Mälardalen University Press Dissertations
No. 214

AUTOMATIC TEST GENERATION FOR INDUSTRIAL CONTROL SOFTWARE

Eduard Enoiu

2016

School of Innovation, Design and Engineering

Copyright © Eduard Enoiu, 2016
ISBN 978-91-7485-291-2
ISSN 1651-4238
Printed by Arkitektkopia, Västerås, Sweden

Mälardalen University Press Dissertations
No. 214

AUTOMATIC TEST GENERATION FOR INDUSTRIAL CONTROL SOFTWARE

Eduard Enoiu

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
tisdagen den 22 november 2016, 13.15 i Pi, Mälardalens högskola, Västerås.

Fakultetsopponent: Professor Mats Heimdahl, University of Minnesota

School of Innovation, Design and Engineering

Abstract

Since the early days of software testing, automatic test generation has been suggested as a way of allowing tests to be created at a lower cost. However, industrially useful and applicable tools for automatic test generation are still scarce. As a consequence, the evidence regarding the applicability or feasibility of automatic test generation in industrial practice is limited. This is especially problematic if we consider the use of automatic test generation for industrial safety-critical control systems, such as are found in power plants, airplanes, or trains.

In this thesis, we improve the current state of automatic test generation by developing a technique based on model-checking that works with IEC 61131-3 industrial control software. We show how automatic test generation for IEC 61131-3 programs, containing both functional and timing information, can be solved as a model checking problem for both code and mutation coverage criteria.

The developed technique has been implemented in the CompleteTest tool. To evaluate the potential application of our technique, we present several studies where the tool is applied to industrial control software. Results show that CompleteTest is viable for use in industrial practice; it is efficient in terms of the time required to generate tests that satisfy both code and mutation coverage and scales well for most of the industrial programs considered.

However, our results also show that there are still challenges associated with the use of automatic test generation. In particular, we found that while automatically generated tests, based on code coverage, can exercise the logic of the software as well as tests written manually, and can do so in a fraction of the time, they do not show better fault detection compared to manually created tests. Specifically, it seems that manually created tests are able to detect more faults of certain types (i.e, logical replacement, negation insertion and timer replacement) than automatically generated tests. To tackle this issue, we propose an approach for improving fault detection by using mutation coverage as a test criterion. We implemented this approach in the CompleteTest tool and used it to evaluate automatic test generation based on mutation testing. While the resulting tests were more effective than automatic tests generated based on code coverage, in terms of fault detection, they still were not better than manually created tests.

In summary, our results highlight the need for improving the goals used by automatic test generation tools. Specifically, fault detection scores could be increased by considering some new mutation operators as well as higher-order mutations. Our thesis suggests that automatically generated test suites are significantly less costly in terms of testing time than manually created test suites. One conclusion, strongly supported by the results of this thesis, is that automatic test generation is efficient but currently not quite as effective as manual testing. This is a significant progress that needs to be further studied; we need to consider the implications and the extent to which automatic test generation can be used in the development of reliable safety-critical systems.

ISBN 978-91-7485-291-2
ISSN 1651-4238

To Raluca

I want to live my life taking the risk all the time that I don't know enough yet. That I haven't read enough, that I can't know enough, that I'm always operating hungrily on the margins of a potentially great harvest of future knowledge and wisdom... take the risk of thinking for yourself. Much more happiness, truth, beauty, and wisdom will come to you that way.

Christopher Hitchens

Acknowledgments

Many people have helped in the writing of this thesis, but none more than my supervisors, Professor Daniel Sundmark, Professor Paul Pettersson and Dr Adnan Čaušević, who are wonderful, wise and inspiring. Huge thanks also to Ola Sellin and the TCMS team in Bombardier Transportation.

I'd like to thank Associate Professor Cristina Seceleanu, Professor Elaine Weyuker, Dr Aida Čaušević and Associate Professor Stefan Stancescu. They have been incredibly supportive throughout my entire higher education. I'm very grateful for their guidance and encouragement.

Big thanks to my family for their love and support. I want to say thank you to my friends at Mälardalen University, who provided that little spark of inspiration.

Finally, I would like to thank the Swedish Governmental Agency of Innovation Systems (Vinnova), the Knowledge Foundation (KKS), Mälardalen University and Bombardier Transportation whose financial support via the Advanced Test Automation for Complex and Highly-Configurable Software-intensive Systems (ATAC) project and ITS-EASY industrial research school, has made this thesis possible.

Västerås, October 2016

List of Publications

Appended Studies[1]

This thesis is based on the following studies:

Study 1. Using Logic Coverage to Improve Testing Function Block Diagrams. Eduard Enoiu, Daniel Sundmark, Paul Pettersson. Testing Software and Systems, Proceedings of the 25th IFIP WG 6.1 International Conference ICTSS 2013, volume 8254, pages 1 - 16, Lecture Notes in Computer Science, 2013, Springer.

Study 2. Automated Test Generation using Model-Checking: An Industrial Evaluation. Eduard Enoiu, Adnan Čaušević, Elaine Weyuker, Tom Ostrand, Daniel Sundmark and Paul Pettersson. International Journal on Software Tools for Technology Transfer (STTT), volume 18, issue 3, pages 335–353, 2016, Springer.

Study 3. A Controlled Experiment in Testing of Safety-Critical Embedded Software. Eduard Enoiu, Adnan Causevic, Daniel Sundmark and Paul Pettersson. Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST), pages 1-11, 2016, IEEE.

Study 4. A Comparative Study of Manual and Automated Testing for Industrial Control Software. Eduard Enoiu, Adnan Causevic, Daniel Sundmark and Paul Pettersson. Submitted to the International Conference on Software Testing, Verification and Validation (ICST), 2017, IEEE.

Study 5. Mutation-Based Test Generation for PLC Embedded Software using Model Checking. Eduard Enoiu, Daniel Sundmark, Adnan Causevic, Robert Feldt and Paul Pettersson. Testing Software and Systems, Proceedings of the 28th IFIP WG 6.1 International Conference ICTSS 2016, volume 9976, pages 155-171, Lecture Notes in Computer Science, 2016, Springer.

Statement of Contribution

In all the included studies, the first author was the primary contributor to the research, approach and tool development, study design, data collection, analysis and reporting of the research work.

[1]All published studies included in this thesis are reprinted with explicit permission from the copyright holders (i.e. IEEE and Springer).

Other Studies

Other relevant papers co-authored by Eduard Enoiu but not included in this thesis:

1. A Study of Concurrency Bugs in an Open Source Software. Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, Eduard Enoiu. IFIP Advances in Information and Communication Technology, Open Source Systems: Integrating Communities, Proceedings of the International Conference on Open Source Systems (OSS), volume 472, pages 16-31, 2016, Springer.

2. Statistical Analysis of Resource Usage of Embedded Systems Modeled in EAST-ADL. Raluca Marinescu, Eduard Enoiu, Cristina Seceleanu. IEEE Computer Society Annual Symposium on VLSI, pages 380-385, 2015, IEEE.

3. Enablers and Impediments for Collaborative Research in Software Testing: an Empirical Exploration. Eduard Enoiu, Adnan Čaušević. Proceedings of the International Workshop on Long-term Industrial Collaboration on Software Engineering, pages 49-54, 2014, ACM.

4. MOS: An Integrated Model-based and Search-based Testing Tool for Function Block Diagrams. Eduard Enoiu, Kivanc Doganay, Markus Bohlin, Daniel Sundmark, Paul Pettersson. International Workshop on Combining Modeling and Search-Based Software Engineering, pages 55 - 60, 2013, IEEE.

5. Model-based Test Suite Generation for Function Block Diagrams using the UPPAAL Model Checker. Eduard Enoiu, Daniel Sundmark, and Paul Pettersson. Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 158 - 167, 2013, IEEE.

6. A methodology for formal analysis and verification of EAST-ADL models. Eun-Young Kang, Eduard Enoiu, Raluca Marinescu, Cristina Seceleanu, Pierre Yves Schnobbens, Paul Pettersson. International Journal of Reliability Engineering & System Safety, volume 120, pages 127–138, 2013, Elsevier.

7. A Design Tool for Service-oriented Systems. Eduard Enoiu, Raluca Marinescu, Aida Čaušević, Cristina Seceleanu. Proceedings the International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA), volume 295, pages 95-100, 2013, Elsevier.

8. ViTAL : A Verification Tool for EAST-ADL Models using UPPAAL PORT. Eduard Enoiu, Raluca Marinescu, Cristina Seceleanu, Paul Pettersson. International Conference on Engineering of Complex Computer Systems (ICECCS), pages 328-337, 2012, IEEE.

Contents

Acknowledgments v

List of Publications vii

I Thesis Summary **1**

1 Introduction . . . 3
2 Background . . . 4
2.1 Industrial Control Software . . . 4
2.2 Programmable Logic Controllers . . . 5
2.3 Test Generation . . . 6
3 Summary of Contributions . . . 9
3.1 Motivation . . . 9
3.2 Research Objective . . . 9
3.3 RG1: Techniques and Tool Implementation . . . 11
3.4 RG2: Applicability . . . 13
3.5 RG3: Cost and Fault Detection Evaluation . . . 14
4 Related Work . . . 16
4.1 Test Generation Technique and Tool Implementation . . . 16
4.2 Empirical Studies . . . 17
5 Conclusions and Future Work . . . 19
6 Outline of the Thesis . . . 20

Bibliography 21

II Studies **29**

1 Using Logic Coverage to Improve Testing Function Block Diagrams 31
1 Introduction . . . 33
2 Preliminaries . . . 35
2.1 FBD Programs and Timer Components . . . 35
2.2 Networks of Timed Automata . . . 35
2.3 Logic-based Coverage Criteria . . . 36
3 Testing Methodology and Proposed Solutions . . . 36
4 Function Block Diagram Component Model . . . 38
5 Transforming Function Block Diagrams into Timed Automata . . . 39

6 Test Case Generation using the UPPAAL Model-Checker . . . 41
7 Logic Coverage Criteria for Function Block Diagrams . . . 42
8 Example: Train Startup Mode . . . 43
8.1 Experiments . . . 44
8.2 Logic Coverage and Timing Components . . . 46
9 Related Work . . . 47
10 Conclusion . . . 47
References . . . 48

2 Automated Test Generation using Model-Checking: An Industrial Evaluation 51
1 Introduction . . . 53
2 Preliminaries . . . 55
2.1 Programmable Logic Controllers . . . 55
2.2 The Compressor Start Enable Program . . . 57
2.3 Networks of Timed Automata . . . 57
2.4 Logic-based Coverage Criteria . . . 59
3 Translation . . . 60
3.1 FBD Structure . . . 60
3.2 Cycle Scan and Triggering . . . 61
3.3 Translation of basic blocks . . . 63
4 Testing Function Block Diagram Software using the UPPAAL Model-Checker . . . 65
5 Analyzing Logic Coverage . . . 67
6 Overview of the Toolbox . . . 69
6.1 User Interface . . . 69
6.2 Toolbox Architecture . . . 73
6.3 PLCOpen XML Standard . . . 74
6.4 Implemented Model Translation . . . 74
6.5 Dynamic Traces - JavaCC - Test Cases . . . 77
7 Experimental Evaluation and Discussions . . . 77
8 Related Work . . . 82
9 Conclusion . . . 83
References . . . 84

3 A Controlled Experiment in Testing of Safety-Critical Embedded Software 87
1 Introduction . . . 89
2 Testing PLC Embedded Software . . . 91
2.1 Specification-Based Testing of IEC 61131-3 Software . . . 92
2.2 Implementation-Based Testing for IEC 61131-3 Software . . . 92
3 Experiment Design . . . 93
3.1 Research Questions . . . 93
3.2 Experimental Setup Overview . . . 94

3.3 Operationalization of Constructs . . . 95
3.4 Instrumentation . . . 97
3.5 Data Collection Procedure . . . 98
4 Experiment Conduct . . . 98
5 Experiment Analysis . . . 99
5.1 Fault Detection . . . 101
5.2 Decision Coverage . . . 103
5.3 Number of Tests . . . 104
5.4 Testing Duration . . . 104
5.5 Cost-effectiveness Tradeoff . . . 107
5.6 Limitations of the Study and Threats to Validity . . . 107
6 Related Work . . . 108
7 Conclusions and Future Work . . . 109
References . . . 109

4 A Comparative Study of Manual and Automated Testing for Industrial Control Software 113
1 Introduction . . . 115
2 Related Work . . . 117
3 Method . . . 118
3.1 Case Description . . . 119
3.2 Test Suite Creation . . . 120
3.3 Subject Programs . . . 121
3.4 Measuring Code Coverage . . . 122
3.5 Measuring Fault Detection . . . 122
3.6 Measuring Efficiency . . . 123
4 Results . . . 126
4.1 Fault Detection . . . 126
4.2 Fault Detection per Fault Type . . . 127
4.3 Coverage . . . 130
4.4 Cost Measurement Results . . . 131
5 Discussions and Future Work . . . 132
6 Threats to Validity . . . 133
7 Conclusions . . . 134
References . . . 134

5 Mutation-Based Test Generation for PLC Embedded Software using Model Checking 139
1 Introduction . . . 141
2 Background and Related Work . . . 142
2.1 PLC Embedded Software . . . 143
2.2 Automated Test Generation for PLC Embedded Software . . . 143
2.3 Mutation Testing . . . 144
3 Mutation Test Generation for PLC Embedded Software . . . 144
3.1 Mutation Generation . . . 145

3.2 Model Aggregation 146
3.3 Mutant Annotation 147
3.4 Test Generation 148
4 Experimental Evaluation 149
5 Experimental Results and Discussion 151
5.1 Discussion 154
6 Conclusions 155
References 156

Part I

Thesis Summary

1 Introduction

Software plays a vital role in our daily lives and can be found in a number of domains, ranging from mobile applications to medical systems. The emergence and wide spread usage of large complex software products has profoundly influenced the traditional way of developing software. Nowadays, organizations need to deliver reliable and high-quality software products while having to consider more stringent time constraints. This problem is limiting the amount of development and quality assurance that can be performed to deliver software. Software testing is an important verification and validation activity used to reveal software faults and make sure that the expected behavior matches the actual software execution [4]. In the academic literature a *test* or a *test case* is usually defined as an observation of the software, executed using a set of inputs. A set of test cases is called a *test suite*. Eldh [20] categorized the goals used for creating a test suite into the following groups: specification-based testing, negative testing, random testing, coverage-based testing, syntax and/or semantic-based testing, search-based testing, usage-based testing, model-based testing, and combination techniques. For a long time these software testing techniques have been divided into different levels with respect to distinct software development activities (i.e., unit, integration, system testing).

If software testing is severely constrained, this implies that less time is devoted to assuring a proper level of software quality. As a solution to this challenge, automatic test generation has been suggested to allow tests to be created with less effort and at lower cost. In contrast to manual testing, test generation is automatic in the sense that test creation satisfying a given test goal or given requirement is performed automatically. However, over the past few decades, it has been a challenge for both practitioners and researchers to develop strong and applicable test generation techniques and tools that are relevant in practice. The work included in this thesis is part of a larger research endeavor well captured by the following quotation:

> *"Test input generation is by no means a new research direction... but the last decade has seen a resurgence of research in this area and has produced several strong results and contributions. This resurgence may stem, in part, from improvements in computing platforms and the processing power of modern systems. However, we believe... that researchers themselves deserve the greatest credit for the resurgence, through advances in related areas and supporting technologies..."*
> *(A. Orso and G. Rothermel, Software testing: a research travelogue (2000–2014), Future of Software Engineering, ACM, 2014.)*

We notice that "*advances in related areas and supporting technologies*" refers in part to contributions to automatic test generation. In the literature, a great number of techniques for automatic test generation have been proposed [54]. The general idea behind these techniques is to describe the test goal in a mathematical model, and then generate a set of inputs for a software program by searching towards the goal of achieving some coverage or satisfying a certain reachability property. The advantage of this approach is that it can be used early in the software development cycle to

reveal software faults and exercise different aspects of a program. However, tools for automatic test generation are still few and far from being applicable in practice. As a consequence, the evidence regarding the mainstream use of automatic test generation is limited. This is especially problematic if we consider relying on automatic test generation for thoroughly testing industrial safety-critical control software, such as is found in trains, cars and airplanes. In these kind of applications, software failures can lead to economical damage and, in some cases, loss of human lives. This motivated us to investigate the use of automatic test generation and identify the empirical evidence for, or against, the use of it in practice when developing industrial control software.

In this thesis we study and develop automatic test generation techniques and tools for a special class of software known as industrial control software. In particular, we focus on IEC 61131-3, a popular programming language used in different control systems.

2 Background

In this section, major aspects of industrial control software and automatic test generation are discussed. The presented aspects are related to current research in the field as well as the research done in this thesis.

2.1 Industrial Control Software

An Industrial Control Software (ICS) is a type of software typically used in industries such as transportation, chemical, automotive, and aerospace to provide supervisory and regulatory control [72]. This type of software is vital to the operation of critical infrastructures. ICS has different characteristics that differ from traditional software. Some of these differences are direct consequences of the fact that the behavioral logic executing in an ICS has a direct effect on the physical world which includes significant risk to the health and safety of human lives, serious environment damage, as well as serious economical issues. ICS have unique performance, reliability and safety requirements and are often running on domain-specific operating systems and hardware. Examples of ICS include Supervisory Control and Data Acquisition (SCADA) software [10], software running on a Distributed Control Systems (DCS) [15], and control programs running on Programmable Logic Controllers (PLCs) [8].

SCADA software is used to control systems scattered geographically, where the control behavior is critical to the system operation as a whole [45]. They are used in water distribution, oil and natural gas pipelines, electrical power grids, and railway systems. DCS are systems based on a control architecture containing a supervisory control level overseeing large numbers of locally integrated software or hardware controllers. PLCs are computer devices used for controlling industrial equipment. Even if software running on a PLC can be used throughout large SCADA and DCS systems, they are often the primary components in smaller control systems used to provide operational process control of such systems as trains, car assembly lines and power plants. PLCs [45] are used extensively in almost all industries. This thesis

is focused on developing techniques for testing the behaviors of industrial control software implemented on PLCs.

2.2 Programmable Logic Controllers

A PLC is a dedicated computer implemented using a processor, a memory, and a communication bus. PLCs contain a programmable memory for storing programs exhibiting different behaviors such as logical, timing, input/output control, proportional-integral-derivative (PID) control, communication and networking, and data processing. The control software is used to monitor signals, sensors or actuators, what parameter ranges are acceptable and reliable, and what kind of response the system should give when any of the parameters are behaving outside their acceptable values. The semantics of software running on a PLC has the following representative characteristics:

- execution in a cyclic loop where each cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs).
- Inputs and outputs correspond to internal signals, sensors, or actuators.

IEC 61131-3 is a popular programming language standard for PLCs used in industry because of its simple textual and graphical notations and its digital circuit-like nature [53]. As shown in Figure 1, blocks in an IEC 61131-3 program can be represented in a Function Block Diagram (FBD). These diagrams form the basis for composing applications. These blocks may be supplied by the hardware manufacturer (e.g., Set-Reset (SR), Select (SEL), Greater-Than (GT), AND and XOR), defined by the user, or predefined in a library (e.g., On-Delay Timer (TON) and Timer-Pulse (TP)). An application translator is used to automatically transform each program to compliant code. A PLC periodically scans an IEC 61131-3 program, which is loaded into the PLC memory. The IEC 61131-3 program is created as a composition of interconnected blocks, which may have inner data communication. When activated, a program consumes one set of inputs and then executes the interconnected blocks to completion. The program runs on a specific PLC hardware.

The IEC 61131-3 [36] standard proposes a hierarchical software architecture for composing any IEC 61131-3 program. This architecture specifies the syntax and semantics of a unified control software based on a PLC configuration, resource allocation, task control, program definition, block repository, and program code [53, 73]. PLCs contain a particular type of blocks called *PLC timers*. These timers are real-time instructions that provide the same functions as timing relays and are used to activate or deactivate a signal or a device after a preset interval of time. There are two different timer blocks (i) On-delay Timer (TON) and (ii) Off-delay Timer (TOF). A timer block keeps track of the number of times its input is either true or false and outputs different signals. In practice many other timing configurations can be derived from these basic timers.

This thesis is focused on developing automatic test generation techniques for testing the functional and timing behaviors of IEC 61131-3 control programs running on

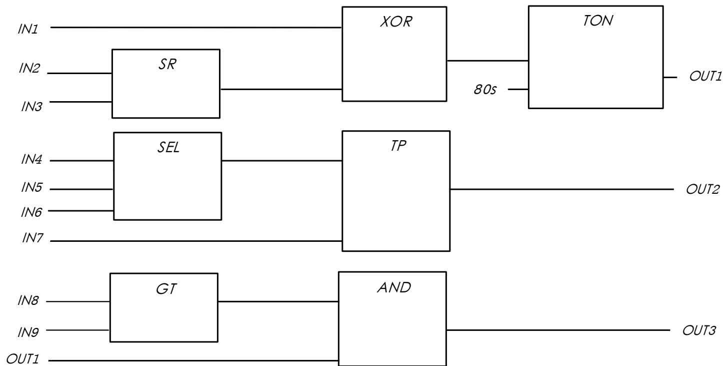

Figure 1: Example of a IEC 61131-3 Program.

PLCs. The following sections will provide details on test generation as a general concept used in this thesis.

2.3 Test Generation

The process of test generation is that of using methods to find suitable test inputs using a description of the test goal that guides towards a certain desirable property. In Figure 2, a typical setting for automatic test generation is shown. This thesis is mainly concerned with algorithmic test generation techniques where the *test goal* is reached automatically, as opposed to techniques that require manual assistance. Such algorithms derive tests for a desired test goal. A test includes inputs that stimulate the software. For PLCs this could be parameters to start the software, a sequence of inputs and the timing when these inputs should be supplied. The *automatic test generation* results in a set of tests called a *test suite*. A *test execution* framework runs the test suite against the *software under test* and produces a *test result*, which is compared to the expected result, imposed by the *requirement*. This results in a *test verdict*, ideally being a pass or a fail.

Requirement-based Testing

Like other software engineering disciplines, many of today's automatic test generation techniques use *requirement models* to guide the search towards achieving a certain goal. Many notations are used for such models, from formal - mathematical descriptions [17] and semi-formal notations such as the Unified Modeling Language (UML) [51] to natural language requirements. Formal requirement models with precise semantics are suitable for automatic test generation [71]. Even so, recent results have showed that natural language is still the dominant documentation format in control and embedded software industry for requirement specification [67] even if engineers are

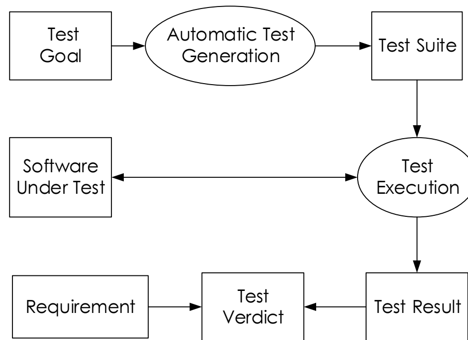

Figure 2: A typical automatic test generation scenario.

dissatisfied with using natural language for requirements specification. This thesis is focusing on specifications expressed in a natural language, as this is still a realistic scenario for testing industrial control software.

A requirement model is an abstraction of a desired software behavior and different types of abstractions are often needed to construct it. Requirement-based testing (also known as specification-based testing) is a technique where tests are derived from a requirement model that specifies the expected behavior of the software. Different test thoroughness measures based on requirement models have been proposed for guiding the generation of suitable tests: conformance testing [47, 32, 75], coverage criteria that are based on the specifications [52, 3, 78], domain/category partitioning [5, 55], just to name a few.

Requirement-based testing requires the understanding of both the specified requirements and the program under test. The specification usually contains preconditions, input values and expected output values [4] and a tester uses this information to manually or automatically check the software conformance with the specification. A test suite should contribute to the demonstration that the specified requirements and/or coverage criteria have indeed been satisfied. Engineering of industrial control software typically requires a certain degree of certification according to safety standards [12]. These standards pose specific concerns on testing (e.g., the demonstration of requirement testing and/or some level of code coverage). In this thesis we seek to investigate the implications of using requirement-based testing for IEC 61131-3 control software.

Implementation-based Test Generation

Implementation-based testing is usually performed at unit level to manually or automatically create tests that exercise different aspects of the program structure. To support developers in testing code, implementation-based test generation has been explored in a considerable amount of work [54] in the last couple of years from code coverage criteria to mutation testing.

Code Coverage Criteria

Code coverage criteria are used in software testing to assess the thoroughness of test cases [4]. These criteria are normally used to determine the extent to which the software structure has been exercised. In the context of traditional programming languages (e.g., Java and C#), decision coverage is usually referred to as *branch coverage*. A test suite satisfies branch coverage if running the test cases causes each branch in the software to have the value *true* at least once and the value *false* at least once. In this thesis, code coverage is used to determine if the test goal is satisfied.

Numerous techniques for automatic test generation based on code coverage criteria [23, 11, 74, 82, 44] have been proposed in the last decade. An example of such an approach is EvoSuite [23], a tool based on genetic algorithms, for automatic test generation of Java programs. Another automatic test generation tool is KLEE [11] which is based on dynamic symbolic execution and uses constraint solving optimization as well as search heuristics to obtain high code coverage. In this thesis, we investigate how an automatic test generation approach can be developed for IEC 61131-3 control software and how it can be adopted for testing industrial control software. Moreover, we evaluate and compare such techniques with manual testing performed by industrial engineers.

Mutation Testing

Recent work [28, 37] suggests that coverage criteria alone can be a poor indication of fault detection. To tackle this issue, researchers have proposed approaches for improving fault detection by using mutation analysis as a test goal. Mutation analysis is the technique of automatically generating faulty implementations of a program for the purpose of examining the fault detection ability of a test suite [16]. During the process of generating mutants one should create syntactically and semantically valid versions of the original program by introducing a single fault or multiple faults (i.e., higher-order mutation [41]) into the program. A *mutant* is a new version of a program created by making a small change to the original program. For example, a mutant can be created by replacing a method with another, negating a variable, or changing the value of a constant. The execution of a test case on the resulting mutant may produce a different output than the original program, in which case we say that the test case *kills* that mutant. The mutation score can be calculated using either an output-only oracle (i.e., strong mutation [79]) or a state change oracle (i.e., weak mutation [34]) against the set of mutants. When this technique is used to *generate* test suites rather than evaluating existing ones, it is commonly referred to as *mutation testing* or mutation-based test generation. Despite its effectiveness [43], no attempt has been made to propose and evaluate mutation testing for PLC industrial control software. This motivated us to develop an automatic test generation approach based on mutation testing targeting this type of software.

3 Summary of Contributions

In this section, the motivation for the thesis is shown based on the challenges and gaps in the scientific knowledge. In addition, the overall research objective is presented and broken down into specific research goals that the thesis work aims to achieve.

3.1 Motivation

Numerous automatic test generation techniques [54] provide test suites with high code coverage (e.g., branch coverage), high requirement coverage, or to satisfy other related criteria (e.g., mutation testing). However, for industrial control software contributions have been more sparse. The body of knowledge in automatic test generation for IEC 61131-3 control programs is limited, in particular in regards to tool support, empirical evidence for its applicability, usefulness, and evaluation in industrial practice. The motivation for writing this thesis stems in part from a fundamental issue raised by Heimdahl [30]:

> *"...reliance on models and automated tools in software development, for example, formal modeling, automated verification, code generation, and automated testing, promises to increase productivity and reduce the very high costs associated with software development for critical systems. The reliance on tools rather than people, however, introduces new and poorly understood sources of problems, such as the level of trust we can place in the results of our automation."*

From a software testing research point of view, this thesis is also motivated by the need to provide evidence that automatic test generation can perform comparably with manual testing performed by industrial engineers. Consequently, we identified the general problem as:

> **The need for a tool-supported approach for testing IEC 61131-3 control software that can be usable and applicable in industrial practice.**

In the next sections, we introduce the research objective of the thesis, present the contributions, and show how the research goals are addressed by the contributions. This research started with the problem of implementing, adopting and using automatic test generation in industrial control software development, and ends with proposing a solution for this problem while building empirical knowledge in the area of automatic test generation.

3.2 Research Objective

The objective of this thesis is to propose and evaluate an automatic test generation approach for industrial control software written in the IEC 61131-3 programming language and identify the empirical evidence for, or against, the use of it in industrial practice.

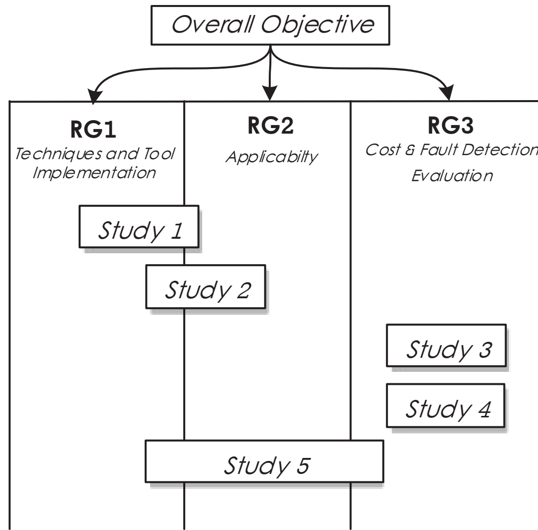

Figure 3: Overview of the how the studies included in this thesis support the three research goals.

As shown in Figure 3 the research was performed during five studies (i.e., Studies 1-5). Combined, these studies provided support to the overall objective of this thesis and a contribution to the body of knowledge in automatic test generation for IEC 61131-3 control software. The studies included in this thesis used different research methods (i.e., case studies and controlled experiments), data collection strategies (unstructured interviews, document analysis and observation) and data analysis statistics.

Practically, the research objective was broken down into three research goals:

RG 1. *Develop automatic test generation techniques for testing industrial control software written in the* IEC 61131-3 *programming language.*

This goal refers to building an approach for automatic test generation that would support IEC 61131-3 control software. This thesis is mainly concerned with test generation techniques where the test goal is based on the implementation itself (i.e., code coverage criteria, mutation testing). In certain application domains (e.g., railway industry) testing IEC 61131-3 software requires certification according to safety standards [12]. These standards recommend the demonstration of some level of code coverage on the developed software. In order to be able to automatically generate tests, a translation of the IEC 61131-3 software to timed automata is proposed. The resulting model can be automatically analyzed towards finding suitable tests achieving some type of code coverage. To tackle the issue of generating tests achieving better fault detection, an approach is proposed that targets the detection of injected

faults for IEC 61131-3 software using mutation testing. Support for this research goal was acquired in studies 1, 2 and 5.

RG 2. *Evaluate the applicability of the proposed automatic test generation techniques in an industrial context.*

Applicability refers to its success in meeting efficiency (e.g., generation time) and test goal requirements (e.g., achieving code coverage). This makes this goal key to determine the value of its feasibility in an industrial context. The results contributing to this goal were acquired primarily from studies 1, 2 and 5.

RG 3. *Compare automatic test generation with manual testing in terms of cost and fault detection.*

This goal addresses how automatically created tests compare to manually written ones in terms of effectiveness and cost. This goal also concerns the difference between designing tests based on requirements and creating test suites solely satisfying code coverage. Given that recent work [28] suggests that code coverage criteria alone can be a poor indication of fault detection, with this goal we seek to investigate overall implications of using manual testing, specification-based testing, code coverage-adequate and mutation-based automatic test generation for IEC 61131-3 industrial control software. The aim of this goal was to provide experimental evidence of test effectiveness in the sense of bug-finding and cost in terms of testing time. Explicit work to contribute to this goal was performed in studies 3, 4 and 5.

In the next section we describe the main contributions of this thesis. The contribution is divided into three parts: techniques and tool implementation, applicability, and evaluation of cost and fault detection.

3.3 RG1: Techniques and Tool Implementation

Code Coverage-Based Test Generation

The thesis work began with the development of an automatic test generation technique that provided initial results in the form of two studies (i.e., study 1 and 2). In study 2 several improvements to the technique proposed in study 1 regarding the model transformation are described. The main objective of these two studies was to show how code coverage can be measured on IEC 61131-3 software and how, by transforming an IEC 61131-3 program to timed automata [2], test suites can be automatically generated. There have been many models introduced in the literature for describing industrial control software and PLCs [18, 1, 9, 49, 65, 70, 29]. One of the most used models is the timed automata formalism. A timed automaton is a standard finite-state automaton extended with a collection of real-valued clocks. The model was introduced by Alur and Dill [2] in 1990 and has gained in popularity as a suitable model for representing timed systems. In this thesis timed automata is used for modeling the functional and timing behavior of PLCs.

As shown in Figure 4, an approach is devised for translating IEC 61131-3 programs to a suitable representation containing the exact functional and timing behavior to

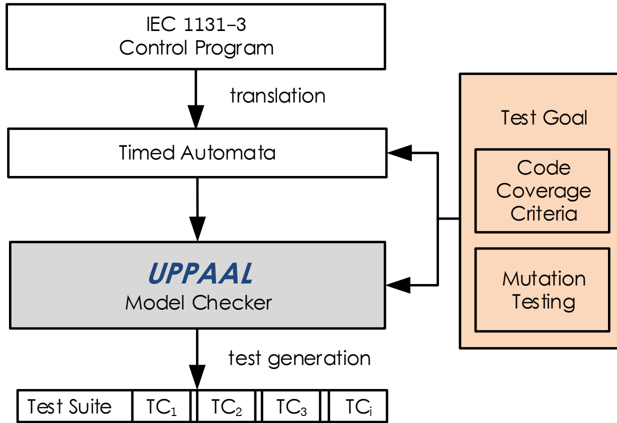

Figure 4: Overview of the implemented automatic test generation techniques.

be used for test generation purposes. Practically, an automatic model–to–model transformation to timed automata is described. The transformation accurately reflects the code characteristics of the IEC 61131-3 language by constructing a complete behavioral model which assumes a *read-execute-write* program semantics. The translation method consists of four separate steps. The first three steps involve mapping all the interface elements and the existing timing annotations. The latter step produces a behavioral description and coverage instrumentation instructions for every standard block in the program. The output of these steps is a network of timed automata which is used by the UPPAAL model-checker [48] for test generation. UPPAAL checks that a reachability property describing the code coverage goal is satisfied and generates a test suite. The main goal with this contribution was to use code coverage as a test goal for generating tests based on the transformed timed automata. This allowed us to further investigate other test generation techniques for IEC 61131-3 programs.

Mutation-based Test Generation

In study 5 a mutation testing technique for IEC 61131-3 programs is implemented. This is achieved by using a specialized strategy that monitors the injected fault behavior in each execution and optimizes towards achieving overall better fault detection. We show how this strategy can be used to automatically generate test cases that detect certain injected faults.

The objective of study 5 was to show a method for generating test suites that detect injected faults and as a consequence improve the goals of automatic test generation for IEC 61131-3 software. This approach is based on a network of timed automata that contains all the mutants and the original program. Overall, the

approach is composed of four steps: mutant generation, model aggregation, mutant annotation and test generation.

The CompleteTest Tool

Study 2 details the development of CompleteTest, a tool for automatic test generation based on the concepts initially shown in study 1. Code coverage criteria can be used by CompleteTest to generate test cases. The main goal of the design of the user interface was to meet the needs of an industrial end user. The function of the user interface is to provide a way for the user to select a program, generate tests for a selection of coverage criteria, visualize the generated test inputs, and determine the correctness of the result produced for each generated test by comparing the actual test output with the expected output (as provided manually by the tool user).

The tool is built from the following modules: an **import editor** used for validating the structure of a provided input file, a **translation plugin** that creates an XML-format accepted by the Uppaal model checker, an Uppaal **server plugin** allowing CompleteTest to connect as a client to the model checker and verify properties against the model, and a **trace parser** that collects a diagnostic trace from the model checker and outputs an executable test suite containing inputs, actual outputs and timing information (i.e., the time parameter in the test is used for constraining the inputs in time).

Further, to advance automatic test generation for IEC 61131-3 software, a study was performed (i.e., study 5) where CompleteTest was extended to support mutation testing, a technique for automatically generating faulty implementations of a program for the purpose of improving the fault detection ability of a test suite [16]. We used CompleteTest tool for testing industrial IEC 61131-3 programs and evaluated it in terms of cost and fault detection.

3.4 RG2: Applicability

A series of studies based on industrial use-case scenarios from Bombardier Transportation show the applicability of using automatic test generation in practice. The results indicate that automatic test generation is efficient in terms of time required to generate tests and scales well for industrial IEC 61131-3 software.

In study 1 an IEC 61131-3 program part of the Train Control Management System (TCMS) provided by Bombardier Transportation is used. This program is transformed to timed automata and the cost needed for generating test suites fulfilling a certain code coverage criteria is measured. Overall, the results of study 1 show that the proposed test generation approach is efficient in terms of generation time and memory. We observed that for more complex logic coverage criteria, test cases are larger in size than for branch coverage. Further, we noted that the use of timer elements influences the test generation cost. This is explained by the fact that these timers are varying the timing of the entire program and therefore increasing the number of execution cycles needed to satisfy certain branches. In study 2 an extensive empirical study of CompleteTest is carried out by applying the toolbox to 157 real-world industrial programs developed at Bombardier Transportation. The

results indicate that model checking is suitable for handling code coverage criteria for real-world IEC 61131-3 programs, but also reveal some potential limitations of the tool when used for test generation such as the use of manual expected outputs. The evaluation in study 2 and 5 showed that automatic test generation is efficient in terms of time required to generate tests that satisfy code and mutation coverage and that it scales well for real-world IEC 61131-3 programs.

3.5 RG3: Cost and Fault Detection Evaluation

We compared automatic test generation with manual testing performed by human subjects both in an academic setting and in industry. This goal aims to bring some experimental evidence to the basic understanding of how automatic test generation compares with manual testing. Automatic test generation can achieve similar code coverage as manual testing performed by human subjects but in a fraction of the time. The results of this thesis support the conclusion that automatically generated tests are slightly worse at finding faults in terms of mutation score than manually created test suites.

A Controlled Experiment

Study 3 was performed in an academic setting comparing requirement-based manual testing and implementation-based test generation (i.e., manual test creation and automatic test generation). We manipulated the context and mitigated different factors that could affect the study's results. This came at the expense of studying the phenomenon in an academic context and not in its actual industrial environment. The experiment design started with the formulation of the research objective that was broken down into research questions and hypotheses. As part of the laboratory session, within a software verification & validation course at Mälardalen University, human subjects were given the task of manually creating tests and generating tests with the aid of an automatic test generation tool (a total of twenty-three software engineering master students took part in this experiment). The participants worked individually on manually designing and automatically generating tests for two IEC 61131-3 programs. Further, the efficiency and effectiveness in terms of fault detection is measured. Tests created by the participants in the experiment were collected and analyzed in terms of fault detection score, code coverage, number of tests, and testing duration. When compared to implementation-based testing, requirement-based manual testing yields significantly more effective test suites in terms of the number of faults detected. Specifically, requirement-based test suites more effectively detect comparison and value replacement type of faults, compared to implementation-based tests. On the other hand, code coverage-adequate automatic test generation leads to fewer test suites (up to 85% less test cases) created in shorter time than the ones manually created based on the specification.

Case Studies

An industrial case study (i.e., study 4) comparing manual test suites created by industrial engineers with test suites created automatically was performed. This case study provided the understanding of automatic test generation and manual testing in its actual context and heavily used industrial resources. Unstructured interviews with several industrial engineers were used to explore the actual usage of manual testing. We analyzed test specification documents to give input to the actual empirical work done in this case study, e.g., manual test cases were analyzed and executed on a set of programs provided by Bombardier Transportation. In addition, a planned observation was used to observe how manual testing was performed at the company. This study provided an overall view of the opportunities and limitations of using automatic test generation in industrial practice.

In addition, study 4 provided support for improvement in selecting test goals and fault detection. Practically, study 4 is a case study in which the cost and effectiveness between manually and automatically created test cases were compared. In particular, we measured the cost and effectiveness in terms of fault detection of tests created using a coverage-adequate automatic test generation tool and manually created tests by industrial engineers from an existing train control system. Recently developed real-world industrial programs written in the IEC 61131-3 FBD programming language were used. The results show that automatically generated tests achieve similar code coverage as manually created tests but in a fraction of the time (an improvement of roughly 90% on average). We also found that the use of an automated test generation tool did not show better fault detection in terms of mutation score compared to manual testing. This underscores the need to further study how manual testing is performed in industrial practice. These findings support the hypothesis that manual testing performed by industrial engineers achieve high code coverage and good fault detection in terms of mutation score. This study suggests some issues that would need to be addressed in order to use automatic test generation tools to aid in testing of safety-critical embedded software.

No attempt has been made to evaluate mutation testing for control software written in the IEC 61131-3 programming language. This motivated us to evaluate further mutation-based test generation targeting this type of software in study 5. For realistic validation we collected industrial experimental evidence on how mutation testing compares with manual testing as well as automatic decision-coverage adequate test generation. In the evaluation, manually seeded faults were provided by four industrial engineers. The results show that even if mutation-based test generation achieves better fault detection than automatic decision coverage-based test generation, these mutation-adequate test suites are not better at detecting faults than manual test suites. However, the mutation-based test suites are significantly less costly to create, in terms of testing time, than manually created test suites. The results suggest that the fault detection scores could be improved by considering some new and improved mutation operators for IEC 61131-3 programs as well as higher-order mutations.

4 Related Work

In this section, we identify key pieces of work that are related to the approach presented in this thesis. This section begins with background information and then discusses recent improvements to automatic test generation, like model checking, and how the approach developed in this thesis benefited from these studies. We also identify other techniques that aim at testing IEC 61131-3 in specific, and discuss how they are related to the techniques developed in this thesis. For the work on comparing manual testing with automatic test generation, we discuss how other studies are related to the results of this thesis.

4.1 Test Generation Technique and Tool Implementation

There have been a number of techniques for automatic test generation developed during the past few years [23, 76, 57, 74]. For example Randoop [57] creates random tests by using feedback information as search guidance. EvoSuite [23] is a tool based on genetic algorithm for Java programs. In the following we briefly describe the techniques and tools mostly related to the CompleteTest tool and automatic test generation using model checkers, presented in this thesis.

Test Generation using Model Checking

A model checker has been used to find test cases to various criteria and from programs in a variety of languages [7, 33]. Black et al. [7] discuss the problems of using a model-checker for automatic test generation for full-predicate coverage. Rayadurgam and Heimdahl [63] defined a formal framework that can be used for coverage-based test generation using a model checker. Rayadurgam et al. [62] described a method for obtaining MC/DC adequate test cases using a model-checking approach. Similarly to CompleteTest, the model is annotated and the properties to be checked are expressible as a single sequence. In contrast to these approaches, we provide an approach to generate test cases for different code coverage criteria that are directly applicable to IEC 61131-3 programs. For a detailed overview of testing with model checkers we refer the reader to Fraser et al. [26].

Test Generation for IEC 61131-3 Control Software

Previous contributions in testing of IEC 61131-3 programs range from a simulation-based approach [64], verification of the actual program code [6, 39] and automatic test generation [40, 80, 38, 69, 19]. The technique in [6] is based on Petri Nets. In comparison to the work in this thesis, they do not cope with the internal structure of the PLC logical and timing behavior. In CompleteTest we showed that Uppaal model-checker can be used for automatic test generation based on code and mutation coverage criteria. The idea of using model-checkers for testing IEC 61131-3 programs is not new [68]. The work in [68] uses the Uppaal TRON for verification of IEC 61131-3 programs, however the translated model is used for requirement-based testing. In contrast to the online model-based testing approach used in [68] in this

thesis we generate tests based on code coverage for offline execution. Recently, several automatic test generation approaches [40, 80, 38, 69, 19] have been proposed for IEC 61131-3 software. These techniques can typically produce tests for a given code coverage criterion and have been shown to achieve high code coverage for various IEC 61131-3 programs. Compared to the work in this thesis, these works are lacking the tool support. In addition, CompleteTest augments previous work in this area with mutation testing.

4.2 Empirical Studies

The number of successful applications of automatic test generation in the literature is large and expanding. It is impossible to survey each of them in this thesis. We therefore restrict ourselves to empirical studies closely related to the work of this thesis.

Requirement-based Test Generation

There is a substantial body of work on automatic requirement-based test generation that examines cost, fault detection and coverage in embedded and control systems across a range of safety-critical industrial systems [26, 14, 59]. Heimdahl et al. [31] found that specification coverage criteria proposed in the literature (i.e., state, transition and decision coverage) are inadequate in terms of model fault detection. Compared to this work, in study 3 we are not using formal specifications and structural test goals but instead natural language requirement specifications created by industrial engineers.

Implementation-based Test Generation

A few studies [21, 35] have been concerned with the fault and failure detection capabilities of automatic test generation based on code coverage criteria. These studies compare these code coverage-adequate tests with random tests and show some rather mixed results. None of these studies investigate the relationship between automatically and manually created tests. Namin and Andrews [50] found that code coverage achieved by automatically generated test suites is positively correlated with their fault finding capability. Recently, Inozemtseva and Holmes [37] found rather low code coverage/fault detection correlation when the number of tests was controlled for. They also found that generating tests based on stronger code coverage criteria does not imply stronger fault-finding capability.

Comparison of Implementation-based and Manual Test Generation

There are studies comparing manual testing with automatic implementation-based test generation. Several researchers have evaluated automatic test generation in case studies [77, 46, 66, 13] and used already created manual tests while several others performed studies using controlled experiments [24, 25, 60, 61] with participants manually creating and automatically generating tests.

Recently, Wang et al. [77] compared automatically generated tests with manual tests on several open-source projects. They found that automatically generated tests are able to achieve higher code coverage but lower fault detection scores with manual test suites being also better at discovering hard-to-cover code and hard-to-kill type of faults. Another closely related study done by Kracht el al. [46] used EvoSuite on a number of open-source Java projects and compared those tests with the ones already manually created by developers. Automatically-generated tests achieved similar code coverage and fault detection scores to manually created tests. Recently, Shamshiri et al. [66] found that tests generated by EvoSuite achieved higher code coverage than developer-written tests and detected 40% out of 357 real faults. The results of this thesis indicate that, in IEC 61131-3 software development, automatic test generation can achieve similar branch coverage to manual testing performed by industrial engineers. However, these automatically generated tests do not show better fault detection in terms of mutation score than manually created test suites. The fault detection rate for automated implementation-based test generation and manual testing was found, in some of the published studies [24, 25, 46, 77], to be similar to the results of this thesis. Interestingly enough, our results indicate that code coverage-adequate tests might even be slightly worse in terms of fault detection compared to manual tests. However, a larger empirical study is needed to statistically confirm this hypothesis.

Fraser et al. [24, 25] performed a controlled experiment and a follow-up replication experiment on a total of 97 subjects. They found that automated test generation performs well, achieving high code coverage but no measurable improvement over manual testing in terms of the number of faults found by developers. Ramler et al. [60] conducted a study and a follow-up replication [61], carried out with master's students and industrial professionals respectively, addressing the question of how automatic test generation with Randoop compare to manual testing. In these specific experiments, they found that the number of faults detected by Randoop was similar to manual testing. However, the fault detection rates for automatic implementation-based test generation and manual specification-based testing were found [60] to be significantly different from the experiments included in this thesis. This could stem from the fact that the subjects used Randoop rather than CompleteTest and that in this thesis they were given more time to manually test their programs compared to previous controlled experiments. By using a more restrictive testing duration, one would expect human participants to show less comprehensive understanding of the task at hand.

Mutation-based Test Generation

Most studies concerning automatic test generation for mutation testing and related to the work included in this thesis have focused on how to generate tests as quickly as possible, improve the mutation score and/or compare with code coverage-based automatic test generation [58, 81, 22, 42]. For example, mutation-based test generation [81] is able to outperform code coverage-directed test generation in terms of mutant killing. Frankl et al [22] have shown that mutation testing is superior to several code

coverage criteria in terms of effectiveness. This fact is in line with the results of study 5. Neither of these studies have looked at comparing mutation testing with manual tests created by humans. The study of Fraser et al. [27] is the only one, that we are aware of, comparing mutation-based test generation with manual testing in terms of fault detection by using manually seeded faults. They report that the generated tests based on mutation testing find significantly more seeded faults than manually written tests. In comparison, study 5 shows that mutation-adequate test suites are not better at detecting seeded faults than manual test suites. This partly stems from the fact that some of these undetected faults are not reflected in the mutation operator list used for generating mutation adequate test suites.

5 Conclusions and Future Work

The automatic test generation techniques presented in this thesis are working on IEC industrial control software and are based on model checking for the purpose of covering the implementation. These techniques are implemented in the CompleteTest tool and used throughout this thesis. There are many tools for generating tests, such as KLEE [11], EvoSuite [23], Java PathFinder [76], and Pex [74]. The use of these tools in this thesis is complicated by the transformation of IEC 61131-3 programs directly to Java or C, which was shown to be a significant problem [56] because of the difficulty to translate timing constructs and ensure the real-time nature of these programs. As a concrete future work, we wish to study some of these tools and their application on IEC 61131-3 control software.

This thesis suggests that automatically generated tests are significantly less costly in terms of testing time than manually created tests. The use of CompleteTest in IEC 61131-3 software development can save around 90% of testing time. The results of this thesis suggest that automatic test generation is efficient. This has interesting implications that need to be further studied. As part of this thesis, we used cost measurements to estimate the efficiency of performing automatic test generation. Nevertheless, a more sophisticated cost model that supports both indirect and direct costs affecting the testing process and a real fault detection evaluation needs to be studied in future work.

The results of this thesis showed that automatically generated tests, based on branch coverage, can exercise the logic of the software as well as tests written manually. However, these automatically generated tests do not show better fault detection compared to manually created tests and it seems that manually created tests are able to detect more faults of certain types (i.e, logical replacement, negation insertion and timer replacement) than automatically generated tests. The results of this thesis suggest the improvement of the test goals used by automatic test generation tools. Implementation-based test generation needs to be carefully complemented with other techniques such as mutation testing. This approach was implemented and used to compare automatic test generation based on mutation testing with manual testing. The resulting tests are still not better than manual tests. As a highlight from these results, there is a need for improving the established list of mutation operators used for mutation testing of IEC 61131-3 control software by the addition of several

other operators. This new list of mutation operators needs to be further evaluated in practice.

The results of this thesis support the claim that automatic test generation is efficient but currently not quite as effective as manual testing. This needs to be further studied; we need to consider the implications and the extent to which automatic test generation can be used in the development of reliable safety-critical industrial control software.

To evaluate the potential application of automatic test generation techniques, several studies are presented where CompleteTest is applied to industrial control software from Bombardier Transportation. Even if the results are mainly based on data provided by one company and this can be seen as a weak point, we argue that having access to real industrial data from the safety-critical domain is relevant to the advancement of automatic test generation. More studies are needed to generalize the results of this thesis to other systems and domains.

6 Outline of the Thesis

The rest of this thesis is divided in five parts: Studies 1, 2, 3, 4, and 5. The main research objective of study 1 was to show how code coverage can be measured on IEC 61131-3 software and how, by transforming an IEC 61131-3 program to timed automata, test suites can be automatically generated using a model checker. In study 2 we present the development of a tool for automatic test generation based on the concepts shown in study 1 and a large case study with more elaborate empirical evaluation of the use and applicability of automatic test generation. In study 3, the objective was to compare specification— and implementation-based testing of control software written in the IEC 61131-3 language. The objective of study 4 is to show experimental evidence on how automated test suite generation could be used in industrial practice and how it compares with, what is considered, rigorous manual testing performed by industrial engineers. Finally, in paper 5 we describe and evaluate an automated mutation-based test generation approach for IEC 61131-3 control software.

Bibliography

[1] R. Alur. “Timed Automata”. In: *Computer Aided Verification*. 1999, pp. 688–688 (cit. on p. 11).

[2] R. Alur and D. Dill. “Automata for Modeling Real-time Systems”. In: *Automata, languages and programming*. Springer, 1990, pp. 322–335 (cit. on p. 11).

[3] Paul Ammann and Paul Black. “A Specification-based Coverage Metric to Evaluate Test Sets”. In: *International Journal of Reliability, Quality and Safety Engineering*. Vol. 8. 04. World Scientific, 2001, pp. 275–299 (cit. on p. 7).

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008 (cit. on pp. 3, 7, 8).

[5] Paul Ammann and Jeff Offutt. “Using Formal Methods to Derive Test Frames in Category-partition Testing”. In: *Conference on Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security*. 1994, pp. 69–79 (cit. on p. 7).

[6] L. Baresi, M. Mauri, A. Monti, and M. Pezze. “PLCTools: Design, Formal Validation, and Code Generation for Programmable Controllers”. In: *International Conference on Systems, Man, and Cybernetics*. Vol. 4. 2000, pp. 2437–2442 (cit. on p. 16).

[7] Paul Black. “Modeling and Marshaling: Making Tests from Model Checker Counter-Examples”. In: *Digital Avionics Systems Conference*. Vol. 1. IEEE, 2000, 1B3–1 (cit. on p. 16).

[8] William Bolton. *Programmable Logic Controllers*. Newnes, 2015 (cit. on p. 4).

[9] Sébastien Bornot, Ralf Huuck, and Ben Lukoschus. “Sequential Function Charts”. In: *Discrete Event Systems: Analysis and Control*. Vol. 569. Springer, 2012, p. 255 (cit. on p. 11).

[10] Stuart Boyer. *SCADA: Supervisory Control and Data Acquisition*. International Society of Automation, 2009 (cit. on p. 4).

[11] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Symposium on Operating Systems Design and Implementation*. Vol. 8. USENIX, 2008, pp. 209–224 (cit. on pp. 8, 19).

[12] CENELEC. “50128: Railway Application–Communications, Signaling and Processing Systems–Software for Railway Control and Protection Systems”. In: *Standard Report*. 2001 (cit. on pp. 7, 10).

[13] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. “Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports”. In: *International Symposium on Software Reliability Engineering (ISSRE)*. 2008, pp. 157–166 (cit. on p. 17).

[14] SJ Cunning and Jerzy W Rozenblit. “Automatic Test Case Generation from Requirements Specifications for Real-Time Embedded Systems”. In: *International Conference on Systems, Man, and Cybernetics*. Vol. 5. 1999, pp. 784–789 (cit. on p. 17).

[15] Raffaello D’Andrea and Geir E Dullerud. “Distributed Control Design for Spatially Interconnected Systems”. In: *Transactions on Automatic Control*. Vol. 48. 9. IEEE, 2003, pp. 1478–1495 (cit. on p. 4).

[16] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”. In: *Computer*. Vol. 11. 4. IEEE, 1978, pp. 34–41 (cit. on pp. 8, 13).

[17] Jeremy Dick and Alain Faivre. “Automating the Generation and Sequencing of Test Cases from Model-based Specifications”. In: *International Symposium of Formal Methods Europe*. 1993, pp. 268–284 (cit. on p. 6).

[18] Henning Dierks. “PLC-Automata: A New Class of Implementable Real-Time Automata”. In: *Theoretical Computer Science*. Vol. 253. 1. Elsevier, 2001, pp. 61–93 (cit. on p. 11).

[19] Kivanc Doganay, Markus Bohlin, and Ola Sellin. “Search Based Testing of Embedded Systems Implemented in IEC 61131-3: An Industrial Case Study”. In: *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 425–432 (cit. on pp. 16, 17).

[20] Sigrid Eldh. “On Test Design”. In: *PhD Thesis, Mälardalen University Press*. Mälardalen University Press, 2011 (cit. on p. 3).

[21] Phyllis G Frankl and Stewart N Weiss. “An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing”. In: *Transactions on Software Engineering*. Vol. 19. 8. IEEE, 1993, pp. 774–787 (cit. on p. 17).

[22] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. “All-uses vs Mutation Testing: an Experimental Comparison of Effectiveness”. In: *Journal of Systems and Software*. Vol. 38. 3. Elsevier, 1997, pp. 235–253 (cit. on p. 18).

[23] Gordon Fraser and Andrea Arcuri. “Evosuite: Automatic Test Suite Generation for Object-oriented Software”. In: *Conference on Foundations of Software Engineering*. ACM, 2011, pp. 416–419 (cit. on pp. 8, 16, 19).

[24] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. “Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study”. In: *Transactions on Software Engineering and Methodology*. Vol. 24. 4. ACM, 2014, p. 23 (cit. on pp. 17, 18).

[25] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. "Does Automated White-Box Test Generation Really Help Software Testers?" In: *International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 291–301 (cit. on pp. 17, 18).

[26] Gordon Fraser, Franz Wotawa, and Paul E Ammann. "Testing with Model Checkers: a Survey". In: *Journal on Software Testing, Verification and Reliability*. Vol. 19. 3. Wiley, 2009, pp. 215–261 (cit. on pp. 16, 17).

[27] Gordon Fraser and Andreas Zeller. "Mutation-driven generation of unit tests and oracles". In: vol. 38. 2. IEEE, 2012, pp. 278–292 (cit. on p. 19).

[28] Gregory Gay, Matt Staats, Michael Whalen, and Mats Heimdahl. "The Risks of Coverage-Directed Test Case Generation". In: *Transactions on Software Engineering*. Vol. 41. 8. IEEE, 2015, pp. 803–819 (cit. on pp. 8, 11).

[29] David Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming*. Vol. 8. 3. Elsevier, 1987, pp. 231–274 (cit. on p. 11).

[30] Mats Heimdahl. "Safety and Software Intensive Systems: Challenges Old and New". In: *Future of Software Engineering*. 2007, pp. 137–152 (cit. on p. 9).

[31] Mats Heimdahl, Devaraj George, and Robert Weber. "Specification Test Coverage Adequacy Criteria= Specification Test Generation Inadequacy Criteria". In: *International Symposium on High Assurance Systems Engineering*. 2004, pp. 178–186 (cit. on p. 17).

[32] Teruo Higashino, Akio Nakata, Kenichi Taniguchi, and Ana R Cavalli. "Generating Test Cases for a Timed I/O Automaton Model". In: *Testing of Communicating Systems*. Springer, 1999, pp. 197–214 (cit. on p. 7).

[33] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. "A Temporal Logic-Based Theory of Test Coverage and Generation". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2002, pp. 327–341 (cit. on p. 16).

[34] William E Howden. "Weak mutation testing and completeness of test sets". In: *Transactions on Software Engineering*. 4. IEEE, 1982, pp. 371–379 (cit. on p. 8).

[35] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. "Experiments of the Effectiveness of Dataflow-and Controlflow-based Test Adequacy Criteria". In: *International Conference on Software Engineering*. 1994, pp. 191–200 (cit. on p. 17).

[36] IEC. "International Standard on 61131-3 Programming Languages". In: *Programmable Controllers*. IEC Library, 2014 (cit. on p. 5).

[37] Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness". In: *International Conference on Software Engineering*. ACM, 2014, pp. 435–445 (cit. on pp. 8, 17).

[38] Marcin Jamro. "POU-Oriented Unit Testing of IEC 61131-3 Control Software". In: *Transactions on Industrial Informatics,* vol. 11. 5. IEEE, 2015 (cit. on pp. 16, 17).

[39] E. Jee, S. Kim, S. Cha, and I. Lee. "Automated Test Coverage Measurement for Reactor Protection System Software Implemented in Function Block Diagram". In: *Journal on Computer Safety, Reliability, and Security.* Springer, 2010, pp. 223–236 (cit. on p. 16).

[40] Eunkyoung Jee, Donghwan Shin, Sungdeok Cha, Jang-Soo Lee, and Doo-Hwan Bae. "Automated Test Case Generation for FBD Programs Implementing Reactor Protection System Software". In: *Software Testing, Verification and Reliability.* Vol. 24. 8. Wiley, 2014 (cit. on pp. 16, 17).

[41] Yue Jia and Mark Harman. "Higher Order Mutation Testing". In: *Information and Software Technology.* Vol. 51. 10. Elsevier, 2009, pp. 1379–1393 (cit. on p. 8).

[42] Bryan F Jones, David E Eyres, and H-H Sthamer. "A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing". In: *The Computer Journal.* Vol. 41. 2. Brazilian Computer Society, 1998, pp. 98–107 (cit. on p. 18).

[43] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In: *International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 654–665 (cit. on p. 8).

[44] Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. "Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing". In: *International Conference on Automated Software Engineering.* IEEE, 2013, pp. 519–528 (cit. on p. 8).

[45] Eric D Knapp and Joel Thomas Langill. *Industrial Network Security: Securing critical infrastructure networks for smart grid, SCADA, and other Industrial Control Systems.* Syngress, 2014 (cit. on p. 4).

[46] Jeshua S Kracht, Jacob Z Petrovic, and Kristen R Walcott-Justice. "Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites". In: *International Conference on Quality Software.* IEEE, 2014, pp. 256–265 (cit. on pp. 17, 18).

[47] Moez Krichen and Stavros Tripakis. "Black-box Conformance Testing for Real-time Systems". In: *International SPIN Workshop on Model Checking of Software.* 2004, pp. 109–126 (cit. on p. 7).

[48] K.G. Larsen, P. Pettersson, and W. Yi. "UPPAAL in a Nutshell". In: *International Journal on Software Tools for Technology Transfer (STTT).* Vol. 1. 1. Springer, 1997, pp. 134–152 (cit. on p. 12).

[49] Tadao Murata. "Petri Nets: Properties, Analysis and Applications". In: *Proceedings of the IEEE.* Vol. 77. 4. IEEE, 1989, pp. 541–580 (cit. on p. 11).

[50] Akbar Siami Namin and James H Andrews. “The Influence of Size and Coverage on Test Suite Effectiveness”. In: *International Symposium on Software Testing and Analysis*. 2009, pp. 57–68 (cit. on p. 17).

[51] Jeff Offutt and Aynur Abdurazik. “Generating Tests from UML Specifications”. In: *International Conference on the Unified Modeling Language*. 1999, pp. 416–429 (cit. on p. 6).

[52] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. “Criteria for Generating Specification-based Tests”. In: *International Conference on Engineering of Complex Computer Systems*. 1999, pp. 119–129 (cit. on p. 7).

[53] M. Öhman, S. Johansson, and K.E. Årzén. “Implementation Aspects of the PLC standard IEC 1131-3”. In: *Journal on Control Engineering Practice*. Vol. 6. 4. Elsevier, 1998, pp. 547–555 (cit. on p. 5).

[54] Alessandro Orso and Gregg Rothermel. “Software Testing: a Research Travelogue (2000–2014)”. In: *Proceedings of the International conference on Software Engineering (ICSE), Future of Software Engineering* (2014), pp. 117–132 (cit. on pp. 3, 7, 9).

[55] Thomas J. Ostrand and Marc J. Balcer. “The Category-partition Method for Specifying and Generating Functional Tests”. In: *Communications of the ACM*. Vol. 31. 6. ACM, 1988, pp. 676–686 (cit. on p. 7).

[56] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. “An Overview of Model Checking Practices on Verification of PLC Software”. In: *Software & Systems Modeling*. Springer, 2014, pp. 1–24 (cit. on p. 19).

[57] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. “Feedback-Directed Random Test Generation”. In: *International Conference on Software Engineering*. IEEE, 2007, pp. 75–84 (cit. on p. 16).

[58] Mike Papadakis and Nicos Malevris. “Automatic Mutation Test Case Generation via Dynamic Symbolic Execution”. In: *International Symposium on Software Reliability Engineering*. 2010, pp. 121–130 (cit. on p. 18).

[59] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. “One Evaluation of Model-based Testing and its Automation”. In: *International Conference on Software Engineering*. 2005, pp. 392–401 (cit. on p. 17).

[60] Rudolf Ramler, Dietmar Winkler, and Martina Schmidt. “Random Test Case Generation and Manual Unit Testing: Substitute or Complement in Retrofitting Tests for Legacy Code?” In: *Euromicro Conference on Software Engineering and Advanced Application*. 2012, pp. 286–293 (cit. on pp. 17, 18).

[61] Rudolf Ramler, Klaus Wolfmaier, and Theodorich Kopetzky. “A Replicated Study on Random Test Case Generation and Manual Unit Testing: How Many Bugs Do Professional Developers Find?” In: *Computer Software and Applications Conference*. IEEE, 2013, pp. 484–491 (cit. on pp. 17, 18).

[62] S Rayadurgam and MPE Heimdahl. “Generating MC/DC Adequate Test Sequences Through Model Checking”. In: *NASA Goddard Software Engineering Workshop Proceedings*. 2003, pp. 91–96 (cit. on p. 16).

[63] Sanjai Rayadurgam and Mats PE Heimdahl. “Coverage Based Test-Case Generation using Model Checkers”. In: *International Conference and Workshop on the Engineering of Computer Based Systems*. 2001, pp. 83–91 (cit. on p. 16).

[64] S. Richter and J.U. Wittig. “Verification and Validation Process for Safety IC Systems”. In: *Nuclear Plant Journal*. Vol. 21. 3. EQES, Inc., 2003, pp. 36–36 (cit. on p. 16).

[65] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004 (cit. on p. 11).

[66] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. “Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges”. In: *International Conference on Automated Software Engineering*. ACM, 2015, pp. 201–211 (cit. on pp. 17, 18).

[67] Ernst Sikora, Bastian Tenbergen, and Klaus Pohl. “Industry Needs and Research Directions in Requirements Engineering for Embedded Systems”. In: *Requirements Engineering*. Vol. 17. 1. Springer, 2012, pp. 57–78 (cit. on p. 6).

[68] L.D. da Silva, L.P. de Assis Barbosa, K. Gorgônio, A. Perkusich, and A.M.N. Lima. “On the Automatic Generation of Timed Automata Models from Function Block Diagrams for Safety Instrumented Systems”. In: *Industrial Electronics*. 2008, pp. 291–296 (cit. on p. 16).

[69] Hendrik Simon, Nico Friedrich, Sebastian Biallas, Stefan Hauck-Stattelmann, Bastian Schlich, and Stefan Kowalewski. “Automatic Test Case Generation for PLC Programs Using Coverage Metrics”. In: *Emerging Technologies and Factory Automation*. IEEE, 2015, pp. 1–4 (cit. on pp. 16, 17).

[70] Ramavarapu Sreenivas and Bruce Krogh. “On Condition/Event Systems with Discrete State Realizations”. In: *Discrete Event Dynamic Systems*. Vol. 1. 2. Springer, 1991, pp. 209–236 (cit. on p. 11).

[71] Phil Stocks and David Carrington. “A Framework for Specification-based Testing”. In: *Transactions on software Engineering*. Vol. 22. 11. IEEE, 1996, pp. 777–793 (cit. on p. 6).

[72] Keith Stouffer, Joe Falco, and Karen Scarfone. “Guide to Industrial Control Systems (ICS) Security”. In: *Special Report*. Vol. 800. 82. NIST, 2011, pp. 16–16 (cit. on p. 4).

[73] J. Thieme and H.M. Hanisch. “Model-based Generation of Modular PLC Code using IEC61131 Function Blocks”. In: *Proceedings of the International Symposium on Industrial Electronics*. Vol. 1. 2002, pp. 199–204 (cit. on p. 5).

[74] Nikolai Tillmann and Jonathan De Halleux. “Pex–White Box Test Generation for. net”. In: *Tests and Proofs*. Springer, 2008, pp. 134–153 (cit. on pp. 8, 16, 19).

[75] Jan Tretmans. “Model Based Testing with Labelled Transition Systems”. In: *Formal Methods and Testing*. Springer, 2008, pp. 1–38 (cit. on p. 7).

[76] Willem Visser, Corina S Pasareanu, and Sarfraz Khurshid. “Test Input Generation with Java PathFinder”. In: *SIGSOFT Software Engineering Notes*. Vol. 29. 4. ACM, 2004, pp. 97–107 (cit. on pp. 16, 19).

[77] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. “Experience Report: How is Dynamic Symbolic Execution Different from Manual Testing? A Study on KLEE”. In: *International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 199–210 (cit. on pp. 17, 18).

[78] Michael Whalen, Ajitha Rajan, Mats Heimdahl, and Steven Miller. “Coverage Metrics for Requirements-based Testing”. In: *International Symposium on Software Testing and Analysis*. 2006, pp. 25–36 (cit. on p. 7).

[79] MR Woodward and K Halewood. “From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues”. In: *Workshop on Software Testing, Verification, and Analysis*. 1988, pp. 152–158 (cit. on p. 8).

[80] Yi-Chen Wu and Chin-Feng Fan. “Automatic Test Case Generation for Structural Testing of Function Block Diagrams”. In: *Information and Software Technology*. Vol. 56. 10. Elsevier, 2014 (cit. on pp. 16, 17).

[81] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. “Test Generation via Dynamic Symbolic Execution for Mutation Testing”. In: *International Conference on Software Maintenance (ICSM)*. 2010, pp. 1–10 (cit. on p. 18).

[82] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. “Combined Static and Dynamic Automated Test Generation”. In: *International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 353–363 (cit. on p. 8).