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

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

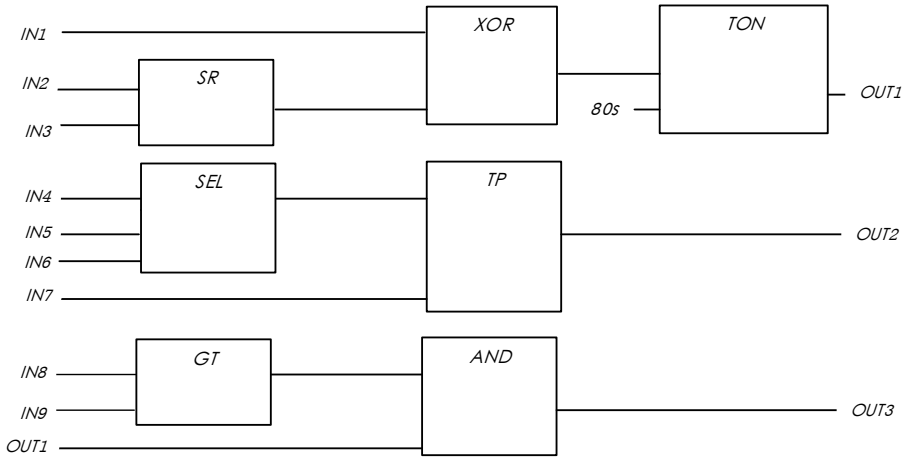


Figure 1: Example of a IEC 61131-3 Program.

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

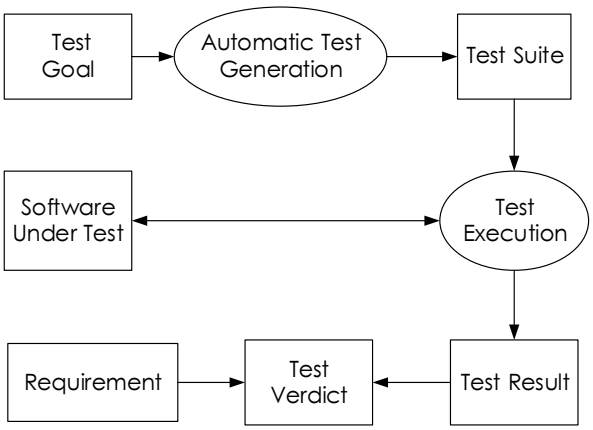


Figure 2: A typical automatic test generation scenario.

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.

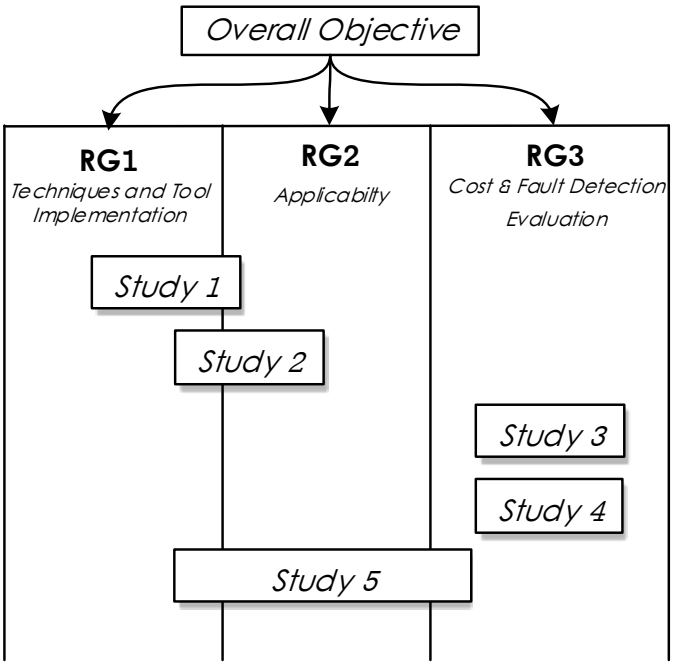


Figure 3: Overview of the how the studies included in this thesis support the three research goals.

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

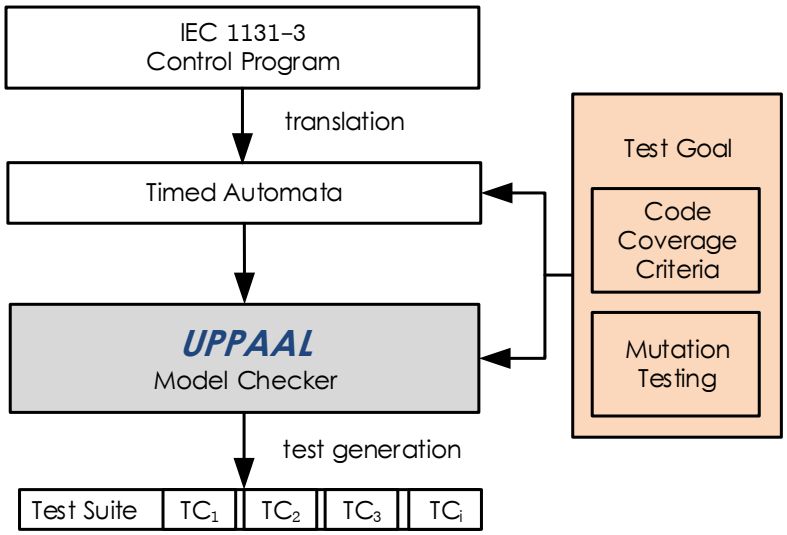


Figure 4: Overview of the implemented automatic test generation techniques.

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).

Part II

Studies

Study 1

Using Logic Coverage to Improve Testing Function Block Diagrams

Eduard Paul 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 1. Using Logic Coverage to Improve Testing Function Block Diagrams

Eduard Paul Enoiu, Daniel Sundmark, Paul Pettersson

Abstract

In model-driven development, testers are often focusing on functional model-level testing, enabling verification of design models against their specifications. In addition, in safety-critical software development, testers are required to show that tests cover the structure of the implementation. Testing cost and time savings could be achieved if the process of deriving test cases for logic coverage is automated and provided test cases are ready to be executed. The logic coverage artifacts, i.e., predicates and clauses, are required for different logic coverage, e.g., MC/DC. One way of dealing with test case generation for ensuring logic coverage is to approach it as a model-checking problem, such that model-checking tools automatically create test cases. We show how logic coverage criteria can be formalized and used by a model-checker to provide test cases for ensuring coverage on safety-critical software described in the Function Block Diagram programming language. Based on our experiments, this approach, supported by a tool chain, is an applicable and useful way of generating test cases for covering Function Block Diagrams.

1 Introduction

Within the last decade model-checking has turned out to be a useful technique for generation of test cases from finite-state models [10]. However, the main problem in using model-checking for testing industrial software systems is the potential combinatorial explosion of the state space and its limited application to models used in practice. Safety-critical and real-time software systems implemented in Programmable Logic Controllers (PLCs) are used in many real-world industrial application domains. One of the programming languages defined by the *International Electrotechnical Commission* (IEC) for PLCs is the *Function Block Diagram* (FBD). Programs developed in FBD are transformed into program code, which is compiled into machine code automatically by using specific engineering tools provided by PLC vendors. The motivation for using FBD as an implementation model comes from the fact that this language is the standard in many industrial software systems, such as rail transport control.

In this paper, our goal is to help testers automatically develop test cases for safety-critical software systems modeled in FBD that require a certain level of certification.

One example of certification includes logic coverage which needs to be demonstrated on the developed programs. There has been little research on using logic coverage criteria for FBD programs in an industrial setting. One way is that logic coverage is analyzed at the code level [6] while tests are designed at the FBD program level, so time-consuming iterations between levels are required. Even if at the code level, logic coverage is used, it would be difficult to standardize the code generation scheme for different PLC tool vendors in order to map directly the criteria to the original FBD program. Hence, in this model-driven environment it is advantageous to move as much testing activity from code level to FBD program level as possible.

As the first contribution of this paper, we present a framework suitable for transforming FBD programs to a formal representation of both its functional and timing behavior. For this, we implement an automatic model–to–model transformation to timed automata, a well known model introduced by Alur and Dill [2]. The choice of timed automata as the target language is motivated primarily by its formal semantics and tool support for simulation and model-checking. Our goal is not to solve all testing issues (e.g., robustness, schedulability, etc.), but to allow the usage of a framework for formal reasoning about logic coverage on FBD programs. The transformation accurately reflects the data-flow characteristics of the FBD 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 formal behavior for every standard component in the FBD program. These steps are independent of timed automata thus are generic in the sense that they could also be used when translating an FBD program to another target language.

As the second contribution, we develop a test case generation technique based on model-checking, tailored for logic coverage of FBD programs. There have been a number of testing techniques used for defining logic coverage using model-checkers, e.g., [5, 18, 19]. However, these techniques are not directly applicable to FBD programs and semantics. We define logic coverage for FBD programs based on the transformed timed automata model. This copes with both functional and timing behavior of an FBD program. This formal definition is necessary for the approach to be applicable to model-checking. We present how a model-checker can be used to generate test cases for covering an FBD program. Based on our experiments, this method is — for the real world models provided by Bombardier Transportation AB — a useful way of generating test cases for logic coverage both in terms of automation and robustness to changes in the FBD programs as monitored by the model-checker.

The paper is organized as follows. Section 2 briefly overviews PLC software, the IEC 61131-3 standard, timed automata and logic coverage. Section 3 describes our overall testing methodology roadmap. Section 4 introduces the modeling approach for FBD programs and Section 5 shows the transformation scheme into timed automata. Section 6 and Section 7 presents the test case generation method required for logic coverage criteria. Next, we apply our method on a Train Startup Mode example in Section 8. In Section 9 we compare to related work, before concluding in Section 10.

2 Preliminaries

This paper describes how to generate test cases that cover the logical structure of FBD programs, by transforming them to networks of timed automata. In this section, we provide some background details on FBD programs, timed automata and logical coverage.

2.1 FBD Programs and Timer Components

PLCs are widely used in control software from nuclear plants to train systems. A PLC is an integrated embedded system that contains a processor, a memory, and a communication bus. Programs execute in a loop, in which the computation follows the *"read-execute-write"* semantics. In this way a PLC reads all inputs, executes the computation without interruption, and then writes to its output. FBD, a PLC programming language standardized by IEC 61131-3, is very popular in the industrial practice because of its graphical notations and its data flow nature [17]. Components in an FBD program are the base for a structured and hierarchical application. They are supplied by the manufacturer, defined by the user, or predefined in a library. An application generator is utilized to automatically transform each component to a C compliant program with its own thread of execution.

The type of systems we are studying contain a particular type of components named *PLC timers*. These timers are output instructions that provide the same functions as timing relays and are used to activate or deactivate a device after a preset interval of time. There are two different timer components (i) On-delay Timer (TON) and (ii) Off-delay Timer (TOF). Basically, a timer counts time-based intervals when the input instruction is true or false. In practice many other time configurations can be derived from this basic timers. In order to study how to generate test cases using a model checker for these types of FBD programs we use a formal representation that can cope with timers and timing information.

2.2 Networks of Timed Automata

A timed automaton is a standard finite-state automaton extended with a finite collection of real-valued clocks. The model was introduced by Alur and Dill [2] and has gained in popularity as a suitable model for real-time systems. We give here a brief summary for readers unfamiliar with timed automata theory.

Let C be a finite set of real-valued clocks and $B(C)$ the set of clock constraints, which are finite conjunctions of atomic guards of the form $x \bowtie n$, where $x \in C$, n is a natural number, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

A *timed automaton* (A) over actions $\mathcal{A}$, atomic propositions P and clocks C is a tuple $\langle N, l_0, E, I, V \rangle$ where N is a finite set of control locations, l_0 is the initial location, $E \subseteq N \times B(C) \times \mathcal{A} \times R^1 \times N$ is the set of edges. In the case of and edge $\langle l, g, a, r, l' \rangle \in E$, we write $l \xrightarrow{g,a,r} l'$ where the label g is a guard of the edge, r is

[1] R denotes the reset set i.e., assignments to manipulate clock- and data variables.

the data- or clock reset assignments of the edge, and a is the action of the edge. $I : N \rightarrow B(C)$ is a function which for each control location assigns an invariant condition and $V : N \rightarrow 2^P$ is a function which for each control location gives a set of atomic propositions true in the location.

The semantics of A is defined in terms of a state transition system, where the state of A is defined as a pair (l, u), where l is a location and $u \in \mathbb{R}^C$ is a clock assignment in C. A state of A depends on its current location and on the current values of its clocks.

We denote by $T(A)$ all traces σ of A starting from the initial state (l_0, u_0) as a sequence of alternating transitions $\sigma = (l_0, u_0) \xrightarrow{a_1} (l_1, u_1) \xrightarrow{a_2} ... \xrightarrow{a_n} (l_n, u_n)$.

A network of timed automata $B_0 \parallel ... \parallel B_{n-1}$ is a parallel composition of n timed automata over C, $\mathcal{A}$ and synchronization functions (i.e., $a!$ is correlative with $a?$). We refer the reader to [1] for more information on the theory of timed automata.

We consider in this paper model-checking algorithms that perform reachability analysis to check for properties of the form $\exists \Diamond \beta$, with respect to a property β of the locations and the values of the clock. $\exists$ is the existential quantifier, and $\Diamond$ is the temporal operator. A reachability property states that there is a path in which β in A is reached. This type of property serves as a basis for formulating various coverage criteria and for deriving properties that could be used by a model-checker to produce test sequences for the timed automaton A.

2.3 Logic-based Coverage Criteria

In this section we briefly describe existing logic-based coverage criteria. In the literature, there are many similar criteria defined, but with different terminology [3]. Also, some definitions of coverage criteria (e.g., MC/DC) have some ambiguities. In order to eliminate the ambiguities and conflicting terminologies, Ammann et al. [4] abstracted logic criteria with a precise definition and formal representation. A *predicate* is an expression that evaluates to a Boolean value. It consists of one or more clauses. A *clause* is a predicate that does not contain any logical operators and can be a Boolean variable, non-Boolean variables used for comparison, or a call to a Boolean function.

Clauses and predicates are used to introduce a variety of coverage criteria. This paper presents three different test criteria, each of which requires a different amount of test cases: (1) *Predicate Coverage (PC)*, (2) *Clause Coverage (CC)*, and (3) *Correlated Active Clause Coverage (CACC)*. These are defined in the next sections in terms of the FBD program. We note that modified condition/decision coverage (MC/DC) is equivalent to CACC and relies on its original definition [4].

3 Testing Methodology and Proposed Solutions

In this section, we describe our approach to automate test-case generation for FBD programs. Logic coverage criteria are used to define what test cases are needed and we use a model-checker to generate test traces. In addition, the formal framework

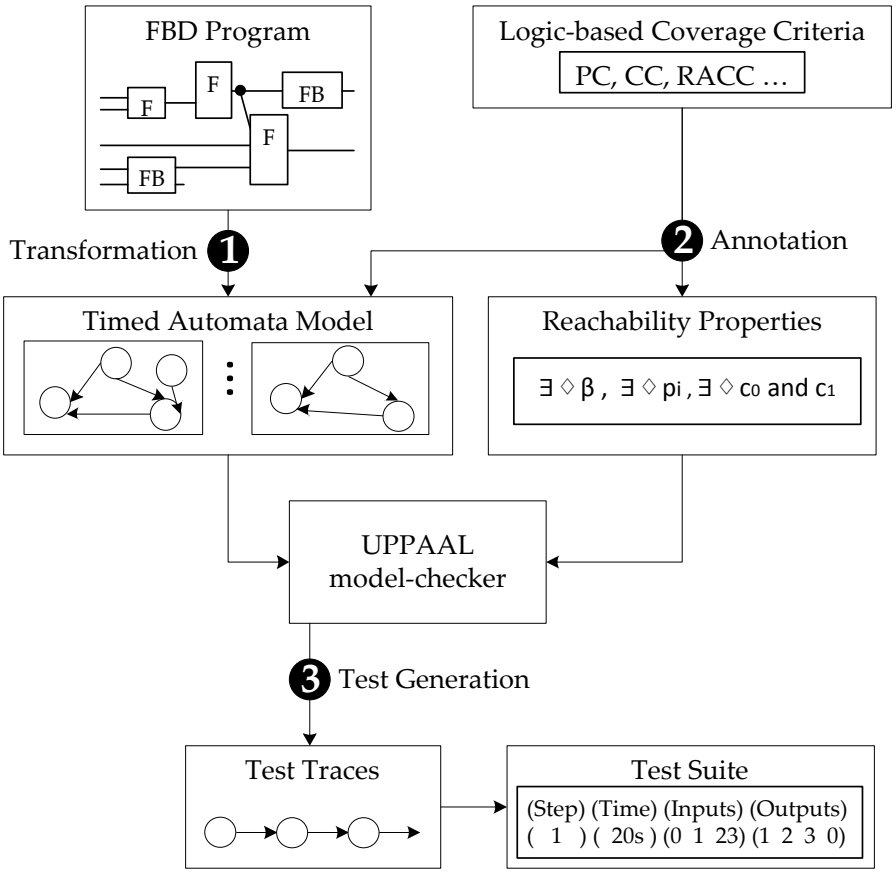


Figure 1.1: Testing Methodology Roadmap

presented in this paper is tailored for FBD programs, and is composed of the following steps, mirrored in Figure 1.1:

1. *Model Transformation.* To test an FBD program we map it to a finite state system suitable for model checking. In order to cope with timing constraints we have chosen to map FBD programs to timed automata.

2. *Logic Coverage Annotation.* We annotate the transformed model such that one can formulate a condition describing a single test case. This is a property expressible as a reachability property used in most model checkers.

3. *Test Case Generation.* We now use the model-checker to generate test traces. To provide a good level of practicality to our work we use a specific model-checker called UPPAAL which is using timed automata as the input modeling language [2]. The verification language supports reachability properties. In order to generate test cases for logic coverage of FBD programs using UPPAAL, we make use of UPPAAL's ability to generate test traces witnessing a submitted reachability property [11]. Currently UPPAAL supports three options for *diagnostic trace generation*: some trace leading to a goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest time delay.

[2]The UPPAAL tool is available at www.uppaal.org.

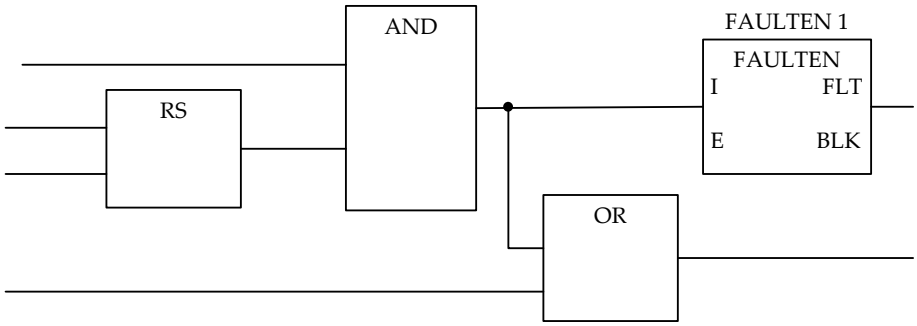


Figure 1.2: An FBD program showing the graphical nature of the language.

While UPPAAL is a viable tool for model checking, it is not tailored to test case generation in practice. We demonstrate how to work around this by automatically generating traces for logic coverage of FBD programs described in timed automata and how we transform these traces to actual test cases. We discuss these steps in further detail in the following sections. First we start by introducing the FBD programs as a finite syntactical representation to describe its component model nature.

4 Function Block Diagram Component Model

An FBD program is a component model which obeys the read-execute-write semantics with a mechanism for monitoring the internal components to determine when the implementation has terminated. Components can be categorized into functions (`FUNC`) and function blocks (`FB`). A `FUNC` does not have any internal state and its output is determined only by the current inputs. *An example of an FBD program depicting a Loadshed Contactor Control is shown in Figure 1.2. Basically the components are equivalent to predicates and instrumentation points shown in a circuit diagram fashion. The system consists of basic functions (e.g., AND, OR) and function blocks (e.g., FAULTEN, RS). In Figure 1.2, AND is a* `FUNC`*. In contrast, FAULTEN is an* `FB` *because it maintains an internal state and produces outputs based on this state and inputs.*

Assume an FBD program defined as the following tuple:

$$FBDProgram \triangleq \langle Name, FE, V, P, Con \rangle,$$

where $Name$ is the program identifier, FE is the set of components defined as the union of `FUNC` and `FB` instances, V is the variable set, defined as the union of input (VI) and output (VO) variables, P is the parameter set, defined as the parameters used internally by the program, and Con is the set of connectors between all components (e.g., `FB` and `FUNC`).

A component in `FE` has an interface, consisting of a name identifier, input and output ports, and a list of parameters. The interface is used to access the component behavior. When the component is activated the behavior is started using the values read on the input ports. When the behavior ends, i.e., when the component

implementation terminates its execution, the output ports are updated. The behavior of a component is typically implemented by a code fragment that updates local variables. We define a component as a tuple $\langle Name, Port, B\rangle$,where $Name$ is the name identifier, `Port` is the set ports, defined as the union of input (IP), output ports (OP), and a list of parameters, whereas B is the behavior description of a component.

Recall that in order to express timing constraints within one component, standard PLC timers are used. The timers in a PLC are operated by an internally generated clock that originates in the processor module. Consider the following PLC timer `TON` defined as a tuple $\langle TON1, (IN, PT, ET, Q), B_t\rangle$, where $TON1$ is the name identifier, IN, PT, ET, and Q are the set of ports and parameters in `Port`, and B_t is the behavior description. This timer component is an attempt to specify its interface and behavior. From a semantic point of view, FBD programs are a special case of deterministic reactive systems. We use more informative notations to denote the actual behavior. In the following section we present several such notations to describe how FBD programs can be handled by the Uppaal model checker.

5 Transforming Function Block Diagrams into Timed Automata

In this section, we introduce the rules that describe the way we transform FBD programs into a network of timed automata, being one step away from test suite generation with the Uppaal tool. Note that the current transformation rules cover one-level hierarchy only. The transformation maps to timed automata all the interface elements FE, V, P, and Con alongside the existing timing annotations within the FBD program. These timing annotations are based on the specifications used from structure and behavioral elements as defined in the FBD language. The transformation process starts by creating a timed automaton for the program description. We place templates of components and list the composed timed automata network representing the FBD program as $FE_1 \parallel ... \parallel FE_n$.

We consider the target model as a network of timed automata named Timed Automata Component Model (TACM) and defined as a tuple as follows:

$$TACM \triangleq \langle Comp, P_{in}, P_{out}, Connections, B_{TACM}\rangle,$$

where $Comp$ is the set of components that TACM contains, P_{in} and P_{out} are the input and output dataflow ports, respectively, and B_{TACM} is the $TACM$'s behavior. If $Comp = \emptyset$ and $Connections = \emptyset$, then $TACM$ is a primitive component.

The mapping is a function $\pi : FBDProgram \rightarrow TACM$, which maps each component to a $TACM$ primitive component, input variables VI to the $TACM$'s component dataflow input ports, output variables VO to the $TACM$'s component dataflow output ports, connectors to the $TACM$'s component connections, and the behavioral specification of a component to B_{TACM}. The execution of a component is modeled as a timed automaton. The following rules establish in more details B_{TACM} with regard to the mapping of an $FBDProgram$ to $TACM$.

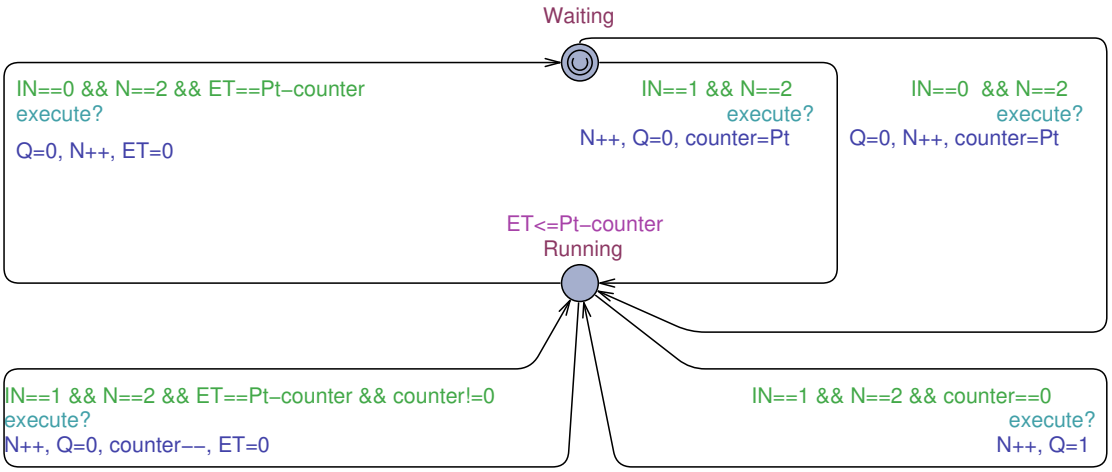


Figure 1.3: Timed Automaton of a TON component.

An FBD program is executed in a loop and the computation follows the run-to completion semantics. The timed automaton of the FBD program contains a clock variable for modeling a delay between the cycles. A cycle starts when the automaton enters the `ReadInputs` node and ends in `UpdateScanTime` node. For a composition $TACM$ the execute operation of each component is extended according to connections and P_{in} and P_{out} variables. A composition is a set of interconnected components closed under a specific execution order. The execution order is automatically defined according to the general rules included in the IEC standard. We use the notion of precedence to describe such dependencies on the convention of reading such FBD programs in a top-to-bottom, left-to-right fashion. For each component we assign a precedence priority to the corresponding timed automaton. A counter is created in this step to represent the execution priority of a component. In this way we ensure that components are executed one by one. After the last component is evaluated, the counter is reset to repeat the scan cycle.

For standard components we assign a timed automaton B_{TACM} with its own logical execution and no internal concurrency. A component is initially *Waiting*, and after performing the read action it starts executing until its internal computation is done. Reconsider the PLC timer `TON` as described in Section 4. A rather straightforward model of the `TON` component is shown as a timed automaton in Figure 1.3. The composition interacts with the TON component via `execute?` action. `TON` is modeled by a standard time on timer that sets the output Q to true if IN variable is true at least as long as the time PT. In this way, we comply with the standard specification of a PLC timer and the structural definition of the program. The timed automaton encapsulates the internal behavior with both functional and timing properties. This means that when we create a TON model we use a separate instantiation of the behavioral model. Also, every instance of TON needs to contain all the variables listed in the interface description and for this reason it is necessary to give each instance of the TON behavioral model a unique identifier.

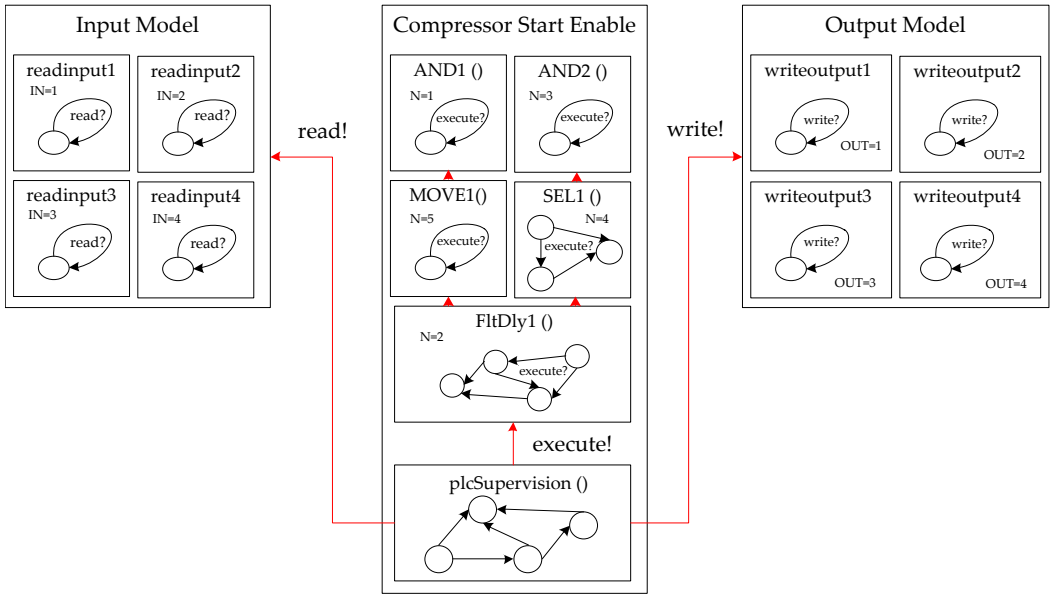


Figure 1.4: Test TA Network for a FBD Program.

6 Test Case Generation using the UPPAAL Model-Checker

As a result of the transformation described in Section 5, we consider that the FBD program is given as a closed network of timed automata as shown in Figure 1.4. This model contains two sub-networks, one modeling the `FBD Program` and the other one modeling its `Environment`. In addition, we consider a completely unconstrained environment that allows all possible interactions between the timed automata network elements. In this way the cycle scan is used to control the FBD program via `read?`, `execute?`, and `write!` actions.

Let us assume the generic timed automata network of an FBD program together with its PLC cycle scan and environment shown in Figure 1.4. A trace produced by the model checker for a given reachability property defines the set of actions executed on the FBD program. An example of a diagnostic trace has the following form:

$$(F_0, E_0) \xrightarrow{a_1} (F_1, E_1) \xrightarrow{a_2} ... \xrightarrow{a_n} (F_n, E_n),$$

where (F_k, E_k) are states of the FBD program and PLC cycle scan with environment constraints, respectively, and a_k are either internal synchronization actions, time-delays or `read?`, `execute?`, and `write!` global synchronizations. For FBD programs the sequence represents only the global synchronizations shown in Figure 1.4. Test cases are obtained by extracting from the diagnostic trace the observable actions `read?`, `execute?`, and `write!`. Obviously a single test case cannot be obtained for every test purpose or criterion. By using a program scan cycle we allow the test suite to be implemented as one or more test sequences separated by resets. To introduce resets in the model, we annotate the PLC cycle scan with a reset transition leading to the initial `ReadInputs` location. On this transition all variables and parameters

(excluding encoded internal variables) are reset to their default value. This reset is hardcoded into the PLC scan cycle for any modeled FBD program in UPPAAL, being an atomic communication between all timed automata.

7 Logic Coverage Criteria for Function Block Diagrams

The basic approach to generating test cases for logic coverage using model-checking is to define a test as a finite execution path. By characterizing a logic coverage criterion as a temporal logic property, model-checking can be used to produce a path for the test obligation.

Ammann et al. [5] argued that criteria such as logic coverage that have constraints involving more than one test trace cannot be handled in this way. The core problem is that each execution is characterized by a temporal formula, and test obligations span multiple runs of the model checker. This means that to ensure model-checking of MC/DC test obligations one should satisfy constraints on multiple runs of the model-checker. However, an FBD program has an implicit control loop, so a reset transition can occur in the program without modifying the transformed timed automata in any way. This reset transition restores the program to its initial state, making it possible to handle test obligations over multiple program executions as a single execution path containing subpaths separated by resets.

By using a translated FBD program, we use logic coverage to directly annotate both the model and the temporal logic property to be checked. We propose the annotation with auxiliary data variables and transitions in such a way that a set of paths can be used as a finite test sequence. In addition, we propose to describe the temporal logic properties as logic expressions satisfying certain logic coverage criteria. Informally, our approach is based on the idea that to get logic coverage of a specific program, it would be sufficient to (*i*) annotate the conditions and decisions in the FBD program, (*ii*) formulate a reachability property for logic coverage, and (*iii*) find a path from the initial state to the end of the FBD program. To apply the criteria, necessary properties for the integration of logic coverage need to be fulfilled.

For each criterion, model checking allows the generation of paths for logical predicates showing test obligations satisfaction. To do so, conditions and decisions have to be formulated as temporal logic formulae. Hessel et al. [12] proposed one way to apply coverage criteria to specifications described in timed automata. We extend this approach to apply it to the conditions and decisions in an FBD program.

Decisions in an FBD program are blocks that can be evaluated to a Boolean value, i.e., true or false. Decisions can be identified from the instrumentation points in the FBD program (e.g., AND block). Let $\{d_i\}$ be the set of decisions in an FBD program and $\{c_{ij}\}$ be the set of conditions in d_i.

DC requires every d_i to evaluate to true and false, and is described by the following two test obligations:

$$o_1 = d_i$$

$$o_2 = \neg d_i$$

These obligations guarantee that each decision d_i evaluates to both true and false, not necessarily along the same execution path.

CC requires two test obligations for each clause c_{ij} in a decision d_i, such that c_{ij} evaluates to both true and false:

$$o_1 = c_{ij}$$

$$o_2 = \neg c_{ij}$$

MC/DC imposes two requirements for test cases. First, for each condition c_{ij} in a decision d_i, test cases must show that c_{ij} determines the value of decision d_i, and second, c_{ij} has to evaluate to true and false. As shown in [3], a condition c_{ij} determines a decision d_i if there is an assignment of values to all the variables in d_i except c_{ij} such that the value of d_i is different for the two values of c_{ij}. This requirement is met if the following logical expression is satisfied [3]:

$$d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)}$$

Combining the two requirements for MC/DC coverage, we have the following two test obligations:

$$o_1 = c_{ij} \wedge (d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)})$$

$$o_2 = \neg c_{ij} \wedge (d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)}).$$

For generating tests for DC, CC, and MC/DC we represent the test obligations over a set of variables monitoring the decisions and conditions as a reachability property. This approach is implemented in the toolbox by automatically creating a temporal logic property used by the model checker to produce tests.

8 Example: Train Startup Mode

In the previous section we presented a technique to compute logic coverage for FBD programs. In the following we show empirically that the performance of our technique is sufficient for practically relevant examples. We have applied our method on a real world example provided by *Bombardier Transportation AB*. We present here how our method is applied to test a part of the MITRAC *Train Control and Management System* (TCMS) provided within the ATAC research project. TCMS is a distributed system, built on open standard *IP-technology* that allows easy integration of control and communication functions for high speed trains. We are concerned with both the transformation of FBD programs to timed automata models and the time and memory used to generate test cases. The tools used for developing these programs are based on the *MULTIPROG* software. The FBD program is transformed using the MOS tool [4] [7].

[3] $d_{i(c_{ij},v)}$ denotes d_i with c_{ij} replaced with v.

[4] MOS is a tool for model-based and search-based testing of safety-critical systems implemented in FBD language, developed at Mälardalen University since 2012.

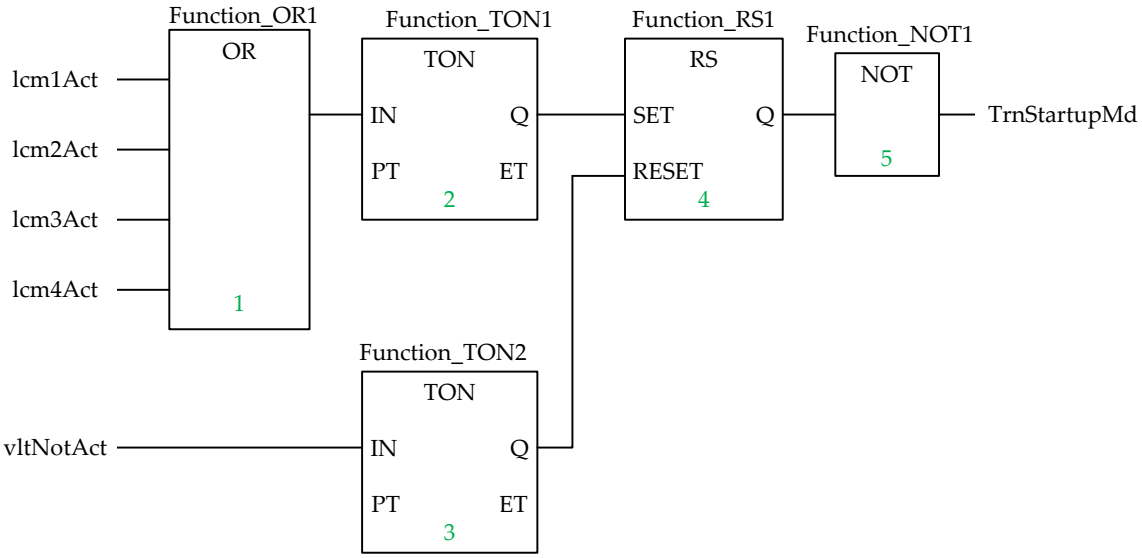


Figure 1.5: Simplified Train Startup Mode modeled as an FBD program.

8.1 Experiments

The experiments reported here are based on an example program, part of TCMS. We use an FBD program of a train Startup Mode System (TSM) and generate test cases for logic coverage. In the process, we describe the FBD program, the program to timed automata model transformation and the annotations made to the model.

The train is built up using motorized cars and intermediate trailer cars with pantographs. These cars are combined to create a fixed 8-cars train set, each with its own complete system for control and propulsion. The task of the train operating in the startup mode is imposed by the controller FBD program depicted in Figure 1.5. When the first Line Converter Module (LCM) is active, the propulsion unit becomes active, i.e. any of the four inputs becomes true. When activating the propulsion system, the program waits an additional five seconds and then sets the output to false, which means that the train is not in the startup mode anymore. If the `NotAct` is true for at least five seconds, the element is reset and the output is set true as in the startup mode.

To validate our approach for generating test cases for logic coverage, we implemented our method in our previously developed MOS tool for analyzing and executing FBD programs. The TSM system is transformed automatically in the fully formal and executable timed automata used by UPPAAL. The TSM system is modeled as a parallel composition of several processes. Several boolean and integer variables are used for recording information: `read!`, `execute!` and `write!` synchronization channels are used to model the execution of the FBD program, `et` is used to keep track of the elapsed time in timer components, `lcm1act, lcm2act, lcm3act, lcm4act` for recording the input variables generated by the LCM input automaton, `notact` for representing the line voltage activation, `TrnStartupMd` for recording the startup mode of the train, p_i and c_i for recording each covered item, and `pt` for recording the delay from the first LCM starts to communicate. The TSM

program has been transformed and checked by UPPAAL model checker for generation of test suites for logic coverage by reachability analysis.

The parallel processes are:

- `plcSupervision` This automaton controls the valid structural information for the other automata. The structure of the FBD program is restricted to reading inputs, execution of the components, and the writing of the outputs.
- `input-i-Act` This automaton non-deterministically generates valid input sequences for the `function-OR1` and `function-TON2` automata. Valid sequences are restricted to boolean values.
- `function-OR1` The `function-OR1` automaton encodes a boolean OR function by reading the input values and returning a true or false value for the next automaton.
- `function-TON1` The TON automaton counts time-based intervals when the input received from the `function-OR1` automaton is true and activates its output after a preset interval five seconds.
- `function-TON2` Similar to `function-TON1` when activating the `NotAct` input for at least five seconds, the element is reset and the output is set true as in the startup mode.
- `function-RS` This automaton is a memory function when we turn the `SET` input port. The `Q` output port stays true until we give another `RESET` signal by setting the output the `function-TON2` automaton to true.
- `function-NOT` The `function-NOT automaton` is outputting the negated value received at it's input port.
- `outputStartupMd` The output startup-mode automaton checks the current value of received from the `function-NOT` automaton. It also updates the values of the variables `out` and `n`.

three synchronization channels, three clock variables used to keep track of the elapsed time in TON component. The *TrainStartupMd* can be true of false depending on the input variables or based on the state of the program.

By introducing the variables p_i and c_i in the parallel process, the model checker is guaranteed to be used for its diagnostic feature. [5]

Table 1.1 shows the generation time (in seconds) for test suites generated from different logic coverage criteria of the TSM example, and the length (number of program cycles) of the generated test suite. We notice that for CACC test cases result in longer traces than for PC and CC. The generation time for CACC is slightly

[5]For reasons of simplicity and clarity in presentation we have chosen only to consider test suites generated from UPPAAL diagnostic trace with breadth first as search order and no state space reduction. However, the generation extends easily to other settings for controlling the behavior of the model-checker.

Table 1.1: Generation time and test suite length for various coverage criteria

Coverage Criterion	Generation Time (seconds)	Test Suite Length (program cycles)
PC	18,04	14
CC	18,21	14
CACC	22,86	25

Table 1.2: Results of obtaining PC of the TSM example with increasing timer elements

Timers	Generation Time (seconds)	Test Suite Length (program cycles)	Memory Usage (MB)
0	0,62	4	15
1	1,54	5	27
2	3,29	7	61
5	6,38	14	122
10	6,79	18	200
50	31	36	520

higher than the number for PC and CC. However, the number of program cycles is twice as high because CACC is combining already generated test suites for PC and CC.

8.2 Logic Coverage and Timing Components

One of the objectives for this experiment is to assess the applicability and scalability of using logic coverage for testing FBD programs with various sizes and complexities. An expected characteristic of the FBD program is its associated timing behavior. For the TSM model, the TON automaton appears to be significantly affecting the generation time. Therefore, we focus the discussion on timer components (e.g., TON, TOF, etc.) because these cases lead to a bigger search space. We modify the program by increasing or decreasing the number of TON components in the TSM model. We observed that an FBD program consisting of ten or more TON components are difficult to cover. This is not surprising as the timing components are varying the timing of the entire model and therefore the number of predicates and clauses in the program. The programming of two or more timers components together in the same FBD program is called *cascading*. From our experiments with timer components in TCMS (over 300 FBD programs), the number of TON and TOF components is always lower than five. Nevertheless we were interested to show that —for the studied program— our method of generating test cases for covering FBD programs is applicable and scalable.

The results, listed in Table 1.2, show that the memory usage increase essentially linearly with the number of timing elements. If we compare test suite length with the generation time, it can be seen that is much cheaper to compute FBD programs for FBDs with less than ten timer components than computing for fifty timer

components. We can try to explain this behavior in the sense that timer components pose restrictions on the solution because it contains more possible behaviors. Thus, searching through more timer components takes longer. We note that the use of timer elements restricts the handling of larger systems, with an increased cost of generation time and used memory.

9 Related Work

Model checkers have been used to produce test cases satisfying various criteria and for programs in a variety of formal languages [5, 13, 8]. Black et al. [5] discuss the problems encountered in using a model-checker for test case generation for full-predicate coverage. They present reasons why model-checking is not directly applicable for generating tests to satisfy logic coverage criteria. In our previous work [9], we overcome this issue by providing a way of generating test cases for logic criteria that are directly applicable to FBD programs. We found that model-checkers are an appropriate technique for automated test generation in terms of performance when used on real-world programs.

For data-flow programming languages such as FBD and Lustre, which describe the relationship between inputs and outputs instead of the control flow of the program, researchers proposed specific coverage metrics based on the structural aspects of the programs [15, 14, 16]. For Lustre, structural coverage metrics are based on the activation condition concept of the language that can be used when data travels from an input edge to an output edge. In addition, Whalen et al. [20] defined an alternative approach to measuring logic coverage for data flow programs called OMC/DC, a combination of MC/DC and an additional obligation to be satisfied such that faults will be observed through a variable monitored by the criteria.

10 Conclusion

In this paper we have shown how test case generation that aims to satisfy logic coverage on Function Block Diagrams can be solved as a model checking problem, by using model checking tools to automatically create traces that can be transformed into executable tests. We described a toolbox in which logic coverage criteria can be formalized and used by a model-checker to generate test cases. We carried out an extensive empirical study of the method by applying the toolbox to 157 real-world industrial programs developed at Bombardier Transportation. The results showed that model checking is suitable for handling logic coverage for real-world FBD programs, and also revealed some potential limitations of the toolbox when used for test generation. The evaluation showed that the toolbox is efficient in terms of time required to generate tests that satisfy logic coverage and that it scales well for most of the programs. Our overall conclusion is that the model-checking approach provides a positive and useful addition to the testing process for FBD programs.

Acknowledgments

The authors would like to thank Elaine Weyuker and Thomas Ostrand for their valuable comments on this work. This research was supported by VINNOVA, the Swedish Governmental Agency for Innovation Systems within the ATAC project.

References

[1] R. Alur. "Timed Automata". In: *Computer Aided Verification*. 1999, pp. 688–688 (cit. on p. 36).

[2] R. Alur and D. Dill. "Automata for Modeling Real-time Systems". In: *Automata, languages and programming*. Springer, 1990, pp. 322–335 (cit. on pp. 34, 35).

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008 (cit. on pp. 36, 43).

[4] Paul Ammann, Jeff Offutt, and Hong Huang. "Coverage Criteria for Logical Expressions". In: *14th International Symposium on Software Reliability Engineering*. 2003, pp. 99–107 (cit. on p. 36).

[5] 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 pp. 34, 42, 47).

[6] 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 p. 34).

[7] Eduard Paul Enoiu, Kivanc Doganay, Markus Bohlin, Daniel Sundmark, and Paul Pettersson. "MOS: An Integrated Model-based and Search-based Testing Tool for Function Block Diagrams". In: *Proceedings of the International Workshop on Combining Modeling and Search-Based Software Engineering*. IEEE, 2013, pp. 55–60 (cit. on p. 43).

[8] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. "Model-based Test Suite Generation for Function Block Diagrams using the UPPAAL Model Checker". In: *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2013, pp. 158–167 (cit. on p. 47).

[9] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. "Using Logic Coverage to Improve Testing Function Block Diagrams". In: *Testing Software and Systems*. Springer, 2013, pp. 1–16 (cit. on p. 47).

[10] 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 p. 33).

[11] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. "Testing Real-time Systems using UPPAAL". In: *Formal Methods and Testing*. Springer, 2008, pp. 77–117 (cit. on p. 37).

[12] Anders Hessel, Kim G Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. "Time-optimal Real-Time Test Case Generation using UPPAAL". In: *International Workshop on Formal Approaches to Software Testing*. 2003, pp. 114–130 (cit. on p. 42).

[13] 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. 47).

[14] 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. 47).

[15] E. Jee, J. Yoo, S. Cha, and D. Bae. "A data flow-based structural testing technique for FBD programs". In: *Information and Software Technology*. Vol. 51. 7. Elsevier, 2009, pp. 1131–1139 (cit. on p. 47).

[16] A. Lakehal and I. Parissis. "Structural Test Coverage Criteria for Lustre Programs". In: *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems*. 2005, pp. 35–43 (cit. on p. 47).

[17] 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. 35).

[18] 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. 34).

[19] 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. 34).

[20] Michael Whalen, Gregory Gay, Dongjiang You, Mats P. E. Heimdahl, and Matt Staats. "Observable Modified Condition/Decision Coverage". In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2013, pp. 102–111 (cit. on p. 47).

Study 2

Automated Test Generation using Model-Checking: An Industrial Evaluation

Eduard Paul 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 2. Automated Test Generation using Model-Checking: An Industrial Evaluation

Eduard Paul Enoiu, Adnan Čaušević, Elaine Weyuker, Tom Ostrand, Daniel Sundmark and Paul Pettersson

Abstract

In software development, testers often focus on functional testing to validate implemented programs against their specifications. In safety critical software development, testers are also required to show that tests exercise, or cover, the structure and logic of the implementation. To achieve different types of logic coverage, various program artifacts such as decisions and conditions are required to be exercised during testing. Use of model-checking for structural test generation has been proposed by several researchers. The limited application to models used in practice and the state-space explosion can, however, impact model-checking and hence the process of deriving tests for logic coverage. Thus, there is a need to validate these approaches against relevant industrial systems such that more knowledge is built on how to efficiently use them in practice. In this paper, we present a tool-supported approach to handle software written in the Function Block Diagram language such that logic coverage criteria can be formalized and used by a model-checker to automatically generate tests. To this end, we conducted a study based on industrial use-case scenarios from Bombardier Transportation AB, showing how our toolbox CompleteTest can be applied to generate tests in software systems used in the safety-critical domain. To evaluate the approach, we applied the toolbox to 157 programs and found that it is efficient in terms of time required to generate tests that satisfy logic coverage and scales well for most of the programs.

1 Introduction

Advances in model-checking tools and technology in the last decade have made it a pragmatically usable technique for test case generation from finite-state models [12]. There have been a number of approaches used for defining logic coverage using model checkers, e.g., [6, 23, 24], however, these techniques are not directly applicable to real-world programs of critical systems. When industrial software systems are being tested, there is still the issue of potential combinatorial explosion of the state space which thereby limits the application to models used in practice.

Many industrial application domains use safety-critical software to implement the behavior of programmable logic controllers (PLCs). One of the programming

languages defined by the *International Electrotechnical Commission* (IEC) for PLCs is the *Function Block Diagram* (FBD). Programs developed in FBD are automatically transformed into program code, which is compiled into machine code by using specific engineering tools provided by PLC vendors. The motivation for using FBDs as a preferred language arises because it is the standard in many industrial software systems, such as in the railway domain. Such systems typically require a certain degree of certification [7], such as some level of logic coverage which must be demonstrated on the developed software. Although all software should aspire to correctness, safety critical software is generally held to a higher standard than other types of systems, which should be reflected in their testing. However, there is no commonly accepted level of test thoroughness for safety-critical software. In this paper, we show how to efficiently generate test cases that achieve several levels of coverage, including MC/DC and decision coverage. It should be noted that the generated tests are not intended to replace requirement-based test design at the FBD program level, but to complement it with a structural perspective.

In our previous work we proposed the use of logical coverage for FBD programs [11] and defined a model-based test generation method based on the Uppaal tool. While this approach is promising, there is a need to validate it using realistic programs of critical systems. To this end, we conduct an experimental evaluation using 157 programs of a train control system, written in the FBD language. We develop a toolbox, named CompleteTest[1], suitable for transforming an FBD program to a formal representation of both its functional and timing behavior. This is done by implementing an automatic model–to–model transformation from FBDs to timed automata. Timed automata, introduced by Alur and Dill [3], were chosen because there is an already existing formal semantics and tool support for simulation and model-checking using Uppaal [21]. The transformation accurately reflects the data-flow characteristics of the FBD 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 final step produces a formal behavior for every standard component in the FBD program. These steps are independent of timed automata and therefore are generic in the sense that they could also be used when translating an FBD program to a different target language. The toolbox uses a test generation technique based on model-checking, tailored for logic coverage of FBD programs. A generated test consists of a sequence of input vectors. As the main purpose of the tool at present is to generate test cases that satisfy coverage criteria, the tool does not generate expected outputs. Expected outputs can be provided manually to the toolbox by a human tester.

The paper is organized as follows. Section 2 provides an overview of PLC software, the IEC 61131-3 standard, timed automata and logic coverage. Section 3 describes the transformation scheme into timed automata. Section 4 and Section 5 present the test case generation method required for logic coverage criteria. In Section 6, we describe the tool box used for testing FBD software and demonstrate its application

[1] CompleteTest is available at `http://www.completetest.org/`.

by showing relevant user scenarios. In Section 7, we evaluate the toolbox on industrial programs in terms of its efficiency and usability. Section 8 describes related work. Section 9 presents our conclusions.

2 Preliminaries

This paper describes a toolbox for generating tests that cover the logical structure of FBD programs, by transforming them first to networks of timed automata. Our technique will be illustrated throughout this paper using a complete, small, but non-trivial FBD program that exhibits many of the features of FBDs. We show how this program can be translated into a timed automaton and used to illustrate the approach, the toolbox evaluation and its practical implications. In this section, we provide some background details on FBD programs, timed automata and logical coverage.

2.1 Programmable Logic Controllers

PLCs are widely used in real-time software for many types of software systems including nuclear plants and train systems. A PLC is an integrated embedded system that contains a processor, a memory, and a communication bus. The semantics of a program running on a PLC has the following representative characteristics:

- programs execute in a cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs).
- Input and output channels correspond to sensors and actuators respectively.

The language can be specified on an implementable subset of timed automata [3]. Dierks [9] proposed a new class of automata suitable for PLCs and this definition is the basis for implementing a model-to-model transformation for PLC software.

FBD, a PLC programming language standardized by IEC 61131-3, is very popular in industry because of its graphical notations and its data flow nature [22]. Blocks in an FBD program form the basis for a structured and hierarchical application. They may be supplied by the manufacturer, defined by the user, or predefined in a library. An application generator is utilized to automatically transform each block to a C compliant program with its own thread of execution. A block cannot be recursive as it cannot call itself [26]. However, blocks may have multiple instances within a program.

Although our description is not limited to a particular PLC software development style for FBD programs, it is exemplified by a generic PLC control application compliant with the IEC 61131-3 standard. A PLC periodically scans an FBD application program, which is loaded into the application memory. As an example of the FBD generic model, we consider first the hierarchical structure of a PLC and the functional integration. An FBD control program is considered as a hierarchical application. The FBD program is created as a composition of interconnected blocks,

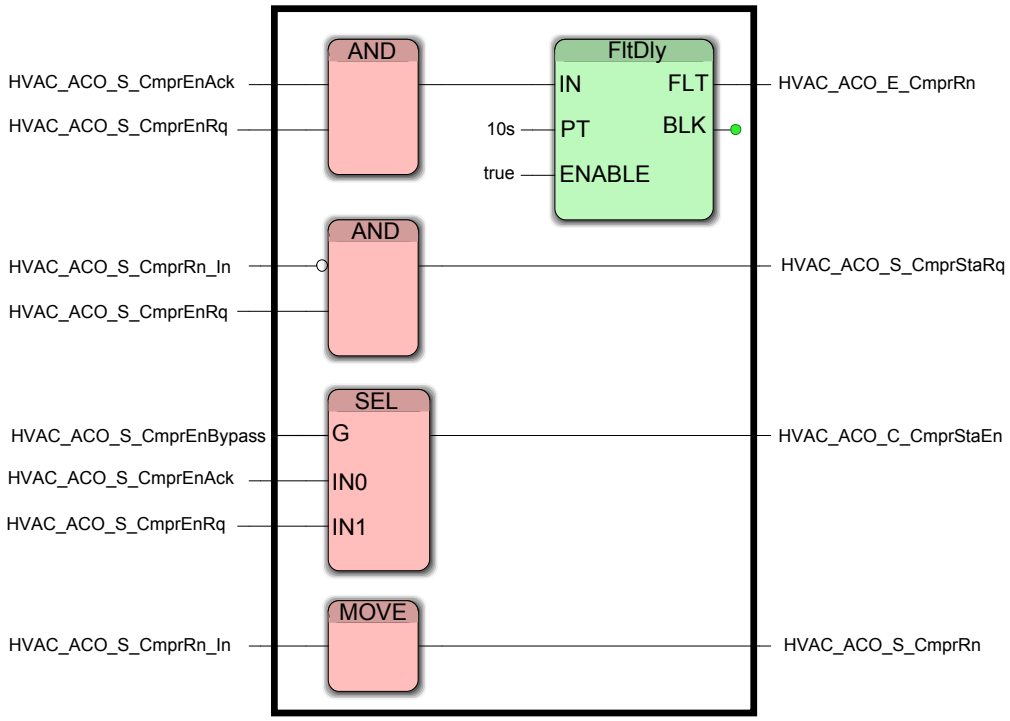


Figure 2.1: Running Example: Compressor Start Enable program showing the graphical nature of the language.

which may have intra-program data flow communication. When activated, a program consumes one set of input data and then executes to completion. The code is used on the specific PLC and is the actual application code from the IEC 61131-3 compliant FBD program.

The IEC 61131-3 standard proposes a hierarchical software architecture for structuring and running any FBD program. This architecture specifies the syntax and semantics of a unified control software based on a PLC configuration, resource allocation, task control, program definition, function and function block repository, and program code [22, 26].

The systems we are studying contain a particular type of blocks called *PLC timers*. These timers are output instructions that provide the same functions as timing relays and are used to activate or deactivate 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 and false and outputs different signals based on these counters. In practice many other time configurations can be derived from these basic timers. In order to study how to generate test cases using a model checker for these types of FBD programs, we use a formal representation that can cope with timers and timing information.

2.2 The Compressor Start Enable Program

The translation scheme, test generation, and logic coverage will be illustrated by translation of a complete, small, but typical FBD program that includes many of the FBD features. Figure 2.1 contains this FBD program for which we will ultimately generate test cases. It was developed by an engineer from Bombardier Transportation responsible for developing train control software in Västerås, Sweden.

The train is made up of motorized cars and intermediate trailer cars with pantographs. These cars are combined to create a fixed eight car train, each with its own complete software control system that applies regulation to a heating and/or air conditioning system. The task of the train operating the ventilation compressor mode is imposed by the controller FBD program depicted in Figure 2.1. The program requests permission to start the ventilation compressor from the auxiliary load control. When granted, it will forward the command to the ventilation controller. The Compressor Start Enable will request permission to start the ventilation compressor. When granted, the signals are forwarded to the ventilation controller.

The request will time out (`HVAC_ACO_E_CmprRn`) when the compressor start signal is acknowledged (`CmprEnAck`) and required (`HVAC_ACO_S_CmprEnRq`) provided the clock is greater than or equal to ten seconds. Additionally, the ventilation should be active (`HVAC_ACO_S_CmprRn`) when the compressor is running (`HVAC_ACO_S_CmprRn_In`). The ventilation request is started (`HVAC_ACO_S_CmprStaRq`) when the compressor is enabled (`HVAC_ACO_S_CmprEnRq`) and the compressor is not running (`HVAC_ACO_S_CmprRn_In`). When the external supply (`HVAC_ACO_S_CmprEnBypass`) is not available, the compressor should be enabled (`CmprStaEn`) only when the compressor is allowed to start from auxiliary load control (`HVAC_ACO_S_CmprEnAck`). If the external supply is available, then the compressor is enabled.

The program consists of basic functions (e.g., AND, SEL, MOVE) and function blocks (e.g., FltDly). In Figure 2.1, AND is a function. In contrast, FltDly is a function block because it maintains an internal state and produces outputs based on this state and inputs. Recall that in order to express timing constraints within one component, standard PLC timers are used. The timers in a PLC are operated by an internally generated clock that originates in the processor module. Consider the following PLC timer `FltDly` defined as a tuple:

$FltDly = \langle FltDly_1, (IN, PT, ENABLE, FLT, BLK), B_t \rangle$,

where $FltDly1$ is the name identifier, IN, PT, $ENABLE$, BLK, and FLT are the set of ports and parameters in `Port`, and B_t is the behavior description. This timer component is an attempt to specify its interface and behavior. From a semantic point of view, FBD programs are a special case of deterministic reactive systems. We use more informative notations to denote the actual behavior. In the following section we present several such notations to describe how FBD programs can be handled by the Uppaal model checker.

2.3 Networks of Timed Automata

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 [3] and has gained

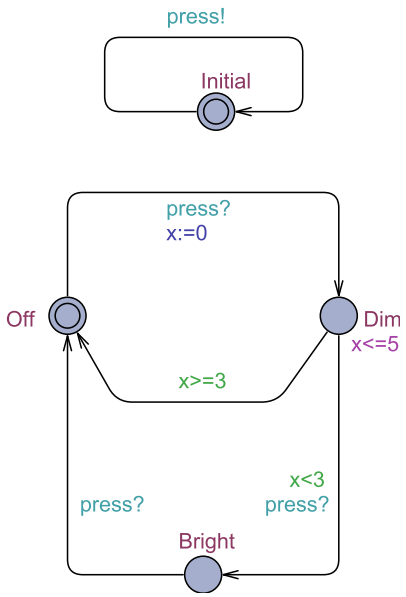


Figure 2.2: Example of a network of timed automata.

in popularity as a suitable model for real-time systems. We consider model checking algorithms that perform analysis to check for a reachability property of the form $\exists \Diamond \beta$. $\exists$ is the existential quantifier, $\Diamond$ is the temporal operator meaning eventually, and β is a formula designed to capture the requirements of a particular type of logic coverage. The reachability property states that there exists a path σ through the states of the timed automaton such that β eventually holds. The property is presented to the model checker, which then attempts to find an actual path that satisfies the property. A path σ that satisfies the reachability property can be converted into a test case that satisfies the desired coverage. We use a timed modal logic to specify properties. The logic may be seen as properties that can be expressed as logical formulae in the Timed Computational Tree Logic (TCTL) [2].

An example of a network of timed automata modeled in UPPAAL is shown in Figure 2.2. The network consists of an automaton of a lamp and an automaton of a user. A network of timed automata in this case can be written as $Lamp \parallel User$. The user operates the lamp by pressing the on/off switch. By pressing the switch, the lamp can be in three possible locations: Off, Dim and Bright. The automaton of the lamp starts at the Switched Off location and contains one clock x. If the user presses the light switch, the lamp switches to Dim and the clock is reset, by the update assignment $x := 0$. When Dim, the lamp remains on as long as the clock is less than or equal to five time units (i.e., invariant $x <= 5$). A state of the automaton Lamp depends on its current location and on the current values of its clocks. If the user presses the light switch before three time units, then the lamp switches to location Bright. In this location, the lamp automaton stays ON until the user presses the light switch again. Both automata synchronize via the actions press! and press? i.e., by sending via channel press! and receiving using press?. Based on the states of the Lamp automaton, one can denote traces starting from the initial state as a sequence of alternating transitions $\sigma = (Off, 0) \xrightarrow{press} (Dim, 0) \xrightarrow{delay} (Dim, 2) \xrightarrow{press} (Bright, 2)$.

We provide a brief summary of the notation and concepts in Appendix, for readers unfamiliar with timed automata theory. Further information can be found in [1].

2.4 Logic-based Coverage Criteria

Coverage criteria are a code-based means of assessing the thoroughness of test cases. They are normally used at the unit test level to check that various aspects of the code structure have been exercised by the test cases. Out of the many criteria that have been defined and studied, we have implemented three logic-based criteria that measure the thoroughness of test coverage of the control flow structure of FBD programs.

FBD program flow is controlled by atomic Boolean expressions called *conditions*, and by *decisions* made up of conditions combined with Boolean operators (NOT, AND, OR, XOR, IMPLIES, EQUIV). A condition can be a single boolean variable, an arithmetic or character comparison with a Boolean value (e.g., $A > B$ or $str1 == str2$), or a call to a function with a Boolean value, but does not contain any Boolean operators. The test generation tool uses the UPPAAL model checker to generate test cases that satisfy three types of logic coverage: *decision coverage DC* (also known as predicate coverage), *condition coverage CC* (also known as clause coverage) and *Modified condition decision coverage (MC/DC).*

A set of tests satisfies decision coverage if running the test cases causes each decision in the FBD program to have the value *true* at least once and the value *false* at least once. Note that for any individual predicate, the true and false values might occur under a single test case or under two different test cases. In general a single test case will exercise more than one decision, and it is possible, but certainly not required, that all decisions in a program might have both values exercised by a single test case. In the context of traditional sequential programming languages, decision coverage is usually referred to as *branch coverage.*

Condition coverage requires test cases that cause each individual condition to be exercised at least once with value true and once with value false. A set of test cases might satisfy either condition coverage or decision coverage, or both of them. Modified condition decision coverage captures the idea that the value of a decision can be controlled by the value of each of its conditions independently of the values of all other conditions. This means that for each individual condition c in a decision, there are sets of values of all the other conditions so that the decision's value differs for the two possible values of the condition c.

For MC/DC each individual condition in each decision should be shown to be able to determine the outcome of the decision during testing. MC/DC is a stronger requirement than CC; any test set that satisfies MC/DC must also satisfy CC. For most non-trivial decisions, MC/DC is also more strict than DC, even though there are decisions for which a MC/DC-satisfactory test set does not satisfy DC. CC, DC and MC/DC, as well as other logic criteria, are defined and exemplified in [8, 4].

```
1  plc = plcSupervision();
2
3  readinput1= input_HVAC_ACO_S_CmprEnRq();
4  readinput2= input_HVAC_ACO_S_CmprRn_In();
5  readinput3= input_HVAC_ACO_S_CmprEnAck();
6  readinput4= input_HVAC_ACO_S_CmprEnBypass();
7
8  block1= Function_AND1();
9  block2= Function_AND2();
10 block3= Function_MOVE1();
11 block4= Function_SEL1();
12 block5= Function_FltDly1();
13
14 writeoutput1= output_HVAC_ACO_C_CmprStaEn();
15 writeoutput2= output_HVAC_ACO_S_CmprStaRq();
16 writeoutput3= output_HVAC_ACO_S_CmprRn();
17 writeoutput4= output_HVAC_ACO_E_CmprRn();
18
19 system plc, readinput1, readinput2, readinput3,
20 readinput4, block1, block2, block3, block4, block5,
21 writeoutput1, writeoutput2, writeoutput3,
22 writeoutput4;
```

Figure 2.3: Interface elements created from structure and behavioral elements from the Compressor Start Enable.

3 Translation

The translation scheme will be illustrated on the running example. After translation, the UPPAAL model checker can be applied to test that the program satisfies the required logic coverage on the FBD program. The translation is performed starting from signals which are translated into global variables shared by the corresponding blocks. Additionally, FBD blocks are mapped to input/output behavior (e.g., functional and timing) between signals. This may be done by using predefined UPPAAL operators, as in the case of *basic blocks* (e.g., `AND, SEL, MOVE`), or by capturing the functionality of more *complex blocks* (e.g., `FltDly`) from their description.

In practice the timed behavior of an FBD program is defined as a network of timed automata, extended with data input and output variables. We first perform an automatic transformation of the FBD program to timed automata that obeys the *read-execute-write* semantics of the FBD program, hence preserving the semantics of FBDs without altering its structure. Next, we specify the execution of each block, and construct a complete timed automata model by the parallel composition of local behaviors.

3.1 FBD Structure

For illustration, we start with the translation of the Compressor Start Enable Program. For each block, a timed automaton is defined for the program description. Templates of components are included and we list the composed timed automata network representing the FBD program as

$$AND_1 \parallel AND_2 \parallel SEL_1 \parallel MOVE_1 \parallel FltDly_1$$

```
chan execute,write, read;

// variable definition for the FBD program
// US_INT OR BOOL VAR_INPUT

// input variable definition
bool HVAC_ACO_S_CmprEnRq;
bool HVAC_ACO_S_CmprRn_In;
bool HVAC_ACO_S_CmprEnAck;
bool HVAC_ACO_S_CmprEnBypass;

// output variable definition;
bool HVAC_ACO_C_CmprStaEn;
bool HVAC_ACO_S_CmprStaRq;
bool HVAC_ACO_S_CmprRn;
bool HVAC_ACO_E_CmprRn;

// internal intermediate variables
bool AND1;
bool AND2;
bool MOVE1;
bool SEL1;
bool FltDly1;
clock BLK;
bool ENABLE;
const int PT=10;
```

Figure 2.4: Input, Output, and Internal Signals translated for the Compressor Start Enable Program.

The top-level structure of the UPPAAL model is shown in Figure 2.3 and represents a parallel composition of several processes corresponding to inputs (lines 3-6), outputs (lines 14-17), and blocks (lines 8-12).

An input named in the program `HVAC_ACO_S_CmprEnAck` will be automatically translated into a timed automata template named `input_HVAC_ACO_S_CmprEnAck()`. When an input signal in FBD has a name, the name is preserved during translation. However, it is often the case that signals in FBDs are not named (e.g., in the Compressor Start Enable program there are simply "*wires*" connecting two blocks). In such a case, the name given to the signal corresponds to the name of the block which produces the signal. For example, the output signal produced by `Function_AND2()` will correspond to an UPPAAL variable named `bool AND2` as shown in Figure 2.4.

Several Boolean and integer variables are used for recording information in the UPPAAL model and are shown in Figure 2.4: `read`, `execute` and `write` synchronization channels are used to hard code the execution of the program, the `BLK` clock variable is used to keep track of the elapsed time in FltDly, other variables (e.g., bool `HVAC_ACO_S_CmprEnRq`) are used for recording the inputs generated by the input automaton, `PT` represents the fault delay, while `ENABLE` records the compressor enable.

3.2 Cycle Scan and Triggering

A block in an FBD has an *interface*, consisting of a name identifier, input and output ports, and a list of parameters. The interface is used to access the block behavior. When the block is activated the *behavior* is started using the values read

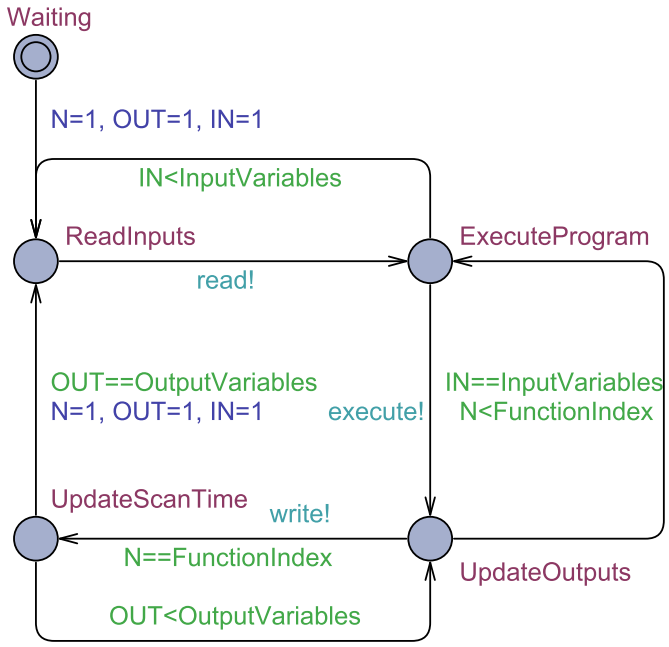


Figure 2.5: Timed Automaton of a Program Cycle Scan and Execution Order.

on the input ports. When the behavior ends, i.e., when the block implementation terminates its execution, the output ports are updated. The behavior of a block is typically implemented by a code fragment that updates local variables. In addition, the program contains a clock variable for modeling a delay between the cycles. We show in Figure 2.5 how a cycle starts when the automaton enters the `ReadInputs` node and ends its computation in `UpdateScanTime` node. For an FBD program, the execute operation of each block is extended according to connections and `IN` and `OUT` variables corresponding to the program inputs and outputs. A program composition is a set of interconnected blocks closed under a specific execution order. The execution order `N` is automatically defined according to the general rules included in the IEC 61131-3 standard. This predetermined order directly dictates the data dependency. Using the program cycle requires deterministic program execution, by restricting the underlying timed transition system. The program is executed in a loop and the computation follows the *run-to completion* semantics. We use the notion of precedence to describe such dependencies on the convention of reading such FBD programs in a top-to-bottom, left-to-right fashion. To show an example of a program cycle scan as shown in Figure 2.5 different actions are executed:

- `read(IN)` for reading variables from `IN`.
- `write(OUT)` for writing variables onto output ports.
- When the execution order holds, the ports are updated by `read(IN)`, and `write(OUT)`.

For the Compressor Start Enable program, the *execution order* is `AND1`, `FltDly1`, `AND2`, `SEL1`, and `MOVE1`. For each block we assign a precedence priority to the

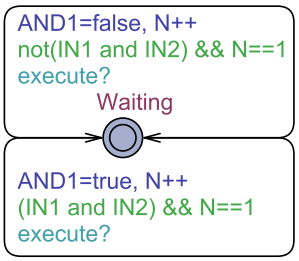


Figure 2.6: An automaton showing the AND logical block.

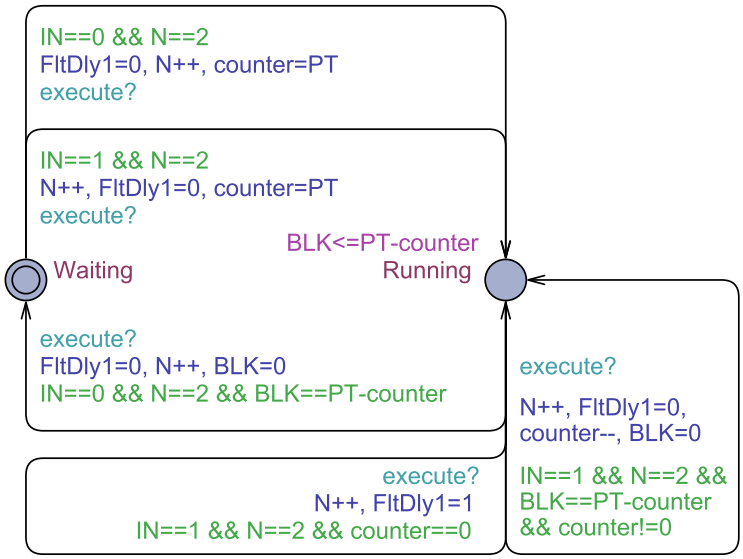


Figure 2.7: A Timed Automaton showing a FltDly timer block.

corresponding timed automaton. A counter is created in this step to represent the execution priority of a block. In this way we ensure that block are executed one by one. After the last block is evaluated, the counter is reset to repeat the scan cycle.

3.3 Translation of basic blocks

Simple FBD blocks are translated into predefined UPPAAL operators. In particular:

- The `Logical Operator` blocks are translated using the logical UPPAAL operators `and`, `not`, `or`.
- The `Arithmetic Operator` blocks are translated using the arithmetic UPPAAL operators `+, =, - , /, *`.
- The `Comparison blocks` are translated using the relational operators UPPAAL `<, >, <=, >=, =`.
- The `Selection` blocks are translated using `if-then-else` statements.

The behavior mapped onto a basic block is modeled as an UPPAAL automaton as shown in Figure 2.6 for an `AND` logical block. The execution of the translated FBD

program is determined in terms of the execution order `N`. A block is therefore initially in location `Waiting`, and after performing the read action it starts executing until its internal computation is done. After completing the write action, which forwards data from the output ports via connections, the block becomes `Waiting` again.

The parallel processes translated for the basic blocks for the Compressor Start Enable program are the following:

- `plcSupervision` The automaton in Figure 2.5 controls the valid structural information for the other automata. The structure of the FBD program is restricted to reading inputs, execution of the components, and the writing of the outputs.

- `input_name` This automaton non-deterministically generates valid input sequences for the translated blocks. Valid sequences are restricted to Boolean and Integer values.

- `block_AND1 and block_AND2` The automaton in Figure 2.6 encodes a Boolean AND function by reading the input values and returning a true or false value for the next automaton.

- `block_SEL1` Selects one of two inputs depending on the value of a Boolean input. Then the translation would be:

  `SEL1= if(G=true) then SEL1=IN1 else SEL1=IN0.`

- `block_MOVE1` This automaton is a memory function when we turn on the input port.

- `output_name` The output startup-mode automaton checks the current value received from the function automaton. It also updates the values of the variables `OUT` and `IN`.

More stateful blocks are translated into UPPAAL automata. In particular:

- The `Bistable` blocks (e.g., `SR` and `RS` latches) are elements whose output depends not only on the current inputs, but also on previous inputs and outputs. These blocks can be implemented using logical, relational UPPAAL operators and `if-then-else` statements.

- The `Edge Detection` blocks are translated using UPPAAL expressions involving Boolean operators.

- The `Counters` blocks are translated by the use of UPPAAL `++` increment and `-` decrement operators.

- The `Timer` blocks are translated as a special automaton that is initially in location `Waiting`. After reading its inputs, it starts executing in location `Running` until its internal computation is done. After computing the on-delay timer, it forwards data to output ports and the block becomes `Waiting` again. One example of a `Timer` block from the Compressor Start Enable program is

shown in Figure 2.7. The `FltDly` automaton counts time-based intervals when the input is true and activates its output after a preset interval of ten seconds. The cycle scan interacts with the timer block via the `execute?` action. The timer sets the output `FltDly1` to true if `IN` variable is true at least as long as the time `PT` and `ENABLE` are set to true.

4 Testing Function Block Diagram Software using the UPPAAL Model-Checker

In this section, we describe an approach to automatically generating tests for FBD programs. Logic coverage criteria are used to define what test cases are needed and we use a model-checker to generate test traces. In addition, the methodology presented in this paper is tailored for FBD programs, and is composed of the following steps, mirrored in Figure 2.8:

1. *Model Transformation* To test an FBD program we map it to a finite state system suitable for model checking. In order to cope with timing constraints we have chosen to map FBD programs to timed automata.

2. *Logic Coverage Annotation* We annotate the transformed model such that a condition describing a single test case can be formulated. This is a property expressible as a reachability property used in most model checkers.

3. *Test Case Generation* We now use the model-checker to generate test traces. To provide a good level of practicality to our work, we use a specific model-checker called UPPAAL which uses timed automata as the input modeling language [2]. The verification language supports reachability properties. In order to generate test cases for logic coverage of FBD programs using UPPAAL, we make use of UPPAAL's ability to generate test traces witnessing a submitted reachability property [14]. Currently UPPAAL supports three options for *diagnostic trace generation*: some trace leading to a goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest time delay.

While UPPAAL is a viable tool for model checking, it is not directly tailored to test case generation in practice. We demonstrate how to work around this by automatically generating traces for logic coverage of the control flow of FBD programs described in timed automata and how we transform these traces to actual test cases. We discuss these steps in further detail in the following sections.

As a result of the translation described in Section 3, we consider that the FBD program is given as a closed network of timed automata as shown in Figure 2.9. This model contains two sub-networks, one modeling the `FBD Program` and the other one modeling its `Input` and `Output` Model. In addition, we consider a completely unconstrained input environment that allows all possible interactions between the timed automata network elements. In this way the cycle scan is used to control

[2]The UPPAAL tool is available at `http://www.uppaal.org`.

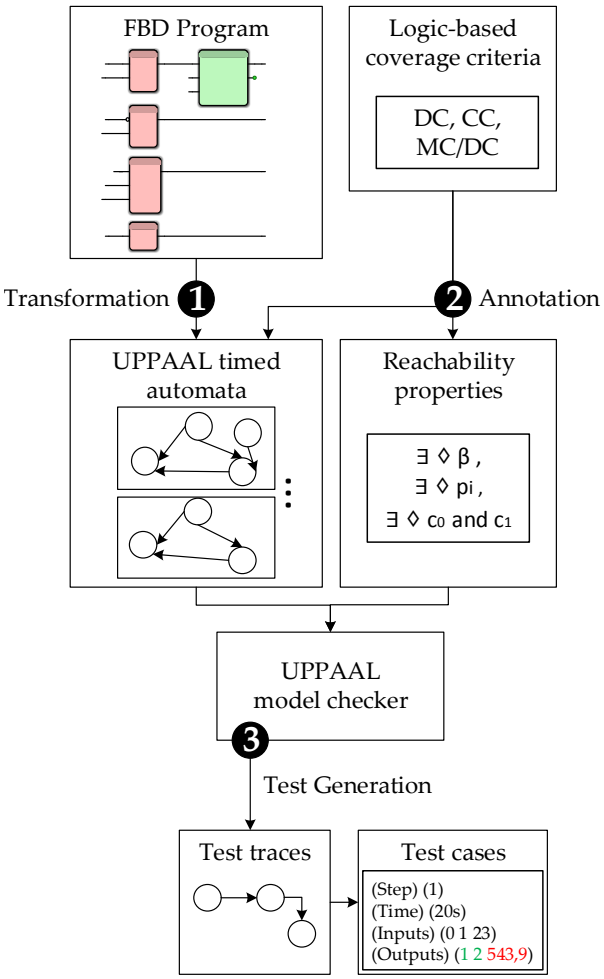


Figure 2.8: Testing Methodology Roadmap.

the FBD program via `read!`, `execute!`, and `write!` actions. This corresponds to synchronization actions implemented in UPPAAL as a hand-shaking synchronization: two automata take a transition at the same time, one will have an a! and the other an a?, a being the synchronization channel.

Let us assume the generic timed automata network of the Compressor Start Enable program together with its cycle scan (`plcSupervision()`) and Input/Output models shown in Figure 2.9. A trace produced by the model checker for a given reachability property defines the set of actions executed on the Compressor Start Enable program which in our case is considered the system model sys. An example of a diagnostic trace has the following form:

$$(sys_0) \xrightarrow{a_1} (sys_1) \xrightarrow{a_2} ... \xrightarrow{a_n} (sys_n),$$

where (sys_k) are states of the FBD program and PLC supervision with input environment constraints, respectively, and a_k are either internal synchronization actions, time-delays or `read!`, `execute!`, and `write!` global synchronizations. For FBD programs, the sequence represents only the global synchronizations shown in Figure 2.9. Test cases are obtained by extracting from the test path the observable

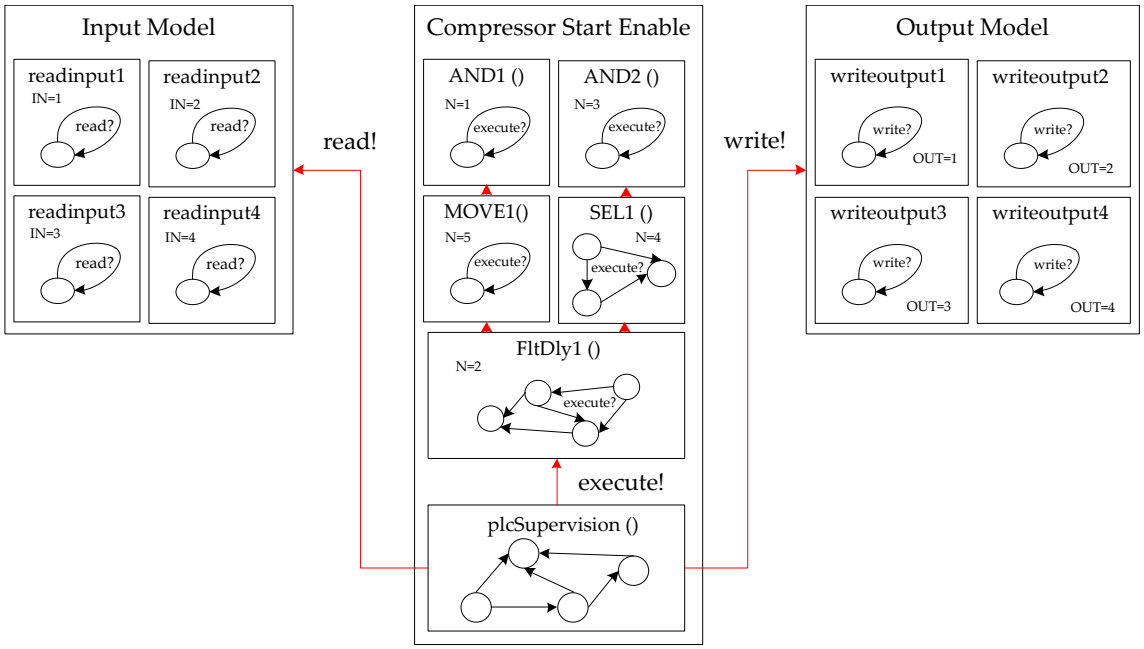


Figure 2.9: Timed Automata Network of the Compressor Start Enable Program.

actions *read!*, *execute!*, and *write!*. Obviously all the test obligations cannot be satisfied by a single test case. By using a scan cycle we allow the test to be implemented as one or more paths separated by resets. To introduce resets in the model, we annotate the cycle scan with a reset transition leading to the initial `ReadInputs` location. On this transition all variables and parameters (excluding encoded internal variables) are reset to their default value. This reset is hardcoded into the PLC supervision for any modeled FBD program in UPPAAL, being an atomic communication between all timed automata.

5 Analyzing Logic Coverage

The basic approach to generating test cases for logic coverage using model-checking is to define a test as a finite execution path. By characterizing a logic coverage criterion as a temporal logic property, model-checking can be used to produce a path for the test obligation.

Ammann et al. [6] argued that criteria such as logic coverage that have constraints involving more than one test trace cannot be handled in this way. The core problem is that each execution is characterized by a temporal formula, and test obligations span multiple runs of the model checker. This means that to ensure model-checking of MC/DC test obligations one should satisfy constraints on multiple runs of the model-checker. However, an FBD program has an implicit control loop, so a reset transition can occur in the program without modifying the transformed timed automata in any way. This reset transition restores the program to its initial state, making it possible to handle test obligations over multiple program executions as a single execution path containing subpaths separated by resets.

By using a translated FBD program, we use logic coverage to directly annotate both the model and the temporal logic property to be checked. We propose the annotation with auxiliary data variables and transitions in such a way that a set of paths can be used as a finite test sequence. In addition, we propose to describe the temporal logic properties as logic expressions satisfying certain logic coverage criteria. Informally, our approach is based on the idea that to get logic coverage of a specific program, it would be sufficient to (*i*) annotate the conditions and decisions in the FBD program, (*ii*) formulate a reachability property for logic coverage, and (*iii*) find a path from the initial state to the end of the FBD program. To apply the criteria, necessary properties for the integration of logic coverage need to be fulfilled.

For each criterion, model checking allows the generation of paths for logical predicates showing test obligations satisfaction. To do so, conditions and decisions have to be formulated as temporal logic formulae. Hessel et al. [15] proposed one way to apply coverage criteria to specifications described in timed automata. We extend this approach to apply it to the conditions and decisions in an FBD program.

Decisions in an FBD program are blocks that can be evaluated to a Boolean value, i.e., true or false. Decisions can be identified from the instrumentation points in the FBD program (e.g., AND block). Let $\{d_i\}$ be the set of decisions in an FBD program and $\{c_{ij}\}$ be the set of conditions in d_i.

DC requires every d_i to evaluate to true and false, and is described by the following two test obligations:

$$o_1 = d_i$$

$$o_2 = \neg d_i$$

These obligations guarantee that each decision d_i evaluates to both true and false, not necessarily along the same execution path.

CC requires two test obligations for each clause c_{ij} in a decision d_i, such that c_{ij} evaluates to both true and false:

$$o_1 = c_{ij}$$

$$o_2 = \neg c_{ij}$$

MC/DC imposes two requirements for test cases. First, for each condition c_{ij} in a decision d_i, test cases must show that c_{ij} determines the value of decision d_i, and second, c_{ij} has to evaluate to true and false. As shown in [4], a condition c_{ij} determines a decision d_i if there is an assignment of values to all the variables in d_i except c_{ij} such that the value of d_i is different for the two values of c_{ij}. This requirement is met if the following logical expression is satisfied [3]:

$$d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)}$$

Combining the two requirements for MC/DC coverage, we have the following two test obligations:

$$o_1 = c_{ij} \wedge (d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)})$$

$$o_2 = \neg c_{ij} \wedge (d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)}).$$

[3] $d_{i(c_{ij},v)}$ denotes d_i with c_{ij} replaced with v.

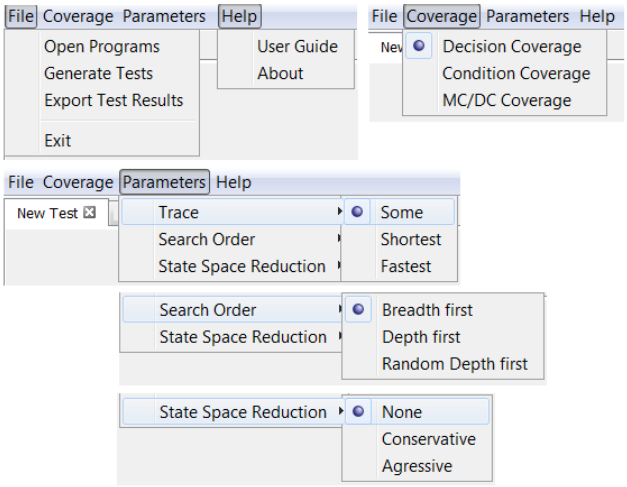


Figure 2.10: User Menu of the Toolbox

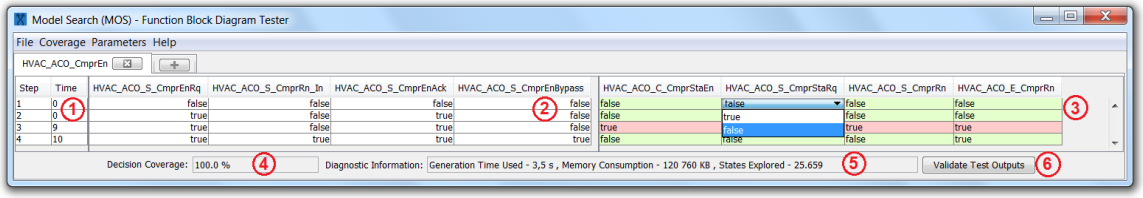


Figure 2.11: Graphical Interface of the Toolbox

For generating tests for DC, CC, and MC/DC we represent the test obligations over a set of variables monitoring the decisions and conditions as a reachability property. This approach is implemented in the toolbox by automatically creating a temporal logic property used by the model checker to produce tests.

6 Overview of the Toolbox

In this section we outline some of the main aspects of the toolbox, including the user interface and the architecture. We also present several technical solutions used in its implementation to fully support the complexity required for model-checking while at the same time presenting a clean and minimal user interface.

6.1 User Interface

The main goal for the design of the user interface was to meet the exact needs of an industrial end user. Although there is a possibility for fine tuning the configuration parameters of the underlying UPPAAL model-checker, most of them are set to default values, making the toolbox immediately ready for use upon startup. Figure 2.10 depicts menu options for the toolbox, listing chosen default values for the parameters and the coverage criteria.

Use-Case Scenario 1: Basic Test Generation

A very basic use-case scenario to get started with the toolbox would consist of:

1. Opening an FBD Progam XML file (File → Open FBD Programs)
2. Generating tests (File → Generate Tests)

These actions cause the tool to attempt to generate a set of test cases that cover all of the decisions. The attempt continues until either all decisions have been covered, or the tool has run for 10 minutes even if there are decisions still uncovered. We found that pragmatically, when the toolbox is applied to FBD programs produced at Bombardier Transportation AB, the model checker has been able to generate tests in 0.05 to 133 seconds. Figure 2.11 depicts an output of the toolbox for this use-case scenario executed on our running example (as defined in Section 2). The figure shows several types of information presented to the user in a table with the test data (points 1,2,3 in the figure), and a set of additional information and actions (points 4, 5 and 6 in the figure). The numbered points in the figure are:

1. Steps and Timing information regarding when the specific test data is provided to the running FBD program.
2. Generated test input data needed to achieve a maximum coverage of the given program.
3. Editable area of the test output data, where the user can provide expected outputs for a specific set of test inputs based on a defined behavior in the requirement. To maintain efficient use of space in the toolbox, expected values for test outputs are provided in the form of a drop-down selection list for boolean values (true/false) or as a text field for other non boolean values (integers, doubles, etc.).
4. Percentage of the logic coverage achieved by using generated tests.
5. Diagnostic information with respect to the time spent on generating tests, memory usage and size of the state space.
6. Optional action to compare expected values with computed ones. Invoking the ”Validate Test Items” button causes the entries in section 3 of the test data table to be colored with green where the expected value matches the computed one, and with red where there is a mismatch. Any subsequent updates to the expected values will automatically update the coloring of that entry.

Table 2.1: Test inputs generated for Decision Coverage (DC) and Condition Coverage (CC) on the running example. In order for decisions to achieve a certain state, test inputs have to be provided for several time units due to the usage of a timer.

Logic Coverage Criteria	Step	Time	HVAC_ACO_S_CmprEnRq	HVAC_ACO_S_CmprRn_In	HVAC_ACO_S_CmprEnAck	HVAC_ACO_S_CmprEnBypas
DC	1	0	false	false	false	false
	2	0	true	false	true	false
	3	9	true	false	true	false
	4	10	true	true	true	true
CC	1	0	false	false	false	false
	2	0	true	true	true	true

Use-Case Scenario 2: Selecting A Logic Coverage Criterion

Tests generated using *Use-Case Scenario 1* aim at achieving maximum decision coverage. If a user would like to use a logic coverage measurement other than the default decision coverage (DC), this can be selected from the "Coverage" menu. Table 2.1 presents test inputs for the running example when the toolbox is using both decision and condition coverage. Since the running example includes a timer function block (`FltDly`), achieving maximum decision coverage is possible only if we provide test inputs for a certain number of time units. In the running example, the `FltDly` function block expects an input value to be *true* for at least 10 seconds. This is why inputs to the program are set to *true* in steps 2, 3 and 4 for decision coverage representing the state of the system at *time=0*, *time=9* and *time=10*. Since there are no observable changes in the way the system behaves between *time=1* and *time=8*, the toolbox does not display those test steps. For an industrial user, this minimization of test steps is very important, because it saves manual effort in providing expected output values for the system under test.

Use-Case Scenario 3: Changing Configuration Parameters

In addition to the basic use-case scenario of the toolbox, a user can perform various configuration changes to the way tests are obtained. This is done

Table 2.2: Manual fault discovery by checking the output (no negated input signal for the AND block in Compressor Start Enable Program). When generating tests with DC for a faulty program, the Compressor Start Request signal will indicate an erroneous false status when the Compressor is not running and there is a request for enabling the compressor.

Step (number of tests)	Time	HVAC_ACO_S_CmprEnRq	HVAC_ACO_S_CmprRn_In	HVAC_ACO_S_CmprStaRq
1 (Original Program)	0	false	false	false
2	0	true	false	true
3	9	true	false	true
4	10	true	true	false
1 (Faulty Program)	0	false	false	false
2	0	true	false	true
3	9	true	false	true
4	10	true	true	false

by modifying the model-checker's settings in the "Parameters" menu of the tool-box. For example, the user can set the search algorithm to be "Breadth First", and/or set the output trace of the model-checker to a "Fastest" one, etc.

Use-Case Scenario 4: Fault Detection in FBD Programs

This example compares the expected values and computed values produced by the program. We created a typical fault in the Compressor Start Enable program, by removing the negated input for the AND block corresponding to the compressor running (`HVAC_ACO_S_CmprRn_In`). Then we generated tests that satisfy DC for both the original program (assumed to be correct) and the faulty program, as shown in Table 2.2 (only three signals are shown because these are the inputs that affect the output). For the original program we observe that the specification described in Section 2.2 agrees with the actual output and therefore in all cases (step 1-4) the output is green. Now by examining the output of the faulty program, the user can determine that the ventilation request is not started (`HVAC_ACO_S_CmprStaRq`) when the compressor is enabled (`HVAC_ACO_S_CmprEnRq`) and the compressor is not running (`HVAC_ACO_S_CmprRn_In`), revealing a bug in the program.

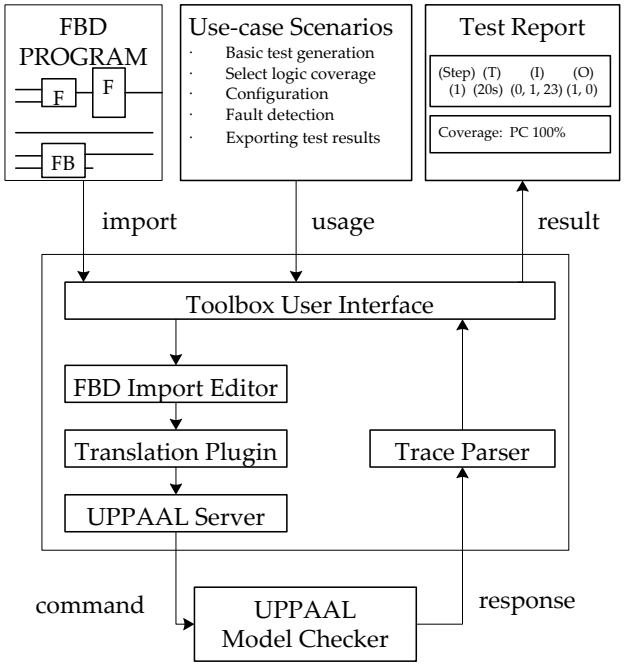


Figure 2.12: Overview of the Toolbox Architecture.

Use-Case Scenario 5: Exporting Test Results

As a final use-case scenario of the toolbox, a user can export the resulting tests in a comma separated values (CSV) format by selecting "File → Export Test Results". In this case, all the information from the test data table is saved, including both computed and expected (i.e., user provided) output values. Such data could be used outside of the toolbox for creation of a custom test report.

6.2 Toolbox Architecture

An overview of the toolbox architecture is presented in Figure 2.12. The actual toolbox was developed as a Java Swing application using the NetBeans integrated development environment and following a modular approach in the design of the toolbox architecture. This resulted in the following modules being part of the toolbox:

- **FBD Import Editor.** This module is used for validating whether the structure of a provided XML file represents a valid PLCOpenXML file containing an FBD Program.
- **Translation Plugin.** Once the FBD Import Editor module has been executed, the PLCOpenXML file containing the FBD Program is translated into an XML-format accepted by the UPPAAL model checker. This translation is carried out by following the rules of translation defined in Section 3.
- **UPPAAL Server.** The UPPAAL Server module is used for external invoking of the UPPAAL model checker. UPPAAL provides support for formal verification

using a client-server architecture, allowing the toolbox to connect as a client to the model checker and verify properties against the model.

- **Trace Parser.** The Trace Parser toolbox module collects diagnostic trace output from the UPPAAL model checker and parses this output into a JavaCC structure corresponding to a set of inputs and outputs for a given model. This parsing mechanism is further explained in Section 6.5.

- **User Interface.** The function of the user interface is to provide a way for the user to communicate with the tool including: (1) the selection of which FBD program to import and generate tests for, (2) the selection of the coverage criterion to be used for test generation, (3) the presentation of generated test inputs, and (4) the determination of 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).

6.3 PLCOpen XML Standard

The PLCOpen XML interchange format for PLC applications is the base for the model translation to timed automata. PLCOpen is a vendor independent standard aiming to provide a common programming interface for the use of the IEC 61131-3 standard. In the toolbox, the XML file used as input for the translation to timed automata is in accordance to the PLCOpen standard defining the FBD programming language. Figure 2.13 depicts an example of a PLCOpen XML file corresponding to the Compressor Enable Program. The program consists of specific XML elements consisting of the program name (lines 5), the interface information (lines 6-20), and the block specification for `AND` and `FltDly` (lines 22-53). The XML scheme is mainly storing program information such the *identifier* for blocks and *dependencies*. As shown in Figure 2.13, `localId` indicates the identifier of a block, and every `refLocalId` in the `connection` tag represents the dependency identifier for the connection to a certain block or input variables. This structural format is used in the implemented translation from FBD to timed automata.

6.4 Implemented Model Translation

We define a translation inside the toolbox, which consists of the formal definition of the FBD language. A program consists of the following elements: composite programs, basic blocks, library blocks, connections, ports, and timing constraints.

The toolbox considers that each modeling element, except for the composite programs, has a set of ports through which it can exchange data. Ports are associated by a set of data types, which are used for data representation, e.g., integer with a specific range. A Port is associated with the same type of data as the associated internal variable.

For an FBD program the read-execute-write semantics means that input ports may only be accessed at the beginning of each computation, and output ports are only written at the end of the computation. Therefore, the behavior is augmented

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="www.plcopen.org/xml/tc6.xsd">
<types><dataTypes/>
<pous>
  <pou name="HVAC_ACO_CmprEn" pouType="fBlock">
    <interface>
      <inputVars retain="false">
       <variable name="HVAC_ACO_S_CmprEnRq">
         <type><BOOL/></type>
       </variable>
       <variable name="HVAC_ACO_SCmprRnIn">
         <type><BOOL/></type>
       </variable>
      </inputVars>
      <outputVars retain="false">
       <variable name="HVAC_ACO_C_CmprStaEn">
         <type><BOOL/></type>
       </variable>
      </outputVars>
    </interface>
    <body><FBD>
      <block typeName="AND" localId="11">
        <inputVariables>
         <variable formalParameter="IN1"
                           negated="true">
           <connection refLocalId="14"></connection>
         </variable>
         <variable formalParameter="IN2"
                           hidden="true">
           <connection refLocalId="13"></connection>
         </variable>
        </inputVariables>
        <inOutVariables/>
        <outputVariables>
         <variable formalParameter="OUT"
                           hidden="true">
         </variable>
        </outputVariables>
      </block>
      <block typeName="FltDly" localId="66">
        <inputVariables>
          <variable formalParameter="IN">
            <connection refLocalId="18"
                    formalParameter="OUT">
            </connection>
          </variable>
          <variable formalParameter="PT">
            <connection refLocalId="21"/>
          </variable>
          <variable formalParameter="ENABLE">
            <connection refLocalId="22"/>
          </variable>
        </inputVariables>
        <inOutVariables/>
        <outputVariables>
          <variable formalParameter="FLT"></variable>
          <variable formalParameter="BLK"></variable>
        </outputVariables>
      </block>
    </FBD></body>
  </pou>
</pous>
</types>
<instances><configurations/></instances>
</project>
```

Figure 2.13: PLCOpen XML format for the Compresor Enable Program

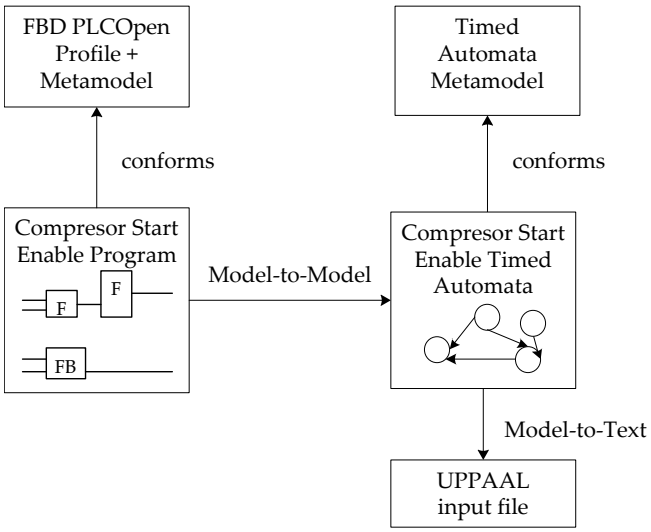


Figure 2.14: Model Export from an FBD Program to UPPAAl Model Checker.

with an external interface. The interface of a block consists of ports and the execution order information. An input port has an associated variable holding the current data values. The internal computation of a block starts with reading all input ports. This internal data is used together with the behavioral model during execution, before writing the variables to the output ports.

We have developed the model transformation shown in Figure 2.14. In order to simplify the semantics of an FBD program, we focus on the PLCOpen language constructs relevant to functional and timing modeling elements.

The PLCOpen language is implemented as an XML profile that provides the ability to describe FBD programs using this profile. The PLCOpen language provides both structural and graphical information needed for implementing the actual translation. The toolbox generates PLCOpen files in an XML format. As shown in Figure 2.14, we introduce the timed automata as the interface between the FBD program and the UPPAAL input model. The Compressor Start Enable Program conforms to the PLCOpen profile and meta-model. The structural translation described in Section 3 maps an FBD program into timed automata. The structure of the timed automata model is the basis of the model to text transformation into the UPPAAL input model.

The modeling elements of an FBD program used in the translation are described in Figure 2.15. These elements represent the structure of the model, the behavior, and the timing information. The meta-model elements provide concepts used in component based design. A `Block` element can be translated with `Type`, `ExecutionOrder` and `Model` elements. Blocks can be composed using connections and ports. Furthermore, a Block element can have a behavioral description as a Model element. The model provided after the translation represents the model annotated with triggering and timing information with assumed functionality.

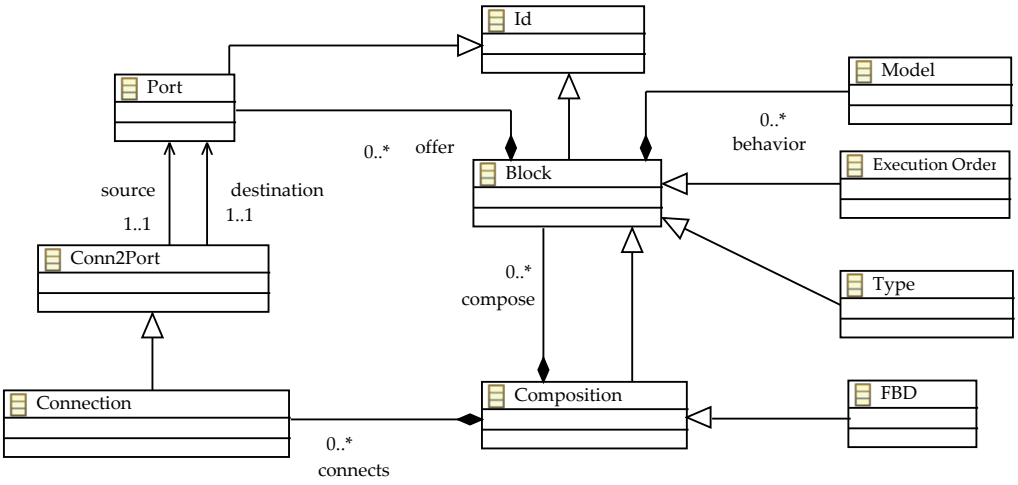


Figure 2.15: Class Diagram representing the meta-model elements of the Function Block Diagram.

6.5 Dynamic Traces - JavaCC - Test Cases

UPPAAL model-checking tool is mainly used for the verification of a certain property of a model, resulting in a affirmative or a negative response. However, it is also possible to obtain a full trace used in the process of verifying that property on a model. An excerpt of such a trace for the running example is shown in Figure 2.16. To interpret dynamic traces generated by UPPAAL, a grammar file was created for JavaCC[4] parser generator. The trace starts with the initial state and is followed by pairs of transitions and states, i.e. the state can be reached from the previous state via the transition. A state in the trace contains locations (lines 3-9), clocks (line 12), internal variables (lines 12-20), decisions and conditions (lines 22-25) in the same order as they appear in the UPPAAL input file. The trace parsing using JavaCC is the process of analyzing the trace, transforming the trace into a state machine, extracting the necessary information (i.e., values of the input and output variables, clock valuation) needed for testing of an FBD program. In the end tests are merged based on the program cycle scan as one or more test cases separated by resets.

7 Experimental Evaluation and Discussions

Our goal in this section is to evaluate the toolbox on industrial FBD programs and to acquire experience regarding its efficiency and usability. We therefore conduct a set of analyses using programs developed by Bombardier Transportation AB in Sweden. The system has been in development for more than two years and uses processes influenced by safety-critical requirements and regulations including the EN 50128 standard [7] which requires different logic coverage levels (e.g., DC and MC/DC). In 2014 its source code was made up of more than 350.000 lines of C code generated from FBD programs. The development teams use both automated and manual testing from unit testing through system testing.

[4]The JavaCC™is available at `https://javacc.java.net/`.

```
State                                                                          1
(                                                                              2
plc.ExecuteProgram readinput1.Process                                          3
readinput2._id10 readinput3.Process                                            4
readinput4._id12  and1.Update                                                  5
and2._id18 fltdly1.Waiting                                                     6
sel1._id15  move1._id16                                                        7
writeoutput1._id5 writeoutput2._id6                                            8
writeoutput3._id7 writeoutput4._id8                                            9
)                                                                              10
                                                                               11
fltdly1.ET<=0 steps=1 HVAC_ACO_S_CmprEnRq=1                                    12
HVAC_ACO_S_CmprRn_In=1 HVAC_ACO_S_CmprEnAck=0                                  13
HVAC_ACO_S_CmprEnBypass=1 HVAC_ACO_C_CmprStaEn=0                               14
HVAC_ACO_S_CmprStaRq=0 HVAC_ACO_S_CmprRn=0                                     15
HVAC_ACO_E_CmprRn=0                                                            16
                                                                               17
AND1=0 AND2=0 FltDly1=0 SEL1=0 MOVE1=0                                         18
                                                                               19
N=1 IN=5 OUT=1                                                                 20
                                                                               21
decisions[0]=0 decisions[1]=0 decisions[2]=0                                   22
decisions[3]=0 decisions[4]=0 decisions[5]=0                                   23
decisions[6]=0 decisions[7]=0 decisions[8]=0                                   24
decisions[9]=0                                                                 25
                                                                               26
fltdly1.counter=0 move1.firstTime=0                                            27
move1.RS_local=0 move1.decision=0                                              28
                                                                               29
Transitions:                                                                   30
 plc.ExecuteProgram->plc.UpdateOutputs                                         31
{ IN == InputVariables, execute!, 1 }                                          32
                                                                               33
and1.Update->and1.Update                                                       34
{ !(HVAC_ACO_S_CmprEnAck && HVAC_ACO_S_CmprEnRq) &&                            35
 N == 1, execute?, AND1 := 0, N++, decisions[0] := 1                           36
}                                                                              37
```

Figure 2.16: An excerpt of a trace in response to a command to UPPAAL for the Compressor Enable Program.

We investigate the following questions regarding the tool's performance:

- *Q1, Efficiency:* What is the time required for the tool to generate tests that satisfy the DC, CC and MC/DC logic coverage criteria?

- *Q2, Coverage:* How close does the tool come to generating tests that achieve 100% coverage of each of the criteria?

The industrial system studied in this paper is the TCMS (Train Control and Management System), developed by Bombardier Transportation AB engineers, which has been deployed to the field. In this research we, have used all TCMS programs written in the FBD standard language resulting in a total of 157 artifacts. Each of the programs is sizable and representative of industrial programs used in the train system's development. Information regarding the size of the system and number of blocks is provided in Table 2.3.

For each program, the tool generated a model version in UPPAAL. Then, for each implementation of a program, the toolbox:

Table 2.3: Information about the 157 subject programs.

	Blocks	Inputs	Outputs	Decisions
Maximum per Program	32	15	29	196
Average per Program	6.9	2.7	5.9	30

Table 2.4: Average, median, minimum, and maximum generation times for 123 of the 157 programs.

	CC	MC/DC	DC
Average Generation Time (s)	1.53	4.54	1.93
Median Generation Time (s)	0.27	0.51	0.34
Minimum Generation Time (s)	0.05	0.06	0.06
Maximum Generation Time (s)	35.37	133.60	72.125

- *Generated test input vectors for three different coverage criteria.* We used a reachability-based approach for generation of tests aimed at satisfying DC, CC and MC/DC. If the model checker is able to find a path to satisfy a reachability property, given that such a path exists, then the approach is guaranteed to generate a test suite that achieves maximum possible coverage of the program.

  Hence, if the model checker succeeds in finding paths to satisfy all the reachability properties for a given criterion, then the method will achieve 100% coverage for that criterion. We have used the Uppaal model checker in our experiments. Our reachability-based test generation approach produces one test for each coverage criterion as our goal is to assess the coverage and efficiency of the toolbox in terms of time to generate tests. To generate the tests, the tool uses the random-depth first search algorithm provided by the Uppaal model checker. The tool terminates the generation by determining the coverage requirements satisfied by each test.

- *Assessed efficiency of each test based on coverage, and collected complexity measures for each program.* We measured the generation time for each program and determined the number of test requirements for each coverage criterion.

To answer Q1 and Q2, the tool generate tests aimed at achieving maximum logic coverage. Since we are using a model checker for generating tests, the toolbox simply produces the maximum achievable coverage with a proof that uncovered test obligations are not coverable. For 123 of the 157 programs (78%) the tool provided tests that covered 100% of the required entities for each of the three coverage criteria. Table 2.4 gives the performance figures in terms of time needed to generate the tests. The generation time for MC/DC averaged approximately twice as long as for DC. The results are summarized as boxplots in Figure 2.17 with the kernel density distribution of the generation time shown in Figure 2.18. The kernel densities estimates for the generation time for DC (red), CC (green) and MC/DC (blue) are plotted on the same graph. It is quite clear on the graph that the distribution of generation times is more variable for MC/DC. It is also worth noting that the generation time modes

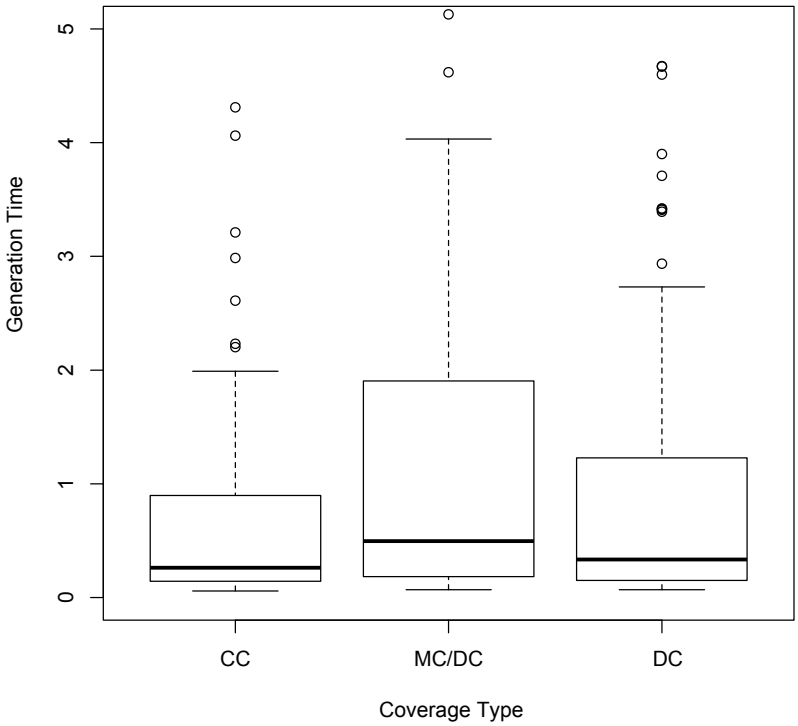


Figure 2.17: Experimental results: Generation Time Distributions.

(i.e., most frequent values in the generation time data set) of a distribution are close to each other for all criteria. We can observe that a few outliers caused the average generation time to greatly exceed the median generation time for all coverage criteria.

For 34 of the 157 programs, the tool did not terminate after running for a substantial period of time. After discussions with engineers from Bombardier Transportation AB regarding the needed time for a tester to provide a set of tests for a desired coverage, we concluded that 10 minutes was a reasonable cut-off point for the model checker to terminate its search. Recall, however, that the aim of these experiments was not to provide measures of test effectiveness in the sense of bug-finding, but instead to evaluate the applicability of using a model checking technique for test generation and its success in meeting coverage requirements. We wanted to work with a realistic cut-off time that could be used in practice if this approach is to be adopted. Therefore, in each case a run of the model checker was terminated after 10 minutes.

As noted above, for 22% of the programs in this study, the tool did not generate the required test suite in an acceptable period of time. To determine the circumstances under which the toolbox does or does not successfully generate test suites that satisfy one of the logic coverage criteria (Q2) we collected the average number of decisions for both the case when the model checker finishes its execution (Case 1) and the case when we forcefully terminated the tool because the running time reached 10 minutes (Case 2). Table 2.5 provides information about these two cases. Case 1 consists of the 78% for which the tool generated tests achieving 100% DC, CC and MC/DC. The number of decisions for Case 1 ranged from 1 to 22 with the average being 5. In contrast, for Case 2, the set of programs for which the tool exceeds the allocated time

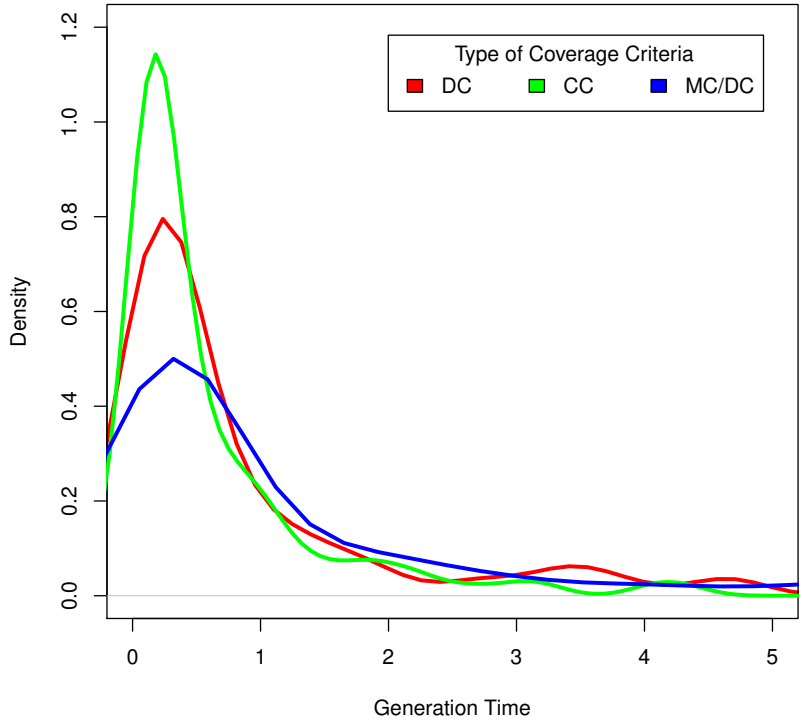


Figure 2.18: Generation Time Distribution by Coverage Criteria.

before generating a test set satisfying the coverage criterion, the decisions ranged from 12 to 196 with the average being 38. This indicates that as the number of decisions increases, the performance deteriorates and the cost of using the tool may become prohibitive. This factor contributes to a scalability issue which results in longer test sequences, especially when generating tests for MC/DC.

It is important to note that during model-checking the reachability-based generation used by the toolbox is guided to achieve a desired coverage, and not to minimize or optimize the test. A generated test may not be the minimal way to satisfy the coverage criterion. However, a generated test might be able to satisfy more than one test obligation. From the point of view of limiting the number of tests generated, we note that our approach would perform better than other approaches including trap property generation [24, 13], which can lead to a large number of duplicate tests because these properties are derived by using the model-checker's ability to generate counterexamples.

Engineers from Bombardier Transportation AB indicated that their certification process involves achieving a minimum of 80% DC for all programs. For 78% of the programs in this study, the tool automatically generated tests achieving 100% DC, CC and MC/DC. For the other 22% of the programs, the results were less satisfactory. The data about the achieved coverage is shown in Table 2.5. As can be seen from this data, the tool generated tests with 82% DC on average. We conclude that we have provided evidence that this is a suitable tool for test generation tailored to FBD programs; it scaled well for most of the programs in this study and it is fully automated. There are, however, some drawbacks. Most importantly, for 22% of the programs, even though the tests generated for the coverage criteria achieved

Table 2.5: Achieved coverage for all Programs.

Case	1	2
Percentage of all Programs	78%	22%
Average DC Achieved	100%	82%
Average CC Achieved	100%	88%
Average MC/DC Achieved	100%	65%

on average at least 65% coverage, we cannot determine whether the remaining test requirements are actually achievable, or if tests satisfying the requirements are longer that the search depth. This is an issue particularly for MC/DC where a fair number of test obligations were not satisfied.

From these experiments, it is clear that the toolbox can be sensitive to the number of decisions and as a consequence to the length of the tests required to achieve the desired coverage. In addition, the number of inputs considered during model checking is affecting the efficiency of the test generation technique. However, model checking does allow one to use a heuristic or meta-heuristic search technique [5, 25] to find the desired tests. We plan to investigate this approach in future work. In addition, the idea of combining symbolic execution or static analysis with model checking to achieve test generation has been proposed [19], and may allow more efficient model checking. Fraser et al. [12] noted that there is a lack of empirical evidence on how these model-checking techniques compare to each other in practice, making it hard to select an appropriate technique for a specific test purpose. We also plan to investigate how various approaches compare in future work.

8 Related Work

Model checkers have been used to produce test cases satisfying various criteria and for programs in a variety of formal languages [6, 16, 10]. Black et al. [6] discuss the problems encountered in using a model-checker for test case generation for full-predicate coverage. They present reasons why model-checking is not directly applicable for generating tests to satisfy logic coverage criteria. In our previous work [11], we overcome this issue by providing a way of generating test cases for logic criteria that are directly applicable to FBD programs. We found that model-checkers are an appropriate technique for automated test generation in terms of performance when used on real-world programs.

For data-flow programming languages such as FBD and Lustre, which describe the relationship between inputs and outputs instead of the control flow of the program, researchers proposed specific coverage metrics based on the structural aspects of the programs [18, 17, 20]. For Lustre, structural coverage metrics are based on the activation condition concept of the language that can be used when data travels from an input edge to an output edge. In addition, Whalen et al. [27] defined an alternative approach to measuring logic coverage for data flow programs called OMC/DC, a combination of MC/DC and an additional obligation to be satisfied such that faults will be observed through a variable monitored by the criteria.

9 Conclusion

In this paper we have shown how test case generation that aims to satisfy logic coverage on Function Block Diagrams can be solved as a model checking problem, by using model checking tools to automatically create traces that can be transformed into executable tests. We described a toolbox in which logic coverage criteria can be formalized and used by a model-checker to generate test cases. We carried out an extensive empirical study of the method by applying the toolbox to 157 real-world industrial programs developed at Bombardier Industries. The results showed that model checking is suitable for handling logic coverage for real-world FBD programs, and also revealed some potential limitations of the toolbox when used for test generation. The evaluation showed that the toolbox is efficient in terms of time required to generate tests that satisfy logic coverage and that it scales well for most of the programs. Our overall conclusion is that the model-checking approach provides a positive and useful addition to the testing process for FBD programs.

Acknowledgments

This research was supported by VINNOVA, the Swedish Governmental Agency for Innovation Systems within the ATAC project and The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC).

Appendix: Networks of Timed Automata

Let C be a finite set of reall-valued clocks and $B(C)$ the set of clock constraints, which are finite conjunctions of atomic guards of the form $x \bowtie n$, where $x \in C$, n is a natural number, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

A *timed automaton* (A) over actions $\mathcal{A}$, atomic propositions P and clocks C is a tuple $\langle N, l_0, E, I, V\rangle$ where N is a finite set of control locations, l_0 is the initial location, $E \subseteq N \times B(C) \times \mathcal{A} \times R^5 \times N$ is the set of edges. In the case of an edge $\langle l, g, a, r, l'\rangle \in E$, we write $l \xrightarrow{g,a,r} l'$ where the label g is a guard of the edge, r is the data- or clock reset assignments of the edge, and a is the action of the edge. $I : N \to B(C)$ is a function which for each control location assigns an invariant condition and $V : N \to 2^P$ is a function which for each control location provides a set of atomic propositions that are true in the location.

The semantics of A is defined in terms of a state transition system, where the state of A is defined as a pair (l, u), where l is a location and $u \in \mathbb{R}^C$ is a clock assignment in C. A state of A depends on its current location and on the current values of its clocks.

A state of an automaton A is defined as a pair (l, u), where l is a location, $u \in \mathbb{R}^C$ a clock assignment in C to a value in $\mathbb{R}_+$, with the initial state (l_0, u_0), where u_0 assigns all clocks in C to zero.

[5] R denotes the reset set i.e., assignments to manipulate clock- and data variables.

The semantics of A is given by the timed transition system $\langle S, s_0, E, V \rangle$, where S is the set of states of A, s_0 is the initial state (l_0, u_0), E is the transition relation defined as follows:

- $(l, u) \xrightarrow{d} (l, u \oplus d)$, where $u \oplus d$ is the result obtained by incrementing all clocks of the automata with the delay amount d such that for any $0 \leq d' \leq d$, the invariant of l holds.

- $(l, u) \xrightarrow{a} (l', u')$, corresponding to taking an edge $l \xrightarrow{g,a,r} l'$ for which the guard g is satisfied by u. The clock valuation u' of the target state is derived from reseting u according to updated r.

We denote by $T(A)$ all traces σ of A starting from the initial state (l_0, u_0) as a sequence of alternating transitions $\sigma = (l_0, u_0) \xrightarrow{a_1} (l_1, u_1) \xrightarrow{a_2} ... \xrightarrow{a_n} (l_n, u_n)$.

A network of timed automata $B_0 \parallel ... \parallel B_{n-1}$ is a parallel composition of n timed automata over C, $\mathcal{A}$ and synchronization functions (i.e., $a!$ is correlative with $a?$). We refer the reader to [1] for more information on the theory of timed automata. We consider a timed modal logic to specify properties. The logic may be seen as properties of A than can be expressed as logical formulae in the Timed Computational Tree Logic (TCTL) [2].

References

[1] R. Alur. "Timed Automata". In: *Computer Aided Verification*. 1999, pp. 688–688 (cit. on pp. 59, 84).

[2] R. Alur, C. Courcoubetis, and D. Dill. "Model-checking in Dense Real-time". In: *Information and computation*. Vol. 104. 1. Elsevier, 1993, pp. 2–34 (cit. on pp. 58, 84).

[3] R. Alur and D. Dill. "Automata for Modeling Real-time Systems". In: *Automata, languages and programming*. Springer, 1990, pp. 322–335 (cit. on pp. 54, 55, 57).

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008 (cit. on pp. 59, 68).

[5] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G Larsen, Paul Petterson, and Judi Romijn. "Guiding and Cost-Optimality in UPPAAL". In: *Spring Symposium on Model-based Validation of Intelligence*. AAAI, 2001, pp. 66–74 (cit. on p. 82).

[6] 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 pp. 53, 67, 82).

[7] CENELEC. "50128: Railway Application–Communications, Signaling and Processing Systems–Software for Railway Control and Protection Systems". In: *Standard Report*. 2001 (cit. on pp. 54, 77).

[8] John Joseph Chilenski and Steven P Miller. "Applicability of Modified Condition/Decision Coverage to Software Testing". In: *Software Engineering Journal*. Vol. 9. 5. IET, 1994, pp. 193–200 (cit. on p. 59).

[9] 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. 55).

[10] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. "Model-based Test Suite Generation for Function Block Diagrams using the UPPAAL Model Checker". In: *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2013, pp. 158–167 (cit. on p. 82).

[11] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. "Using Logic Coverage to Improve Testing Function Block Diagrams". In: *Testing Software and Systems*. Springer, 2013, pp. 1–16 (cit. on pp. 54, 82).

[12] 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. 53, 82).

[13] Angelo Gargantini and Constance Heitmeyer. "Using Model Checking to Generate Tests from Requirements Specifications". In: *Software Engineering ESEC/FSE*. 1999, pp. 146–162 (cit. on p. 81).

[14] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. "Testing Real-time Systems using UPPAAL". In: *Formal Methods and Testing*. Springer, 2008, pp. 77–117 (cit. on p. 65).

[15] Anders Hessel, Kim G Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. "Time-optimal Real-Time Test Case Generation using UPPAAL". In: *International Workshop on Formal Approaches to Software Testing*. 2003, pp. 114–130 (cit. on p. 68).

[16] 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. 82).

[17] 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. 82).

[18] E. Jee, J. Yoo, S. Cha, and D. Bae. "A data flow-based structural testing technique for FBD programs". In: *Information and Software Technology*. Vol. 51. 7. Elsevier, 2009, pp. 1131–1139 (cit. on p. 82).

[19] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. "Generalized Symbolic Execution for Model Checking and Testing". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 553–568 (cit. on p. 82).

[20] A. Lakehal and I. Parissis. “Structural Test Coverage Criteria for Lustre Programs”. In: *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems*. 2005, pp. 35–43 (cit. on p. 82).

[21] 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. 54).

[22] 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 pp. 55, 56).

[23] 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. 53).

[24] 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 pp. 53, 81).

[25] Kevin Seppi, Michael Jones, and Peter Lamborn. “Guided Model Checking with a Bayesian Meta-Heuristic”. In: *Fundamenta Informaticae*. Vol. 70. 1. 2006, pp. 111–126 (cit. on p. 82).

[26] 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 pp. 55, 56).

[27] Michael Whalen, Gregory Gay, Dongjiang You, Mats P. E. Heimdahl, and Matt Staats. “Observable Modified Condition/Decision Coverage”. In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2013, pp. 102–111 (cit. on p. 82).

Study 3

A Controlled Experiment in Testing of Safety-Critical Embedded Software

Eduard Paul 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 3. A Controlled Experiment in Testing of Safety-Critical Embedded Software

Eduard Paul Enoiu, Adnan Causevic, Daniel Sundmark and Paul Pettersson

Abstract

In engineering of safety critical systems, regulatory standards often put requirements on both traceable specification-based testing, and structural coverage on program units. Automated test generation techniques can be used to generate inputs to cover the structural aspects of a program. However, there is no conclusive evidence on how automated test generation compares to manual test design, or how testing based on the program implementation relates to specification-based testing.

In this paper, we investigate specification— and implementation-based testing of embedded software written in the IEC 61131-3 language, a programming standard used in many embedded safety critical software systems. Further, we measure the efficiency and effectiveness in terms of fault detection. For this purpose, a controlled experiment was conducted, comparing tests created by a total of twenty-three software engineering master students. The participants worked individually on manually designing and automatically generating tests for two IEC 61131-3 programs. Tests created by the participants in the experiment were collected and analyzed in terms of mutation score, decision coverage, number of tests, and testing duration. We found that, when compared to implementation-based testing, specification-based testing yields significantly more effective tests in terms of the number of faults detected. Specifically, specification-based tests more effectively detect comparison and value replacement type of faults, compared to implementation-based tests. On the other hand, implementation-based automated test generation leads to fewer tests (up to 85% improvement) created in shorter time than the ones manually created based on the specification.

1 Introduction

The IEC 61131-3 language [18] is a programming standard for process control software, commonly used for Programmable Logic Controllers (PLCs) in the engineering of embedded safety-critical software (e.g., in the railway and power control domains) [28]. Engineering of this type of systems typically requires a certain degree of certification according to safety standards [4]. These standards pose specific requirements on program testing for both specification-based testing and implementation-based testing (e.g., the demonstration of some level of implementation coverage on the

developed software). Several studies [8, 35, 19, 7] have looked at how to generate test input data achieving high implementation coverage for a domain-specific language like IEC 61131-3. Generally, implementation-based testing techniques automatically generate a set of tests that, when fed to the system under test, systematically exercises the implementation (e.g, covering all branches). However, there is little evidence on the extent to which such techniques contribute to the development of reliable systems. Given that recent work [15] suggests that implementation coverage criteria alone can be a poor indication of testing effectiveness, we seek to investigate the implications of testing safety-critical embedded software. In addition, there is some evidence [14] to suggest that testing is still performed, to some extent, manually by industrial practitioners. In this context, we study the behavior of manual and automated test generation.

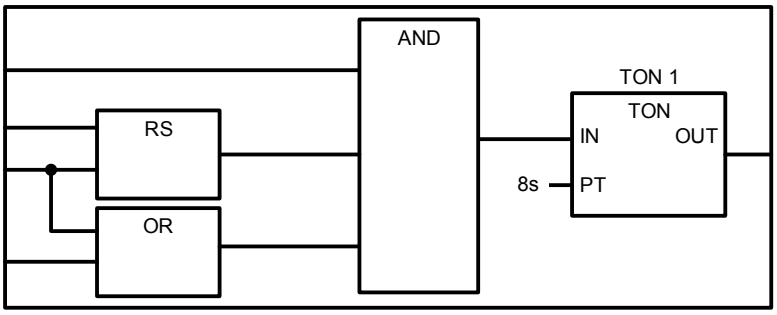
In this paper, we seek to compare the efficiency and effectiveness of testing programs written in IEC 61131-3 by comparing tests manually written by human subjects based on a specification, tests manually written based on the implementation and tests produced with the help of an automated test generation tool named CompleteTest [8] based on the implementation. The research objective can be stated as follows:

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

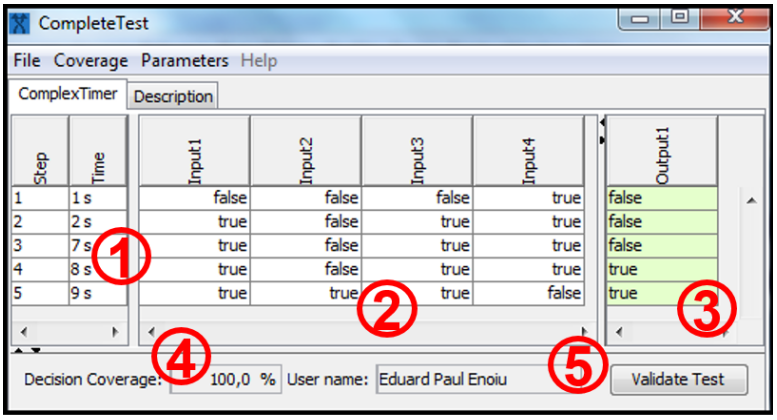
To address this objective, an experiment was conducted with master students enrolled in a software verification and validation course at Mälardalen University during autumn 2014. Twenty-three master level students in software engineering took part as subjects in a controlled experiment. The subjects were given two IEC 61131-3 programs and were asked to construct tests manually based on a specification, and with the help of an automated implementation-based test generation tool. In addition, students were asked to manually create tests for covering the implementation. All tests created during the experiment were analyzed using the following metrics: mutation score, decision coverage, number of tests, and testing duration.

The results of this study show that tests created manually based on a specification are more effective, in terms of fault detection, than tests created based on the implementation either manually or with the help of an automatic test generation tool. Generally, compared to the implementation-based tests, specification-based tests more effectively detect comparison and value replacement type of faults. Our results also show that tests created manually based on the specification perform significantly better than randomly generated tests of equal size (up to 31% more faults found). Additionally, we found that implementation-based automated test generation leads to less costly tests in terms of number of tests and testing duration than either manual specification-based testing or manual implementation-based testing. Finally, implementation-based automated tests perform better, in terms of faults found, than random tests of equal size.

We draw two conclusions from these results. First, specification-based testing does, for the twenty-three study participants and the programs used in the experiment, yield effective tests relative to their size. Second, the use of automatic test generation tools coupled with implementation coverage criteria required by some safety standards



(a) An example of an FBD program showing the graphical nature of the language.



(b) Graphical Interface of CompleteTest

Figure 3.1: The CompleteTest tool, generating tests for an FBD program.

should be carefully studied further.

Our results highlight the need for more research in how different test design techniques for embedded software can influence the efficiency and effectiveness of testing this type of software.

2 Testing PLC Embedded Software

Programmable Logic Controllers (PLC) are real-time systems used in numerous industrial domains, i.e., nuclear plants and train systems. A program running on a PLC [21] executes in a cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs). Function Block Diagram (FBD) [18] is an IEC 61131-3 language, that is very popular in automation industry for programming PLC software [21]. An FBD programmer uses graphical notations and describes the program in a data flow manner. As shown in Figure 3.1a, blocks (e.g., RS, OR, AND and TON) and connections between blocks are the basis for creating an FBD program. These blocks are supplied by the PLC manufacturer, defined by the user, or predefined in a library. An application generator is utilized to automatically translate each FBD program to a compliant executable program

with its own thread of execution.

The motivation for using IEC 61131-3 FBD as the target language in this study comes from the fact that it is the programming standard [21] in many embedded systems, such as PLCs in the railway and power domain. According to a Sandia National Laboratories study [28] from 2007, PLCs are widely used in a large number of industries with a global market of approx. $ 8.99 billion.

2.1 Specification-Based Testing of IEC 61131-3 Software

In testing IEC 61131-3 FBD programs in the railway domain, the engineering processes of software development are performed according to safety standards and regulations [4]. Specification-based testing of FBD programs is mandated by the EN 50128 standard to be used to design tests. This process requires the understanding of both the specified requirements and the FBD program. The specification contains preconditions, input values and expected output values [1]. The tester checks the FBD program conformance with every statement in the specification. Each test should contribute to the demonstration that a specified requirement has indeed been satisfied.

Software specifications can be expressed in a variety of forms from natural languages, semi-formal languages to full formal representations. Recent results have showed that natural language is still the dominant documentation format in embedded software industry for requirement specification [30]. In this experiment we are focusing on specification-based testing using functional specifications expressed in a natural language, as this is a realistic scenario for testing FBD programs.

2.2 Implementation-Based Testing for IEC 61131-3 Software

Coverage criteria are an implementation-based means of assessing the thoroughness of tests. They are normally used at the unit test level to manually or automatically create tests that exercise different aspects of the implementation structure. In the railway domain, EN 50128 safety standard [4], recommends a certain level of implementation coverage on the developed IEC 61131-3 FBD software (e.g., decision coverage which is also known as branch coverage).

Even if implementation-based tests can be created manually, this process can be tedious and error prone because of its nature. As an alternative, automated implementation-based test generation is a research direction that has received much attention lately [3, 11, 32]. Specifically for the IEC 61131-3 FBD software, CompleteTest [8] is an automated test input generation tool which automatically produces tests for a given coverage criterion. The tool stops searching for test inputs when it achieves 100% coverage or when a stopping condition is achieved (i.e, timeout or out of memory). The user interface of CompleteTest is shown in Figure 3.1b. The interface shows several types of information presented to the user. The numbered points in Figure 3.1b represent:

1. steps and timing information regarding when the specific test input is provided to the program,

2. generated test inputs needed to achieve a maximum coverage for the given program,

3. editable area of the test outputs where the user can provide expected outputs for a specific set of test inputs based on a defined behavior in the specification,

4. percentage of the code coverage achieved by the generated test inputs, and

5. an action to compare expected outputs with the actual ones, computed by the program under test.

A generated test consists of a timed and ordered sequence of inputs. As the main purpose of CompleteTest at present is to generate tests that satisfy a certain coverage criteria, the tool does not generate expected outputs. Expected outputs are provided manually in the user interface, shown in Figure 3.1b, by a human tester.

3 Experiment Design

In this section we report the description of the performed controlled experiment. Additional details on the study (e.g, instruction material and programs used) can be found at the experiment website for replication and review purposes[1].

3.1 Research Questions

We defined the following research questions as a starting point in the experiment design:

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

RQ2: Are manually-written specification-based tests more costly to perform (prepare, write, execute and check the result) than tests manually or automatically created based on the implementation of the program under test?

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

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

[1]We provide all experimental material of this study at the following website `http://www.testinghabits.org/completetest/`

3.2 Experimental Setup Overview

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

Study Subjects

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

Table 3.1: Study Objects: "LOC" refers to the number of XML code lines contained on each of the programs, "NOD" refers to the number of decision outcomes.

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

Object Selection

The objects of study were chosen manually, based on the following criteria:

- The programs should have a natural language specification that is understandable and sufficiently rich in details for a tester to write executable tests.
- The programs should represent different types of real testing scenarios in different areas where the IEC 61131-3 standard is used.
- The programs should be developed using the IEC 61131-3 FBD language.
- The CompleteTest tool should be able to automatically generate tests for the programs. This excludes programs for which the underlying search engine does not support the data types used (i.e., strings).

We investigated the industrial libraries provided by Bombardier Transportation AB, a leading, large-scale company focusing on development and manufacturing of trains and railway equipment, used in our earlier studies [8, 9]. From a total of 157 artifacts we identified 14 candidate programs matching our criteria. The programs should not be trivial, yet fully manageable to test within 90 minutes and no domain-specific knowledge should be needed to understand the programs. We then assessed the relative difficulty of the identified programs by manually writing and automatically generating tests using CompleteTest. This process resulted

in the identification of one suitable program written in the IEC 61131-3 FBD programming language. This program, named *Fan Control*, was selected from a train control management system developed by industrial engineers from Bombardier Transportation AB in Sweden. The system is in development and uses processes influenced by safety-critical requirements and regulations including the EN 50128 standard [4]. In addition, we searched through previous research studies on testing IEC 61131-3 software. This process resulted in the identification of another suitable program written in IEC 61131-3 FBD. This program, is a function used in a nuclear power plant controlling the shutdown system for calculating *th_ X_ Trip*, as taken from the paper by Jee et al. [20] (Figure 1 in [20]). In the rest of the paper, this program is named *X Trip*. Details on the programs used in the experiment can be found in Table 3.1. We note here that an FBD program is written in a graphical environment that can be saved in an PLCOpen XML format[2].

For both programs a specification document written in natural language is available and contains all necessary detailed requirements of the program. All things considered, natural language specifications are less understood than code with regard to size and complexity [5] and therefore we are reporting here just the specification document size. The specification document for the *Fan Control* program is a collection of natural language functional requirements that contains 236 words. This document is created by an industrial requirements engineer in Bombardier Transportation AB. On the other hand the specification document for the *X Trip* program has 103 words and contains requirements expressed in natural language as described in the paper by Jee et al. [20].

3.3 Operationalization of Constructs

In this experiment, we compare the effect of using different test techniques on the implementation coverage, *effectiveness* and *efficiency* of the resulting tests.

Decision Coverage. Implementation coverage criteria are used in software testing to assess the thoroughness or adequacy of tests [1]. These criteria are normally used at the code level to assess the extent to which the program structure has been exercised by the tests.

Out of the many criteria that have been defined, logic coverage [10] can be used to measure the thoroughness of test coverage for the structure of FBD programs. The flow in an FBD program is largely controlled by atomic Boolean connections called *conditions*, and by blocks called *decisions* made up of conditions combined with Boolean operators (not, and, or, xor). A condition can be a single Boolean variable, an arithmetic comparison with a Boolean value (e.g., $out1 > in2$), or a call to a function with a Boolean value, but does not contain any Boolean operators. A set of tests satisfies decision coverage if running the tests causes each decision in the FBD program to have the value *true* at least once and the value *false* at least once.

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

In the context of traditional sequential programming languages, decision coverage is usually referred to as *branch coverage*.

In this experiment, implementation coverage is operationalized using decision coverage criteria. For the selected study objects, the EN 50128 standard [4] requires different implementation coverage levels (e.g., statement coverage). For the object developed by Bombardier Transportation AB, engineers developing IEC 61131-3 software indicated that their certification process for FBD programs, as the ones selected for this experiment, involves achieving high decision coverage. In this experiment we use decision coverage as the criterion for which tests are automatically generated. A coverage score indicator of the created tests is obtained for each individual solution.

Effectiveness. Mutation analysis is the technique of creating faulty implementations of a program (usually in an automated manner) for the purpose of examining the fault detection ability of a test [6]. A mutation score is calculated by automatically seeding faults to measure the fault detecting capability of the written tests. Using this approach we obtain a mutation score indicator of the created tests for each individual solution. During the process of generating mutants, a mutation tool typically creates syntactically and semantically valid versions of the original program by introducing a single fault into the program. As exhaustive categorization of all possible faults that may occur when using the FBD language is impractical, we rely on previous studies that looked at commonly occurring FBD faults [23, 29]. By considering these specific faults we used the following mutation operators:

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

To generate mutants, each of the mutation operators was automatically applied to each program element whenever possible. In total, for both objects, 138 mutants[3] (faulty programs based on LRO, CRO, NIO and VRO operators) were generated by automatically introducing a single fault into the correct implementation. We

[3] 38 and 100 mutants were created for *X Trip* and *Fan Control*, respectively.

computed the mutation score using an output-only oracle against the set of mutants. For both objects, we assessed the fault-finding effectiveness of each set of tests by calculating the ratio of mutants killed to the total number of mutants.

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

- *Duration:* Number of minutes spent on testing the program. This surrogate measure of cost includes the following actions: preparing, writing, executing the tests, and checking the expected versus actual outputs.

- *Number of tests:* This metric is defined by the number of created tests. Recall from Section 2 that each FBD program operates as a large loop receiving input and producing output. In this way, a generated set of tests is thus a finite number of steps, with each step (i.e., test) corresponding to a set of test inputs, actual and expected outputs.

3.4 Instrumentation

Two sessions were organized for the sake of the experiment: the first one for writing tests manually based on the specification and the other one for implementation-based manual and automated testing:

- *Session 1.* The subjects were given the task to test (to the extent they consider sufficient based on a given specification) two programs already implemented. They were instructed to read the specification and create tests to provide evidence that each behavior specified has been covered. The subjects were not grouped and the specification document needed for testing the program was provided digitally and in written form.

- *Session 2.* The subjects were given the task to test (to achieve full decision coverage) the same two programs tested in Session 1 by (*i*) manually creating tests to achieve full decision coverage and (*ii*) by automatically creating tests to achieve full decision coverage. The CompleteTest tool was used to generate, execute and check tests. Before commencing session 2, a short tutorial of approximately 10 minutes on IEC 61131-3 and FBD language syntax was provided to the subjects in order to avoid further problems with subjects' unfamiliarity with the concepts used. The tutorial included screencasts demonstrating programming and testing of FBD programs both manually and automatically using CompleteTest.

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

3.5 Data Collection Procedure

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

4 Experiment Conduct

Once the experiment design was defined, the requirements for executing the experiment were in place. Session 1 and 2 were held one week apart from each other and preceded by a theoretical lecture on specification-based testing and implementation-based testing respectively. These theoretical lectures were held two days before each session.

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

The subjects worked individually during the experiment; the first two authors of this paper briefly interacted with the participants to ensure that everybody had sufficient understanding of the involved tools without getting involved in the writing of the solution. All subjects used machines provided in the university premises of the same hardware configuration. The experiment was fixed to three hours per lab session. To complete the assignments in both sessions, the subjects were given the same time to work on testing the programs according to the given instructions. For measuring the mutation score, the achieved decision coverage, the number of tests and the testing duration, we provided a template to enforce the usage of the same reporting interface. By having a common template for test reporting we eased the process of performing the data collection and analysis.

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

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

Table 3.2: Results for each metric and for both programs. We report several statistics relevant to the obtained results: minimum, median, mean, maximum and standard deviation values.

(a) X Trip

Metric	Method	*Min*	*Median*	*Mean*	*Max*	*SD*
Mutation score (%)	SMT	68,42	97,37	93,94	100,00	8,99
	IMT	57,89	73,68	72,31	92,11	9,01
	IAT	63,16	71,05	72,54	84,21	5,66
Decision coverage (%)	SMT	92,86	100,00	99,15	100,00	2,27
	IMT	85,71	100,00	97,21	100,00	4,16
	IAT	100,00	100,00	100,00	100,00	0,00
No. of Tests	SMT	6,00	32,00	33,08	95,00	21,48
	IMT	3,00	3,0	5,13	16,00	3,32
	IAT	3,00	5,0	4,82	6,00	1,11
Duration (min.)	SMT	17,40	59,78	58,43	120,90	25,11
	IMT	10,07	27,57	30,05	54,08	12,18
	IAT	0,78	3,87	4,49	9,58	1,94

(b) Fan Control

Metric	Method	*Min*	*Median*	*Mean*	*Max*	*SD*
Mutation score (%)	SMT	97,00	98,00	98,57	100,00	1,34
	IMT	80,00	84,00	88,76	100,00	7,31
	IAT	85,00	92,00	90,78	98,00	4,08
Decision coverage (%)	SMT	92,86	100,00	97,83	100,00	3,36
	IMT	78,00	100,00	96,73	100,00	6,18
	IAT	100,00	100,00	100,00	100,00	0,00
No. of Tests	SMT	8,00	10,00	10,04	17,00	1,94
	IMT	3,00	4,00	5,96	15,00	2,99
	IAT	5,00	6,00	5,83	7,00	0,78
Duration (min.)	SMT	11,35	29,42	31,85	61,85	12,75
	IMT	12,33	27,58	26,43	45,25	7,20
	IAT	2,05	3,67	4,10	9,30	1,93

5 Experiment Analysis

This section provides an analysis of the data collected in this experiment. In analyzing the data, we followed the guidelines on statistical procedures for assessing randomized algorithms in software engineering provided by Arcuri and Briand [2].

For each program under test and each testing technique (i.e., SMT stands for *Specification-based Manual Testing*, IMT is short for *Implementation-based Manual Testing*, and IAT stands for *Implementation-based Automatic Testing*), each subject in our study provided a set of tests. These tests were used to conduct the experimental analysis. For each set of tests produced, we derived four distinct metrics: mutation score, decision coverage, number of tests, and testing duration. These metrics form

Table 3.3: Results of the experiment. For each metric we calculated the effect size of each method compared to each other. We also report the p-values of a Wilcoxon-Mann-Whitney U-tests with significant effect sizes shown in bold.

(a) X Trip

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

(b) Fan Control

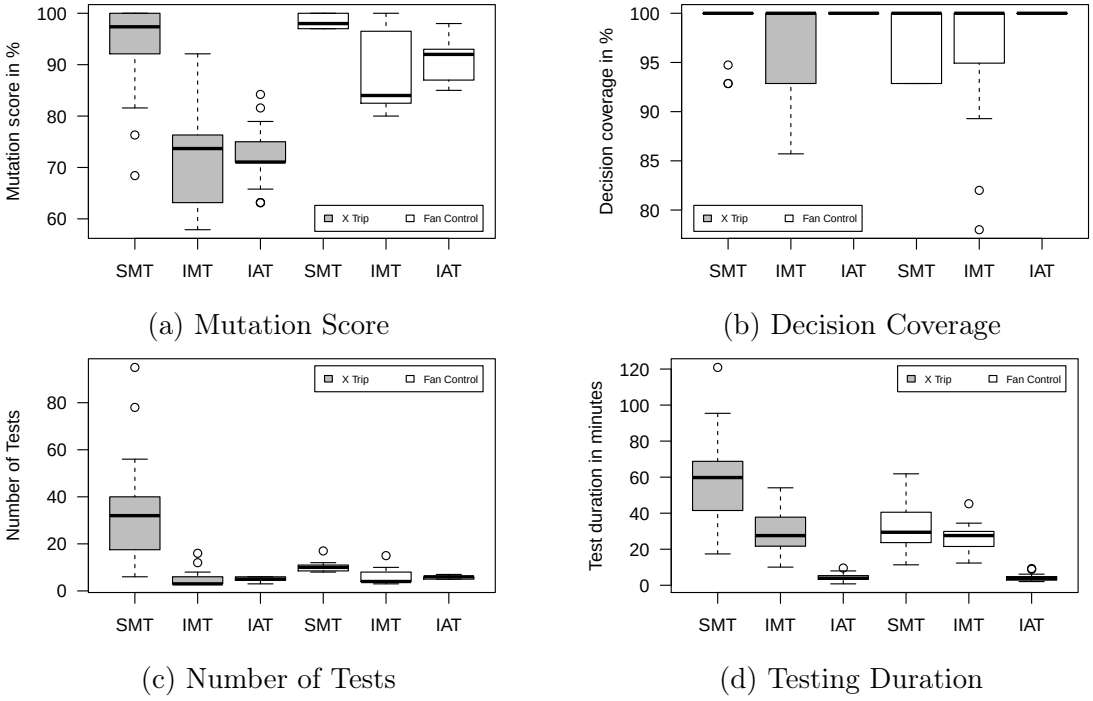
Metric	Method	Effect Size	p-value
Mutation score	SMT IMT	**0,848**	< 0,001
	IMT IAT	0,398	0,205
	SMT IAT	**0,923**	< 0,001
Decision coverage	SMT IMT	0,511	0,859
	IMT IAT	0,359	0,004
	SMT IAT	0,359	0,004
Number of Tests	SMT IAT	**0,844**	< 0,001
	IMT IAT	0,366	0,087
	SMT IAT	**0,958**	< 0,001
Duration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

the basis for our statistical analysis towards the goal of answering the research questions from Section 3.1. Statistical analysis was performed using the R software [31].

Table 3.2 lists the detailed statistics on the obtained results, like minimum values, median, mean and standard deviation. The results of this study are also summarized in the form of boxplots in Figure 3.2.

Our observations are drawn from an unknown distribution. To evaluate if there is any statistical difference between each testing technique without any assumption on the distribution of the collected data, we use a Wilcoxon-Mann-Whitney U-test [17], a non-parametric hypothesis test for determining if two populations of samples are drawn at random from identical populations. This test is used for checking if there is any statistical difference among the three groups for each metric. In addition, the Vargha-Delaney test [33] was used to calculate the standardized effect size, which is a non-parametric effect magnitude test that shows significance by comparing two

populations of samples and returning the probability that a random sample from one population will be larger than a randomly selected sample from the other. According to Vargha and Delaney [33] statistical significance is determined when the effect size measure is above 0,71 or below 0,29.



(a) Mutation Score

(b) Decision Coverage

(c) Number of Tests

(d) Testing Duration

Figure 3.2: Test metrics comparing specification-based manual testing (SMT) against implementation-based manual testing (IMT) and implementation-based automated testing (IAT); boxes spans from 1st to 3rd quartile, black middle lines mark the median and the whiskers extend up to 1.5x the inter-quartile range and the circle symbols represent outliers.

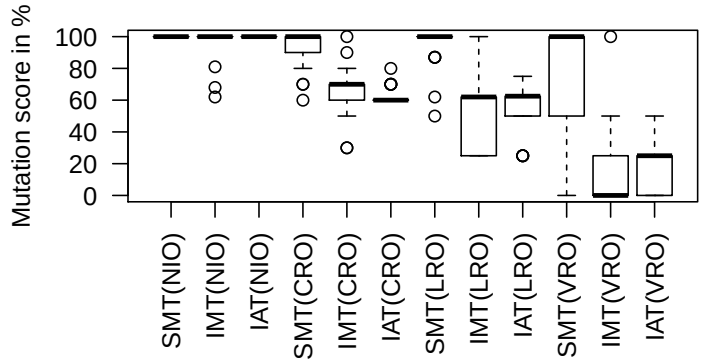
For each metric, we calculated the effect size of SMT, IMT and IAT. To this end, we reported in Table 3.3 the p-values of these Wilcoxon-Mann-Whitney U-tests with statistical significant effect sizes shown in bold.

5.1 Fault Detection

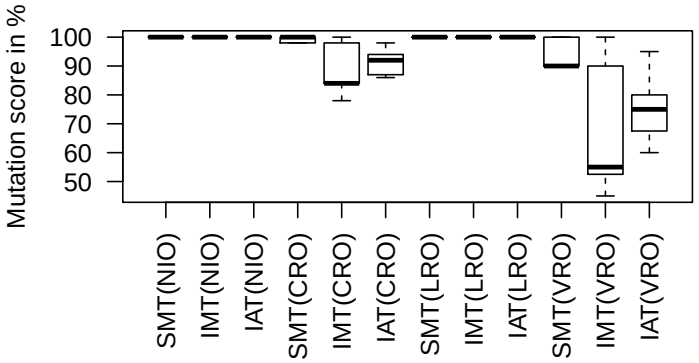
For both programs, as shown in Figure 3.2a, the fault detection scores of tests manually written based on the specification (SMT) were superior to tests written based on the implementation with statistically significant differences between SMT and IMT or IAT (effect size of over 0,844 in Table 3.3). For example, tests written for *X Trip* using SMT show an average fault detection of 93,94% compared to 72,31% for IMT and 72,54% for IAT. For *Fan Control*, SMT tests detect in average 98,57% of the faults versus 88,76% for IMT and 90,78% for IAT. None of the cases show any

statistically significant differences in fault detection between IMT and IAT (at 0,05), as the lowest p-value is equal to 0,205 for *Fan Control*.

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



(a) X Trip



(b) Fan Control

Figure 3.3: Mutation scores comparing specification-based manual testing (SMT) against implementation-based manual testing (IMT) and implementation-based automated testing (IAT); NIO is the negation insertion operator, CRO is the comparison block replacement operator, LRO is the logical block replacement operator, and VRO is the value replacement operator.

A question emerging from these results concerns why tests written using specification based manual testing are far better than the ones written using implementation-based testing. For the purpose of shedding some light on this matter, we investigated if these results could be explained by the fact that tests generated based on the implementation are particularly weak in detecting certain type of faults. More precisely, we examined what type of mutants were killed by tests written using SMT

to tests written using IMT and IAT. For each of the mutation operators described in Section 3, we examined the faults detected by each technique for both programs. The results of this analysis are shown in Figure 3.3 in the form of box plots. For the *Fan Control* program, both negation type of faults (NIO) and logical type of errors (LRO) are 100% detected by all three testing techniques. This shows that, for this program, all LRO and NIO injected faults are easily detected by every participant' test. On the other hand, tests written using SMT detect, on average, 7,9% more comparison type of faults (CRO) than tests produced using IAT. The increase is bigger for value replacement type of faults (VRO) with tests produced using SMT detecting, in average, 19,1% more faults than IAT. For the *X Trip* program, the situation is relatively similar, with SMT detecting more comparison (with 30% more faults in average), logical (with 38% more faults in average) and value replacement faults (with 51% more faults in average) than IAT. For both programs the NIO type of faults are detected by the majority of tests produced using all three testing techniques.

To further investigate the differences in fault detection for different mutation operators, we looked at one particular set of tests automatically generated using IAT by one of the participants using the CompleteTest tool for the *X Trip* program. These specific five tests achieve 81,58% mutation score with seven mutants not being detected. This set of tests achieved 100% decision coverage on the non-mutated version of the *X Trip* program. Interestingly enough the tests exercise all decisions also on the mutated program except for one mutant on which the tests achieved just 92,85% decision coverage. There is an obvious reduction in achieved coverage of the generated set of tests for that specific mutant but not for the other mutants. To determine if this behavior stems from the generation of poor tests and what tests would improve the mutation score, we observed that one extra test targeting the detection of the value replacement fault in the *X Trip* program would detect this specific mutant and, as a byproduct, all comparison replacement mutants. In addition, three extra tests were created targeting the detection of the remaining undetected logical replacement mutants. With a final set of tests of nine, all mutants were detected. In this case, the addition of four tests targeting the detection of the remaining faults has improved the fault-finding effectiveness. As a secondary result this particular example shows that for achieving better tests one should not solely rely on a decision coverage criterion alone.

5.2 Decision Coverage

As seen in Figure 3.2b, for both *X Trip* and *Fan Control* programs, the use of CompleteTest (IAT) entails 100% decision coverage (which is natural, as covering all decisions is the search objective for the test generation). Considering the effect sizes and the corresponding p-values in Table 3.3, results for both programs are not strong in terms of effect size and we did not obtain any significant statistical difference for decision coverage. The results for both programs matched our expectations: even if IAT achieves tests for both programs satisfying 100% decision coverage, tests written using SMT achieved relatively high coverage (in average 99,15% for *X Trip*

and 97,83% for *Fan Control*). This shows that, for the two programs studied in this experiment, SMT achieves high implementation coverage for both programs. This is likely due to the relatively limited complexity of the studied programs. It is possible that a more complex program would yield greater coverage differences between tests written using SMT and IAT or IMT.

5.3 Number of Tests

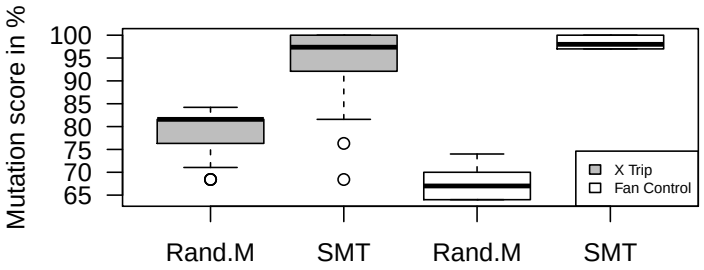
Based on the results highlighted in Figure 3.2c, the use of IAT and IMT consistently results in less number of tests for both programs compared to SMT. This is perhaps most pronounced for IAT, for which we can see in average less number of tests with 42% to 85,5% when using the CompleteTest tool than SMT. Examining Table 3.3, we see the same pattern in the statistical analysis: standardized effect sizes being higher than 0,844, with p-values below the traditional statistical significance limit of 0,05. The effect is the strongest for the Fan Control program with a standardized effect size of 0,958. It seems that a human tester, given sufficient time will create much more tests using SMT than IMT or IAT. This can be explained, for IAT, by considering that CompleteTest tool optimizes for decision coverage. It is likely that specification-based manual testing (SMT) will in practice achieve more tests for a similar level of coverage.

Answer RQ2: The use of implementation-based testing results in less number of tests than the use of specification-based testing.

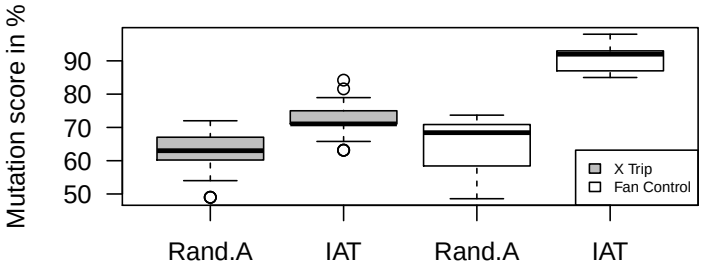
A question emerging from these results concerns why the number of tests written using specification-based manual testing is higher than the number of tests written using implementation-based testing. To investigate the effect of the number of tests on fault-finding effectiveness we produced purely random tests of equal size as the ones created by the participants using SMT (see Figure 3.4a) and purely random tests of equal size as the ones generated by participants using IAT (see Figure 3.4b). In this way we controlled random tests for their number. The results are shown in Figure 3.4 as box plots. For all programs, random generated tests with the same size as SMT and IAT are less effective in terms of mutation score than tests written using SMT and IAT, respectively. Overall, this indicates that tests produced using SMT are good indicators of test effectiveness, with a mutation score larger on average by 15% to 31% compared to random tests of equal size. When comparing random tests with implementation-based automated tests, we can observe from Figure 3.4 that, for both programs, decision coverage alone is a better indicator of tests effectiveness than random tests of equal size. In addition, we provided evidence that SMT is a good indicator of test effectiveness with factors other than the number of tests impacting the testing process.

5.4 Testing Duration

Analyzing testing duration is partially related to the number of tests analysis, but this metric gives a slightly different picture as the effort per created tests is not

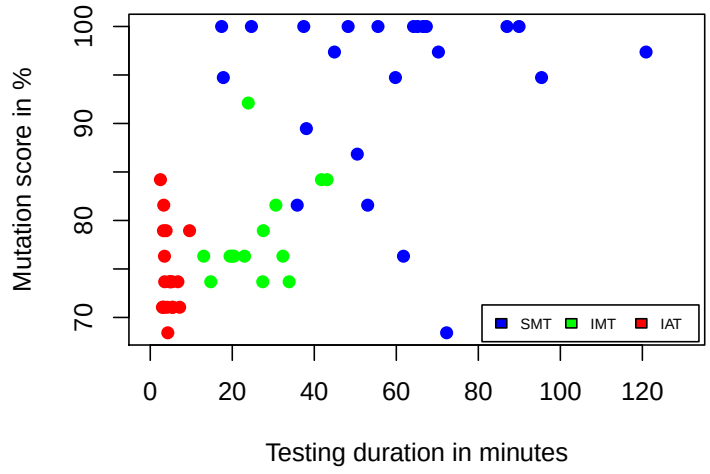


(a) Mutation score comparison between manually created tests based on the specification (SMT) and pure random tests of the same size as SMT (Rand.M).

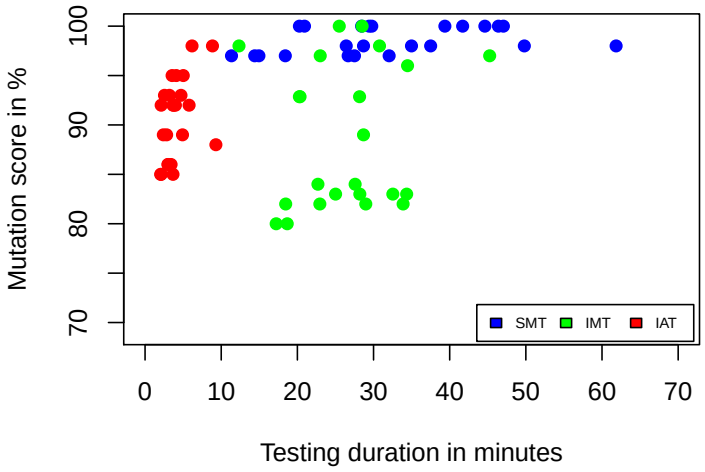


(b) Mutation score comparison between automatically generated tests (IAT) and pure random tests of the same size as IAT (Rand.A)

Figure 3.4: The effect of the number of tests on mutation score using random tests of the same size as the ones created by the study subjects.



(a) X Trip



(b) Fan Control

Figure 3.5: The relation between cost and effectiveness for tests manually written based on the specification (SMT), tests manually written based on the implementation (IMT) and implementation-based tests generated automatically using CompleteTest (IAT).

necessarily constant over the different techniques under investigation. As seen in Figure 3.2d, the duration of writing tests using CompleteTest (IAT) is consistently significantly lower than for manually derived tests based on the specification (SMT). First, consider the data related to both programs (Figure 3.2d); CompleteTest assisted subjects have a shorter completion time (from 85,5% to 15,5% shorter in average) over specification-based manual testing (SMT) and implementation-based manual testing (IMT). Examining Table 3.3, we observe that there is enough evidence to claim that these results are statistically significant with p-values below the traditional statistical significance limit of 0,05 and a standardized effect size of 0,958.

Answer RQ2: Implementation-based automated tests are less costly in terms of testing duration than manually created specification-based manual tests or implementation-based manual tests.

5.5 Cost-effectiveness Tradeoff

One important question in software testing is how the use of the investigated testing techniques affect the cost-effectiveness relation. In Figure 3.5 we show the relation between cost and effectiveness for tests written using SMT, tests written using IMT and tests generated with IAT. We use a proxy measure for cost, duration time (preparation, creation, execution of tests and checking the results) and a surrogate measure for effectiveness, namely mutation score. Obviously for both programs the ideal scenario would be to have low values for duration time while achieving high mutation scores. As shown in Figure 3.5a, for the *X Trip* program, the set of tests derived using SMT provided a good mutation score (93,94% in average) and an inconsistent testing duration that spans from 17,40 minutes to 120,90 minutes. On the other hand, tests derived using IAT are significantly consistent in terms of testing duration (between 0,78 minutes to 4,49 minutes) while achieving lower mutation scores than SMT (72,31% in average) but similar to the effectiveness shown by test written using IMT. In addition, Figure 3.5b shows that the achieved mutation scores for SMT are very consistent for both programs even if this comes at the price of having expensive tests in terms of testing duration. Specifically for *X Trip*, tests generated using IAT are cheap (completion time between 2,05 to 9,30 minutes) with fairly good fault-detection capability between 85,00% to 98,00% mutation score.

5.6 Limitations of the Study and Threats to Validity

External Validity. All of our subjects are students and have limited professional software development experience. This fact has been shown to be of somehow minor importance in certain conditions in a study by Höst et al. [16] with software engineering students being good substitutes in experiments for software professionals. Furthermore, in the light of our results regarding specification-based testing being better at fault detection than implementation-based testing, we see no reason why the use of professionals in our study would yield a completely different result. Testing

professionals with experience in IEC 61131-3 FBD software would intuitively write better tests at detecting common faults than tests written by student subjects.

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

Construct Validity. In our study we automatically seeded faults to measure the fault detecting capability of the written tests. While it is possible that faults created by industrial developers would give different results, there is scientific evidence [22] to support the use of injected faults as substitutes for real faults.

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

6 Related Work

Among the various fields of research in software testing, automated test generation has gain a considerable amount of work [24] in the last couple of years. Automated test generation techniques are used for generating a set of input values for a program, typically with the final aim of fulfilling a certain coverage criteria or reachability property.

A wide range of techniques for automated test input generation [11, 34, 25, 32] have been proposed in the last decade to replace or complement manual testing and are mainly targeting object-oriented programs. For example Randoop [25] creates random test inputs by using feedback information as guidance. EvoSuite [11] is a tool based on genetic algorithm for Java programs. The CompleteTest tool is using model checking and it is tailored to testing IEC 61131-3 FBD programs used in embedded software development.

While the application of automated test generation has been increasing the last few years, there have been a few studies involving human subjects that are addressing the question of how these techniques compare to manual specification-based testing. Ramler et al. [26] conducted a study and a follow-up replication [27], carried out with master students and industrial professionals respectively, addressing the question of how automated testing compare to manual testing. In these specific experiments, they found that the number of faults detected by the automated testing tool was similar to manual testing. Recently, Fraser et al. [12, 13] performed a controlled experiment and a follow-up replication experiment on a total of 97 subjects. They found that automated test generation, and specifically the EvoSuite tool, leads to high code

coverage but no measurable improvement over manual testing in terms of number of faults found by developers. Fault detection rate between automated implementation-based test generation and manual specification-based testing was found, in some of the studies [13, 26], to be relatively different from our experiment. This could stem from the fact that the subjects were given more time to manually test their programs compared to previous controlled experiments. By using a more restrictive testing duration, we would expect human participants to show less comprehensive understanding of the task at hand.

7 Conclusions and Future Work

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

The results of this experiment indicate that while the use of implementation-based automated testing yields high structural coverage and improves the number of tests and the testing time over specification-based manual testing, this is not reflected in the ability of the written tests to detect more faults. Our results shows the need to take caution in selecting test generation objectives when using tools for automated test input generation, as well as continued research in establishing more effective test adequacy criteria.

To perform a full, in-depth, study on testing embedded software, the experiment would need to be performed in an industrial setting on a larger number of programs.

Acknowledgments

This research was supported by The Knowledge Foundation (KKS) through the following projects: (20130085) Testing of Critical System Characteristics (TOCSYC), Automated Generation of Tests for Simulated Software Systems (AGENTS), and the ITS-EASY industrial research school.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008 (cit. on pp. 92, 95).

[2] Andrea Arcuri and Lionel Briand. "A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering". In: *Software Testing, Verification and Reliability*. Vol. 24. Wiley, 2014 (cit. on p. 99).

[3] 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 p. 92).

[4] CENELEC. “50128: Railway Application–Communications, Signaling and Processing Systems–Software for Railway Control and Protection Systems”. In: *Standard Report*. 2001 (cit. on pp. 89, 92, 95, 96).

[5] Alexander Dekhtyar, Jane Huffman Hayes, and Tim Menzies. “Text is Software Too”. In: *International Workshop on Mining Software Repositories*. (2004), pp. 22–26 (cit. on p. 95).

[6] 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 p. 96).

[7] 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 p. 90).

[8] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. “Automated Test Generation using Model Checking: an Industrial Evaluation”. In: *International Journal on Software Tools for Technology Transfer*. Vol. 18. 3. Springer, 2014, pp. 335–353 (cit. on pp. 90, 92, 94).

[9] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. “Model-based Test Suite Generation for Function Block Diagrams using the UPPAAL Model Checker”. In: *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2013, pp. 158–167 (cit. on p. 94).

[10] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. “Using Logic Coverage to Improve Testing Function Block Diagrams”. In: *Testing Software and Systems*. Springer, 2013, pp. 1–16 (cit. on p. 95).

[11] 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. 92, 108).

[12] 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 p. 108).

[13] 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. 108, 109).

[14] Vahid Garousi and Junji Zhi. “A Survey of Software Testing Practices in Canada”. In: *Journal of Systems and Software*. Vol. 86. 5. Elsevier, (2013), pp. 1354–1376 (cit. on p. 90).

[15] 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 p. 90).

[16] Martin Höst, Björn Regnell, and Claes Wohlin. "Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment". In: *Empirical Software Engineering*. Vol. 5. Springer, 2000 (cit. on p. 107).

[17] David Howell. *Statistical Methods for Psychology*. Cengage Learning, 2012 (cit. on p. 100).

[18] IEC. "International Standard on 61131-3 Programming Languages". In: *Programmable Controllers*. IEC Library, 2014 (cit. on pp. 89, 91).

[19] E. Jee, J. Yoo, S. Cha, and D. Bae. "A data flow-based structural testing technique for FBD programs". In: *Information and Software Technology*. Vol. 51. 7. Elsevier, 2009, pp. 1131–1139 (cit. on p. 90).

[20] 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 p. 95).

[21] K.H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010 (cit. on pp. 91, 92).

[22] 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. 108).

[23] Younju Oh, Junbeom Yoo, Sungdeok Cha, and Han Seong Son. "Software Safety Analysis of Function Block Diagrams using Fault Trees". In: *Reliability Engineering & System Safety*. Vol. 88. 3. Elsevier, 2005, pp. 215–228 (cit. on p. 96).

[24] 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 p. 108).

[25] 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. 108).

[26] 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. 108, 109).

[27] 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 p. 108).

[28] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. "Control System Devices: Architectures and Supply Channels Overview". In: *Sandia Report SAND2010-5183*. Sandia National Laboratories, 2010 (cit. on pp. 89, 92).

[29] Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae. "Empirical Evaluation on FBD Model-based Test Coverage Criteria using Mutation Analysis". In: *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 465–479 (cit. on p. 96).

[30] 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. 92).

[31] R Development Core Team. *R: A language and environment for statistical computing. http://www. R-project. org. The R Foundation for Statistical Computing, Vienna, Austria, (2005).* URL: `http://www.%20R-project.%20org` (cit. on p. 100).

[32] 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. 92, 108).

[33] András Vargha and Harold D Delaney. "A critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong". In: *Journal of Educational and Behavioral Statistics*. Vol. 25. 2. Sage Publications, 2000, pp. 101–132 (cit. on pp. 100, 101).

[34] 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 p. 108).

[35] 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 p. 90).

Study 4

A Comparative Study of Manual and Automated Testing for Industrial Control Software

Eduard Paul Enoiu, Adnan Causevic, Daniel Sundmark and Paul Pettersson

Submitted to the International Conference on Software Testing, Verification, and Validation (ICST), 2017, IEEE.

The paper was reformatted for uniformity, but otherwise is unchanged.

Study 4. A Comparative Study of Manual and Automated Testing for Industrial Control Software

Eduard Paul Enoiu, Adnan Causevic, Daniel Sundmark and Paul Pettersson

Abstract

The recent maturation of techniques and tools for automated test generation offers the opportunity to produce test suites with high code coverage in a matter of seconds. Nonetheless, it is not very well studied how such test suites compare to manually written ones in terms of cost and effectiveness. To a certain extent such comparisons have been made using open-source programs. However, evidence on how automated coverage-directed test generation could be used for industrial software is sparse. This is particularly true for industrial control software, where strict requirements on both specification-based testing and code coverage typically are met with rigorous manual testing. To address this issue, we conducted a case study in which we compared the cost and effectiveness between manually and automatically created test suites. In particular, we measured the cost and effectiveness in terms of fault detection of test suites created using a coverage-directed automated test generation tool and test suites manually created by industrial engineers for an existing train control system. We used recently developed real-world industrial programs written in the IEC 61131-3 FBD, a popular programming language for developing safety-critical systems using programmable logic controllers. The results show that automatically generated test suites achieve similar code coverage as manually created test suites, but in a fraction of the time (an average improvement of roughly 90%). We also found that the use of an automated test generation tool does not result in better fault detection in terms of mutation score compared to manual testing. Specifically, manual test suites more effectively detect logical, timer and negation type of faults, compared to automatically generated test suites. The results underscore the need to further study how manual testing is performed in industrial practice. We suggest some improvement opportunities for supporting the use of automated test generation tools in testing of industrial control software.

1 Introduction

Testing is an important activity in engineering of safety-critical control software. In certain application domains (e.g., the railway industry) engineering software requires certification according to safety standards [3]. These standards mandate the use

of specification-based testing and recommends the demonstration of some level of code coverage on the developed software. In this way, a developer needs to check software conformance with the specification: each test case should contribute to the demonstration that a specified requirement has indeed been satisfied. To achieve a certain level of code coverage, the designed test cases, when fed to the system under test, should systematically exercise the implementation (e.g., covering all branches). Naturally, developers want their test cases to be of high quality: test cases should be cost-effective and good at detecting faults. To support developers in testing, researchers have proposed different approaches for producing good test cases. In the last couple of years a wide range of techniques for automated test generation [10, 36, 28, 34] have been explored with the goal of complementing manual testing. Even though there is some evidence suggesting that automatically generated test suites may even cover more code than those manually written by developers [11], this does not necessarily mean that these tests are effective in terms of detecting faults. As manual testing and automated code coverage-directed test generation are fundamentally different and each strategy holds its own inherent limitations, their respective merits or demerits should be analyzed more extensively in comparative studies.

In this paper, we empirically evaluate automated test generation and compare it with test suites manually created by industrial engineers on 61 programs from a real industrial train control system. This system contains software written in IEC 61131-3 [15], a popular language in safety-critical industry for programming control software [19, 29]. We have applied a state-of-the-art test generation tool for IEC 61131-3 software, CompleteTest [9], and investigated how it compares with manual testing performed by industrial engineers in terms of code coverage, cost and fault detection.

Our case study indicates the following main results:

1. For IEC 61131-3 safety-critical control software, automated test generation can achieve similar code coverage as manual testing performed by industrial engineers but in a fraction of the time.

2. Even when achieving full code coverage, automatically generated test suites are not necessarily better at finding faults than manually created test suites. In our case study, 56% of the test suites generated using CompleteTest found less faults than test suites created manually by industrial engineers. Overall, 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.

3. Automatically generated test suites are significantly less costly in terms of testing time than manually created test suites. The use of automated test generation in IEC 61131-3 software development can potentially save around 90% of testing time.

These results point out important issues that need to be addressed in order to use automated test generation tools in testing safety critical control software.

Investigating the quality of the test suites revealed the need for improving the detection of faults. We found that there are more manually created test suites that are effective at detecting certain fault types than the automatically generated test suites. By using these fault types in addition to structural properties as the coverage criterion used by an automated test generation tool, we could generate more effective test suites. This can be achieved by considering an automated approach to generate test suites that can detect these specific faults. We argue that based on the results of this study, automated test generation could potentially be improved and be used in industrial practice for IEC 61131-3 software development to aid manual testing.

2 Related Work

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. In the software testing literature [1] a *test case* is a test executing the program, corresponding to a set of test inputs and expected outputs. A set of test cases is called a *test suite*. In this way, a test suite is thus a finite number of test cases executing one after the other.

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 software developers in testing of their programs, automated test generation has been explored in a considerable amount of work [26] in the last couple of years. Numerous techniques for automated test generation [10, 2, 36, 34, 39, 21] have been proposed in the last decade to complement manual testing. Many of these techniques mainly target object-oriented programs. Specifically, EvoSuite [10] is a tool based on genetic algorithms, for search-based testing of Java programs. Another automated test generation tool is KLEE [2] which is based on dynamic symbolic execution and uses constraint solving optimization as well as search heuristics to obtain high code coverage.

In the context of developing safety-critical control software, IEC 61131-3 [19] has become a very popular programming language used in different control systems from traffic control software to nuclear power plants. Several automated test generation approaches [18, 38, 17, 32, 9, 7] have been proposed in the last couple of years for IEC 61131-3 software. These techniques can typically produce test suites for a given code coverage criterion and have been shown to achieve high code coverage for different IEC 61131-3 industrial software projects.

While high code coverage has historically been used as a proxy for the ability of a test suite to detect faults, recently Inozemtseva et al. [16] and Gay et al. [12] found that coverage should not be used as a measure of quality mainly because of the fact that it is not a good indicator for fault detection. In other words, the fault detection capability of a test suite might rely more on other test design factors than the extent to which the structure of the code is covered. For example, in the safety-critical control software domain, software testing processes are influenced by different safety standards mandating and recommending different test design techniques, and one might speculate that other factors may affect the test suite

quality. This motivated us to investigate a thorough comparison between manual testing and implementation-based automated test generation.

There are studies investigating the use of both manual testing and automated implementation-based testing of real-world programs. Several researchers have performed case studies [37, 22, 30] and focused on already created manual test suites while others performed controlled experiments [11] with human participants manually creating and automatically generating test suites. In 2015, Wang et al. [37] compared KLEE-based test suites with already created manual test suites on several open source programs. They found that automatically generated test suites 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. [22] used EvoSuite on a number of open-source Java projects and compared those test suites with the ones already manually created by developers. EvoSuite-based test suites achieved similar code coverage and fault detection scores to manually created test suites. In addition, Fraser et al. [11] performed a controlled experiment and a follow-up replication experiment on a total of 97 subjects. They found that automated test generation, and specifically the EvoSuite tool, leads to high code coverage but no measurable improvement over manual testing in terms of number of faults found by developers. EvoSuite was used in another study by Shamshiri et al. [30] which found that test suites automatically generated achieved higher code coverage than developer-written test suites and detected 40% out of 357 real faults from different open-source projects.

These results kindled our interest in studying how manual testing compares to automated implementation-based test generation in an industrial safety-critical control software domain. For such systems, there are strict requirements on both traceable specification-based and implementation-based testing. Is there any evidence on how these code coverage-directed automated tools compare with, what is perceived as, rigorous manual testing?

3 Method

We designed a case study according to the method shown in Figure 4.1. From a high level view we started the case study by considering:(i) manual test suites created by industrial engineers and a tool for automated test generation named CompleteTest, (ii) a set of real industrial programs from a recently developed train control management system (TCMS), (iii) a cost model and (iv) a fault detection and code coverage metric. Consequently, we aimed to answer the following research questions:

- *RQ1: Are automatically generated test suites able to detect more faults than tests suites manually created by industrial engineers?*
- *RQ2: Are automatically generated test suites less costly than tests suites manually created by industrial engineers?*

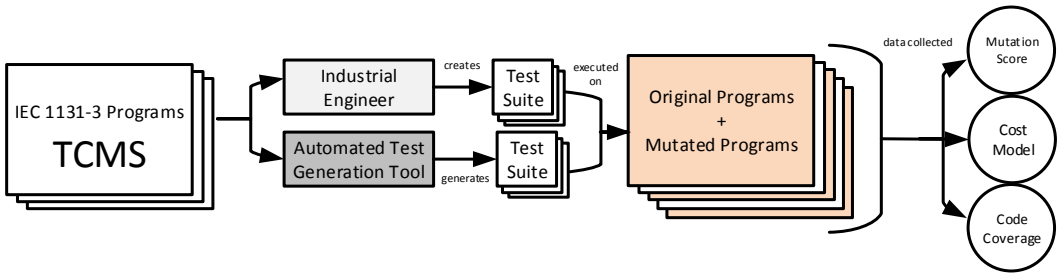


Figure 4.1: Overview of the experimental method. For each program in the Train Control Management System (TCMS), test suites are collected, generated and executed on both the original and the mutated programs.

We considered an industrial project containing IEC 61131-3 programs and for each selected program, we executed the test suites produced by both manual testing and automated test generation and collected the following measures: code coverage in terms of achieved decision coverage, the cost of performing testing and the mutation score as a proxy for fault detection. In order to calculate the mutation score, each test suite was executed on the mutated version of the original TCMS program to determine whether it detects the injected fault or not. This section describes in detail the case study procedure.

3.1 Case Description

The studied case is an industrial system actively developed in the safety-critical domain by Bombardier Transportation, a leading, large-scale company focusing on development and manufacturing of trains and railway equipment. The system is a train control management system (TCMS) that has been in development for several years and is engineered with a testing process highly influenced by safety standards and regulations. TCMS is a distributed control system with multiple types of software and hardware components, and is in charge of much of the operation-critical, safety-related functionality of the train. TCMS runs on Programmable Logic Controllers (PLC) which are real-time controllers used in numerous industrial domains, i.e., nuclear plants and avionics. The system allows for integration of control and communication functions for high speed trains and contains all functions controlling the train. These functions are developed as software programs using an IEC 61131-3 graphical programming language named Function Block Diagram (FBD) [15].

A program running on a PLC [19] executes in a loop where every cycle contains the reading of input values, the execution of the program without interruption and the update of the outputs. As shown in Figure 4.2, predefined logical and/or stateful blocks (e.g., bistable latch SR, OR, XOR, AND, greater-than GT and timer TON) and connections between blocks represent the behavior of an FBD program. These blocks are supplied by the hardware manufacturer or defined by a developer. PLCs contain particular types of blocks, such as timers (e.g., TON) that provide the same functions as timing relays and are used to activate or deactivate a device after a

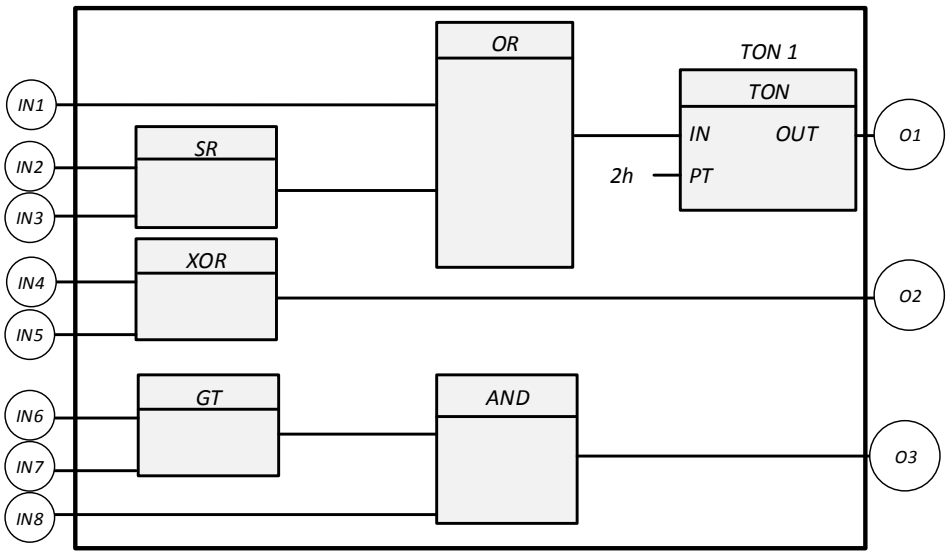


Figure 4.2: A program with eight inputs and three outputs written using the IEC 61131-3 FBD programming language.

preset interval of time. An FBD program is translated to a compliant executable PLC code. For more details on this programming language we refer the reader to the work of John et al. [19].

3.2 Test Suite Creation

As a baseline, we used manual test suites created by industrial engineers in Bombardier Transportation from a TCMS project delivered already to customers. A test suite created for an FBD program contains a set of test cases containing inputs, expected and actual outputs and timing information (i.e., Time parameter in the test suite is expressing timing constraints within one program).

Manual test suites were collected by using a post-mortem analysis [5] of the test data available. In testing IEC 61131-3 FBD programs in TCMS, the engineering processes of software development are performed according to safety standards and regulations. Specification-based testing is mandated by the EN 50128 standard [3] to be used to design test cases. Each test case should contribute to the demonstration that a specified requirement has indeed been covered and satisfied. Executing test cases on TCMS is supported by a test framework that includes the comparison between the expected output with the actual outcome. The test suites collected in this study were based on functional specifications expressed in a natural language.

In addition, we used test suites automatically generated using an automated test generation tool. For the programs in the TCMS system, EN 50128 recommends the implementation of test cases achieving a certain level of code coverage on the developed IEC 61131-3 FBD software (e.g., decision coverage which is also known as branch coverage). To the best of our knowledge, CompleteTest [9] is the only available automated test generation tool for IEC 61131-3 FBD software that produces tests for a given coverage criterion. As input for the test case generation,

the tool requires a standard PLCopen XML implementation of the program under test. CompleteTest supports different coverage criteria with the default criterion being decision coverage. The tool stops searching for test inputs when it achieves 100% coverage or when a stopping condition is achieved (i.e, timeout or out of memory). CompleteTest uses the Uppaal [23] model-checker as the underlying search engine and can be used both in a command-line and a graphical interface. A developer using CompleteTest can automatically generate test suites needed to achieve a maximum achievable code coverage for a given FBD program and can manually provide the expected outputs for a specific test case based on the defined behavior written in the specification. The tool will automatically run the tests and compare expected outputs with the actual ones, computed using the program under test.

It should be noted that in case CompleteTest is unable to achieve full coverage for a given program (which may happen since some coverage items may not be reachable, or that the search space is too large), a cutoff time is required to prevent indefinite execution. Based on discussions with engineers developing TCMS regarding the time needed for CompleteTest to provide a test suite for a desired coverage, we concluded that 10 minutes was a reasonable timeout point for the tool to finish its test generation. As a consequence, the tests generated after this timeout is reached will potentially achieve less than 100% code coverage.

3.3 Subject Programs

We used a number of criteria to select the subject programs for our study. We investigated the industrial library contained in TCMS provided by Bombardier Transportation. Firstly, we identified a project containing 114 programs. Next, we excluded 32 programs based on the lack of possibility to automatically generate test cases using CompleteTest, primarily due to the fact that those programs contained data types or predefined blocks not supported by the underlying model checker (i.e. string and word data types). The remaining 82 programs were subjected to detailed exclusion criteria, which involved identifying the programs for which engineers from Bombardier Transportation had created tests manually. This resulted in 72 remaining programs, which were further filtered out by excluding the programs not containing any decisions or logical constructs (since these would not be meaningful to test using logic criteria). A final set of 61 programs was reached. These programs contained on average per program: 825 lines of IEC 61131-3 FBD code, 18 decisions (i.e., branches), 10 input variables and 4 output variables.

For each of the 61 programs, we collected the manually created test suites. In addition we automatically generated test suites using CompleteTest for covering all decisions in each program. As a final step we generated additional test cases for all 61 programs using random test suites.

3.4 Measuring Code Coverage

Code coverage criteria are used in software testing to assess the thoroughness of test cases [1]. These criteria are normally used to assess the extent to which the program structure has been exercised by the test cases. In this study, code coverage is operationalized using the decision coverage criterion. For the TCMS system the EN 50128 safety standard [3] requires different code coverage levels (e.g., statement coverage). For the programs selected in this study and developed by Bombardier Transportation AB, engineers developing IEC 61131-3 software indicated that their certification process involves achieving high decision coverage. In the context of traditional programming languages (e.g., Java and C#), decision coverage is usually referred to as *branch coverage*. A decision coverage score is obtained for each individual test suite. A test suite satisfies decision coverage if running the test cases causes each decision in the IEC 61131-3 program to have the value *true* at least once and the value *false* at least once. Decision coverage was previously used by Enoiu et al. [9] to measure the thoroughness of code coverage for the specific structure of IEC 61131-3 programs. In this study we used our own tool implementation to collect the decision coverage achieved by each test suite.

3.5 Measuring Fault Detection

Fault detection was measured using mutation analysis. For this purpose, we used our own tool implementation to generate faulty versions of the subject programs. To describe how this procedure operates, we must first give a brief description of mutation analysis. Mutation analysis is the technique of creating faulty implementations of a program (usually in an automated manner) for the purpose of examining the fault detection ability of a test suite [6].

A *mutant* is a new version of a program created by making a small change to the original program. For example, in an IEC 61131-3 program, a mutant is created by replacing a block with another, negating a signal, or changing the value of a constant. The execution of a test suite on the resulting mutant may produce a different output as the original program, in which case we say that the test suite *kills* that mutant. A mutation score is calculated by automatically seeding mutants to measure the mutant detecting capability of the written test suite. We computed the mutation score using an output-only oracle (i.e., expected values for all of the program outputs) against the set of mutants. For all programs, we assessed the fault-finding capability of each test suite by calculating the ratio of mutants killed to the total number of mutants. Just et al. [20] showed that if a test suite can detect or kill most mutants, it can also detect real software faults, thus providing evidence that the mutation score is a fairly good proxy for real fault detection ability.

In the creation of mutants we rely on previous studies that looked at commonly occurring faults in IEC 61131-3 software [25, 31]. We used these common faults in this study for establishing the following mutation operators:

- *Logic Block Replacement Operator (LRO).* Replacing a logical block with another block from the same function category (e.g., replacing an AND block

with an OR block).

- *Comparison Block Replacement Operator (CRO).* Replacing a comparison block with another block from the same function category (e.g., replacing a Greater-Than (GT) block with a Greater-or-Equal (GE) block).

- *Arithmetic Block Replacement Operator (ARO).* Replacing an arithmetic block with another block from the same function category (e.g., replacing a maximum (MAX) block with a subtraction (ADD) block).

- *Negation Insertion Operator (NIO).* Negating an input or output connection (e.g., an input variable *in* becomes NOT(*in*)).

- *Value Replacement Operator (VRO).* Replacing a value of a constant variable connected to a block (e.g., replacing a constant value ($const = 0$) with its boundary values (e.g., $const = -1$)).

- *Timer Block Replacement Operator (TRO).* Replacing a timer block with another block from the same function category (e.g., replacing a Timer-On (TON) block with a Timer-Off (TOF) block).

To generate mutants, each of the mutation operators was systematically applied to each program element wherever possible. In total, for all of the selected programs, 5161 mutants (faulty programs based on ARO, LRO, CRO, NIO, VRO and TRO operators) were generated by automatically introducing a single fault into the original implementation.

3.6 Measuring Efficiency

Many factors affect the cost of testing IEC 61131-3 FBD programs. According to Leung and White [24], testing involves two cost types: direct and indirect costs. A direct cost includes the implementer time for performing all activities related to unit testing and the machine resources such as the test infrastructure. Indirect costs include: the management of the testing process, test tool development, and execution history. Ideally, the effort is captured by measuring the time required for performing different testing activities. However, since this is a post-mortem study of a now-deployed system and the development was undertaken a few years back, this was not practically possible in our case. Instead, efficiency was measured using a cost model that captures the context that affects the testing of IEC 61131-3 software. We focused on the unit testing process as it is implemented in Bombardier Transportation for testing the programs selected in this case study. Figure 4.3 presents a timeline depicting the unit testing process for a single program. The EN 50128 safety standard requires that a software design is produced for each program and that each program is then tested using the design as the test oracle. For the TCMS system used in this case study, the test specification, execution and reporting are performed by the implementer of the IEC 61131-3 software. In the rest of the paper we will not use the cost of creating the design of the software because even if

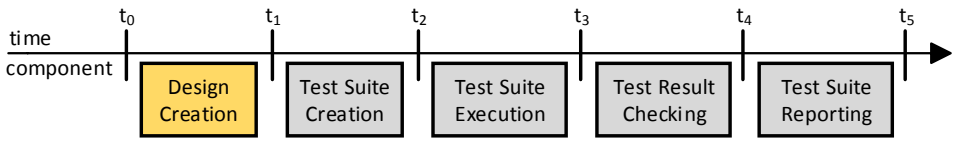


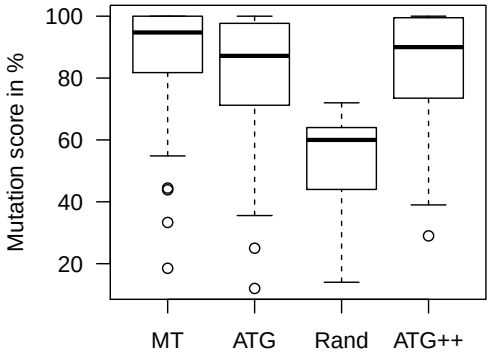
Figure 4.3: Program Testing Cycle for IEC 61131-3 FBD software in TCMS.

Table 4.1: Constituent cost components and factors that cause the cost to vary when using both manual testing (MT) and automated test generation (ATG). T represents the number of test cases created in a single test suite, δ, ε, α and τ represent the average estimated time an engineer spends to manually create, execute, check and report a test case, respectively.

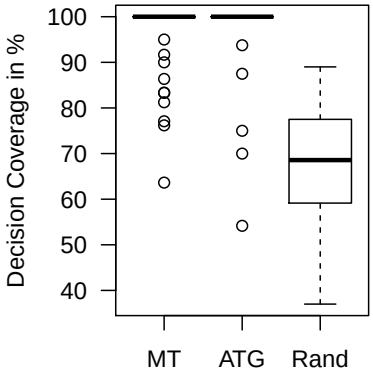
Component	MT	ATG
Test Creation (C_δ)	$\delta \cdot T$	t_e
Test Execution (C_ε)	$\varepsilon \cdot T$	
Test Result Check (C_α)	$\alpha \cdot T$	$\alpha \cdot T$
Test Reporting (C_τ)	$\tau \cdot T$	t_r

it varies with the size and characteristics of each program and requirement, the cost is constant between manual and automated test generation.

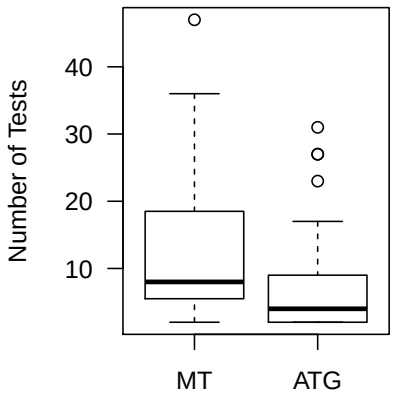
In the cost model, we concentrated on the following components (mirrored in Table 4.1): the test creation cost (C_δ) represents the cost of writing the necessary test suite, the test execution cost (C_ε) represents the cost of executing a test suite, test result check cost (C_α) represents the cost of checking the result of the test suite, and test reporting cost (C_τ) represents the cost of reporting a test suite. The cost model does not include the required tool preparation. However, preparation entails exporting the program to a format readable by CompleteTest and opening the resulting file. This effort is comparable to that required for opening the tools needed for manual testing. To formulate a cost model incorporating the cost components shown in Table 4.1, we must measure costs in identical units. To do this, we recorded all costs using a time metric. For manual testing all costs are related to human effort. For automated test generation the cost of checking the test result (i.e., C_α in Table 4.1) is related to human effort with the other costs (i.e., the combined generation and execution time t_e and the reporting time t_r when using CompleteTest) measured in machine time needed to compute the results. In this case study we consider that all cost components are related to the number of test cases. The higher the number of tests cases, the higher are the respective costs. We assume this relationship to be linear (δ, ε ,α and τ in Table 4.1 are factors representing the average time spend by an engineer in each cost component for a test case). Practically, we measured the costs of these activities directly as an average of the time taken by three industrial engineers (working at Bombardier Transportation implementing some of the IEC 61131-3 programs used in our case study) to perform manual testing.



(a) Mutation score comparison



(b) Decision coverage achieved



(c) Number of Test Cases

Figure 4.4: Mutation score, achieved code coverage and number of test cases comparison between manually created test suites (MT), automatically generated test suites (ATG), pure random test suites (Rand) of the same size as the ones created manually by industrial engineers, and coverage-adequate tests with equal size as manual tests (ATG++); boxes spans from 1st to 3rd quartile, black middle lines mark the median and the whiskers extend up to 1.5x the inter-quartile range and the circle symbols represent outliers.

Table 4.2: Results for each metric. We report several statistics relevant to the obtained results: minimum, median, mean, maximum and standard deviation values.

Metric	Test	Min	$Median$	$Mean$	Max	SD
Mutation Score(%)	MT	18,51	94,73	86,30	100,00	19,66
	ATG	25,00	89,47	82,93	100,00	18,37
Coverage (%)	MT	63,63	100,00	96,33	100,00	8,08
	ATG	54,16	100,00	97,45	100,00	8,65
# Tests	MT	2,00	8,00	12,80	47,00	10,57
	ATG	2,00	4,00	7,42	31,00	7,35

4 Results

This section provides an analysis of the data collected in this case study. For each program and each testing technique considered in this study we collected the produced test suites. The overall results of this study are summarized in the form of boxplots in Figure 4.4. Statistical analysis was performed using the R software [33]. In Table 4.2 we present the mutation scores, coverage results and the number of test cases in each collected test suite (i.e., *MT* stands for manually created test suites and *ATG* is short for test suites automatically generated using CompleteTest). This table lists the minimum, median, mean, maximum and standard deviation values. As our observations are drawn from an unknown distribution, we evaluate if there is any statistical difference between *MT* and *ATG* without making any assumptions on the distribution of the collected data. We use a Wilcoxon-Mann-Whitney U-test [14], a non-parametric hypothesis test for determining if two populations of data samples are drawn at random from identical populations. This statistical test was used in this case study for checking if there is any statistical difference among each measurement metric. In addition, the Vargha-Delaney test [35] was used to calculate the standardized effect size, which is a non-parametric magnitude test that shows significance by comparing two populations of data samples and returning the probability that a random sample from one population will be larger than a randomly selected sample from the other. According to Vargha and Delaney [35] statistical significance is determined when the effect size is above 0.71 or below 0.29.

4.1 Fault Detection

How does the fault detection of manual test suites compare with that of automatically generated test suites based on decision coverage? For all programs, as shown in Figure 4.4a, the mutation scores obtained by manually written test suites are higher in average with 3% compared with the ones achieved by automatically generated test suites. However, there is no statistically significant difference at 0.05 as the p-value is equal to 0.087 (effect size 0.600 in Table 4.3). Consequently, a larger sample size, as well as additional studies in different contexts, would be needed to obtain more confidence to claim that automatically created test suites are actually worse than manually created test suites.

Table 4.3: For the mutation score we calculated the effect size representing the magnitude of the difference between manual testing (MT), automated test generation (ATG) and random testing (Rand). We also report the p-values of a Wilcoxon-Mann-Whitney U-tests with significant effect sizes shown in bold.

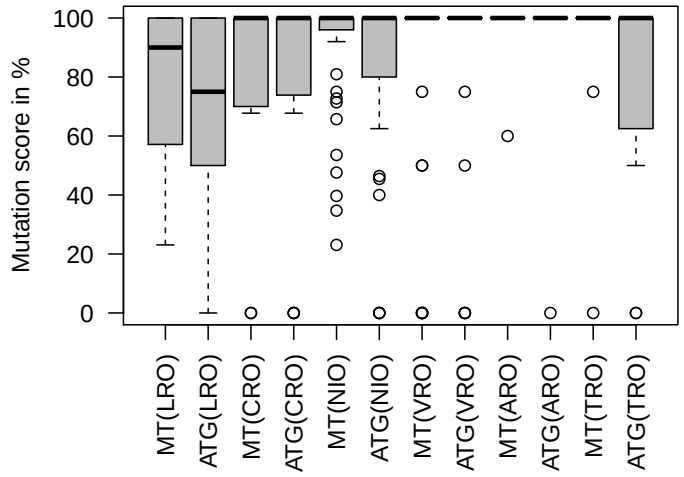
Measure	Method	Effect Size	p-value
Mutation Score	MT ATG	0.600	0.087
	MT Rand	**0.897**	< 0.001

Answer RQ1: Using automatically generated test suites does not yield better fault detection than using manually created test suites.

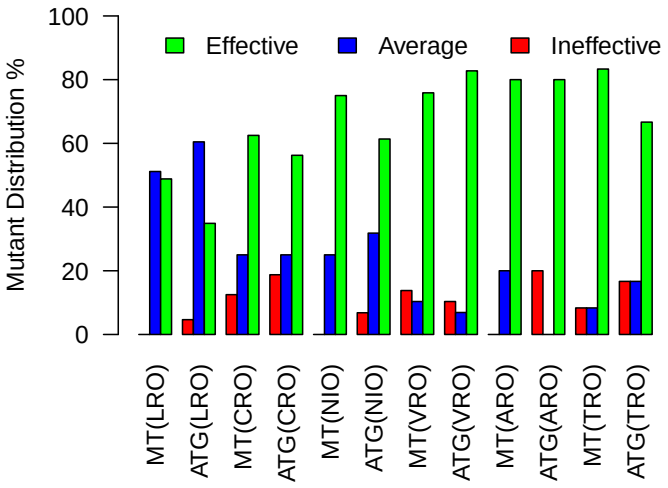
The difference in effectiveness between manual and automated testing could be due to differences in test suite size. As shown in Figure 4.4c, the use of automated test generation results in less number of test cases than the use of manual testing (a difference of roughly 40%). To control for size, we generated purely random test suites of equal size as the ones manually created by industrial engineers (*Rand* in Figure 4.4a) and coverage-adequate test suites with equal size as manual test suites (ATG++ in Figure 4.4a). Our results suggest that fault detection scores of manually written test suites are clearly superior to random test suites of equal size, with statistically significant differences (effect size of 0.897 in Table 4.3). In addition, even coverage-adequate test suites with equal size as manual test suites (ATG++ in Figure 4.4a) are not showing better fault detection than the ones manually created. This shows that the effect of reduced effectiveness for automated test generation is not only due to smaller test suites. This is not an entirely surprising result. Our expectation was that manual test suites would be similar or better in terms of fault detection than automatically created test suites based on decision coverage. Industrial engineers with experience in testing IEC 61131-3 FBD programs would intuitively write good test cases at detecting common faults. Our results are not showing any statistically significant difference in mutation score between manual test suites and CompleteTest-based test suites.

4.2 Fault Detection per Fault Type

To understand how automatically generated test suites can be improved in their fault detection capability, we examined if these tests suites are particularly weak or strong in detecting certain fault types. We concern this analysis to what type of faults were detected by both manual testing and CompleteTest. For each mutation operator described in Section 3.5, we examined what type of mutants were killed by tests written using manual testing and CompleteTest. The results of this analysis are shown in Figure 4.5a in the form of box plots with mutation scores broken down by the mutation operator that produced them. There are some broad trends for these mutation operators that hold across all programs considered in this study. The



(a) Mutation scores comparing manual testing (MT) against implementation-based automated test generation (ATG).



(b) Distribution of different types of mutants killed by manual versus automated test generation and categorized according to their effectiveness level.

Figure 4.5: Mutation analysis results per fault type: LRO is the logical block replacement fault type, CRO is the comparison block replacement fault type, NIO is the negation insertion fault type, VRO is the value replacement fault type, ARO is the arithmetic block replacement fault type and TRO is the timer block replacement fault type.

fault detection scores for arithmetic (ARO), value (VRO) and comparison (CRO) replacement fault types are not showing any significant difference between manually created test suites and automatically generated test suites. On the other hand, test suites written using manual testing detect, on average, 12% more logical type of faults (LRO) than test suites generated automatically using CompleteTest. The increase is slightly similar for negation (NIO) and timer (TRO) replacement type of faults with manually written test suites detecting, in average, 13% more NIO and TRO fault types than automatically generated test suites. Overall, it seems that one of the reasons behind manual testing success, seems to do with its strong ability to kill mutants from certain operators.

Manual test suites are detecting more logical, timer and negation type of faults than automatically generated test suites.

We further classified all manual and automated test suites into three types as follows:(i) Effective: Test suites that score above 95% on a specific fault type. Test suites achieving this kind of mutation score have been shown [13] to be very effective at finding faults. (ii) Average: Test suites that score between 5 and 95% on a specific fault type. (iii) Ineffective: Test suites that score lower than 5% on a specific fault type. Each test suite type has meaningful implications. For example, effective tests indicate strong fault detection effectiveness per fault type while ineffective tests indicate weak fault detection for a certain type of fault. The results of this analysis are shown in Figure 4.5b in the form of bar plots. For logical (LRO) faults, 48% of the manual test suites are effective at detecting this type of faults, an improvement of 13% over automatically generated test suites. We found more manual test suites being effective at detecting comparison (CRO) replacement and negation insertion (NIO) faults (an improvement of 6% and 13%, respectively) over automatically generated test suites. The increase is bigger for timer replacement (TRO) faults with manual testing having 16% more effective test suites than automated test generation. On the other hand, for value replacement (VRO) fault types, we found more effective automatically generated test suites than manually created test suites (an improvement of 6%). For arithmetic (ARO) faults, there is no difference between manual and automatically-generated effective test suites. In addition, in Figure 4.5b, we can observe that automated test generation results in more ineffective tests compared to manual testing when considering logical replacement (LRO), negation insertion (NIO) and timer replacement (TRO) fault types.

To identify the reasons behind the differences in mutation score per fault type between manual testing and CompleteTest, we investigated deeper the nature of each mutation operator. For both the negation insertion and the timer replacement fault type it seems that CompleteTest with branch coverage as the stopping criterion, achieves a poor selection of test input conditions with too few test cases being produced; a certain input value that fails to kill the NIO and TRO type of mutant could have been made more robust with further test inputs. Logical replacement type of faults where an AND block is replaced by an OR block and vice versa tends to be relatively trivial to detect by both manual testing and CompleteTest. This

Table 4.4: For code coverage we calculated the effect size representing the difference between manual testing (MT), automated test generation (ATG) and random testing (Rand). We also report the p-values of a Wilcoxon-Mann-Whitney U-tests with significant effect sizes shown in bold.

Measure	Method	Effect Size	p-value
Coverage	MT ATG	0.449	0.192
	MT Rand	**0.971**	< 0.001

comes from the fact that these faults are detected by any test cases where the inputs evaluate differently and the change is propagated to the output of the program. This does not mean that all logical faults can be easily detected. Consider an LRO type of mutant where OR blocks are replaced by XOR. The detection of this type of fault is harder, with manual test suites detecting 24% more LRO type of mutants where a logical block is replaced by XOR than test suites generated automatically. The detection in this case happens only with one specific test case that propagates the change in the outputs. It seems that manual testing has a stronger ability to detect these kind of logical faults than automated test generation because of its inherent limitation of only searching for inputs that are covering the branches of the program.

4.3 Coverage

As seen in Figure 4.4b, for the majority of programs considered, manually created test suites achieve 100% decision coverage. Random test suites of the same size as manually created ones achieve lower decision coverage scores (in average 61%) than manual test suites (in average 96%). The coverage achieved by manually created test suites is ranging between 63% and 100%. As shown in Table 4.2, the use of CompleteTest achieves in average 97% decision coverage. Results for all programs (in Table 4.4) show that differences in code coverage achieved by manual versus automatic test suites are not strong in terms of any significant statistical difference (with an effect size of 0.449 and a p-value of 0.192). Even if automatically generated test suites are created by CompleteTest having the purpose of covering all decisions, these test suites are not showing any significant improvement in achieved coverage over the manually written ones.

Overall, we confirm that the code coverage scores achieved by CompleteTest-based test suites are similar to the ones created manually by industrial engineers. While developing TCMS programs, engineers manually writing test suites have to demonstrate the use of specification-based testing while maintaining a certain degree of decision coverage. After discussions with three engineers developing IEC 61131-3 FBD software at Bombardier Transportation, functional specifications seem to be the main source of information for performing manual testing in our case study. From our results one question arises: Is high decision coverage achieved by test suites just a byproduct of performing specification-based manual testing? This underscores the need to study further how manual testing is actually performed in practice and what

Table 4.5: Cost measurement results for both manual testing and automated test generation using CompleteTest.

(a) Manual Testing

Cost (min.)	*Min*	*Median*	*Mean*	*Max*	*SD*
C_δ	13,2	52,8	84,5	310,2	70,5
C_ε	6,6	26,4	42,2	155,1	35,2
C_α	5,0	20,0	32,0	117,5	26,7
C_τ	1,0	4,0	6,4	23,5	5,3
C_{total}	25,8	103,2	165,2	606,3	137,8

(b) Automated Test Generation using CompleteTest

Cost(min.)	*Min*	*Median*	*Mean*	*Max*	*SD*
$C_\delta + C_\varepsilon$	0,003	0,012	1,120	10,900	3,185
C_α	5,000	10,000	18,563	77,500	18,597
C_τ	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
C_{total}	5,003	15,007	19,684	77,683	18,433

makes it so good at achieving high code coverage and fault detection.

4.4 Cost Measurement Results

This section aims to answer RQ2 regarding the relative cost of performing manual testing versus automated test generation. The conditions under which each of the strategies is more cost effective were derived. The cost variables presented in Section 3.6 are measured in time (i.e., minutes) spent, and their calculation depends on several cost components.

For manual testing (MT) the total cost C_{total} involves only human resources. We interviewed three engineers working on developing and manually testing TCMS software and asked them to estimate the time (in minutes) needed to create (δ), execute (ε), check the result (α) and report a test suite (τ). All engineers independently provided very similar cost estimations. We averaged the estimated time given by these three engineers and based on the formulae in Table 4.1 we calculated each individual cost using the following average estimations: $\delta = 6.6$, $\varepsilon = 3.3$, $\alpha = 2.5$ and $\tau = 0.5$. The overall cost measures are reflected in Table 4.5. In addition, for automated test generation the total cost of performing automated test generation involves both machine and human resources. We calculated the cost of creating, executing and reporting test suites for each program, by measuring the time required to run CompleteTest, and the time required to execute each test case (i.e., t_e in Table 4.1). For the cost of checking the test result we used the average time needed by three industrial engineers to check the results using manual testing ($\alpha = 2.5$). The resulting cost measures are reflected in Table 4.5.

Analyzing the cost measurement results is directly related to the number of test cases giving a picture of the effort per created test case. As seen in Table 4.5, the cost of performing testing using CompleteTest is consistently significantly lower

than for manually created tests; automatic generated tests have a shorter testing cost (145.5 minutes shorter testing time in average) over manual testing.

Answer RQ2: Automatically generated tests are significantly less costly in terms of testing time than manually created tests.

Based on these results, we can clearly see that when using automatic test generation tools, the creation, execution and reporting costs are very low compared to manual testing. While these cost are low, the cost of checking the results is human intensive and is a consequence of the relative difficulty to understand each created test.

5 Discussions and Future Work

Developers of safety critical control software use different test design techniques for testing their programs. We showed in Section 4 the results obtained from a case study performed at Bombardier Transportation, a large-scale company focusing on development of trains. The programs considered in this study have been in development and are used in different train products all over the world. The obtained results could prove useful for both practitioners, tool developers and software testing researchers. To further explore the results of our case study we considered the implications for future work and the extent to which automated test generation can be used in the development of reliable systems.

Our results indicate that, in IEC 61131-3 software development, automated test generation can achieve similar decision coverage to manual testing performed by industrial engineers. However, these automatically generated test suites are not yielding better fault detection in terms of mutation score than manually created test suites. The fault detection rate between automated implementation-based test generation and manual testing was found, in some of the published studies [11, 22, 37], to be relatively similar to our results. Interestingly enough, our results indicate that CompleteTest-based test suites might even be slightly worse in terms of fault detection compared to manual test suites. However, a larger empirical study is needed to statistically confirm this hypothesis.

Our study is the first to consider fault detection per fault type in an industrial context to understand how automatically generated test suites can be enhanced. From our results we highlight the need for improving the goals used by automated test generation tools for creating test suites. Code coverage-based test generation needs to be carefully complemented with other techniques such as mutation testing. We found that there are more manually created test suites that are effective at detecting certain type of faults than automatically generated test suites. By considering generating test suites that are detecting these fault types one could improve the goals of automated test generation by using a specialized mutation testing strategy. This needs to be carefully considered in future studies.

As part of our study, we used cost measurements to estimate the efficiency of performing automated test generation. Our study suggests that automatically generated test suites are significantly less costly in terms of testing time than

manually created test suites. The use of CompleteTest in IEC 61131-3 FBD software development can potentially save around 90% of testing time. The fact that automated test generation is faster, cheaper and possibly as good as manual testing, stands as a significant progress in aiding developers performing unit testing.

6 Threats to Validity

In our study we automatically seeded mutants to measure the fault detection capability of the written tests. While it is possible that faults created by industrial developers would yield different results, there is some scientific evidence [20] to support the use of injected faults as substitutes for real faults.

There are many tools (e.g., KLEE [2], EvoSuite [10], Java PathFinder [36], and Pex [34]) for automatically generating tests and these may give different results. The use of these tools in this study is complicated by the transformation of IEC 61131-3 programs directly to Java or C, fact shown to be a significant problem [27] because of the difficulty to transform timing constructs and ensure the real-time nature of these programs. Hence, we went for a tool specifically tailored for testing IEC 61131-3 programs. To the best of our knowledge, CompleteTest is the only openly available such tool.

The results are based on a case study in one company using 61 programs and manual test suites created by industrial engineers. Even if this number can be considered quite small, we argue that having access to real industrial test suites created by engineers working in the safety-critical domain can be representative. More studies are needed to generalize these results to other systems and domains.

We note here that the cost of automatically testing IEC 61131-3 FBD programs is heavily influenced by the human cost of checking the test result. We assumed that the average time of checking the results per test case for automated testing is the same as in the case of manual testing. In practice, this might not be the real situation. A test strategy, that requires every decision in the program to be executed, could contain test cases that are not specified. This might increase the cost of checking the test case result. If we assume that the cost of checking the result for automatic tests is different by one order of magnitude, α (in Table 4.1) could be about ten times different in quantity. Hypothetically, the cost of automatically generating test suites could be slightly higher (with 20 minutes in average) than manually testing. A more accurate cost model would be needed to obtain more confidence to claim that CompleteTest actually is worse in terms of testing time than manual testing.

A final threat is that we did not use other stronger criteria than decision coverage for automatically generating test suites, such as MCDC and its variants [4, 1]. Intuitively, MCDC-based test suites would show better fault detection than decision coverage-based test suites. We did not consider these criteria in our study for two reasons. First, engineers from Bombardier Transportation AB testing the programs considered in this study suggested that their certification process recommends the use of decision coverage for assessing the thoroughness of their test suites. Second, a recent study [8] on the complexity of both safety-critical and non-critical Java programs has shown that MCDC and similar criteria are only needed on a small

fraction of programs containing more complex branches or decisions. We therefore argue that using decision coverage for automatically generating test suites is a suitable and realistic scenario for many programming languages.

7 Conclusions

In this paper we investigated, in an industrial context, the quality and cost of performing manual testing and automated test generation. The study is based on 61 real-world programs from a recently developed safety-critical control software and manual test suites produced by industrial professionals. We posed our first research question RQ1 (i.e., *Are automatically generated test suites able to detect more faults than test suites manually created by industrial engineers?*) in order to understand how good in terms of fault detection are automatically generated tests by comparing them with manual test suites. Our results do not confirm the hypothesis that automatically generated test suites are better at finding faults in terms of mutation score than manually created test suites. In addition, we showed that the effect of reduced fault detection for automated test generation is not only due to smaller test suites. Overall, it seems that manual testing shows a stronger ability to detect faults of certain type than automated test generation. With regard to the second research question RQ2 (i.e., *Are automatically generated test suites less costly than test suites manually created by industrial engineers?*) we aimed to bring some industrial experimental evidence to the basic understanding of how automated test generation compare in terms of testing cost with manual testing. Our results suggest that automated test generation can achieve similar decision coverage as manual testing performed by industrial engineers but in a fraction of the time.

Acknowledgments

This research was supported by The Knowledge Foundation (KKS) through the following projects: (20130085) Testing of Critical System Characteristics (TOCSYC), Automated Generation of Tests for Simulated Software Systems (AGENTS), and the ITS-EASY industrial research school.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008 (cit. on pp. 117, 122, 133).

[2] 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. 117, 133).

[3] CENELEC. "50128: Railway Application–Communications, Signaling and Processing Systems–Software for Railway Control and Protection Systems". In: *Standard Report*. 2001 (cit. on pp. 115, 120, 122).

[4] John Joseph Chilenski and Steven P Miller. "Applicability of Modified Condition/Decision Coverage to Software Testing". In: *Software Engineering Journal*. Vol. 9. 5. IET, 1994, pp. 193–200 (cit. on p. 133).

[5] Reidar Conradi and Alf Inge Wang. *Empirical methods and studies in software engineering: experiences from ESERNET*. Vol. 2765. Springer, 2003 (cit. on p. 120).

[6] 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 p. 122).

[7] 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 p. 117).

[8] Vinicius HS Durelli, Jeff Offutt, Nan Li, Marcio E Delamaro, Jin Guo, Zengshu Shi, and Xinge Ai. "What to Expect of Predicates: An Empirical Analysis of Predicates in Real World Programs ". In: *Journal of Systems and Software*. Vol. 113. Elsevier, 2016, pp. 324–336 (cit. on p. 133).

[9] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. "Automated Test Generation using Model Checking: an Industrial Evaluation". In: *International Journal on Software Tools for Technology Transfer*. Vol. 18. 3. Springer, 2014, pp. 335–353 (cit. on pp. 116, 117, 120, 122).

[10] 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. 116, 117, 133).

[11] 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. 116, 118, 132).

[12] 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 p. 117).

[13] Moheb Ramzy Girgis and Martin R Woodward. "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria". In: *Proceedings of the Workshop on Software Testing*. IEEE, 1986 (cit. on p. 129).

[14] David Howell. *Statistical Methods for Psychology*. Cengage Learning, 2012 (cit. on p. 126).

[15] IEC. "International Standard on 61131-3 Programming Languages". In: *Programmable Controllers*. IEC Library, 2014 (cit. on pp. 116, 119).

[16] 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 p. 117).

[17] Marcin Jamro. "POU-Oriented Unit Testing of IEC 61131-3 Control Software". In: *Transactions on Industrial Informatics,* vol. 11. 5. IEEE, 2015 (cit. on p. 117).

[18] 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 p. 117).

[19] K.H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010 (cit. on pp. 116, 117, 119, 120).

[20] 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 pp. 122, 133).

[21] 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. 117).

[22] 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. 118, 132).

[23] 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. 121).

[24] Hareton KN Leung and Lee White. "A Cost Model to Compare Regression Test Strategies". In: *Software Maintenance*. IEEE, 1991, pp. 201–208 (cit. on p. 123).

[25] Younju Oh, Junbeom Yoo, Sungdeok Cha, and Han Seong Son. "Software Safety Analysis of Function Block Diagrams using Fault Trees". In: *Reliability Engineering & System Safety*. Vol. 88. 3. Elsevier, 2005, pp. 215–228 (cit. on p. 122).

[26] 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 p. 117).

[27] 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. 133).

[28] 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. 116).

[29] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. "Control System Devices: Architectures and Supply Channels Overview". In: *Sandia Report SAND2010-5183*. Sandia National Laboratories, 2010 (cit. on p. 116).

[30] 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 p. 118).

[31] Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae. "Empirical Evaluation on FBD Model-based Test Coverage Criteria using Mutation Analysis". In: *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 465–479 (cit. on p. 122).

[32] 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 p. 117).

[33] R Development Core Team. *R: A language and environment for statistical computing. http://www. R-project. org. The R Foundation for Statistical Computing, Vienna, Austria, (2005)*. URL: `http://www.%20R-project.%20org` (cit. on p. 126).

[34] 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. 116, 117, 133).

[35] András Vargha and Harold D Delaney. "A critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong". In: *Journal of Educational and Behavioral Statistics*. Vol. 25. 2. Sage Publications, 2000, pp. 101–132 (cit. on p. 126).

[36] 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. 116, 117, 133).

[37] 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. 118, 132).

[38] 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 p. 117).

[39] 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. 117).

Study 5

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

Eduard Paul 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.

The paper was reformatted for uniformity, but otherwise is unchanged.

Study 5. Mutation-Based Test Generation for PLC Embedded Software using Model Checking

Eduard Paul Enoiu, Daniel Sundmark, Adnan Causevic, Robert Feldt and Paul Pettersson

Abstract

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

1 Introduction

Software testing is an important verification and validation activity used to reveal software faults and make sure that actual software behavior matches its expected behavior

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

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

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

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

2 Background and Related Work

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

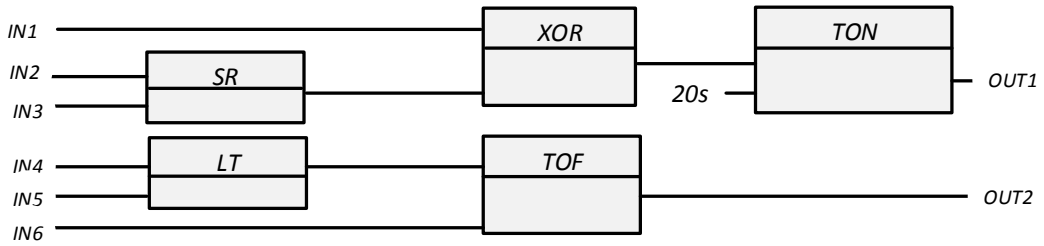


Figure 5.1: An FBD program with six inputs and two outputs.

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

2.1 PLC Embedded Software

Safety-critical embedded systems implemented using Programmable Logic Controllers (PLCs) are used in many industrial application domains such as electric, transportation, chemical, pharmaceutical, etc.. One of the programming languages defined by the *International Electrotechnical Commission* (IEC) for PLCs is the Function Block Diagram (FBD) language [16]. Programs developed in FBD are compiled into program code, which in turn is compiled into machine code by using specific engineering tools provided by PLC vendors. The motivation for using FBD as the target language in this study comes from the fact that it is the de facto standard in many industrial systems [26], such as the ones in the railway transportation domain. Programs running on a PLC execute in a loop, in which the iteration follows the *"read-execute-write"* semantics. FBD is popular because of its graphical notations and its usefulness in applications with a high degree of data flow between control components. As shown in Figure 5.1, predefined logical and/or stateful blocks (i.e., SR, XOR, TOF, LT and TON in Figure 5.1) and signals (i.e., connections) between blocks represent the behavior of an FBD program. The blocks are supplied by the hardware manufacturer or defined by a developer. PLCs contain particular types of blocks called timers (e.g., TON and TOF) that provide the same functions as timing relays in electrical circuits and are used to activate or deactivate a device after a preset interval of time. For more details on this programming language we refer the reader to the work of John et al. [20].

2.2 Automated Test Generation for PLC Embedded Software

In general, automated test generation has been explored in a considerable amount of work [25] in the last couple of years. Numerous techniques for automated test generation using code coverage criteria (e.g., [9, 4]) have been proposed in the last decade, since test suites can be created and executed with reduced human effort and cost. However, for domain specific languages used in embedded software development, contributions have been more sparse. For IEC 61131-3 software, a few automated test generation approaches [30, 18, 28] have been proposed in the last couple of years,

but currently there is a lack of tool support. In our previous work, we developed an automated test input generation approach and tool named CompleteTest [8], which automatically produces test suites for a given coverage criterion and an IEC 61131-3 program written using the FBD language. CompleteTest supports different code coverage criteria with the default criterion being decision coverage.

2.3 Mutation Testing

Recent work [13, 17] suggests that coverage criteria alone can be a poor indication of fault detection in testing. To tackle this issue, researchers have proposed approaches for improving fault detection by using mutation analysis as a test criterion. Mutation analysis is the technique of automatically generating faulty implementations of a program for the purpose of examining the fault detection ability of a test suite [6]. A **mutant** is a new version of a program created by making a small change to the original program. The execution of a test case on the resulting mutant may produce a different output as the original program, in which case we say that the test case **kills** that mutant. The mutation score is calculated using either an output-only oracle (i.e., strong mutation [29]) or a state change oracle (i.e., weak mutation [15]) against the set of mutants. For all programs, one needs to assess the fault-finding effectiveness of each test suite by calculating the ratio of mutants killed to total number of mutants. When this technique is used to *generate* test suites rather than evaluating existing ones, it is commonly referred to as *mutation testing* or mutation-based test generation. Despite its effectiveness [21], to the best of our knowledge, no attempt has been made to propose and evaluate mutation testing for PLC embedded software written in the IEC 61131-3 FBD programming language. This motivated us to develop an automated test generation approach based on mutation testing targeting this type of software.

3 Mutation Test Generation for PLC Embedded Software

Within the last decade *model-checking* has turned out to be a useful technique for generation of test cases from models [10]. In this paper, we describe an approach to automatically generate test suites using a model checker based on mutation testing for PLC embedded software. Overall, the approach is composed of the following steps, mirrored in Figure 5.2:

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

2. Model Aggregation. The second step (described in detail in Section 3.2) is used for combining a program and the set of mutants into a single model. The

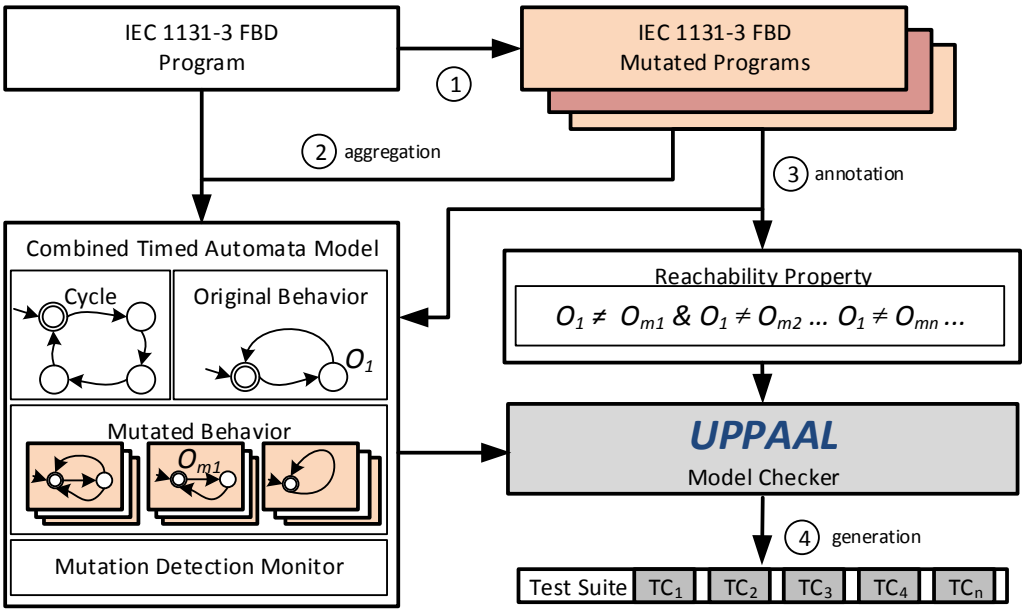


Figure 5.2: Overview of mutation testing for IEC 61131-3 FBD programs.

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

3. Mutant Annotation. The third step (described in Section 3.3) involves the annotation of the combined model with instrumentation instructions for the detection of each mutant. This means that the mutation detection monitor is used to record the mutant execution and detection, thus for all mutants a property is created for checking the detection of mutants.

4. Test Suite Generation. The fourth step (described in Section 3.4) requires the use of the Uppaal model checker [22] to generate a set of test cases satisfying the detection of mutants by using the model checker's ability to export abstract traces witnessing a submitted property.

3.1 Mutation Generation

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

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

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

3.2 Model Aggregation

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

using this already developed translation for obtaining the model needed for running mutation-based test generation. Let M be a finite set of of mutants, each of which contains one syntactic change in the original program P. The model aggregation step is applied as follows:

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

As a result of the model aggregation step we consider that the combined model is a closed network of timed automata. This model, briefly shown in Figure 5.2, contains four processes, two modeling the program and its mutants and the other two supervising the overall execution and monitoring the mutant detection. To show an example of an aggregated model cycle scan, different actions are executed: `read(IN)` for reading input variables, `write(OUT)` for updating the output variables, and `write(OUT(`m_i`))` for updating the duplicated output variables corresponding to each mutant m_i. When the execution order holds, the input variables are updated and the execution continues to the next block.

3.3 Mutant Annotation

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

For annotation, it should be noted that there are different interpretations of how to implement mutation analysis. The most common implementation, called strong mutation deals with the comparison of the original and mutated program outputs at the end of the execution cycle. Another way is weak mutation, which compares

the state of the program immediately after the execution of the mutated part of the program. As these implementations can be useful in their different interpretation of mutation analysis, our approach employed both approaches.

Weak Mutation.

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

Strong Mutation.

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

3.4 Test Generation

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

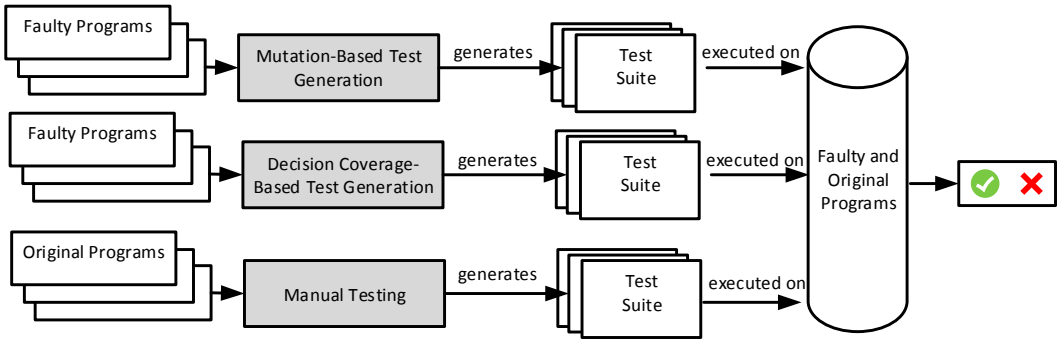


Figure 5.3: Overview of the experimental setup used to perform the case study.

test cases containing inputs, actual outputs and timing information (i.e., the time parameter in the test suite is expressing timing constraints within one program).

4 Experimental Evaluation

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

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

The case study setup is shown in Figure 5.3. From a high level view we started the case study by collecting: (i) a set of real industrial programs from a recently developed train control management system (TCMS), and (ii) manual test suites created for the above programs by industrial engineers. The studied programs were already thoroughly tested and are currently used in a set of operational trains. For all programs, test suites were also generated for weak mutation, strong mutation and decision coverage (as detailed below).

In order to measure fault detection, realistic faulty versions of the programs under test are required. However, the data set did not contain any information about what faults occurred during development, as Bombardier Transportation AB does not keep any such data in a format that could be directly collected post-mortem at this level of testing. To overcome this issue, several engineers from Bombardier were asked to manually create a number of faults for the programs considered in this study. We obtained faults from engineers at Bombardier Transportation manually introducing relevant faults in some of the programs considered in this study. Since mutation-based test generation is using an existing program implementation to guide

the search, we automatically generate all tests suites using the seeded faults instead of the original programs. This corresponds to the realistic situation where an engineer has made a fault located in the program to be tested. In summary, we used a TCMS system containing 61 programs provided by Bombardier Transportation AB. These programs contained on average per program: 828 lines of IEC 61131-3 FBD code, 22 decisions (i.e., branches), 11 input variables and 5 output variables.

Manually Seeding Faults.

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

Test Generation

For each faulty program, we ran mutation and decision-coverage test generation ten times using a random-depth-first search (RDFS) strategy with random seed (i.e., test suites are varying from run to run), each test generation run with a stopping time limit for the search of 10 minutes. The stopping criteria for the search is three-fold: achieving 100% mutation score, reaching the time limit of 10 minutes, or getting a memory exception. We chose a time limit of 10 minutes for the sake of this experiment. In addition, we used manual test suites created by industrial engineers in Bombardier Transportation from a TCMS project delivered already to customers. Manual test suites were collected by using a post-mortem analysis of the test data available. The test suites collected in this study were based on functional specifications expressed in a natural language. Practically, we considered the original TCMS programs and for each faulty program, we executed the test suites produced

by manual testing for the original program. Finally, for all test suites we collected the following measures: generation time, execution time, number of test cases and fault detection score. In order to calculate the fault detection score, each test suite was executed on both the original program and its faulty counterpart. In case the results differed between the executions, the fault was considered to be detected.

Measuring Cost.

We measured the cost of performing testing focusing on the unit testing process as it is implemented in Bombardier Transportation for testing the programs selected in this case study. For the TCMS system, the creation and execution of test cases is performed by the implementer of the IEC 61131-3 software. In the cost measure, we use *the creation cost*, *the execution cost*, and *the result check cost*. The cost does not include the required tool preparation, the reporting and the maintenance of the test suite. We consider that all cost components related to human effort are depended to the number of test cases. The higher the number of tests cases, the higher are the respective costs. We assume this relationship to be linear with a constant factor representing the average time spend by an engineer in each cost component for a test case. Practically, we measured the costs of these activities directly as an average of the time taken by three industrial engineers (working at Bombardier Transportation implementing some of the IEC 61131-3 programs used in our case study) to perform manual testing.

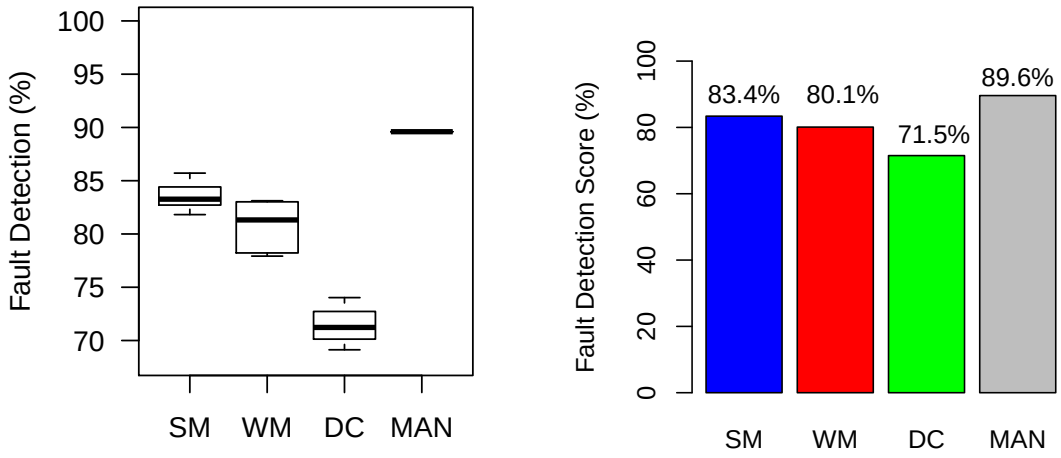
5 Experimental Results and Discussion

The case study presented us with a fault detection score and a cost measurements for each of the collected test suites (i.e., manually created test suites by industrial engineers (MAN), mutation-adequate test suites (i.e., weak-mutation testing (WM), strong-mutation testing (SM)) and automatically generated test suites based on decision coverage (DC)). The overall results of this study are summarized in the form of boxplots[1] in Figure 5.4 and 5.5.

Fault Detection.

To answer RQ1 regarding the fault detection, in terms of detection of manually seeded faults, we focused on comparing all DC, WM, SM and MAN test suites. For all programs, as shown in Figure 5.4, the fault detection scores obtained by manual written test suites are higher in average with 9% and 6% than those achieved by weak mutation and strong mutation respectively. The difference in fault detection is slightly greater between strong-mutation testing and decision coverage-adequate testing (i.e., a difference of almost 12% on average). To understand how manual test suites achieve better fault detection than mutation-adequate test suites, we examined if the test suites are particularly weak or strong in detecting certain type

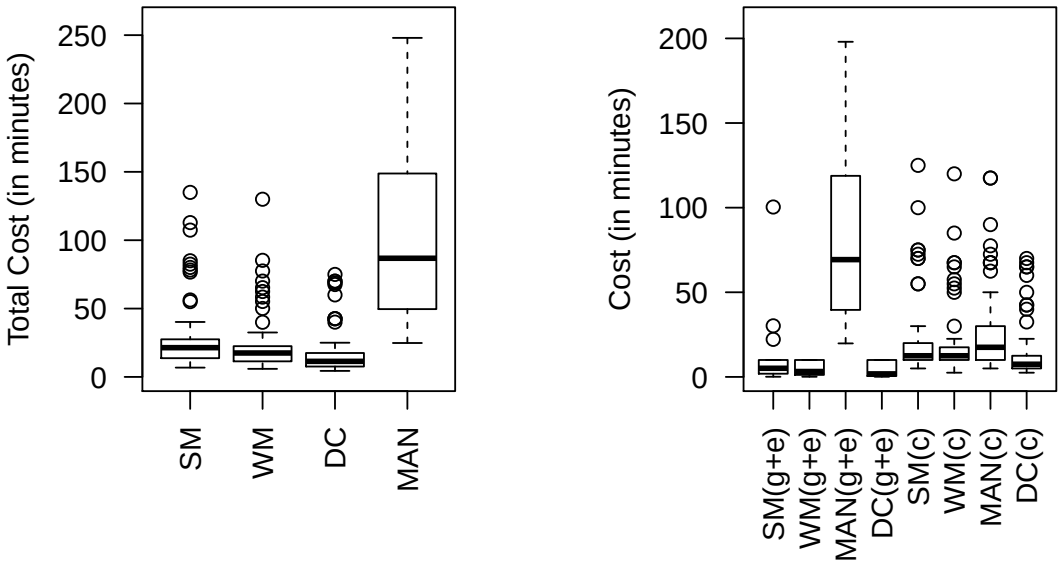
[1]boxes spans from 1st to 3rd quartile, black middle lines mark the median and the whiskers extend up to 1.5x the inter-quartile range and the circle symbols represent outliers.



(a) Overall Fault Detection Comparison. (b) Average Fault Detection Score.

Figure 5.4: Fault detection results for manual testing (MAN), decision coverage-directed test generation (DC), weak mutation testing (WM) and strong mutation testing (SM).

of faults. We concern this analysis to what kind of faults were detected by manual testing and not by strong-mutation test generation. From a total of 77 faults, we identified eight faults (i.e., for exemplification purposes these faults are named Fault 1-8) that were not detected by any strong-mutation test suite while being detected by manual test suites. To produce meaningful results the remaining 69 faults are not included in this fault detection analysis because there is no consistent difference between manual and strong-mutation test suites. There are some broad trends for eight faults that can be used for explaining at least the difference in fault detection between manual and strong-mutation testing. Test suites written using manual testing are able to detect all of these eight faults. Mutation test suites are achieving a poor selection of test inputs produced for detecting certain faulty behaviors; for six faults, strong mutation testing generated test suites achieving 100% mutation score while for the remaining two faults, the model checker was unable to find a test suite detecting all mutants, given the 10 minutes time limit. It seems that manual testing has a stronger ability to detect these faults than mutation testing because of its inherent advantage of relying also on the specification of the program under test. For four of the faults, multiple changes in the program have been seeded (e.g, two or more blocks and variables have been replaced, deleted or inserted). For example, Fault 1 contains three changes combining three simpler faults corresponding to the application of CRO and VRO mutation operators. Fault 2 contains a combination of seeded changes corresponding to the creation of mutants using LRO and NIO mutation operators. In addition, Faults 3 and 4 contain multiple changes that were



(a) Overall Cost Results

(b) Detailed Cost Results

Figure 5.5: Cost measurement results for manual testing (MAN), decision coverage-directed test generation (DC), weak mutation testing (WM) and strong mutation testing (SM).

not captured by previously defined mutation operators. On the other hand, four of the faults are first order faults containing only one change in the program. A feedback loop signal connecting one of the outputs of the programs with one of the blocks was seeded in Faults 5 and 6. On the other hand, Fault 7 contains an extra logical block that was added to the original program while in Fault 8 a constant variable has been replaced to a non-boundary value. As a direct result, we discuss in Section 5.1 the improvement of mutation-based test generation for PLC software by considering additional mutation operators not considered before in the literature [23, 27] to model possible faults.

Cost.

We interviewed three engineers working on developing and manually testing TCMS software and asked them to estimate the time (in minutes) needed to create, execute and check the result of a test suite. All engineers independently provided similar cost estimations. We averaged the estimated time given by these three engineers and we calculated each individual cost using the following constants: 6.6 minutes for the creation of a test case, 3.3 minutes for the execution of a test case and 2.5 minutes for the checking of the result of a test case. Practically, for answering RQ2, we used these constants and the number of test cases in each test suites to represent the average time spend by an engineer to manually test each program. In addition,

for mutation-based and decision coverage-directed test generation the total cost involves both machine and human resources. We calculated the cost of generating and executing a test suite by directly measuring the time required by the tool to run the test generation and the time required to execute each test case. For the cost of checking the test result we used the same average time as for manual testing (i.e. 2.5 minutes for the checking of the result of a test case). The resulting cost measures are reflected in Figure 5.5. The cost of performing testing using mutation testing either weak or strong is consistently significantly lower than for manually created test suites; automatically generated test suites have a smaller testing cost (110 and 115 minutes shorter testing time on average for WM and SM respectively) than the cost of using manual test suites. A more detailed cost measurement would be needed to obtain more confidence in the cost results obtained in this study.

5.1 Discussion

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

Improving Mutation Testing for PLC Programs.

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

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

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

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

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

Mutation Testing using Model Checking.

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

6 Conclusions

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

Acknowledgments

This research was supported by The Knowledge Foundation (KKS) through the following projects: (20130085) Testing of Critical System Characteristics (TOCSYC), Automated Generation of Tests for Simulated Software Systems (AGENTS), and the ITS-EASY industrial research school.

References

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. "Model-Checking for Real-Time Systems". In: *Logic in Computer Science*. 1990, pp. 414–425 (cit. on p. 146).

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008 (cit. on p. 142).

[3] 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. 155).

[4] 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. 142, 143).

[5] CENELEC. "50128: Railway Application–Communications, Signaling and Processing Systems–Software for Railway Control and Protection Systems". In: *Standard Report*. 2001 (cit. on p. 142).

[6] 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 p. 144).

[7] Richard A Demillo and Jefferson A Offutt. "Constraint-based automatic test data generation". In: *Transactions on Software Engineering*. Vol. 17. 9. IEEE, 1991, pp. 900–910 (cit. on p. 142).

[8] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. "Automated Test Generation using Model Checking: an Industrial Evaluation". In: *International Journal on Software Tools for Technology Transfer*. Vol. 18. 3. Springer, 2014, pp. 335–353 (cit. on pp. 144, 146).

[9] 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. 142, 143).

[10] 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 p. 144).

[11] 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. 142).

[12] Angelo Gargantini. "Using Model Checking to Generate Fault Detecting Tests". In: *International Conference on Tests and Proofs*. 2007, pp. 189–206 (cit. on p. 155).

[13] 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 p. 144).

[14] Jens Godskesen, Brian Nielsen, and Arne Skou. "Connectivity Testing Through Model-Checking". In: *International Conference on Formal Techniques for Networked and Distributed Systems*. 2004, pp. 167–184 (cit. on p. 155).

[15] 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. 144).

[16] IEC. "International Standard on 61131-3 Programming Languages". In: *Programmable Controllers*. IEC Library, 2014 (cit. on pp. 143, 147).

[17] 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. 142, 144).

[18] Marcin Jamro. "POU-Oriented Unit Testing of IEC 61131-3 Control Software". In: *Transactions on Industrial Informatics,* vol. 11. 5. IEEE, 2015 (cit. on p. 143).

[19] 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. 155).

[20] K.H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010 (cit. on p. 143).

[21] 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 pp. 142, 144).

[22] 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 pp. 145, 146).

[23] Younju Oh, Junbeom Yoo, Sungdeok Cha, and Han Seong Son. "Software Safety Analysis of Function Block Diagrams using Fault Trees". In: *Reliability Engineering & System Safety*. Vol. 88. 3. Elsevier, 2005, pp. 215–228 (cit. on pp. 145, 153).

[24] Vadim Okun, Paul E Black, and Yaacov Yesha. "Testing with Model Checker: Insuring Fault Visibility". In: *Transactions on Systems*. Vol. 2. 1. 2003, pp. 77–82 (cit. on p. 155).

[25] 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 p. 143).

[26] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. "Control System Devices: Architectures and Supply Channels Overview". In: *Sandia Report SAND2010-5183*. Sandia National Laboratories, 2010 (cit. on p. 143).

[27] Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae. "Empirical Evaluation on FBD Model-based Test Coverage Criteria using Mutation Analysis". In: *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 465–479 (cit. on pp. 145, 153).

[28] 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 p. 143).

[29] 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. 144).

[30] 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 p. 143).