing files from CODESYS v2.3. With a PLCopen XML file, the SEAFOX tool parses the file to find all input variable names and data types. These are then stored in the tool for further use. Our goal was to modify SEAFOX to behave in the same way with an exp-file from CODESYS v2.3 by implementing another parser that can handle this kind of file format. As mentioned in Section 6.2.1, the tool contains a limitation with not being able to keep input variables from several function blocks separated. This limitation will also apply when importing an exp-file.

We started by first studying how the exp-file from CODESYS v2.3 is built by examining exported files from different CODESYS example projects. We found that it is a structured text file where each function block is separated from the other and holds its own input and output variables. SEAFOX is written in the C# language, and there are no built-in functions in C# to parse an exp file format, as there is for xml files. Therefore, we approached a solution of reading the file line by line, searching for the input variables. The input variables in a function block are located in a section starting with "VAR_INPUT" and ending with "END_VAR", as shown in Figure 7 where we can see this section with the two input variables in_one and in_two. The solution is thereby to search for the line "VAR_INPUT" and then store the following information as variable names and data types using the same data structures and logic as the already implemented xml-parser until the line "END_VAR" is read. This solution was implemented in a new parser that is used when the SEAFOX tool is addressed with an .exp-file.

The solution increase the generalizability of the SEAFOX tool to be used with IEC 61131-3 PLC software in CODESYS and is an answer to RQ1.1. It is however not a part of the final tool chain since decisions were made to work with CODESYS v3.5, which imports and exports PLCopen XML.

```
7  FUNCTION_BLOCK FB_Sum
8  VAR_INPUT
9      in_one: UINT;
10     in_two: UINT;
11 END_VAR
12 VAR_OUTPUT
13     out_result : UINT;
14 END_VAR
15 (* @END_DECLARATION := '0' *)
16 out_result := in_one + in_two;
17 END_FUNCTION_BLOCK
```

Figure 7: A function block represented in an exp-file. The name and datatype of the input variables are located between VAR_INPUT and END_VAR.

During the modification, we also extended the generalizability by adding data types to SEAFOX. The tool originally supported the data types REAL, INT, and BOOL, and the pairwise option of the tool was in a previous thesis [13] implemented with also including UINT, USINT, LONG and ULONG. We extended the base choice option with these data types, and random choice with only UINT and USINT. The function in SEAFOX for creating a random number relies on the built-in random function in C#, which returns an integer [24]. This was still manageable when creating a function that returns a random number for UINT and USINT, but since LONG and ULONG can

contain much larger values it would require defining a new algorithm for randomising these. Defining such an algorithm lies outside the scope of this thesis. Further, we extended the tool to support TIME and WORD as a data type for all algorithms. We based our translation of the data types according to CODESYS help manual [25]. The supported data types are shown in Table 2 where the extensions we implemented in the tool is highlighted with an orange background. Furthermore, the source code for our modified version of SEAFOX is available in a GitHub repository[5].

Table 2: Supported data types per algorithm in SEAFOX. X = Supported. - = Not supported. X = our extensions to the tool.

Data Type	Random Choice	Base Choice	Pairwise
BOOL	X	X	X
REAL	X	X	X
INT	X	X	X
WORD	X	X	X
TIME	X	X	X
USINT	X	X	X
UINT	X	X	X
LONG	-	X	X
ULONG	-	X	X

6.3. Selection of the Test Script Execution Tool

The next part of a tool chain that needed to be enhanced was the possibility to execute tests in CODESYS IDE. In this section we review all the examined tools that were available for test execution in CODESYS, to answer RQ1.2. We also motivate why the tool that we selected is the most appropriate one for our thesis.

6.3.1. APTest

The first tool that we studied was the Advanced POU Testing framework (APTest) [17]. It is written in one of the IEC 61131-3 programming languages - Structured Text (ST) language and is built as a unit-testing tool. APTest is implemented as POU-based framework with a supplementary test library that is provided to support test writing with assertions. When integrated into CODESYS environment, test library further can be used for writing the tests for separate POUs using its ready-made functions. APTest library consists of one static program called PRG_TestCase that controls all the test cases, and several sets of functions used for diverse testing approaches such as assertions, that are mainly used for data comparisons with expected outcomes. Framework itself consists of several advanced POU-based unit testing features described more in detail in [17].

As we found out, the APTest framework is only compatible with CODESYS version 2.3, and to be able to use the framework for testing the PLC programs appropriately, it is required to set-up an internal state machine that will update the assertion states in every PLC cycle. Since the framework itself is written in ST language all the tests can be written in the same or any other IEC 61131-3 standard supported language. Although, the creator of the APTest tool [17] states that the framework is relatively easy to apply and use, there is a certain level of PLC programming skills needed to create a completely working test environment. Hence, we chose not to further investigate this test execution tool as an option for our thesis.

6.3.2. CfUnit

CfUnit [26] is an open-source unit testing framework that is compatible with CODESYS v3.5. After downloading and installing the framework into CODESYS, it can be added as a library in a CODESYS project to run tests on the application by adding test suites to the project [27]. To

[5]https://github.com/acn18/DVA331-SEAFOX-02

test a FB, a test function block containing one or more function calls to test cases needs to be written. This is the test suite for the FB. Test cases are implemented as methods where the FB under test is invoked and the inputs and expected outputs for each test are defined. The test code is then written with CfUnit-assert methods where the expected and actual output is compared. These methods also include an error message that will be displayed if the assertion fails. A test can have one or more assert methods. To run the tests, a test program where the test suites are called upon needs to be written and added to the application's task configuration. When the application runs, the test program will run all test suites in the test program and generate an xml-file with the result. The result is also visible in the CfUnit Logger on the device.

CfUnit has, at the time of writing, 34 different assert methods that compare if data types and values, as well as values in arrays, are equal to each other [28]. There are also two assert methods for checking if a condition is true or false. There are no functionalities that allows test case values to be imported into the application, which means that all inputs and expected outputs needs to be added manually unless an external solution is created.

During the upcoming test execution process, we discovered some limitations with the CfUnit tool. Firstly, a PLC can implement timers, such as TON blocks [29], which can be set to delay the block for a certain amount of time. The output from a TON block is False until the timer has enumerated to an input time value. Only after that, the block can be set to True. However, CfUnit only take the input values of the timer into account when generating the output in the test. It does not wait for the timer to enumerate. Next, a maximum of 100 test cases can be run at once in a test suite. Lastly, there are limitations in the documentation of failures when several assertions are used in the same test suite. If a test case include several assertions, only the first failed assertion in the test case will produce an error message. Any failed assertions after that in that specific test case will not be documented. Another relevant aspect is that all tests are executed independently of each other which is not necessarily a limitation, although important to understand when performing tests on the system.

6.3.3. CODESYS Test Manager

The CODESYS Test Manager is an Add-on tool that comes as a part of a tool bundle for CODESYS Professional Development edition [30]. Test Manager is created to allow you as a programmer write and execute automated tests in CODESYS environment. This testing tool is only compatible with CODESYS version 3.5 or higher. Test cases can be written as test scripts or as unit-tests using any of the IEC 61161-3 standard supported textual languages. Testing tool can support tests written for different kind of objects, such as applications or IEC libraries. The specifics of the tool integration and how to run the tests in CODESYS can be found in Test Manager Data Sheet that is available to download from [30]. We did not choose to investigate this testing tool further or use it in our thesis because it is not available as an open-source tool.

6.3.4. Testing Tool selection

Based on availability and user support options between these three testing tools, we chose to include CfUnit as the part of the presented tool chain. The characteristics of all the studied tools are showed and compared in Table 3. Once we had established this important part of the tool chain that will be used for test code writing according to its library as well as for test suite execution, we needed to find a way how to import additional test cases into the test code in CODESYS IDE.

Table 3: The comparison of testing tool characteristics.

Characteristics	APTest [17]	CfUnit [26]	Test Manager [30]
Compatible with CODESYS version	2.3	3.5	3.5
Free of charge	Yes	Yes	No
Test language	ST IEC 61131-3	ST IEC 61131-3	IEC 61131-3
Type of tests	Unit	Unit	Unit
User support*	No	Yes	Yes

* e.g Tutorial, FAQ

6.4. Implementation of Test Scripts

When the SEAFOX tool has generated test case input values, they are saved in a csv-file. To be able to use these values for automated test case creation, we needed to integrate the content of the csv-file into the respective test cases that would be created based on a template from previously exported PLCopen XML file from CODESYS (Step 3 and 4 in Figure 5). We chose to automate this process by providing a script, that would read the csv-file row by row and create new elements of XML file tree that will correspond to the new test cases in program test code. We wrote the script in Python programming language, because CODESYS IDE v3.5 supports the execution only of python scripts directly in the programming environment. Although, it seemed like an obvious step to take to enhance the automation process of the test case integration in PLC programs, the script execution directly in CODESYS did not give the expected results. To run the tests with CfUnit testing tool in CODESYS, it would require adjustments of the program structure by manually moving the test cases to the relevant test suits. Hence, we decided to run the script as an additional external process, that creates a new PLCopen XML file with relevant test cases, and then the file could be imported into CODESYS IDE (Step 5 in Figure 5) ensuring that the new test cases are added to the test code into the correct places.

In the implementation of our script, we followed the pseudo-code in Algorithm 1, that demonstrates the process of the new test case creation while processing the rows from a csv-file. Input values for those test cases are set during the iteration process over each created data element's variable tags. In the python script, we used a well established python library called *xml.etree.ElementTree* that is made specially for parsing, creating and modifying xml files.

Algorithm 1 Pseudo code of python script

```
1: Use xml.etree.parse(FB_under_test) to establish tree as xmlTree
2: Use xmlTree.getroot() to establish root as root
3: for element in xmlTree do
4:     if element includes namespace then
5:         Remove Namespace
6:     end if
7: end for
8: for row in csv-file do //Create new test case
9:     for element in xmlTree/.../inputVariables do //Set input values
10:        if type == TIME then
11:            simpleValue ← "#T" + row[i] + "ms"
12:            i ← i + 1
13:        else
14:            simpleValue ← row[i]
15:            i ← i + 1
16:        end if
17:    end for
18:    for element in root/.../Method do //Set method name
19:        name ← methodname
20:    end for
21:    for element in root/.../Method/body/ST do //Set method body text
22:        Update method name in method body text
23:        methodName_List.append(methodName)
24:    end for
25:    newTestCases_List.append(new test case from xmlTree)
26: end for
27: root/.../pou/body/ST ← methodNames
28: Clear old test case in xmlTree
29: root/.../pou/addData ← newTestCases_List
30: Write xmlTree to outputFile
```

The process starts with taking a PLCopen XML file with the FB test suite in it and parsing the file to find a root tag in the file tree structure (lines 1 to 2). Then for each element in xml-Tree (lines 3 to 7), remove the name-space from element's tags, if such is found. The next iteration (lines 8 to 26) is over the rows in a csv-file that includes three nested for loops, where each is responsible for setting the relevant values for different element tags:

- For each element's input variable, set local variable values (lines 9 to 17). If the variable type is TIME, a special syntax is used, otherwise, value is set as it is from the csv-file located in the current row at given index *i*.
- For each element, set the *Method name* (lines 18 to 20).
- For each element, update method name in *Method body* text and add this name in separate list *methodName_List* (lines 21 to 24).

At line 25, before closing the for loop, all the created elements or new test cases are added in the list *newTestCases_List*. At the line 26, list with test case method names is added to a FB body text. This is an important step to make sure that all the test cases will be executed from the connected FB test suite.

Before writing data to a new PLCopen XML file, the template test case from the FB test suite is removed and instead the list with new test cases is added into a file structure as additional tree data elements (lines 27 to 30).

The overall task of this python script is to merge together two input files with the relevant information taken from both of them, create new test cases or elements, and create a new PLCopen XML file. The script is the final part of our tool chain integration and the answer to RQ1.3. The source code of the script is available in a GitHub repository[6].

[6]https://github.com/lae18001/DVA331-TEST-CONCRETIZER

7. Conduct of the Experiment

In order to create a test automation process that would provide the needed data to measure the specified code coverage afterwards, we needed to prepare a setting that consists of several steps that were already briefly described in Section 5. Here we will describe each step of the experiment process in more detail. The overview of the whole process is shown in Figure 8.

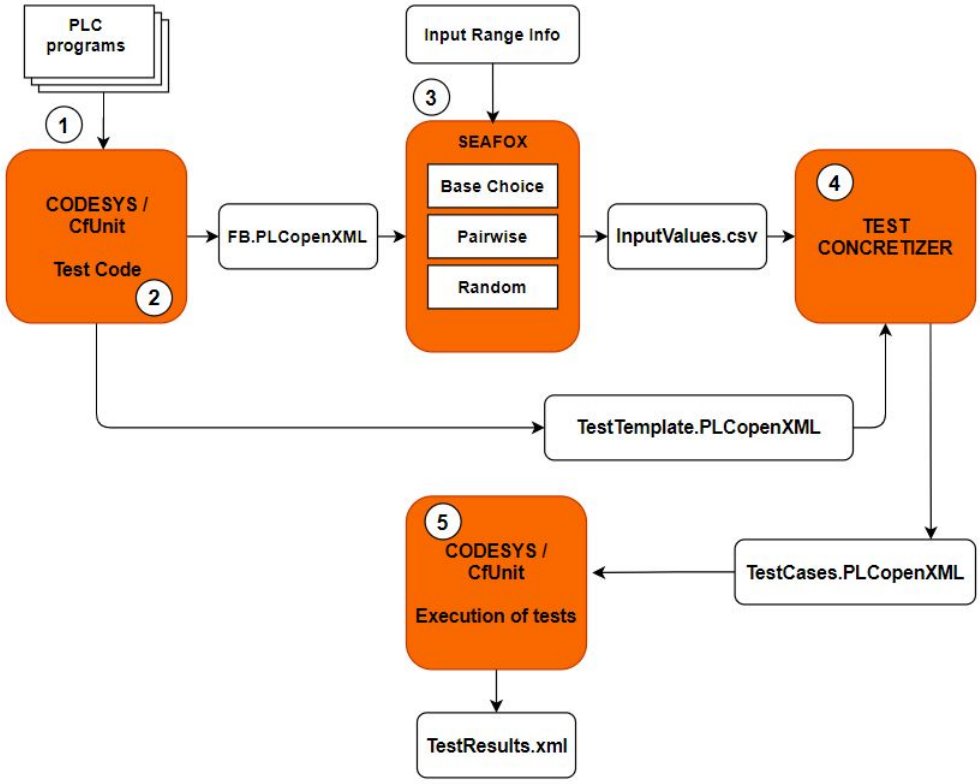

Figure 8: The experimental process.

7.1. Selecting the Programs

The evaluation of the test suites in our created tool chain was measured using observable decision coverage. Therefore, each FB that we tested needed to have at least one output that evaluated to a Boolean value (i.e., returned True or False). Furthermore, there had to be a minimum of two input values to conduct the pairwise algorithm in SEAFOX and the data types had to be supported by the SEAFOX tool. These constraints resulted in a thorough investigation of the provided programs before writing the test cases (Step 1 in Figure 8).

Out of the 18 industrial FBs provided to us, a subset of nine FBs could be used in our experiment. Some of the programs containing these FBs are realistic programs used in industry and were provided by an automation company. Seven FBs were rejected because they did not have a minimum of two inputs or a Boolean as output. Further, two FBs were rejected due to complex data types with nested data structures, which could not be used with SEAFOX to create test case inputs. However, two FBs with complex data types were possible to modify by decomposing the data structures into data types that could be used with SEAFOX without changing the logical structure of the source code and is hence included in the selection.

The final test execution set contained a variation of FBs that consists of both small and large input sizes between 2-41 variables, a variety of 1-3 Boolean outputs, FBs written in FBD and ST languages containing from one block or a few lines of code up to seven blocks or several hundred lines of code. Table 4 shows an overview of the characteristics of the selected programs.

Table 4: The variety of characteristics in the selected FBs for the experiment.

Type	FB_1	FB_2	FB_3	FB_4	FB_5	FB_6	FB_7	FB_8	FB_9
Data type inputs*	B,I,R	W	B	B	B,T	B,R	B,I	B	B,I,T
Programming language	ST	ST	FBD	FBD	FBD	ST	FBD	FBD	FBD
No. of input variables	41	2	8	5	4	28	4	4	4
No. of decision outputs	3	1	1	2	3	1	1	3	1
Lines of code	273	9	-	-	-	144	-	-	-
No. of blocks	-	-	7	6	5	-	7	4	3
No. of connectors	-	-	3	3	3	-	1	3	1

* B = BOOL, I = INT, R = REAL, T = TIME, W = WORD

7.2. Test Code writing in CODESYS

Test cases in CODESYS IDE were written following the instructions by CfUnit [27] (Step 2 in Figure 8), with a change to the initialisation of input variables as well as including the input variables instead of fixed values in the call to the FB to be tested. One test case per test suite was created as a template that was later used with the script and the exported file from SEAFOX to generate all the test cases for the test suite. All input variables in the test case are initialised with a dummy value for enabling the script to set the test case inputs (see Figure 9).

CfUnit provides an assertion method that checks if the condition in the assertion is True. If it is False, an assertion error is created. We used this assertion to compare if the expected return value equalled the actual return value. This method was used throughout the test cases for all function blocks to measure the decision coverage. Due to the limitation in the documentation of failures when several assertions are used in the same test case, as previously mentioned in Section 6.3.2, we created separate test cases for each Boolean output of a FB.

The expected output needs to be manually added to the test case. This value will be the same for every test case in the test suite that is later created with the script. We decided that the expected outcome was assumed to be False in every test case. By having the same value for all test cases, we did not need to manually change the values in the later created test suite. Also, measuring the decision coverage was made easier since there is no need to keep track of which outcome that was expected for each test case when analysing the results. The main point of the coverage is to determine how many outcomes that were taken, which can be achieved without knowing the real expected outcome.

After creating the test case and corresponding FB with the test suite holding this one test case, the test suite is exported as a PLCopen XML file to be used with the script.

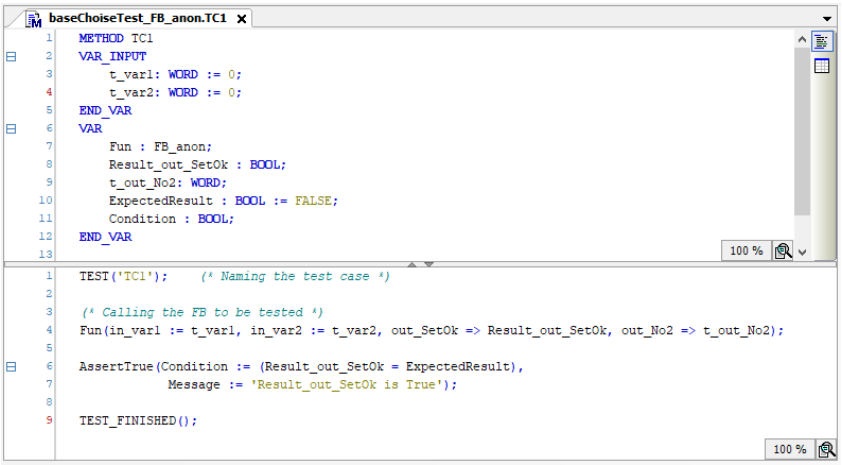

Figure 9: A test case in CODESYS IDE. The input variables are initialised with a dummy value that will later be replaced with a proper test case input through the execution of the script. An assertion that compares the actual and expected result is used to measure the decision coverage.

7.3. Parameter Input Range Specification for SEAFOX

For the SEAFOX tool to generate relevant input values, a user needs to load a file into the tool containing the specific FB that will be tested (as described in Section 6.2.1), and then manually enter the parameter ranges for each variable that this FB takes as input. Every FB has a specific number of input variables and those variables can even be of different types. That is why we needed to come up with a strategy how to specify the ranges for each individual variable (Step 3 in Figure 8).

Our strategy is based on analysis of the code for each FB in the test execution set. More specifically, we tried to understand the logic while also looking for the comments in the code that would lead to a better understanding of what kind of ranges would be appropriate to use for each variable in that FB. If there were already set variable values in the code we used those to partition the input space by setting the ranges around the boundaries of each value. The set value was also used as the base value when the Base choice method in SEAFOX was chosen. Based on an article by Östrand and Balcer [31] we defined the following steps to take for selecting the input space range for each FB:

1. Analyse the code based on comments and logic. Identify and categorise:
   - parameters
   - characteristics of each parameter, e.g.:
     - preset values for parameters
     - comments that indicates values for parameters
     - parameter with similar purpose
   - Interactions between parameters
2. Partition the categories into choices
   - identify the possible values for each category and add these to the parameters input space range
     - preset values are added along with the closest value above and below
     - values found in comments
     - parameters with similar purpose are chosen to have the same input space range
     - values that influence the found interactions
   - identify Base choice value
3. Determine constraints among the choices
   - Identify constraints that can make the program malfunction, e.g., division with 0 and remove these from the input space range (these will interrupt the test execution)
4. As an additional step, we also ensured that all data types that are numbers include the value zero, a negative value, and a positive value, unless any previous step prevented it.

If there were no indication of what kind of values would be expected for the variable, we made an assumption based on commonly used practise for software testing [9, pp. 234-237]. For example, for INT, WORD or REAL types we would always include a range of zero as it is an exceptional case in the software testing practice. For the same types we would add randomly generated numbers for negative and positive values inside the range of allowed values for the type itself. If using the Base choice method for the variables INT, WORD or REAL without any constraints in the code for the possible base value, we would use the smallest randomly generated number. Whereas, for the type TIME the base value would always be set to zero, to allow more frequent variable flow in the cases where the TON block was included.

For the Boolean variables, the parameter range can only be from zero to one, and when using Base choice method, we set the base value to be True or 1, unless there were other indications in the code to choose otherwise. We tried to choose the smallest ranges possible for all the function blocks to avoid exceptionally large test suites to be generated due to limitations in CfUnit regarding test

suites containing more than 100 test cases. The whole spectrum of the input space ranges for each FB is available in our GitHub repository[7].

Once the input ranges had been defined, we generated the test case inputs in SEAFOX. One set of test cases for Pairwise and Base choice methods respectively, and three different sets for each Random method, as described in Section 5.2.1. These were saved as csv-files and stored in a folder already containing the test suite with the test case template.

7.4. Merging the CSV file and PLCopen XML

Merging test case inputs from csv-file and a test template from PLCopen XML is the next step in our experiment set up (Step 4 in Figure 8), to create additional test cases for the relevant test suits. The user required interaction and the workflow of the script execution is demonstrated in Figure 10.

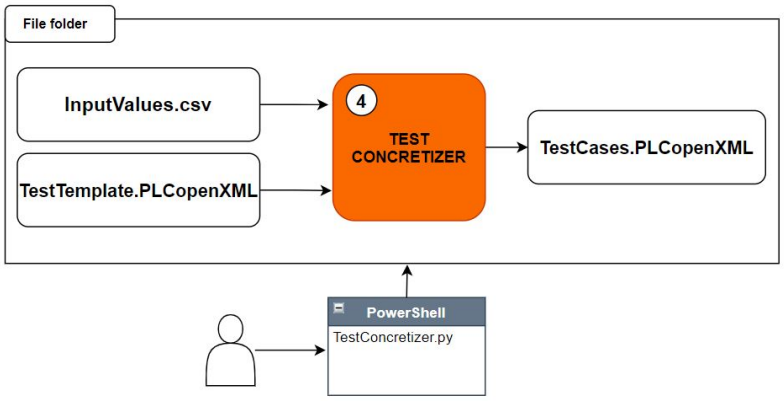

Figure 10: The Workflow of the Script execution.

By using the script, for merging process to begin, we first needed to specify the names of the input files, i.e. the csv-file holding all the input values, and the PLCopen XML file that will be parsed and used as a template for the new test case creation (see Figure 11, row 8 and 11). The name of the output PLCopen XML file that will be created containing all the newly generated test cases also requires to be specified in the script (see Figure 11, row 14). In order to execute the script, all the previous mentioned input files and the script itself needs to be located in the same file folder on a computer. Then the script is ready to be run through the *Command Prompt* or *Windows Power Shell*. If the execution in of the script went as expected, the number of created test cases will be printed out on the screen as well as the message about successfully created new xml-file. The execution of the script in Windows Power Shell and a folder containing the needed files and the created output file are depicted in Figure 12.

testConcretizer.py

```
1  import csv
2  from xml.etree import ElementTree as etree
3  import copy
4
5  ###-------FILE NAMES NEED TO BE CHANGED FOR EVERY FB--------###
6
7  # file with generated input values from SEAFOX
8  input_file = "testcases_FB_anon.csv"
9
10 # exported PLCopenXML file from CODESYS with FB_Test_suite as an example test case
11 file_to_parse = "baseChoiceTest_FB_anon.xml"
12
13 # File that will be imported back to CODESYS with new test cases
14 new_file_name = "output.xml"
15
16 m_name = "TC"          #----CHANGE THE METHOD NAME----#
```

Figure 11: A snippet of the script. Names of input files are specified on row 8 and 11, and the output file on row 14.

[7]https://github.com/acn18/DVA331-SEAFOX-02

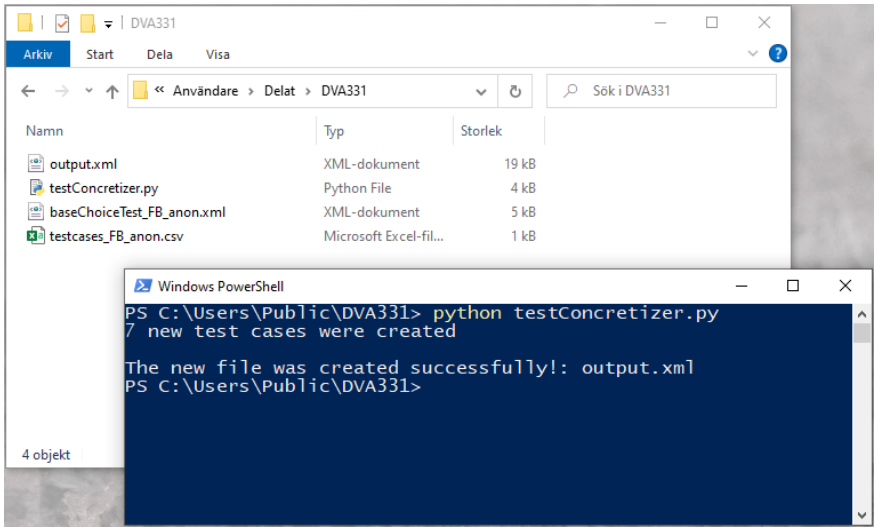

Figure 12: All files used with the script must be located in the same folder to run the script. A confirmation of the merge is visualised in the shell after executing the script.

7.5. Test execution in CODESYS

The device in CODESYS IDE for our experiment is set up to be a standard device with a name: CODESYS *Control Win V3*. This device and the PLC used for simulation needs to be started on a computer before the test program could be run. In *Communication Settings* for the device, the Gateway and the PLC are connected allowing to run the code in CODESYS simulation environment.

The last step of the experiment process (Step 5 in Figure 8) is to import the newly created test cases into the CODESYS IDE. To do so, these steps needs to be followed thoroughly:

- Mark the folder with the test suite where the additional test cases shall be imported
- Open *Project/Import PLCopenXML...*, choose the file to import.
- Select the test cases to import (preferable all). Confirm with *OK*.
- In the pop-up window, select the option *Replace the existing object (for all conflicts)*. Confirm with *OK*.

As shown in the example in Figure 13, the numbered method names as new test cases are imported in the test suite function block *baseChoiceTest_FB_anon* in the folder that was specified when the importing process was started. Now it should be possible to execute the test cases by calling the *CfUnit.RUN()* from the main test program TEST_PRG. When the test code was written, in the main program a variable had to be declared and initiated as the FB that corresponds to the test suite that will be executed (see Figure 13, in the program code, row 4).

The results of the executed test suite are displayed in the Device/Log window if the *Logger* is set to CfUnit, in the same way as demonstrated in Figure 14. The exact same results are also saved locally on the computer as an xml file that is automatically overwritten with the new test results after each test execution. We collected the data after each test execution of displayed results manually by entering the number of test cases that passed or failed in tables created separately for each FB and their respective decisions.

With previously described steps and the order of those demonstrated in Figure 8, the overall experiment can be repeated with the same settings we used for independent variables, i.e., the algorithms in SEAFOX (Pairwise, Base choice and Random) and the FB, under test so that in the process they would measure the same dependent variable (the decision coverage) as outcome. In Section 8, the outcome is presented and analysed further.

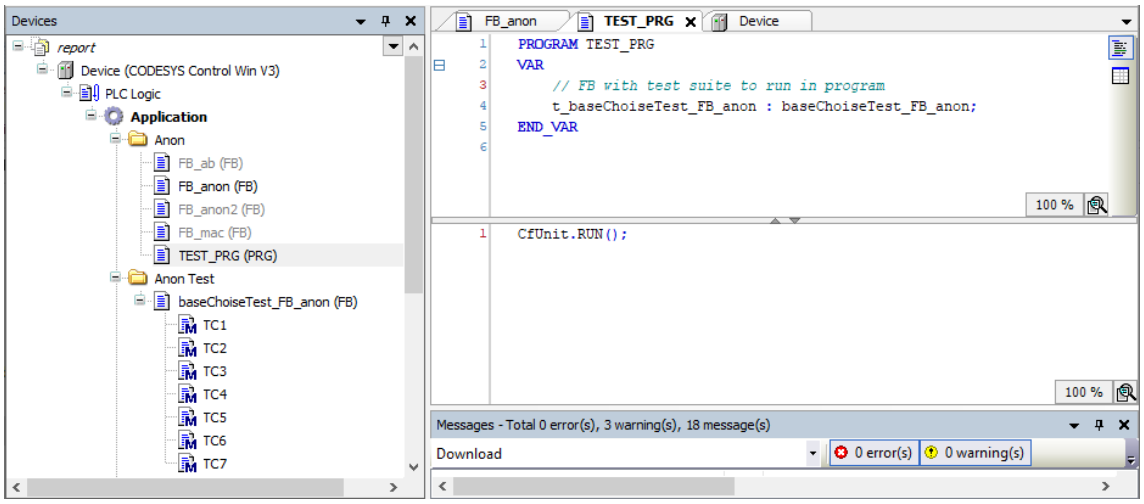

Figure 13: The Test Program with a test suite for function block *FB_anon* containing 7 test cases.

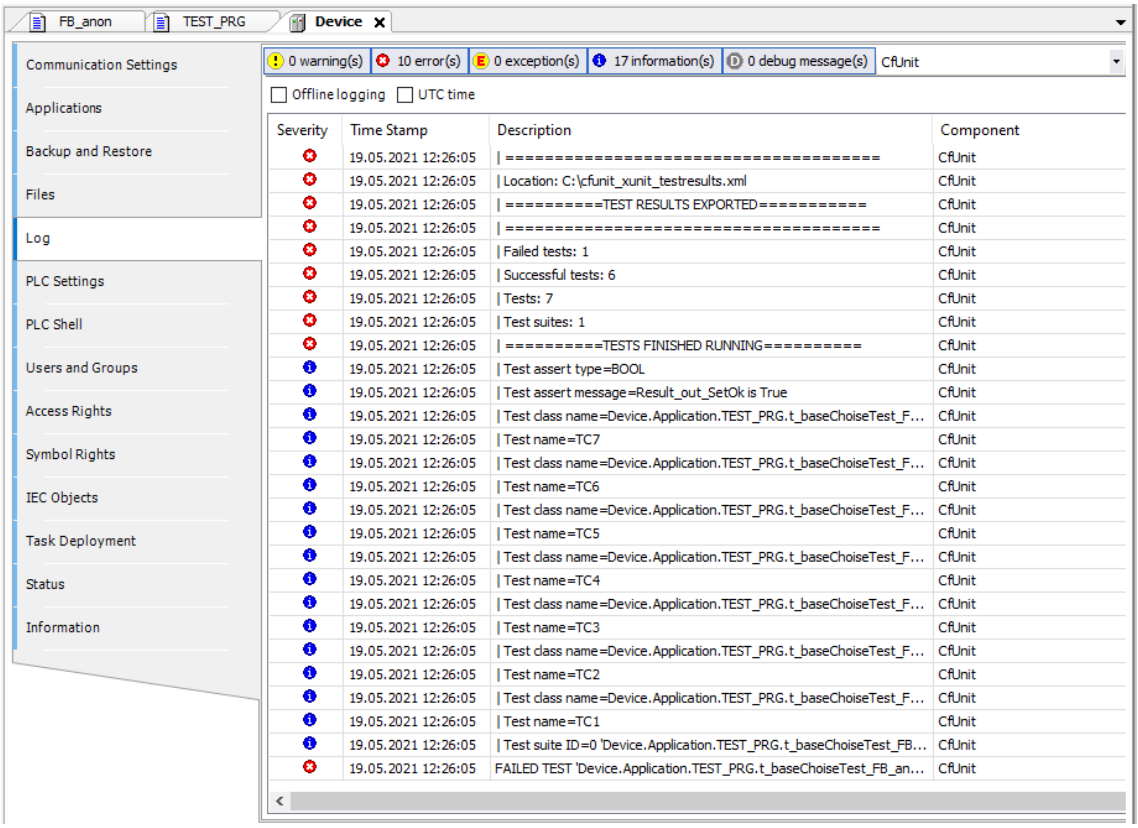

Figure 14: The Device Log printout after test execution in CODESYS IDE

8. Experiment Results

We organised the collected data from the experiment in tables and diagrams which are presented, described, and analysed in this section. Each Boolean output in the experiment was measured individually, as previously described in Section 7.2. If an FB had several Boolean outputs, one test suite per output was created and an average value for the test suite was computed as the decision coverage for the FB before any further calculations were performed. For example, if an FB has two outputs which were measured in two test suites that achieved 50% and 100% respectively, the presented decision coverage for the FB is 75%.

An overview of the decision coverage achieved for each method with their respective mean, minimum, maximum, and median values in percentage is presented in Table 5 in Section 8.1. These results are further displayed as boxplots in Figure 15. Table 6 in Section 8.2 show a different view of the collected data, where the data for each function block is displayed. Throughout the tables and figures, Random(PW) stands for using the random method and generating a test suite of the same size as the pairwise test suite for that function block. In the same way, Random(BC) stands for using the random method and generating a test suite of the same size as the Base choice test suite for that function block. Random 10, Random 30, Random 60, and Random 90 stands for a test suite with 10, 30, 60, and 90 test cases respectively created with the random method. The results in Table 5 shows that all methods obtained an average decision coverage larger than 90%.

8.1. Comparison Between Methods

As seen in Table 5, the tool chain achieved 90% or larger decision coverage on average regardless of the method used. On average, all methods yield 94% decision coverage, which confirms our hypothesis of achieving at least 93%. Random 90 achieved the highest decision coverage among the methods with 98.15%, closely followed by Random 60 that achieved 97.53%. Overall, the Random method performed better than Pairwise and Base choice. This is also reflected in the minimum coverage achieved. Pairwise and Base choice had the lowest measured decision coverage with achieving 50% in one test execution. No other method achieved this low, which we can see in Table 5 and Figure 15. The minimum coverage result can however be misleading since the result of the Random methods is an average of three test executions per function block. This hides the fact that even the random methods did achieve 50% in some test executions. The complete raw data where this is visualised can be found in our GitHub repository[8].

The results for the mean values in Table 5 is the answer to RQ2. The minimum mean decision coverage was measured at 90.74% and the highest at 98.15% when executing test cases on the function block provided to us. The average level of decision coverage achieved based on the methods used in this experiment is 94%. This average value is calculated out of the mean values for each method in Table 5.

Table 5: Decision Coverage in percentage per method used in the experiment.

Algorithm	Mean (%)	Min (%)	Max (%)	Median (%)
Pairwise	90.74	50.00	100.00	100.00
Base Choice	90.74	50.00	100.00	100.00
Random(PW)	94.15	72.33	100.00	100.00
Random(BC)	95.04	77.67	100.00	100.00
Random 10	91.93	77.67	100.00	100.00
Random 30	93.83	66.67	100.00	100.00
Random 60	97.53	83.33	100.00	100.00
Random 90	98.15	83.33	100.00	100.00
Total Mean	94.01	70.13	100.00	100.00

Random 10 achieved 91.93% in mean decision coverage, which is the lowest coverage among the random methods. This was however a rather expected outcome. Since larger test suites will

[8]https://github.com/acn18/DVA331-SEAFOX-02

have a higher possibility of creating more combinations of inputs, it was expected that the random methods with larger test suites should perform better in terms of decision coverage than the ones with a smaller. An interesting test suite size result is that Random 10 achieved a higher decision coverage than both Pairwise and Base choice. Comparing the test suite sizes between the methods (see Table 6), we see that approximately half of Pairwise and Base choice test suites have ten test cases or less, and only two test suites are much larger than this. Even though the results indicate that a small randomly generated test suite can outperform both Pairwise and Base choice methods, the results in our experiment mainly cover smaller test suites also for Pairwise and Base choice and hence can not be generalised for Pairwise or Base choice test suite of all sizes. The smaller test suites in our experiment is a result of our intention to keep the input ranges small to avoid exceptionally large test suites to be generated due to the limitations in CfUnit with test suites containing more than 100 test cases.

Another interesting observation is that Random(PW) and Random(BC) achieved higher decision coverage than Pairwise and Base choice respectively. Analysing the performance of the function blocks when comparing these methods reveals that poorly set base choice values and the influence of TON blocks could be a cause for these differences. This is further analysed in Section 8.2.

The boxplot representation in Figure 15 shows how the data is distributed and is a complement to Table 5. The boxes stretch over the second and third quartile with the mean value marked as an x. The whiskers indicate variability outside the box, and any outliers are marked with a filled circle. The median is marked with a line, but since all medians in the data collection are 100%, this is not visible in the figure. We can see in Figure 15 that Random(PW) and Random(BC) had a larger variety in the results that lies within the lower range of the achieved coverage compared to Pairwise and Base choice, since they have whiskers that spreads further out. They did not however have any outliers, which indicates that the results are more evenly distributed. Comparing the random methods Random 10, 30, 60, and 90, we can see that the decision coverage improves as the test suite grows.

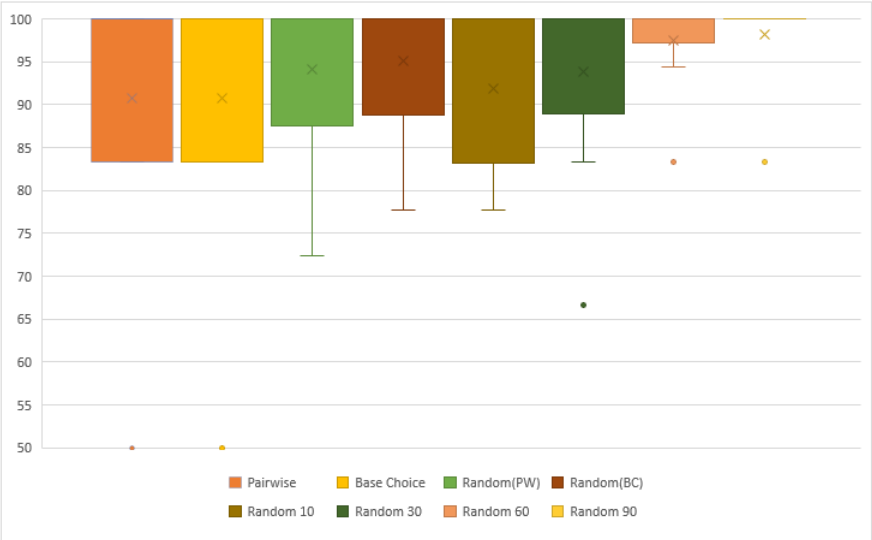

Figure 15: Decision coverage achieved by each method visualised with boxplots.

8.2. In-Depth Comparison Between Function Blocks

In the previous section, we presented the results that are directly connected to RQ2. Here, we will go more into detail about the differences between the results while also focusing more on the individual function blocks. Table 6 show an overview of the collected data. This table is organised with the tested function blocks per row, and the different methods used when generating the test suites, as well as the input size and number of decisions outputs per function block presented

in the columns. The six different test suite sizes conducted when using the random method are divided into Random(PW), Random(BC) and Random in the table, whereas the Random column is further divided into the test suite sizes 10, 30, 60, and 90. The test cases columns present the number of test cases in the test suite, and the coverage is presented in percentage.

Table 6: The number of test cases per test suite and a value of decision coverage achieved in percentage for each FB.

Function Blocks	No. of inputs	No. of decisions	Pairwise		Base Choice		Random(PW)		Random(BC)		Random	
			Test Cases	Coverage (%)	Test Cases	Coverage (%)	Test Cases	Coverage (%)	Test Cases	Coverage (%)	Test Cases	Coverage (%)
FB_1	41	3	59	100	170	83	59	100	170	100	10	78
											30	94
											60	94
											90	100
FB_2	2	1	16	100	7	100	16	100	7	100	10	100
											30	100
											60	100
											90	100
FB_3	8	1	6	100	9	50	6	100	9	100	10	100
											30	100
											60	100
											90	100
FB_4	5	2	6	100	6	100	6	92	6	100	10	100
											30	100
											60	100
											90	100
FB_5	4	3	6	83	6	100	6	72	6	78	10	83
											30	100
											60	100
											90	100
FB_6	28	1	57	100	108	100	57	100	108	100	10	100
											30	100
											60	100
											90	100
FB_7	4	1	14	100	10	100	14	100	10	100	10	83
											30	100
											60	100
											90	100
FB_8	4	3	6	83	5	83	6	83	5	78	10	83
											30	83
											60	83
											90	83
FB_9	4	1	25	50	12	100	25	100	12	100	10	100
											30	67
											60	100
											90	100

While studying the results in Table 6, we noticed that most function blocks scored 100% decision coverage with at least one method. FB_2 and FB_6 stands out with scoring 100% decision coverage with every method. FB_8 on the other hand stands out in the opposite direction with never scoring 100% decision coverage in any of test executions. When examining the FB_8 itself, we found that one of the decisions was dependent on a TON block with the input time set to 14 milliseconds. As described in Section 6.3.2, a TON block can only return True after the timer has enumerated the input time value, which in these tests can only occur if the input time is zero. The decision in FB_8 that was dependent on the TON block could therefore never produce a True outcome and could hence only reach a coverage level of 50%. The TON blocks are also implemented in FB_5 and FB_9. However, in these cases, the time input is presented as input variable that we set in the test execution and not as a preset hard-coded value. This allows for the function blocks to take both True and False outcome depending on the inputs. Although, when examining the results for these function blocks, we found that they had a lower degree of decision coverage than other function blocks of similar size. This indicates that getting the combination of input variables to produce test cases that covers all outcomes of decisions can be more difficult with a TON block than with simpler Boolean blocks, such as AND, OR etc.

The function blocks that had a higher number of decisions also in general had test cases with a lower degree of decision coverage. Out of the three function blocks that had three decisions,

none of them achieved 100% coverage in more than half of the methods used. In comparison, the function blocks that only had one decision achieved 100% coverage in at least three quarters of the used methods.

The number of input variables does however not indicate any significant impact on the decision coverage. Although the results for FB_1 (which has the highest number of inputs) had low decision coverage, we believe that this is more likely due to other factors such as that there are three decision outputs from the function block. In FB_6 that also has a decent number of inputs but only one output, all methods achieved 100% coverage. Therefore, we interpret the results as an indication that the number of inputs are not a major influence that affect the decision coverage.

The combination of a high number of inputs and outputs could however have an impact on the decision coverage since we see a larger variation in the results in FB_1 compared to any of the other function blocks. Specifically, Random 10 achieved a low coverage in this function block which can indicate that the combination of large input size, large output size, and small test suite size has a strong impact on the decision coverage.

Another aspect that we focused on while examining the results was if there were any differences between Pairwise or Base Choice method compared to their Random counterparts. We found that Random(BC) achieved higher coverage than Base Choice in both FB_1 and FB_3. These were two of the more complex function blocks in the test execution set, and where the indications of good base choice values were very limited. The base choice value should preferably correspond to the normal behaviour of the system [22]. If the base choice values when generating base choice test suites are not set with an understanding of their impact, the system could be operating with an abnormal behaviour which can affect the number of decisions taken. Poorly chosen base choice values is a reasonable assumption for the variation of the results between the Base choice and Random(BC) methods in these function blocks. This is further based on that most Random methods also managed to achieve a higher coverage while having a smaller test suite than Base choice.

In the opposite way, the Base choice method managed to achieve full coverage in FB_5 while Random(BC) did not, nor pairwise or Random(PW). Base choice also achieved full coverage in FB_9, which Random(PW) also did, but not Pairwise. Both of these function blocks implemented a TON block where the timer was one of the input variables, and neither of the function blocks were complex in their construct. This allowed for good base choice values to be set, where we could foresee that both outcomes of all decisions would be taken. The difficulties with TON blocks can be the reason why Random(PW) performed better than Pairwise, since we can see that Pairwise did not achieve full coverage in any of the function blocks that implemented a TON. In addition, FB_5 implemented a block where three input variables needed to be True at the same time, which can be difficult with the Pairwise method, but easily managed when choosing base choice values. As mentioned in Section 7.3, we have used the comments and our knowledge when examining the code to find good input space ranges and base choice values. The results in Table 6 indicates that this has been an influential factor in achieving good levels of coverage.

Summing up, our observations are that most function blocks achieved full decision coverage with at least one of the methods used. In the one case where this was not achieved, a timer (TON) with a non-changeable value was the issue that prevented a higher level of coverage. Variations are found between methods for some of the function blocks. One reason for these variations can be the impact of either poorly, or well set base values in the base choice method. Another aspect can be a small test suite that shall cover many inputs and decisions. Comparing the achieved coverage between function blocks with different number of decisions, we noticed that function blocks with more decisions on average had a lower decision coverage. Further, the number of input variables did not directly reflect in the decision coverage, although a combination of large number of inputs with a large number of Boolean outputs might do.

9. Discussion

In this section, we discuss our solution by relating to the order of previously presented research questions. In the first part, we address RQ1 with its sub-questions, and the second part is discussing the achieved results by answering RQ2. Lastly, we discuss the possible threats to validity of our research work.

9.1. The Tool Chain Integration and its Potential Use in Practice

Based on working integration possibilities between the IEC 61131-3 Control Software Development tool CODESYS and the combinatorial testing tool SEAFOX, we have achieved a way to facilitate the PLC testing process with our tool chain. The proposed tool chain can be used to create test cases, generate new input values for these using SEAFOX, be used to automate the additional test case creation and integration, as well as for test execution.

We started our research process by adapting a combinatorial testing tool SEAFOX in a way that it could be integrated with CODESYS IDE v2.3 by translating test case inputs. The exp-file parser was implemented in SEAFOX resulting in a solution to RQ1.1. Although, we did not use this added functionality further in our thesis, we are reasoning that such a possibility gives an additional value to the SEAFOX tool and a more general use and more ways to generate test cases for various PLC software programs developed in the field.

By studying the options for test execution in CODESYS v2.3, we came across the same findings as Hofer and Russo [17]; there were practically no options for automated testing compatible with IEC 61131-3 standard used in industry. They presented an advanced POU testing framework - APTest. We discovered the limitations of using APTest in CODESYS v2.3. Instead we decided to consider working with a newer version of CODESYS IDE – v3.5. This decision led us to discover an open-source testing tool - CfUnit, that contributed to our tool chain in response to RQ1.2. We think that CfUnit, in its current form, is too limited to fully meet the needs of industrial testing as the total number of test cases usually can exceed the limit of one hundred. The other limitations we discovered, especially the difficulties with the TON block, would make it problematic to fully test a PLC programs in a real-time simulation environment using only the current CfUnit functionality.

Accepting the constraints with CfUnit, we moved forward and focused on integration of the input values generated by SEAFOX into relevant test cases. We realised that the process of input value integration would benefit from a script that could automate the additional test case creation. We mention here that the number of test cases could become quite large, depending on the input generation method used. The implementation of the script is linked to RQ1.3. We know that executing our script requires user interaction in terms of the file name specification, which can affect the overall efficiency of the test creation process, but it still can save a lot of time, especially when bigger test suites could be generated.

In the way we propose our tailored tool chain, it provides a more generalizable PLC program testing in industries that follows IEC 61131-3 standards, as well as an improved automation of test execution in CODESYS IDE.

9.2. Decision Coverage Levels Achieved by Combinatorial Testing

After the tool chain was completed, we performed an experiment by running test cases in CODESYS IDE using the tool chain. The result of our experiment shows that over 90% decision coverage can be achieved using the created tool chain with several combinatorial testing techniques: Pairwise, Base choice, and Random. On average, the generated test cases for all methods used in the experiment yield 94% coverage, which exceeds our initial hypothesis of achieving at least 93%. Nonetheless, if we only focus on Pairwise and Base Choice methods that achieved 90.74% coverage, our hypothesis did not hold, but the coverage results are still high. This does not necessarily indicate that the tool chain does not meet its needs, although it hints at the need of investigating improvements to the techniques used. The analysis of the results imply that a higher understanding of good base values and finding ways to overcome the limitations within the CfUnit tool could improve the measured decision coverage.

We found that the random testing methods in our experiment achieved higher coverage levels on average than both Pairwise and Base choice. When focusing only on the set sizes 10, 30, 60, and 90 for the Random method, we clearly see that the coverage improves with the test suite size. We consider this result not surprising since a larger test suite should have a higher chance of covering more combinations. When the number of input variables and the input range for the FB under test is small, it is likely that a large random test suite can even manage to generate all possible combinations.

Our results confirms the conflicting results that Ghandehari et al. [5] studied. Whereas their study demonstrated that pairwise testing in most cases was more efficient than random testing, we add weight to the other side of the Libra where we suggest that the random testing not only performs equal as, but even better than pairwise testing. This contribution to the research field suggests that even more research is needed to further bring more evidence on the comparison between these techniques. The limitations with the TON block can affect the results, in particular the Pairwise method since this method did not achieve full coverage in any of the function blocks where a TON was used. Before making any strong claims regarding the performance of the techniques on IEC 61131-3 standard code in CODESYS IDE, this limitation in the tool should be addressed.

The decision coverage achieved using the Pairwise method is only three percentage points below the results of Charbachi et al. [4], but is not comparable to the results of Enoiu et al. [16]. Average decision coverage levels for Random 90 however was not far behind the results reported by [16]. The SEAFOX tool was used in [4] which makes this an interesting comparison. The similar results in this thesis indicate that the limitations in the CfUnit tool might not have affected the end result on a large scale. Enoiu et al. [16] on the other hand used the CompleteTest tool for test case generation. Not reaching the same level of decision coverage as in [16] while being comparable to the coverage levels in [4] point towards the efficiency of CompleteTest being a tool that creates test cases based on a criterion rather than an inefficiency in the tool chain. It should also be stated that the decision coverage was measured differently in our thesis compared to other studies [4], [16], which should be taken into account when comparing the results.

One factor that can have an influence on the achieved coverage is the chosen range for the input space. If we look more closely at each FB, we see though that all but one FB achieved full coverage with at least one of the combinatorial methods used. This demonstrates that the chosen ranges include the values needed to cover a large number of possible decisions. Even though the ranges were selected based only on the source code, they still appear to be adequate. Improving or extending the ranges might improve the coverage since a larger input space would give a larger test suite with more combinations of inputs. We focused on keeping the test suites small since there are limitations in CfUnit with regards to large test suites. Boolean values always had both 0 and 1 included in the range, but the ranges for other data types were kept as small as possible while still covering all partitions of inputs. Although we did reach an average coverage above 90%, one can still argue that a higher decision coverage could be possible by extending the test suites or changing the input ranges. Since creating the test cases is an automated process in the tool chain, the number of test cases in a test suite does not impose any extra work for the person performing the test (not taking the limitation in the tool into account).

The ability of setting a base choice value also became important in some cases, when the coverage was compared between the combinatorial techniques. We noticed that in the cases where more than two inputs were needed to be set to a specific value for a block to become True, the possibility to set these values with a base choice that favoured the True outcome was beneficial to reach full coverage. This further exposed the importance of setting good base choice values. Using complex function blocks would require a higher experience or different basis than the source code to set these values in a beneficial manner.

The results of this thesis, both in terms of the created tool chain and the result of the experiment, contributes to the knowledge in this research field. By automating the test case creation in CODESYS IDE with the TEST CONCRETIZER, we can guide users of CODESYS and CfUnit by reducing the time needed during the testing process for test creation. The results of the experiment show that decision coverage achieved using the tool chain is in some aspects similar to other previous studies [4], [16]. The experiment is however rather small in terms of programs used and further research is needed to generalise the results within the field.

9.3. Threats to Validity

The test case inputs for our experiment were generated using algorithms in the SEAFOX tool. Other combinatorial testing tools on the market (e.g. ACTS that was used by [3] or PICT used by [5]) might produce different test suites than SEAFOX. Another factor that influences the test case generation is the chosen input space. We modelled the input space ranges to the best of our knowledge, but these values could be set differently if modelled by for example an experienced software engineer. Both of these factors could affect the results of the experiment.

As previously mentioned in section 6.3.2, CfUnit uses assertions in all of the test methods in their library. These can only test outputs from the FB, and the decision coverage in this thesis is hence measured only based on these decisions. By measuring also internal decisions, results could be different.

Another limitation with CfUnit is the inability to enumerate timers during test execution, which can also have had an affect on the result. For example, when a TON block in a function block is implemented, its input variable (of type TIME) sets the timer, and the only way for the output variable of this block to be True is when the enumerator (that is zero from the start) reaches an equal value with this input variable. CfUnit only takes into account the set value of the TON input variable and generates output as False every time the input variable has other value than zero. If the enumerator during the test execution could increase its value, a different levels of decision coverage of this block might be measured. Using another tool that is not bound to this limitation might lead to different results.

We performed the experiment using a set of only nine function blocks, which in total contains 16 observable decisions. Although our tests were performed with a variety of input sizes, data types, and function block types, this is not likely to be sufficient enough to make general claims based on these results. Using programs from different industrial companies containing variety of simple and complex function blocks would be needed to make the results more generalizable within the field.

10. Conclusions

In this thesis, we investigated different possibilities for integrating the test execution process for IEC61131-3 PLC software in CODESYS IDE using the combinatorial testing tool SEAFOX for test case generation and CfUnit - for test execution. Testing of industrial software is in some cases a neglected activity in industry and there is little support for automated testing of PLCs. The goal of the thesis is to establish a working integration by proposing an automated combinatorial testing approach to testing in CODESYS IDE as well as the tool chain evaluation in terms of the decision coverage levels achieved using this approach. We defined two research questions to help us achieve our goal. The questions were focused on finding solutions on how to first modify SEAFOX to handle CODESYS export file format that would enable test cases to be generated. In addition, a tool for executing test cases in CODESYS, and integrating the test cases from SEAFOX exported csv-file with the selected tool for testing was achieved. Lastly, we evaluated the tool chain by means of measuring the Observable Decision Coverage achieved when running test cases.

We found just a few available testing tools that are compatible with CODESYS. Only three tools were found and evaluated out of which we selected CfUnit that is open-source and compatible with CODESYS v3.5, where it integrates seamlessly as a library. Being able to work with CODESYS v3.5 instead of CODESYS v2.3 gives advantages such as the option of exporting and importing PLCopen XML files. This option also eliminated the need to modify SEAFOX. Nevertheless, a modification was performed to enable the use of SEAFOX with CODESYS v2.3. Next necessity was an integration between CfUnit and the test cases exported as a csv-file from SEAFOX to enable an automated process for test case integration in CODESYS IDE. Our solution to this is a python script that takes a test suite with one test case as a PLCopen XML file, and the test case inputs for an arbitrary number of test cases as a csv file and merges these two inputs. The result is a PLCopen XML file that contains a test suite with the newly created test cases. With this integration in place, an automated test generation process was developed.

Test cases created using the integrated tool chain achieved on average 94% Observable Decision Coverage. These test cases are created using Random, Pairwise, and Base choice combinatorial techniques. Random test cases achieved a higher decision coverage than Pairwise and Base choice. All methods achieved scores of 90% or higher. The highest decision coverage of 98.15% was achieved when generating 90 test cases with the Random method. The decision coverage achieved when generating test cases using the Pairwise algorithm in SEAFOX is relatively close to the coverage levels reported by Charbachi et al. [4]. This brings confidence that the tool chain is comparable with other works within this field of research.

We encountered limitations with CfUnit when using certain CODESYS standard blocks as well as in the test execution. These limitations make it problematic to fully test PLC programs in a real-time simulation environment using only the CfUnit testing tool functionality. Moreover, the python script requires some minor manual work when used, which can affect the overall efficiency of the test generation process. Despite these drawbacks, we consider the tool chain to be a contribution to the automation of test creation and execution in CODESYS IDE. In our tool chain, it is fairly easy to create a large number of test cases for a test suite while using combinatorial testing techniques.

11. Future Work

In this thesis, when we integrated the tool chain, we chose to write our own script that is used to automate the additional test case creation. The test script generator creates a new PLCopen XML file containing the test cases that are expected to be imported back to the CODESYS IDE. To further improve the automation of the testing process, it would be of a great benefit if the script used to import additional test cases would be written to support execution directly in CODESYS IDE, since IDE provides such an option. It could eliminate unnecessary external file creation and additional user interaction when working with the tool chain.

In addition, we were not able to test all of the function blocks in the provided programs due to their complex data types used as input variables. To allow the testing of more advanced FBs with complex nested data types, it would be beneficial to further develop SEAFOX by implementing a complex data type handler. This handler should be able to disassemble the nested data types to the level that they could be recognised as the IEC 61131-3 standard data types and then generate the input values for those. So far, the SEAFOX tool has been developed to handle a limited number of IEC 61131-3 standard data types. To be able to include larger programs with a wide range of standard data types used in PLC software programming, the SEAFOX could also be extended to support more of these data types as well.

When it comes to the evaluation of the created tool chain, it would be interesting to see what other experimental set-ups could be exercised. Would the parameter input ranges specified in other ways affect the results amongst available combinatorial input generation methods in SEAFOX? Also, it would be interesting to find out the extent to which choosing different base values for the Base choice method affect the results.

References

[1] K.-H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*, 2nd ed. Berlin, Heidelberg, Germany: Springer-Verlag, 2010.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey on automated software test case generation," *Journal of Systems and Software*, vol. 86, pp. 1978–2001, 2013.

[3] S. Ericsson and E. Enoiu, "Combinatorial Modeling and Test Case Generation for Industrial Control Software Using ACTS," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Lisbon, Portugal, 2018, pp. 414–425.

[4] P. Charbachi, L. Eklund, and E. Enoiu, "Can pairwise testing perform comparably to manually handcrafted testing carried out by industrial engineers?" in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Prague, Czech, 2017, pp. 92–99.

[5] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. Kacker, and R. Kuhn, "An empirical comparison of combinatorial and random testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, Cleveland, OH, USA, 2014, pp. 68–77.

[6] W. Bolton, *Programmable logic controllers*, 4th ed. Oxford, England: Newnes/Elsevier, 2006.

[7] D. H. Hanssen, *Programmable logic controllers : a practical approach to IEC 61131-3 using CODESYS*, 1st ed. Chichester, England: Wiley, 2015.

[8] V. Eldijk, "XML Exchange — PLCopen," Accessed on: 2021-03-30. [Online]. Available: https://plcopen.org/technical-activities/xml-exchange

[9] I. Sommerville, *Software engineering*, 10th ed. Essex, England: Pearson, 2016.

[10] "Iso/iec/ieee international standard - systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, 2010.

[11] R. N. Kuhn, Richard D.and Kacker and Y. Lei, *Introduction to Combinatorial Testing*, ser. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Boca Raton, USA: CRC Press, 2013.

[12] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM computing surveys*, vol. 43, no. 2, pp. 1–29, 2011.

[13] P. Charbachi and L. Eklund, "Pairwise testing for PLC embedded software," BSc thesis, Department of Innovation Design and Engineering, Mälardalens University, Västerås, Sweden, 2016. [Online]. Available: http://mdh.diva-portal.org/smash/get/diva2:938297/FULLTEXT01.pdf

[14] E. Enoiu, A. Čaušević, T. Ostrand, E. Weyuker, D. Sundmark, and P. Pettersson, "Automated test generation using model checking: an industrial evaluation," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 335–353, 2016.

[15] J. J. Chilenski, K. J. Hayhurst, L. K. Rierson, and D. S. Veerhusen, *A Practical Tutorial on Modified Condition/Decision Coverage*. Hampton, VA, USA: National Aeronautics and Space Administration, 2001.

[16] E. P. Enoiu, A. Cauevic, D. Sundmark, and P. Pettersson, "A controlled experiment in testing of safety-critical embedded software," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Chicago, IL, USA, 2016, pp. 1–11.

[17] F. Hofer and B. Russo, "IEC 61131-3 Software Testing: A Portable Solution for Native Applications," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 6, pp. 3942–3951, 2020.

[18] M. Jamro, "POU-Oriented Unit Testing of IEC 61131-3 Control Software," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 5, pp. 1119–1129, 2015.

[19] CODESYS GmbH, "Codesys," Accessed on: 2021-04-14. [Online]. Available: https://www.codesys.com/

[20] K. Säfsten and M. Gustavsson, *Research methodology : for engineers and other problem-solvers*, 1st ed. Lund, Sweden: Studentlitteratur AB, 2020.

[21] M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Proceedings of the 2013 International Conference on software engineering*, ser. ICSE '13, San Francisco, CA, USA, 2013, pp. 102–111.

[22] P. Ammann and J. Offutt, "Using formal methods to derive test frames in category-partition testing," in *Proceedings of COMPASS'94 - 1994 IEEE 9th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 1994, pp. 69–79.

[23] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," *Empirical Software Engineering*, vol. 11, no. 4, pp. 583–611, 2006.

[24] Microsoft, "Random.next method (system)," Accessed on: 2021-04-26. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.random.next

[25] CODESYS, "Data types," Accessed on: 2021-04-26. [Online]. Available: https://help.codesys.com/webapp/_cds_struct_reference_datatypes;product=codesys;version=3.5.16.0

[26] "CODESYS Forge - CfUnit: CODESYS Unit Test Framework / Home / Home," Accessed on: 2021-04-07. [Online]. Available: https://forge.codesys.com/prj/cfunit/landing/Home/

[27] "CODESYS Forge - CfUnit: CODESYS Unit Test Framework / Tutorial / Tutorial," Accessed on: 2021-04-07. [Online]. Available: https://forge.codesys.com/prj/cfunit/home/Tutorial/

[28] "CODESYS Forge - CfUnit / API Reference / v1.0.0.0," Accessed on: 2021-04-07. [Online]. Available: https://forge.codesys.com/prj/cfunit/wiki/v1.0.0.0/

[29] CODESYS, "TON," Accessed on: 2021-05-10. [Online]. Available: https://help.codesys.com/webapp/ton;product=codesys;version=3.5.11.0

[30] CODESYS GmbH, "Codesys test manager," Accessed on: 2021-04-14. [Online]. Available: https://store.codesys.com/codesys-test-manager.html?

[31] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.