Evaluation of Structural Testing Effectiveness in Industrial Model-driven Software Development

Author: Mahdi Sarabi
Email: msi10001@student.mdh.se
Date: June 2012
Carried out at: Bombardier Transportation AB

Advisors at Mälardalen University:
Eduard Paul Enoiu
Daniel Sundman

Advisors at Bombardier Transportation:
Johan Bergman
Anders Öberg

Examiner: Paul Pettersson
Abstract

Software testing is a crucial but cost consuming practice in the development of safety critical software-intensive systems. In industry software testing techniques are chosen heuristically rather than scientifically. Researchers have studied for many years which test method is more efficient and effective in an industrial setting. Nevertheless, test design decisions still remain a challenge when applied in industry. Structural testing effectiveness is influenced by program code structure and the software development method used for designing and implementing industrial software-intensive systems. In the field of structural testing, the industrial approach toward different development methods and programming standards is studied insufficiently in the literature. In this thesis, we have found that applying structural testing on the actual generated source code from a Function Block Diagram (FBD) conforming to the IEC 61131-3 industrial standard is not effective. The FBD model is transformed to an FBD program code composed of almost exclusively assignments with Boolean expressions and function calls, which are not detected in code coverage analysis. In this thesis, we use the FBD model as a good candidate for measuring structural coverage, since it expresses the system behavior and it is the model from which the actual target code is generated. We propose a new automated method with potential application to our case study that evaluates the test suite based on structural coverage criteria. This leads to a shorter lifecycle with more effective structural testing for model-driven software development.
Acknowledgments

I would like to thank my supervisors Eduard Paul Enoiu and Daniel Sundmark from Mälardalen University, as well as Johan Bergman and Anders Öberg from Bombardier Transportation AB for their comments, recommendations and guidance during this thesis. They dedicated lots of time and effort to help me successfully finish this thesis. I am very thankful for all the discussions we had in our meetings.

Besides, I would like to express my thankfulness to Ola Sellin, Paul Pettersson and Daniel Flemström who supported me in all ways for making this thesis possible. I am grateful for their help.

In addition, I would like to acknowledge Lars Persson who supplied me with the examples from Bombardier Transportation AB used in this thesis.

Last but not least, I would like to take this opportunity to thank my beloved wife Jenan, and my parents for their enduring support.

Västerås, June 2012
Acronyms

CCU-C       Central Computing Unit Comfort
CCU-D       Central Computing Unit Diagnostics
CCU-O       Central Computing Unit Operation
CCU-S       Central Computing Unit Safe
D-U         Definition – Use
FBD         Function Block Diagram
HMI         Human Machine Interface
HW          Hardware
IP          Internet Protocol
MC/DC       Modified Condition/Decision Coverage
MITRAC      Product name for the train control system
MLM         MainLine and Metros – Bombardier division
MVB         Multi-function Vehicle Bus
MWT         MultiProg Work Tool – TCMS programming tool
PPC         ProPulsion and Control – Bombardier division
SW          Software
TCMS        Train Control and Management System
# Table of Contents

Table of Contents .................................................................................................................. 5  
Introduction .......................................................................................................................... 7  
  1.1 Motivation and Problem Statement .................................................................................. 7  
  1.2 Contribution .................................................................................................................... 8  
  1.3 Methodology ................................................................................................................... 9  

Literature Review on Structural Testing .................................................................................. 10  
  2.1 Adequacy Criteria ......................................................................................................... 10  
  2.2 Structural Testing .......................................................................................................... 10  
  2.3 Control-flow Representation of the Code ...................................................................... 11  
    2.3.1 Code Segment ..................................................................................................... 12  
    2.3.2 Predicate and Condition ................................................................................... 12  
    2.3.3 Control Flow Graph Representation .................................................................. 12  
  2.4 Structural Coverage Criteria .......................................................................................... 13  
    2.4.1 Statement Coverage .......................................................................................... 13  
    2.4.2 Branch Coverage .............................................................................................. 14  
    2.4.3 Condition Coverage ........................................................................................... 15  
    2.4.4 Path Coverage ................................................................................................... 18  
    2.4.5 Data-flow Coverage ........................................................................................... 18  
  2.5 State-of-practice in Structural Testing ............................................................................ 19  
    2.5.1 Why Structural Coverage is Difficult in Practice? .............................................. 20  
    2.5.2 Test Suite Size and Code Coverage Relation ...................................................... 20  
    2.5.3 There is No Guarantees for Absence of Faults ................................................. 20  
  2.6 Coverage and Software Quality Relation ....................................................................... 20  
  2.7 Distribution of Faults .................................................................................................... 21  
  2.8 Subsumption Relation of Coverage Criteria .................................................................. 21  

Case Study Background ........................................................................................................ 23  
  3.1 TCMS ............................................................................................................................ 23  
    3.1.1 CCU-O Software .............................................................................................. 24  
  3.2 TCMS Software Development Tools and Methods ....................................................... 24  
  3.3 TCMS Testing Process .................................................................................................. 25  
    3.3.1 SoftTCMS .......................................................................................................... 26  
    3.3.2 TestCocoon ......................................................................................................... 26  
    3.3.3 Component Testing Method ............................................................................... 27  

Structural Testing Based on the FBD Program Code ............................................................. 30  
  4.1 IEC 61131-3 Function Block Diagram Model to Code Transformation ............................ 30  
  4.2 Structural Testing Behavior on the FBD Program Code ................................................. 31  
  4.3 Branching Points in the FBD Program Code ................................................................. 32  
  4.4 Explicit Control Statement VS. Boolean Expression Assignment ................................ 33
4.5 Statement Coverage Behavior on the FBD Program Code .......................................................... 34
4.6 Branch Coverage Behavior on the FBD Program Code ............................................................... 34
4.7 MCDC Behavior on the FBD Program Code .............................................................................. 34
4.8 Coverage Criteria Comparison Cased on the FBD Program Code Structure ....................... 35
4.9 Estimation of Achievable Coverage on the FBD Program Code .................................................. 36
4.10 Evaluation of a Commercial Code Coverage Tool (TestCocoon) and Requirements ........... 36
4.11 Effectiveness Analysis of Code Coverage Criteria on the FBD Program Code .................... 37
4.12 Chapter Summary .................................................................................................................. 39

Structural Testing Based on the FBD Model .................................................................................. 40

5.1 Manual Method for Applying Structural Testing on a FBD Model .............................................. 40
5.2 Evaluation of Applying Manual Structural Coverage on a FBD Model .................................. 42
5.3 New Automatable Test Suite Evaluation Based on a Coverage Criterion Method .................. 42

5.3.1 A Methodology for Test Suite Evaluation Based on a Coverage Criterion ......................... 42
5.3.2 Evaluation of Using Test Suite Evaluation Based on a Coverage Criterion ....................... 46
5.3.3 Application Example: Evaluation of Test Suite Based on a Coverage Criterion ................. 47

Conclusion ....................................................................................................................................... 49
Future Work ..................................................................................................................................... 51
References ........................................................................................................................................ 52
Chapter 1

Introduction

Testing is an unavoidable activity in the software development process for all types of software intensive systems. Testing is primarily used for verification purposes in the software industry. Different methods and strategies for testing come in different forms, usage methods, and application targets in terms of the type of software system to be developed or maintained. Particularly, in safety critical systems, testing is very important because poor quality software can cause bugs and could lead to malfunctioning devices and maybe catastrophic failure. Very often Programmable Logic Controllers (PLCs) are used in industrial safety-critical systems such as railway and avionics.

As with most other software, testing safety-critical software systems is not a single activity; it is a process that begins when the software starts and continues beyond implementation towards software maintainability. This activity is present in all phases of the software development life cycle. This introduces different testing types (i.e. Black Box, Gray Box, White Box) and levels (i.e. Regression, Acceptance, System, Integration, Module, or Unit) [1]. Applying these testing types and going through different levels does not mean we have fault free software, because faults could be still uncovered. The question remains when should we stop testing the software? Testing is never enough. That is why there are several types and levels for testing, including structural testing. Structural testing is a type of testing based on the structure of the source code or the model of the implemented code and measures the coverage percentage of the code or the model structure as exercised by the test cases. This means that a structural coverage indicator should be used for evaluation purposes. Structural testing can be used to increase the thoroughness of testing and provide a coverage measure. Nevertheless, this does not mean that testing driven by coverage criteria is enough. Also, we argue that there has been little rigorous research and practice on applying test criteria on PLC development process.

In this thesis, we study structural testing techniques, their usage, their actual specification, and their application on a safety-critical type of software named Function Block Diagram (FBD). This language is one of the five programming languages defined by the International Electrotechnical Commission (IEC) [2] for PLCs. FBD programming language conforms to the IEC 61131-3 industrial standard and is one of the most widely used standard languages to implement safety-critical software [3]. Also, we discuss how to facilitate a more systematic method of determining the structural adequacy of FBD programs using state-of-practice tool chains used by Bombardier Transportation AB in their component testing process of their Train Control Management System (TCMS).

1.1 Motivation and Problem Statement

In safety-critical industry there is common for companies (i.e. Bombardier Transportation AB) to use a specific software development process and method, although the software development tool follows a detailed safety standard.
Traditionally there exists a gap between PLC development and structural test criteria, making it difficult for testers to use structural indicators without relying mostly on simulation methods. There is a need in safety-critical software industry on applying structural testing on the FBD software implementation that conforms to the IEC 61131-3 standard. FBD programs are transformed into FBD implementation code, which is compiled into PLC machine code automatically by using specific engineering tools provided by PLC vendors. The actual implementation code is generated usually automatically via a transformation tool from FBD models designed in a graphical editor. The development method and the code structure impact the effectiveness of the used structural testing method and the applicable coverage criterion.

In the FBD model to FBD program code transformation the actual code is optimized, and therefore the model or the requirement specification might not be reflected in the actual code structure. It may be the case in practice that there is a requirement for applying structural testing based on this automatically generated code. Using structural testing on the FBD program code can be preferred at first, since this is the traditional suggestion and there are many commercial code coverage tools specifically build for different programming languages.

Alternatively, applying structural coverage on the FBD model that conforms to the PLC development process and finding a suitable methodology could be more appropriate if applying structural testing on the FBD program code is not successful. Further, it is difficult to tell whether applying structural testing based on the FBD model is not the only option, although finding an optimal solution could be interesting and could be a possible way to achieve effective structural testing during safety-critical system development process.

1.2 Contribution

Based on the problems outlined in the previous section, we formulate our contributions in terms of reason. We have identified three reasons behind the aim of using structural testing and our suggestions should provide the means for integrating them into an industrial setting in the actual testing process:

- **R1**: To satisfy standard safety requirements and obtain higher quality software;
- **R2**: To have an indicator for testing progress and a common interface between the testers and designers/implementers;
- **R3**: To have more thorough tests and a test thoroughness indicator.

This thesis reviews the literature for structural testing that are useful in an industrial setting. We assessed different structural criteria and we analyze their industrial applicability in a safety-critical case study provided by Bombardier Transportation AB. We have studied the factors limiting the application of structural testing on the FBD program code generated from an FBD model. The results of this study are showing that applying structural testing on the FBD program code is not effective due to the code structure and how applying structural testing on the FBD model can support decisions for effective component testing. Subsequently, we reflect on the use of an evaluation tool based on structural test criteria in an industrial setting. We have suggested two methods based on the FBD model as alternatives for applying structural testing. The first method is manual and requires
some effort from the tester. The second method is an automatable evaluation of a test suite based on a coverage criterion.

1.3 Methodology

In this thesis like on any other research work we have started with a literature study and we went through several steps and ended up with solutions, experiments and a prototype tool to prove the applicability of the solution. We have performed a search process and a study of the related academic and industrial research papers in the following databases: IEEE Xplore, ACM digital library, Springer, and Google Scholar. The list of primary studies was collected based on the usage of the following keywords: structural coverage, code coverage, adequacy criteria, adequacy criteria /structural coverage/ code coverage (industry/ advantage/ disadvantage/ evaluation/ survey/ metric).

Also, we have studied several specific documents and manuals from Bombardier Transportation AB. We used the gained knowledge from the literature review for the analysis process that we have performed on the software testing process used at Bombardier Transportation AB. The process was investigated and analyzed thoroughly. After the literature study and analysis, we used the following tasks:

- **T1**: Study the code coverage behavior of three adequacy criteria on the FBD program code;
- **T2**: Evaluating the code coverage behavior on the FBD program code;
- **T3**: Identify possible solutions / methods;
- **T4**: Investigate and create appropriate experiments using the suggested solutions;
- **T5**: Verify and validate the solutions;
- **T6**: Automate and implement a prototype solution;
- **T7**: Report the solutions, the solution analysis and the actual approach;
Chapter 2

Literature Review on Structural Testing

One of the tasks that we have identified in the previous chapter is to review the available coverage metrics in the literature. First we focus on identifying, evaluating, and interpreting the available research studies relevant to structural testing. For many years high code coverage was considered as a convenient metric for assessing test adequacy [4,5]. Soon, the need for more sophisticated or adapted testing criteria was required. Therefore, several testing criteria based on code coverage were defined in the academic literature, ranging from program statements, to decisions, paths and data-flow relations [6].

2.1 Adequacy Criteria

Pezzè and Young [6] define the adequacy criteria as a set of test obligations. The test suite satisfies an adequacy criterion if all tests passed and every test obligation is satisfied by at least one of the test cases. To measure the adequacy criteria, there is the need of inspecting the code or the model elements, which are the test obligations, as much as possible in order to measure the testing adequacy. This is achieved by introducing an indicator based on the ratio of the inspected elements to the total elements.

Most of the adequacy criteria studied by researchers did not find place in the industry. According to Zhu, adequacy criteria are categorized into three main categories: structural, error-based, and fault-based [7]. Surveys show that only some basic structural criteria, such as statement and branch coverage, find its way actually into industrial practice. The industrial usage of other criteria is very limited or close to non-existent [8,9]. Some researchers mention cost as a major reason preventing software industry going towards an effective way of using adequacy criteria [7], while others also blame the difficulty of understanding and using this kind of techniques.

2.2 Structural Testing

Testing based on code structure is called structural testing, in which we judge test suite thoroughness based on the structure of the program [6]. In this case structural testing is defined as a type of specification based testing as follows:

“Structural testing is still testing product functionality against its specification. Only the measure of thoroughness has changed.” [6]

Structural testing is a set of metrics for evaluating the thoroughness of a test suite and a way of measuring coverage, based on how precisely and exhaustively the tests exercise certain aspects of the structure of the program. This means that structural testing is more specific and detailed than functional testing. Since faults are
distributed all over the program a more thorough test suite, such as a test suite applied to every statement in the code, is likely to reveal the existing faults.

Furthermore, structural testing is a way to complement functional testing (black-box testing). In black-box testing, test suites are created from the actual requirements in order to adequately test the behavior of the program-under-test without using the internal structure of the implementation. Unfortunately, there is no direct method of determining the adequacy of a black-box test suite [10]. Instead, the adequacy of such suites is determined by different criteria, which are based on either the source code [1,11] or models [12,13] by using executable artifacts.

For quantifying black-box testing, since executable artifacts are used and are measuring indirectly the adequacy of a test suite, the result is difficult to interpret. Assuming that the executed artifact yields coverage, if the implementation is missing functionality, a weak black-box test suite would not detect the missing functionality. In the other hand, poor coverage requires huge effort for interpretation of the measurement result [10], because this coverage requires to be justified or more test cases should be created in order to achieve higher percentage of the testing coverage. The higher effort requirement is due to black-box testing usage and unawareness of the code structure.

This thesis is based on the work of Pezzè and Young [6]. Their work shows three advantages of using structural testing for the quality managers:

1. Evaluate test suites.
2. Determine when to terminate testing.
3. Identify portions of the code that require additional testing.

Pezzè and Young also consider structural testing as a complementation for functional testing in the sense that some faults are not possible to be found by functional tests, such as code used in defensive programming, which is called infeasible path. Structural testing also helps to improve the documentation of a system specification by making it more complete. It is not a surprise to have differences between specification and implementation.

Another reason for using structural testing is to increase confidence in the thoroughness of testing, since it is automatable and measurable. In addition, the measurement results could be used as progress indicator or a criterion of completion. Structural testing also could be used to identify redundant test cases. As a result we could have a more comprehensive test suite with less redundant test cases.

2.3 Control-flow Representation of the Code

Control flow graph is widely used for analysis, testing and representation of programs and it had been studied for many years [14,15,7,16,17]. Most structural coverage criteria rely on control flow graph (model), which gives a graph representation of the code. Elements of the graph correspond to different code parts, which are code segment, predicate, and condition [18].

We use in the following sections a source code example, which is depicted in Figure 1. The code example is separated based on the elements of the code. The capital letters are the element labels, which will be explained further in the following subsections.
2.3.1 Code Segment

A code segment consists of multiple statements that are executed sequentially with no conditional execution in between. Code segments are represented often in one single block (node), but some researchers prefer to define a node for each statement [16]. Therefore, once a code segment is atomic, it means that once a node of the graph is entered all the associated statements will be executed. A predicate (i.e., a method exit, a loop control, a “break”, and a "goto" statement) can separate two sequential segments. The last statement (i.e. predicate or exit expression) in the segment is selecting the next segment code. In Figure 1(a) the labels A, C, E and F are referring to code segments. Notice that segment F is considered as a code segment. The segmentation is caused by the implicit ELSE branch of the IF statement, although it is not written in the code. That means the implicit ELSE branch causes segment and the segment followed by the ELSE segment, which is F in Figure 1(a) as an individual segment.

2.3.2 Predicate and Condition

Predicate is decomposed from one or many conditions that result on a true or false evaluation. The condition is a Boolean operator in the predicate expression. Predicates are part of the control statements such as if, case, do, while, do until or for. The labels B and D in Figure 1(a) are showing the predicates and Figure 1(b) is showing the decomposition of these predicates to conditions, which are B, C, E, F in Figure 1(b).

2.3.3 Control Flow Graph Representation

Control Flow Graph (CFG) is composed of nodes and edges that connect nodes. The nodes and the edges in the CFG correspond to code segments and branches caused by the control statements in the code.

Control flow graph can be represented graphically by using:

1. Nodes with label for the intermediate nodes
2. Node with only outbound arrow for entry node
3. Node with only inbound arrow for exit node
4. Arrows for edges
In Figure 2 we show models as the control flow graphs of our running example for both branch and condition separation. Each control or condition node has more than one outbound edge. Also, condition nodes have two edges and the control nodes for the IF statement have two edges. For some control statements such as “switch” we have more than two outbound edges represented.

2.4 Structural Coverage Criteria

Researchers have defined several criteria for structural coverage, while a few of them are used in the software industry. Since we focus on the train industry domain, only the coverage criteria that are of interest for the EN 50128 standard are studied.

2.4.1 Statement Coverage

One of the simplest metrics used as adequacy criteria is the statement coverage. In statement coverage each statement must be executed at least once. Statement coverage is a measure calculated as the ratio of statements executed by the test suite to the total number of statements in the program under test, which could be a simple module or component, as it is defined as follows:

\[
Coverage = \frac{\text{No. executed statements}}{\text{No. total statements}}
\]

Coverage is associated to the test suite adequacy. That means the tester should write a test suite that covers all statements at least once. Since we consider statement coverage based on control flow, the program is treated as a control flow model and all the multiple statements that are executed sequentially are often represented by basic blocks in a control flow model. This means 100% block coverage implies 100% code coverage.
Figure 3. A path that satisfies statement coverage.

A possible path sufficient for 100% statement coverage is depicted in Figure 3, which is as a path shown in the red color. A possible input that can satisfy the coverage is a = 2, b = 0, c = 4.

**Statement Coverage Strength & Limitation.** Statement coverage is a weak measurement but is relatively easy to achieve. Although, if the test suite can achieve 100% statement coverage that does not imply the execution of all branches in the program. As the Figure 3 shows, the blue edges, which are branches in the code, are not executed. Statement coverage is completely insensitive to conditions and logical operators showed in the code. Therefore problems related to conditions are not detected. Reaching the termination condition in loops cannot be detected, but we can report whether the loop body is executed. For example it could be the case that “consecutive switch labels” are not detected, because statement coverage assumes testing one switch label is as good enough as another label [19].

**2.4.2 Branch Coverage**

Branch or decision coverage is a criteria indicating whether every branch or decision point in the code is evaluated to true and false at least once. For instance both branches of “if” statement must evaluated to true and false, and for the switch statement each case must be examined. Therefore we must write a test suite that results in a true and a false outcome for each compound predicate or decision at least once. Branch coverage is measured by the ratio between the branches tested by the test suite and the total number of branches in the program under test, as defined follows:

\[
\text{Coverage} = \frac{\text{No. executed branches}}{\text{No. total branches}}
\]

Similar to the statement coverage, branch coverage is also based on control flow model. As mentioned earlier nodes in the control flow are equivalent to the statements in the code. To satisfy branch coverage on the model every edge must executed at least once. Therefore coverage of all nodes also satisfied. This means branch coverage includes statement coverage.
Figure 4 shows an example for branch coverage. The code requires at least two test cases to satisfy the branch coverage, while only one test case is satisfying the statement coverage. But on the other hand, branch coverage cover satisfies coverage of both nodes and edges. Examples of two inputs that can achieve branch coverage on the above example are: \(a=2, b=2, c=-1\) and \(a=3, b=0, c=1\).

**Branch Coverage Strength & Limitation.** Similar to the statement coverage, branch coverage also cannot identify how many times the loop iterated or whether is reached the terminating condition. The branch metric is relatively simple but more effective than statement metric since it includes coverage of if-statements, switch-statement cases, exception handlers, and interrupt handlers [19]. The branch metric considers the compound predicate as an atomic statement. Therefore branches within Boolean expressions which occur due to short-circuit operators are ignored [19]. We say there is a short circuit in a Boolean expression when the second argument is only executed or evaluated if the first argument is not enough to determine the value of an expression.

### 2.4.3 Condition Coverage

Condition coverage analyses the conditions of compound predicate in more detail and examines every condition individually. Four varieties of condition coverage are defined in [6], which are basic condition coverage, branch and condition coverage, MCDC and Compound condition coverage. We select MCDC to be investigated more. It can be considered that MCDC is more practical, effective and used in the highest level of safety critical system development [20]. Furthermore, MCDC subsumes all the mentioned criteria but the compound condition coverage.

**Basic Condition Coverage.** This criterion requires every individual condition in a predicate or in a compound Boolean expression to be evaluated to both true and false [21]. This metric covers all statements but not all branches. To clarify why it does not cover the branch coverage, assume the statement that we had in our example: IF \(((a==2) \mid \mid (c>1))\), to satisfy the basic condition coverage, it is enough to have \((a==2)\) evaluated to both true and false, and \((c>1)\) also evaluated to both true and false. We might have the following case True \mid \mid \text{False, False} \mid \mid \text{True} that satisfies the basic condition coverage but not the branch coverage since it is not examining the both branches of the “IF” statement.
Branch and Condition Coverage. This criterion combines the branch coverage with basic condition coverage [21]. Therefore, in addition to every individual condition that must be evaluated to true and false, the control predicate also must evaluate to true and false. Consequently, branch and condition coverage subsumes both branch coverage and basic condition coverage.

Compound Condition Coverage. This criterion requires that every combination of condition outcome in each decision to be covered. This means that the entire truth table for conditions in a compound Boolean expression or predicate [21]. Therefore there will be at most $2^N$ combinations for a predicate with $N$ conditions, which is causing the test suite size to grow exponentially. On the other hand it subsumes all metrics that we discussed so far.

| a=1 && b=0 | a=2 || c>1 | a | b | c |
|-----------|----------|-----|---|---|---|
| F         | F        |     | -1| -1| -1|
| F         | T        |     | -1|  0|  2|
| T         | F        |     |  2| -1| -1|
| T         | T        |     |  2|  0|  2|

Table 1. Test cases which are satisfying compound condition coverage.

We assume the “IF” statement from the example code in Figure 1: IF (a>1) && (b==0)). Since we have two conditions it requires $2^2$ combination. The total required test cases for our example are in number of 8, since we have two IF statements with a predicate composed of two conditions. The test cases are shown in Table 1. The colored flows in Figure 5 show the compound condition coverage on the CFG with sample test cases.

Modified Decision Condition Coverage (MCDC). For safety critical systems where the implemented program is mostly built from combined predicates, the complexity of the expression becomes a concern. MCDC is directly testing the combined Boolean
expressions effectively compared to functional testing and lower level of structural testing [22] without increasing the test cases number exponentially as it is the case in other type of condition coverage metrics [20]. In other words, MCDC is a way of finding important combinations among all other combinations. The important combinations are the ones which change one of the basic conditions in a compound condition can change the value of the whole compound condition [6]. MCDC could be satisfied by N+1 test cases for N basic conditions [6]. MCDC has three criteria that must be satisfied [20]:

1. Every condition in a decision in the program has taken on all possible outcomes at least once;
2. Every point of entry and exit in the program has been invoked at least once;
3. Each condition has been shown to independently affect the decision's outcome;

MCDC is basic condition coverage “C”, a decision coverage “DC”, and an additional condition “M” that means that every condition must independently affect the decision’s output. Therefore MCDC includes condition and decision coverage and it is included by compound condition coverage [6].

We assume the “IF” statement from the example code in Figure 1 and we apply MCDC on it: IF (a>1) && (b==0)). MCDC is selecting only the important combinations from the full truth table that we had in compound condition coverage. Important combinations are the ones that can affect the result. Therefore the red colored combination in Table 2 is removed from test cases. Also the running example in Table 2 shows that for 2 (N) condition we only require 3 (N+1) test cases, although the difference is only one test case in this example but it will differ very much with a bigger number of conditions in a compound expression.

<table>
<thead>
<tr>
<th>a&gt;1</th>
<th>b==0</th>
<th>Result</th>
<th>a&gt;1 &amp;&amp; b==0</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>One change does not affect the result</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>Only one change can affect the result</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Explanation of MCDC with an example.

We apply the same concept to the example from compound condition coverage. The Table 3 shows the same test data combinations in the running example by considering the effect of “don’t care” conditions, which are conditions that do not change the expression result. Therefore, we removing the redundant combinations and also we make sure that every individual condition can change the result. Any number could replace “X” in the value table because it does not affect the result. This effect is also called short circuit.

Although the total number of test cases for the example code remains the same in this case, we see that number of test cases required for satisfying “IF” statements is decreasing.
Table 3. Explanation of MCDC with short circuit.

| a && b = 0 | a || c > 1 | a | b | c |
|------------|------------|----|----|----|
| F          | F          | -  | X  | -1 |
| -          | -          | -  | -  | -1 |
| T          | F          | T  | 2  | -1 |
| T          | T          | -  | 2  | 0  |

2.4.4 Path Coverage

For the path coverage to be satisfied all possible paths in the code must be tested. Zhu, defines path coverage as follows: "Set $P$ of execution paths satisfies the path coverage criterion if and only if $P$ contains all execution paths from the begin node to the end node in the flow graph" [7].

The path is considered to be a sequence of decisions or branches in the control flow graph and it is not examining the individual condition in the compound predicates. Recalling that, edges are the branches connecting the nodes of the control flow and the nodes are block of statements that executed sequentially. Branches have several combinations and are introducing many paths, in complex code with many control statements where the number of paths grow exponentially. A code with $N$ control statements has $2^N$ distinct paths. Path coverage is measured by the ratio of paths tested by the test suite to the total number of paths in the code under test as follows:

$$ Coverage = \frac{\text{No. of executed paths}}{\text{Total No. of paths}} $$

Figure 6 shows an example of path coverage for our running example code. By comparing the path coverage and compound coverage graphs we can find that there are some paths such as ABCDF shown in Figure 5 is not covered with compound coverage criteria. This means that path coverage is not subsumed by any of the above-mentioned criteria.

2.4.5 Data-flow Coverage

In literature, some researchers [6] do not consider data-flow coverage as structural testing, while some others consider it as coverage metric from structural testing. Structural testing analyzes the code syntactically while the data flow coverage analyzes the code semantically, because it analyzes the data relations and dependencies.

Figure 6. Sample execution paths.
Data flow consists of two types of nodes. First type of nodes is where variables receive value called “Definition” (D), showed in Figure 7 by the green circles, and second type of nodes is where the variables’ value are used called “Use” (U), showed in Figure 7 by the red circles. “Definitions” are considered to be points like assignment statement and input statement. While “Uses” are points such as expression, output statement and return value. “Uses” are in two types: one that affects the control flow of the system called predicate use (p-use) and the other type that does not have immediate effect on the control flow called computational use (c-use).

The advantage of this metric is that the paths correspond exactly to the way that the code handles data. But in the other hand is does not subsume decision coverage and it is more complex compared to other criteria. The examples shown here are taken from [23].

![Figure 7. Explanation of DU data flow.](image)

2.5 State-of-practice in Structural Testing

The most commonly used in practice coverage criteria are statement coverage, decision coverage and condition coverage [24]. Usually coverage data is collected by automatic instrumentation, since manual instrumentation is error-prone and time consuming. The coverage analyzer, which is part of typical test automation environment, processes the collected data from the instrumented code and generates reports as a feedback for the next actions. There exist several settings for structural testing environment and it is not always an integrated test automation environment system. Nevertheless, it has the main parts, which are the code instrumenter, the coverage analyzer and the test execution environment. An “in-house” developed test execution environment includes usually a coverage analyzer as an integrated part. The feedback from the coverage analyzer could be used for selection of new test data. This means that the structural testing criteria are used for automatic or manual test case generation. It is recommended not to drive test cases based on the coverage criteria intending to satisfy coverage. Instead, it is recommended to get the advantage of both functional testing and structural testing and develop functional test suite based on the speciation and measure coverage against a structural metric. In this way, the metric is considered as a test suite adequacy criterion, rather than a test case selection criterion [25].

Obtaining high level code coverage is not always possible due to existence of infeasible paths, which are not satisfiable with any input data. Also we must take in consideration that code reuse and component based software development, since the
reused elements are more general than the strictly required specification, it introduces unavoidable infeasible paths especially if the reused parts are big. These portions of code must be justified, which is huge effort, and not counted in the coverage [26].

2.5.1 Why Structural Coverage is Difficult in Practice?

Although code coverage is not a new area and has been present since early 80s, it is still a hot issue. The code coverage did not raise that much interest in the software industry due to two main reasons that makes code coverage difficult to apply in the industry. First, it is due to difficulty of identifying the additional test cases that satisfy the uncovered criteria and execute the specific statement, branch, condition or other elements [27]. Second, it is difficult to have accurate coverage value in case of infeasible paths, dead code or data dependency in the code. Infeasible elements require extra effort to be proved and should not be counted in coverage measurement since they are elements that are impossible to be executed by any input condition [6,27].

2.5.2 Test Suite Size and Code Coverage Relation

Researchers have studied intensively the relation between code coverage, test suite size and effectiveness. Their experiments results show that code coverage does not depend on the size of the test suite only, and having a good coverage does not mean that we have an effective test suite. The experiments indicate that both controlled growth of test suite and increasing in coverage might improve the effectiveness of tests [28].

The ultimate goal for structural testing is to minimize the test suite size used to examine all the test requirements, and achieve code coverage with effective tests. One test case is more effective than another if it detects more faults than another does. Nevertheless, a test suite with more test cases may or may not be more effective than a test suite with fewer test cases.

2.5.3 There is No Guarantees for Absence of Faults

Similar to any other testing methods, structural testing with 100% coverage does not guarantee that the code is free from defects although if we could execute all control flow elements. For example, it could not reveal a fault when the faulty statement is not corrupted with some test values. As a result, its execution may not lead to failure or the corrupted state may not propagate through execution to eventually result in a failure. Therefore, the fault may remain undiscovered until the code is being executed with some appropriate values causing failure.

To have a better understanding of a faulty statement that could remain hidden until tested by the appropriate values. Let’s say C=A/B in a code that B could have any integer value. We could test this statement by many test values but it cannot reveal the error unless “B=0”. In our example divide by zero error is raised only if it is tested by the appropriate value.

2.6 Coverage and Software Quality Relation

In the topic of test adequacy measurement there is the question of when we should stop testing. Several researchers [26,29,30] have studied the relation between coverage and finding faults (software quality) by trying to quantify the relation. The
mathematical models [26] and the experimental results [29], [31] show that linear increase in code coverage results exponential improvement in the software quality, although 100% coverage can’t guaranty absence of defects. Some other researchers were more careful with defining the relation between the coverage and the software quality (effectiveness of test). They support the mentioned claim that linear relation between coverage and software quality does not exist, by their experiments, but that a nonlinear relation exist [28].

Figure 8. Ratio between code coverage and software quality [29].

The results from [29] are presenting the relation between the code coverage, statement coverage, and the software quality, as depicted in Figure 8. The experimental result of software quality growth by increasing the code coverage is not linear as expected and it is showed to have exponential behavior. The graph also shows that software quality rate growth faster for higher coverage percentage.

2.7 Distribution of Faults

Researchers studied distribution of faults in the code trying to find patterns for fault distribution, in order to ease finding the defects and resolve them. Barry and Basili [32] observed that concentration of faults in some areas of the code is higher and only correction of faults for that section can have high impact on improvement of the overall software quality. They state their observation, as follows: "About 80 percent of the defects come from 20 percent of the modules and about half the modules are defect free". Other studies [33,34,35] are supporting Barry and Basili’s hypothesis, by presenting evidence of Pareto-like distribution of faults, small number of modules causing majority of defects, in industrial systems. The study found that faults are heavily concentrated in small number of files.

This observation can guide the testing process to allocate more resources to the error prone modules, which are responsible for most of the defects [35]. The same study also proposes to apply more detailed coverage analysis only on modules with higher density of faults, since some coverage analysis are more expensive than others.

2.8 Subsumption Relation of Coverage Criteria

Another method for comparing coverage criteria is the subsumption relation. The relationship is defined as follows: “Criterion A is said to subsume criterion B if, and only if, every test set that satisfies A also satisfies B” [36]. Coverage criteria subsume relations were discussed in [6,20,37]. Branch/decision Coverage subsumes (includes)
statement coverage, Path coverage subsumes branch coverage, and Data-flow coverage enhances path coverage. Figure 9 shows the subsumption hierarchy summarized as in [20] and [6].

![Figure 9. Structural coverage subsumption hierarchy.](image)

With this summarization laid out, the issues pertaining to the possible structural testing solutions (T1) become more complex due to complex nature of the system-under-test. We can infer that although coverage criteria have been used for decades, their descriptions are often vague and vary depending on the artifact they are applied to. To study in more depth this, it is often desirable to adopt a case study on which to use test coverage measurements (T3, T4).
Chapter 3

Case Study Background

Bombardier is a Canadian group with approximately 64,600 employees all over the world and a leading actor in manufacturing trains and aircrafts. The group is divided into three companies: Transportation, Aerospace and Capital. This thesis is carried out at Bombardier Transportation AB, located in Västerås, in the MLM division. The MLM division adapts the Train Control and Management System (TCMS) hardware and software developed at other division for each particular train project. MLM is also responsible for testing software developed by the MLM division, as well as the project implementation of the complete TCMS.

The experience of structural testing in MLM division was at its beginning stages when this thesis started. Further, we plan to contribute in developing good structural testing practice for PLC software development of FBD programs. One of the problems on fulfilling this goal is that any evaluation or analysis used during the process should cope with different requirements. Although the proposed study is not limited to a particular PLC software development platform, it is exemplified by a generic PLC control application compliant to the IEC 61131-3 standard in a real railway application. V300 ZEFIRO, is a multinational railway project posed to very high-speed trains for traffic in the north of Italy. This thesis is based on tools, methods, and artifacts developed during this project.

3.1 TCMS

We consider BOMBARDIER MITRAC Train Control and Management System (TCMS) as the target system for evaluating and implementing structural testing. TCMS is a high capacity, infrastructure backbone built on open standard IP-technology that allows easy integration of all control and communication requiring functions on-board the train. TCMS is a distributed system all over the train and it is involved in controlling almost all functionalities in the train such as collecting line voltage, controlling the train engines, opening and closing the train doors and upload of diagnostic data. TCMS consist of several intelligent units such as CCUs, running train control software, connected by different communication links, such as MVB- and IP networks. The TCMS platform is layered based on functional and performance requirements and each layer have different requirements on safety, performance and functionality. The Bombardier layer model for TCMS consists of Safety, Operations and Comfort. The safety layer contains all functionality that requires high levels of safety. Safety related units are shown yellow and the other units by gray in Figure 10.
3.1.1 CCU-O Software

Figure 10 shows the different functionality based intelligent units, which are CCU-O, CCU-D and CCU-S that corresponds to operation, diagnostic and safety accordingly. Our main interest in this thesis is to examine and study the testability and code coverage of the software running on CCU-O. The software running on CCU-O is in charge of the normal train control and functionality. The software in CCU-O is implemented by a method and a tool developed at Bombardier Transportation AB based on IEC 61131-3 standard. The reader can read more explanations in the next section mainly about the tools and methods used in the CCU-O software development.

3.2 TCMS Software Development Tools and Methods

The development tool used at Bombardier Transportation AB for TCMS is called 1131-EDIT (MWT) and it is based on the commercial product MULTIPROG from K&W Software. A screenshot from the MWT tool is shown in Figure 11. This tool is a development environment according to the IEC 61131-3 standard. 1131-EDIT is a software development environment that provides block programming which is the recommended way at Bombardier, although structured text and sequential function chart is accepted by the standard.

IEC 61131-3 defines a number of standard functions composing Function Block Diagrams (FBDs). Function examples are, AND, OR, ADD or CMP, and they can work on several data types, e.g. BOOL, INT, REAL and STRING. It is also possible to write code for CCU-O in the C language. These codes can then be integrated into the 61131 programs and accessed via special custom designed function blocks. The developer receives the design for a function, referred to as a component at Bombardier Transportation, and is the developer who prepares component design specification as part of the component implementation in the MWT tool from the requirements. The components are implemented by combination of standard function blocks or/and C code blocks.
The development tool MWT is providing diverse features and is easing all the required tasks for the actual implementation from the beginning to the end of component implementation and integration. MWT tool includes the component design specification, handling the component development, and FBD model to FBD program code transformation. The process of converting the FBD model into the FBD program code is called model to code transformation. This code must be complied for the target by another tool.

3.3 TCMS Testing Process

There are several levels of test of TCMS, which are carried out in several departments. The testing must comply with the EN50128 standard for development of a railway control systems. Since this thesis is carried out in the MLM department, we are only interested in testing done in this division and more specifically the focus is on TCMS component testing. MLM is responsible for testing software developed by the MLM department, as well as the project implementation of the complete TCMS system. Table 4, summarizes the test levels and the tests carried out at each level which are mostly requirement based.
### Table 4. Summary of the test levels defined at Bombardier MLM division.

<table>
<thead>
<tr>
<th>Test basis</th>
<th>Supporting documentation</th>
<th>Test level</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCMS Software Requirements</td>
<td>TCMS Software Architecture</td>
<td>TCMS release test</td>
</tr>
<tr>
<td>Specification</td>
<td>TCMS Software Design</td>
<td>TCMS overall test</td>
</tr>
<tr>
<td></td>
<td>TCMS Software Interface Specification</td>
<td>TCMS Sw/Hw integration test</td>
</tr>
<tr>
<td></td>
<td>TCMS component design specification</td>
<td>TCMS software integration test</td>
</tr>
<tr>
<td></td>
<td>HMI component design specification</td>
<td>HMI component test</td>
</tr>
</tbody>
</table>

#### 3.3.1 SoftTCMS

SoftTCMS is a software test environment, which is used by the testers/implementers/designers in the early stage during component testing. The SoftTCMS application is able to simulate input and output signals to and from the software component under test.

SoftTCMS also compiles the CCU C code, generated by MWT tool, for Windows OS software in order to be used on a regular desktop computer. Figure 12 shows how the graphical programming looks like in the SoftTCMS platform.

#### 3.3.2 TestCocoon

TestCocoon\(^1\) is an open source test coverage analysis tool integrated with SoftTCMS. TestCocoon includes code coverage measurements, by instrumenting the FBD implementation code. A FBD program code before instrumented by the tool by replacing all bitwise operators with Boolean operators. All the operators in the FBD

---

\(^1\) Version 1.6.14
program code are bitwise operators and TestCocoon is only working by Boolean operator. While running test cases, TestCocoon is measuring the coverage in the behind and provides report after test completion. The measurement results from TestCocoon are provided to a graphical report generator developed in Bombardier Transportation that shows the results applied to the model with colored signals. TestCocoon have the coverage criteria for statement, branch and condition coverage and it does not support MCDC.

3.3.3 Component Testing Method

Component test specification (test cases) is derived from component design specification, which consists of component description (Figure 14) and input/output/constant parameters (Figure 15).

Figure 14. Component design specification.

The tester has to read the component design specification, which comes with every component, and to write test cases according to it. Usually the initial test case is to set all inputs to False and then write test cases in the following format: “SET in_variable_1 to true then check that out_variable is False”. To illustrate the component testing process more thoroughly component “AUX_BCV_LdShdCt” is selected (Figure 13) as an example used in explaining the steps in the process.
Figure 15. Component design specification - parameters.

The test process starts by writing a component test specification from a component design specification. This document contains set of test cases written as steps and having two parts which are action and reaction. Figure 16 is an example component test specification for “AUX_BCV_LdShdCt” component. The following steps are used during this method:

1- The test steps in the component test specification are performed by the designer/implementer in SoftTCMS environment with/without TestCocoon integration, explained in the next section.

2- If the tests were not successful the bug is fixed and the component is tested again.

Figure 17 shows the component testing flow chart, including the usage of a traditional test coverage evaluation. Generally speaking, one of the time consuming activities of testing is running the test cases manually and it must be done accurately. The test cases might require to be run several times due to existence of bugs or completing the test suite to achieve better code coverage.
The experiences we gained during the TCMS case study are summarized in the next chapters (T1-T7). First we contemplate on the usefulness of structural testing on a complete PLC-based system. We envision further that the FBD software development process includes effective structural test coverage and can be enriched with a systematic way of dealing with coverage assessment before the actual test execution the target hardware. One of the problems on fulfilling this is that any evaluation or analysis for the TCMS case study should cope with circumstances under which coverage measurements on artifacts may be restricted because of changing requirements.

Figure 17. Traditional testing process flow chart.
Chapter 4

Structural Testing Based on the FBD Program Code

After studying structural coverage with its various criteria and getting to know the TCMS case study together with the development and testing process and tools used in practice, in this chapter we discuss the application challenges of this research knowledge on this industrial testing process specific to the FBD language. In this chapter, we will discuss how three test criteria evaluation can be used on TCMS case study (T1), what the results of the evaluations are (T2-T5), and how the architectural and structural-inherent problems for structural testing is considered in general at the lowest level of FBD software development for PLCs.

4.1 IEC 61131-3 Function Block Diagram Model to Code Transformation

In order to find the adequacy of code coverage tools performance over the FBD program code we need to understand the code and the model-code relationship. The train functionalities are modeled first and then transformed to C code and optimized during transformation, therefore it is important to analyze this transformation. The FBD program code includes many libraries, general parameter setting and functions that correspond to components in MWT, with the same component name. The code is generated using a simple subset of C language and only includes limited keywords. Keywords such as “for”, “while” and “switch” does not exist in the FBD program code.

The relation between function blocks that build the components and the code is identifiable, as depicted in Figure 18. Each function block corresponds to a code segment in the FBD program code. A One-to-One relation is identified in most cases. Function blocks such as “SEL” generates more than one code segment.

![Figure 18. Function block to code mapping.](image)

Some function blocks in the FBD model such as “SR” state and “RS” state, “TON” timer, “TOF” timer and “TP” timer, “F TRIG” edge detection and “R TRIG” edge detection, “LIMIT” selection, “MAX” selection and “MIN” selection are
transformed into functions and function calls with the provided parameters. Figure 18 shows the mapping between some of the function blocks and the code lines in a FBD program code.

4.2 Structural Testing Behavior on the FBD Program Code

The railway standard (EN50128) is addressing number of structural coverage criteria. In this section we investigate using these criteria on the FBD program code. In the following sections the scope is component level testing that corresponds to a function in the actual code. The FBD components are translated to functions in the FBD program code. The functions vary in their internal structure, some are free of control statement and others contain many control statement such as “If” statements. The control statements are causing segmentation in the code. We define the segment as the branch that includes statements. To reach the statements, the branch requires to be evaluated to true and false. There is another type of branch that does not include statements inside the branch, which is explained in Section 4.6. This type of branches is not considered as segments.

Figure 19. Graphical representation (FBD model) of “TC_ACT_TnlMd” component.

The studied code showed that only control statements are causing code segmentation in a function. The control statements are generated from function blocks such as “SEL”, “JUMP” output, “RETURN” and hand written C code functions with “if/else” control statements that transformed to “if/goto” by MWT when the code is generated. If we consider the component “TC_ACT_TnlMd” (Figure 19) as an example, we see the function code and then we generate the control flow graph. The FBD program code of the component “TC_ACT_TnlMd” and its code segmentation is depicted in Figure 20. The CFG representation of the code example in Figure 20 is shown in Figure 21. This code and CFG example is to be used through the rest of this experiment.
4.3 Branching Points in the FBD Program Code

By inspecting the FBD program code one could find two types of branching points. The first type is the ordinary control statement that is causing branches, which we refer to them by explicit control statement. The second type is the assignments with Boolean expressions. These expressions similarly to the code with control statements are causing branches in the code execution with the difference in the representation format. Figure 22, explains how an assignment like “a = (b && d),” expanded to code with explicit control statements. The expanded assignment shows that every assignment with Boolean expression is making branches and it must be treated as decision points. There are a few papers [20,38,39] that are mentioning about these types of assignments and consider them as branching points and indeed these are affecting the code logic and flow in combination with the explicit control statements in the code.
4.4 Explicit Control Statement VS. Boolean Expression Assignment

In the previous sections the code branches and conditions were split into two categories. The first category includes the branches and conditions due to explicit control statement. While the second category includes the branches caused by Boolean expression assignments, which is relatively more difficult to achieve code coverage.

We randomly select 10 components and calculate the ratio of the conditions in the explicit control statement over the total number of operand. “data->ACT” in the “E” segment of Figure 20 shows a condition in an “if” statement. We call the Boolean variables in the compound Boolean expressions as operands. For example “data->TC_ACT_C_TnlMdSelDm1 & data->TC_ACT_S_CabActDm1” in the “D” segment of Figure 20 shows two operands in Boolean expression assignment. The conditions are considered operands also. The ratio of total number of conditions over the total number of operand is interesting since it can tell if all the signals in the components are tested adequately only applying code coverage on explicit control statements. The signals are represented by operands and conditions in the FBD program code.

Table 5. Ratio of conditions in the control statements to the total number of conditions

<table>
<thead>
<tr>
<th>Component name</th>
<th>Condition/Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>0/4</td>
</tr>
<tr>
<td>FAULTDLV</td>
<td>7/9</td>
</tr>
<tr>
<td>TC_ACV_TwMd</td>
<td>1/21</td>
</tr>
<tr>
<td>TC_ACV_CplMd</td>
<td>1/29</td>
</tr>
<tr>
<td>TC_ACV_PrkMd</td>
<td>25/74</td>
</tr>
<tr>
<td>LATCH1_BO</td>
<td>3/5</td>
</tr>
<tr>
<td>TC_ACV_TnlMd</td>
<td>2/8</td>
</tr>
<tr>
<td>TC_ACV_WshMd</td>
<td>2/27</td>
</tr>
<tr>
<td>TC_ACT_EnOprMd</td>
<td>2/19</td>
</tr>
<tr>
<td>TC_A_T2</td>
<td>4/4</td>
</tr>
</tbody>
</table>

Table 5 shows the low effectiveness indicator for the test cases that can achieve code coverage by only considering conditions in the control statements. The FBD program code is written in a way that uses the least number of conditional statements. Therefore, using above mentioned coverage criteria on only conditions in the control statements does not force effective test cases. Because the ratio of
conditions in the control statements to conditions in the Boolean expressions is low that means we are only testing small portion of the conditions.

4.5 Statement Coverage Behavior on the FBD Program Code

According to the statement coverage definition, all nodes in the control flow graph must be examined to achieve the full coverage.

**Statement Coverage for Explicit Control Statement.** Functions with control statements are causing code segmentation. There is no rule to calculate the statement coverage percentage in advance. Since the code does not implement nested control statements, the maximum number of required test case to achieve 100% statement coverage is equal to the number of control statements multiplied by two. It is sufficient to count the occurrence of them in the component and multiply them by two to find the required number of test cases for achieving statement coverage.

**Statement Coverage for Boolean Expression Assignments.** Functions without control statements are defined with no code segmentations from the statement coverage perspective. Since there are no segmentations all nodes are traversed once the function is executed. Consequently 100% statement coverage is achieved.

4.6 Branch Coverage Behavior on the FBD Program Code

The definition for branch coverage says that every branching or decision point in the code must be evaluated to true and false at least once. The branching points that mentioned in 4.3 do not affect statement coverage but it does affect the branch coverage.

**Branch Coverage for Explicit Control Statement.** From section 4.5, we saw that for the two function categories in order to obtain statement coverage all control statements must be evaluated to true and false. Similar to statement coverage, all edges of the control flow graph must be traversed in order to achieve full branch coverage. We can conclude that applying statement coverage or branch (decision) coverage requires equal effort and satisfying 100% statement coverage implies 100% branch coverage for explicit control statements. This is forced by the code implementation method and structure.

**Branch Coverage for Boolean Expression Assignments.** Each component code (function) is implemented by number of these assignments. The assignments are branches that could be either true or false. Therefore a function call with random input can achieve exactly 50% branch coverage, since all assignments will be assigned by either true or false.

4.7 MCDC Behavior on the FBD Program Code

MCDC is a type of coverage that is in relation with conditions. According to the following definitions from aviation standard [40] and railway standard [31] conditions are decision or part of a decision if the decision is compound Boolean expression. We have identified two type of branching (decision) point in section 4.6. Therefore it is mandatory that conditions in the two branch category be identified and exercised by MCDC for this type of code coverage.
**MCDC for Explicit Control Statement.** There are only two types of control statements in the code which are “if/goto” and “condition ? (MWT_BOOL) 1 : (MWT_BOOL) 0;”, some are with single condition and some others with compound predicate. The ones with single condition the MCDC is achieved when the branch condition is achieved with the same number of test cases. This means a single test case can achieve maximum 50% MCDC coverage for predicates with one condition but the percentage decreases for predicates with more than one condition. Number of test cases for other predicates with more than one conditions in control statement depends on the number of conditions in the expression. For example for a compound predicate with 4 conditions we require 4+1 test cases according to MCDC criteria to achieve 100% MCDC.

**MCDC for Boolean Expression Assignments.** Considerable percentage of our code is built from Boolean expression assignments; therefore achieving MCDC requires huge effort. Furthermore, the existence of local variables complicates the application of MCDC since we have to trace local variable to its definition in order to find the missing test cases. For example Figure 23 shows the use of local variables Boolean expression assignment. The two first Boolean expressions are assigned to local variables and then they are used in another Boolean expression assignment. If code coverage analysis tool shows that the third line is missing test cases with respect to MCDC criteria then we need to trace back the local variables to find the appropriate input data that can satisfy MCDC.

Number of required test cases for Boolean expression to be satisfied by MCDC depends on the number of conditions in the expression. The minimum number of conditions in the Boolean expression is two. Therefore minimum 2+1 test cases are required for every assignment. As a result by one test case that achieves statement coverage, we could achieve less than 33% of MCDC for the Boolean expression assignments. Lower coverage percentage increase per test case indicates higher thoroughness for the coverage criteria.

### 4.8 Coverage Criteria Comparison Based on the FBD Program Code Structure

We have analyzed three coverage criteria on the FBD program code in the previous sections. The coverage percentage by one test case of this analysis for a function without control statement corresponds to a component in MWT is summarized in Figure 24.

<table>
<thead>
<tr>
<th></th>
<th>Statement Coverage</th>
<th>Branch Coverage</th>
<th>MCDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum coverage by one test case</td>
<td>100%</td>
<td>50%</td>
<td>&lt;33%</td>
</tr>
<tr>
<td>Maximum coverage by one test case</td>
<td>100%</td>
<td>50%</td>
<td>33%</td>
</tr>
</tbody>
</table>

Figure 24. Code coverage achievement percentage by single test case.
This comparison shows that statement coverage is not a good indicator for test thoroughness because any random test input can achieve 100% statement coverage. On the other hand, branch coverage achieves 50% by any random test data but increasing the coverage percentage requires more thorough test cases. Therefore 50% must be considered as an initial point in branch coverage. The percentage for MCDC shows that this criterion is more thorough and suite better the testing process of the FBD program code. Achieving 33% by one test case using MCDC is very optimistic, since this is valid only if all Boolean expression assignments have two conditions. For example an expression with four conditions will only reach 20% MCDC, \( \frac{1}{(4+1)} \times 100 = 20\% \), by one random test case.

### 4.9 Estimation of Achievable Coverage on the FBD Program Code

In the structural coverage measurement a target percentage number must set and to aim for. The percentage number must be reasonable and achievable. Therefore the test target must be analyzed structural coverage achievability.

The structure of the FBD program code at the function\-component level is simple and there is no infeasible path from structural point of view, such as exception handling or default cases that will never execute. The function code might include constant parameters, which might lead to reduce the target coverage percentage. In cases these constant parameters are acting as conditions in the Boolean expression assignments; therefore reduction of the required test cases must be considered. That means in an expression with three conditions which one of them is constant parameter, the total number of the required test cases to achieve MCDC will be three instead of four. Consequently the target percentage is 75% instead of 100%.

### 4.10 Evaluation of a Commercial Code Coverage Tool (TestCocoon) and Requirements

Structural testing environment at Bombardier Transportation AB is provided by integration of TestCocoon and SoftTCMS. The SoftTCMS environment is facilitated in a way that enables benefiting from TestCocoon in the ordinary SoftTCMS environment without introducing new complexity. By evaluating TestCocoon we found that it is a suitable tool since it can detect all type of conditions in the code, explained in Section 4.6. The drawback of using this tool is that it cannot distinguish between the conditions that must be checked and the conditions that are alias (local variables) for a compound Boolean expression. The code uses some local variables that hold the value of compound Boolean expressions. As a result we will have two copies of the same expression. Therefore it can’t provide accurate indicator number for the testing progress. The other drawback noticed for this tool, is the short circuit operation (refer to 0 for short circuit definition). The source code is implemented by bitwise operators and it forces the full evaluation of the code while replacing the bitwise operators by logical operators introduces short circuit. This could lead to the creation of extra test cases. The graphical report makes this issue obvious. Also this tool is insensitive to comparisons and does not evaluate them. While the comparisons logic (LG, GT, EQ) have a big portion of our code.

The last weakness that we found in the in TestCocoon than it does not support MCDC natively. Nevertheless, it supports branch and condition coverage, which means less test cases and less effectiveness compared to MCDC. Table 6 summarizes the advantage and the disadvantage of using TestCocoon on the FBD program code.
4.11 Effectiveness Analysis of Code Coverage Criteria on the FBD Program Code

The introduced activities to the testing lifecycle must be effective. Therefore we have studied the behavior of several code coverage criteria on the FBD program code but the question that we intend to answer here is: How effective are the code coverage criteria on our code?

The code is generated automatically from a FBD model developed by MWT tool. Testing MWT tool and FBD model to FBD program code transformation is not in the scope of this thesis. Therefore we assume that the tool is tested well, the code conforms to the model, and the FBD program code is safe. Thus we only focus on testing the functional specification and we are not intending to test the code for bugs that leads to errors such as memory errors, type errors, exceptions or any type of error that is not logical error.

In the FBD model to FBD program code transformation the code is optimized and some function blocks are transformed to functions and function calls. As depicted in Figure 25 that is the FBD program code of “AUX_BCV_LdShdCt” component, which was shown in Figure 13, the highlighted lines show the equivalent function calls for function blocks such as “RS” and “FAULTDLY”. In this example we have two function blocks that are transformed to one function code, called with different parameters. In cases that a function is called from several places in the code it is infeasible to distinguish the coverage percentage of the function code by each function call. As a result we will have high code coverage without adequate testing of the component functional specification. We show in Figure 26 the function “FAULTDLY” is called with different parameters in the code. The first function call is marked by blue and the second is marked by yellow. The code in the right side shows the function code of the called function from the left side code.

<table>
<thead>
<tr>
<th>All condition types detected</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid short circuit</td>
<td>NO</td>
</tr>
<tr>
<td>Avoid redundant conditions</td>
<td>NO</td>
</tr>
<tr>
<td>Evaluating comparisons</td>
<td>NO</td>
</tr>
<tr>
<td>MCDC Supported</td>
<td>NO</td>
</tr>
</tbody>
</table>

Table 6. Evaluation of TestCocoon.
When a function with several branching points is called, there might only some branches of the function that are being evaluated. The right side code in the running example shows a possible evaluation of branches by different colors corresponds to the function calls at the left side in one execution. The blue parts correspond to the parts of the code that evaluated by the first function call. The yellow parts are the parts that evaluated by the second function call and the gray color means the code is
evaluated by both function calls. There are also some parts which are not marked that means those parts are not evaluated in the first code execution. Therefore, the coverage percentage is calculated for the function from both function calls. This cannot provide a good indicator whether the function is tested adequately according to functional specification. An individual coverage measurement for every function call can provide a better possibility for testing system behavior and functional specification.

4.12 Chapter Summary

In this section we presented our analysis results of applying structural testing on the FBD program code. We found that MCDC is the most relevant criteria to be used with the FBD program code because of its strict relation with the defined conditions. Also, statement coverage is not helpful because any test suite would easily exercise every statement in the FBD program code. Both type of conditions included in the control statement and the Boolean expression assignments must be considered with the applied code coverage.

The analysis of implementing structural coverage on the FBD program code tells us that it cannot help with testing the functional specification adequately. This is due to code optimization and transformation of function blocks to functions and function calls. While the definition tells us, “Structural testing is still testing product functionality against its specification. Only the measure of thoroughness has changed.” [4] This is exactly our interest and we are not concerned in testing the code to find errors in the code itself, since the FBD model to FBD program code transformation is assumed to be already tested. We have studied the application of code coverage criteria on the FBD program code and we have studied its weakness. The functional specification is modeled in MWT, therefore it could be a better option for implementing structural coverage on the component model rather than on functions in the FBD program code. Furthermore, the test evaluation is telling us that it cannot adequately help testing the functional specification of the FBD programs. This is due to FBD program code optimization of tracking the function calls. Obviously, we investigate further the application of structural testing directly on the FBD model.
Chapter 5

Structural Testing Based on the FBD Model

We have seen in the previous chapter that applying structural testing on the FBD program code is not effective. Therefore in this chapter we study applying structural testing on the FBD model. Since the FBD model is designed from the component design specification and is used to generate the FBD program code automatically, it could be an interesting artifact for structural testing of functional specification. We propose two methods that can identify the function blocks that are not tested adequately with respect to a coverage criterion. The FBD model itself is not an executable artifact, therefore the execution result must be mapped to the model or using other methods to apply structural coverage on the model such as static analysis approach. In the two methods that we propose, using a FBD model as a medium in structural testing is different from the traditional methods since there is no requirement (R1) for instrumentation of the code or the model to measure the structural coverage.

We can further evaluate the adequacy of a test suite based on a coverage criterion before going to the test execution on the actual target system. The main result of this section is the provided discussions and the proposed method used to find the inadequately tested parts and generating the coverage percentage in advance (T6-T7).

5.1 Manual Method for Applying Structural Testing on a FBD Model

We aim from manual structural coverage based on FBD model to implement structural testing for software components or units that cannot be instrumented such as encrypted code or software running on a target without an extra tool. Also this method could be used for structural testing in general. In this method, first we test the component without structural coverage concern. After finishing testing we make sure manually that the tests were adequate according to a coverage criterion. Similar method is proposed in [20,38,39]. The method consists of two parts with seven steps in total. The first part is the ordinary testing that was discussed in Section 3.3.3, which are steps 1 to 3. The second part from step 4 to 7 is more in relation with structural coverage. The proposed method is as following:

1- Write test specification Figure 27.
2- Run the test specification in the simulated environment (SoftTCMS) and observe the reaction.
3- If the test was satisfactory go to the next steps.
4- Write the “Actions” (input data) and the “Reactions” observed output on the signals in the schematics Figure 28.
5- Calculate the mid signals combination from the test input Figure 28.
6- Compare the input and the output combination of each function block with the coverage criteria.
a. Branch: Every signal must be evaluated to true and false

b. MCDC: Every logical function block must satisfy MCDC criteria and other non-logical blocks must be tested by Branch criteria.

7- Test cases based on the functionality requirement for the component must be added to achieve coverage for the function blocks that missing test case according to the coverage criterion.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Set input signal Battery Bus Voltage to 9V and also the signal Battery Bus Voltage L.</td>
<td>Check that the output signal Battery Bus Voltage is 9V and that the signal Battery Bus Voltage L is NOT set (FAUL)</td>
</tr>
<tr>
<td>2</td>
<td>Set input signal Battery Bus Voltage to 9V and also the signal Battery Bus Voltage H.</td>
<td>Check that the output signal Battery Bus Voltage is 9V and that the signal Battery Bus Voltage H is NOT set (FAUL)</td>
</tr>
<tr>
<td>3</td>
<td>Set input signal Battery Bus Voltage to 9V and also the signal Battery Bus Voltage L.</td>
<td>Check that the output signal Battery Bus Voltage is 9V and that the signal Battery Bus Voltage L is set (TRUE)</td>
</tr>
<tr>
<td>4</td>
<td>Set input signal Battery Bus Voltage to 9V and also the signal Battery Bus Voltage H.</td>
<td>Check that the output signal Battery Bus Voltage is 9V and that the signal Battery Bus Voltage H is set (TRUE)</td>
</tr>
<tr>
<td>5</td>
<td>Set input signal Battery Bus Voltage to 9V and also the signal Battery Bus Voltage L.</td>
<td>Check that the output signal Battery Bus Voltage is 9V and that the signal Battery Bus Voltage L is set (TRUE)</td>
</tr>
<tr>
<td>6</td>
<td>Set input signal Battery Bus Voltage to 9V and also the signal Battery Bus Voltage H.</td>
<td>Check that the output signal Battery Bus Voltage is 9V and that the signal Battery Bus Voltage H is set (TRUE)</td>
</tr>
<tr>
<td>7</td>
<td>Set input signal Battery Bus Voltage to 0V, set input signal Battery Bus Voltage L and also the signal Battery Bus Voltage H.</td>
<td>Check that the output signal Battery Bus Voltage is 0V and that the signal Battery Bus Voltage L is set (TRUE) and also the signal Battery Bus Voltage H is set (TRUE)</td>
</tr>
</tbody>
</table>
For example in Figure 28 the “OR” block is not tested adequately according to MCDC. The block has been evaluated by inputs \{F,F\} and \{F,T\} but it was not evaluated by \{T,F\} combination to satisfy 100% MCDC. Therefore the component must be tested by an input data that generates this combination for this block or it must be reasoned that this combination is not possible. In this example by testing the component with input data less than 0, such as -1, we can generate the \{T,F\} combination for the block and achieve 100% MCDC coverage.

5.2 Evaluation of Applying Manual Structural Coverage on a FBD Model

The advantage of this method is that it does not require additional tool for implementing structural testing. Also it could be used in cases that we are not able to instrument the code. The main disadvantage of this method is the manual operations that introduce lots of effort and it is time consuming. Furthermore the manual methods are usually error prone. This method also has another disadvantage. In case of finding missing test cases, retesting the component is required. Retesting the component might be required several times, which are time consuming and depending on the coverage percentage that we aim for and missing number of the test cases. This method has the flow chart showed in Figure 17. The other suggestion in this thesis is automated and overcomes the need of retesting.

5.3 New Automatable Test Suite Evaluation Based on a Coverage Criterion Method

Similar to the manual method for structural testing, the method is not based in code or model instrumentation. The method is based on observation and evaluation. The idea is to evaluate the adequacy of the test suite based on a coverage criterion for a FBD model before going for test on the target. That means only good enough test suites must go for test, since testing is time consuming.

5.3.1 A Methodology for Test Suite Evaluation Based on a Coverage Criterion

This method develops more the manual method presented in the earlier chapter and makes it automatable. The same concept of dealing with input and output combination for every function block is used. All function blocks in a component could have Boolean or numeric or both values as inputs and as a result could have
Boolean or numeric values on output. Therefore we could make a value table for every block by knowing the provided inputs from the test suite and the generate outputs for every block from the block functional property.

On the other hand, every function block requires number of tests to be adequately tested according to a certain structural coverage criteria. These required tests for achieving coverage could be represented as a value table for every type of function block. For example if we have two “OR” and “AND” function blocks we require two value tables showing the minimum test combinations needed to have an adequate test suite based on a criterion. The method of test suite evaluation based on a coverage criterion is explained in the following steps:

1. Preparing reference value table for every function block type according to a structural coverage criterion, explained in page 43.
2. Writing test specification from component design specification.
3. Transform the D-U relations in the FBD model to expressions, explained in page 44.
4. Supply the test input to the D-U expressions of the FBD model to generate the value tables for every function block in the model, explained in page 44.
5. Comparing and finding the missing input/output combinations in the generated value table compared to the reference value table. Explained in page 45.
6. Calculating coverage percentage
7. Taking appropriate action such as adding test cases and reevaluating the test suite, going to step 2.
8. Running the adequate test suite on the target. This step is only taken again if there is a mistake in the model design.

**Reference Value Table for Every Function Block.** For each function block in a component a value table that corresponds to the required input and output combinations satisfying a structural coverage criterion is generate. The value table is used as a reference value table for all function blocks with the same type of blocks that the value table is generated from. The value tables for the function blocks are in form of inputs and outputs arrays. For example a “RS” function block in Figure 29, which is a latch, needs five tests with combinations showed in Figure 30 to achieve full MCDC [39]. In states like “RS” the test execution sequence can make difference in the output and by testing with the wrong sequence we might not achieve the full coverage although if we have all the required input combinations, the five test combinations in the example Figure 30. Therefore the comparison between the test suite value tables and the criterion value tables must be on combination of both inputs and outputs.

![Figure 29. “RS” function block.](image-url)
D-U Relations to Expressions Transformation. Every FBD model (or component model as it referred to in Bombardier Transportation AB) consists of one or several inputs and outputs. The outputs are depending on the FBD model inputs and the FBD model internal logics. Therefore, every FBD model output could be expressed in a defined expression format. The dependency between the FBD model input and output called Definition-Use (D-U) data flow. Then in the other words the Definition-Use (D-U) data flow for every output parameter is converted to a defined expression format, D-U data flow explained in section 2.4.5.

For better explanation we select the “AUX_BCV_BatCt” component as an example showed in Figure 31. The component output is marked by green circle. The output is depending on the internal logic in addition to the input which is marked by the red circle. Simple equivalent expression to this dependency could be an expression like:

```
orb(2, AUX_BCV_C_CiCh, 10, tonb(1, AUX_BCV_S_DcSdWnr, 60, 30), 1)
```

The red highlights correspond to function blocks, the expression starts with the most outer block. The blue highlights are the input parameters and the rest unmarked items in the expression are IDs used for component items tracing back.

Generation of Test Suite Value Table Based on a FBD Model. Since we have the FBD model transformed to a symbolic expression, we can generate the value tables for each function block from the expressions fed by the test suite. For a better explanation on the generation of the value tables, we use the FBD model in the following example instead of the equivalent expression representation. Sample test cases are shown in Figure 32 and they are written by green color on the model in Figure 33. These values generate value tables for each of the function blocks in the FBD model. The value table shown in Figure 33 belongs to “OR” block. The generated value tables are to be compared with the reference value tables, which are the coverage criterion value tables for each function block type.
Comparing and Finding the Missing Input / Output Combinations. The remaining combination of subtracting the criterion value tables from the test suite value tables is the missing test cases for the function blocks. We try to explain this by using an example. The value table for “OR” function block in Figure 33 has three identical combinations. Which are {F,F} -> F, {T,F} -> T and {T,T} -> T. The values in the “{}” are the inputs and the value on the right after “->” is the output. If we consider a reference table with MCDC for an “OR” function block we would have {F,F} -> F, {T,F} -> T and {F,T} -> T combinations. Comparing these combinations with the combinations generated from the test suite in Figure 33 we find that the combination {F,T} -> T is missing in the test suite. Therefore the “OR” block requires one more test input combination to be adequately tested according to MCDC. Figure 34 shows the missing test combinations for the function blocks written next to each block according to the provided test suite in Figure 32 and MCDC.
5.3.2 Evaluation of Using Test Suite Evaluation Based on a Coverage Criterion

There are several advantages for using test suite evaluation compared to the traditional methods and commercial structural coverage tools, pointed as following:

1. The testing lifecycle will be faster than the traditional lifecycle described in Section 3.3.3 and Figure 17. The new lifecycle with the test suite evaluation is showed in Figure 35. By using this method, the component testing process can be optimized. The process can change in a way that, one of the time consuming activities which is “Run test specification” is taken out from the nested loop. This advantage becomes much more visible when there is a bottleneck for accessing specific testing resource. Such as hardware environment used for testing.

2. Identifying the not or not well tested parts of the FBD model or FBD program code by only doing test suite evaluation.

3. Knowing the coverage percentage in advance, before running the test suite on the target.

4. This method is easing to have test suite progress indicator, always at any time without running the test suite.

5. This method is a good solution for special language standards such as MultiProg IEC 611313, where code coverage tools are a few or does not exist.

6. The obtained results from a commercial structural coverage tool require more interpretation compared to this method. While this method tells what is expected to have fuller coverage while the existing tools tell that the code portion requires more tests.
There are two disadvantages or limitations with this method that could be addressed. The method is only applicable for the FBD model driven software and the method is not implementable for other types of software development. Second, the method shows the places that are lacking tests with also telling the missing tests for every function block but not for test cases for the component.

5.3.3 Application Example: Evaluation of Test Suite Based on a Coverage Criterion

We aim to provide means for using the previously mentioned method by developing a proof-of-concept tool on test coverage measurements. Based on our evaluation the tool should take the FBD model, in specific format, and the test suite as inputs and then outputs the missing test inputs for every function block according to a structural coverage criterion. The tool is developed as proof of concept for this method. After providing the FBD model and the test suite to the tool, it can evaluate the test suite based on MCDC. Figure 36, shows the actual evaluation of a test suite for the “AUX_BCV_BatCt” component. The top part of the GUI shows the expression representation of the FBD model. This is an important step for making the model interpretable for the tool. There are two grids in the lower part of the GUI. The toper grid shows the test suite for testing the component and the lower grid shows the missing test cases obtained by the tool. The prototype tool is implemented using C# programming language for the interface. Also, PostgreSQL, which is an open source Database Management System (DBMS), is used for implementing the actual method. Also the FBD model, the test suite and the reference value tables are stored in the database. The test suite and the value tables generated from the test suite are traceable on the model. Therefore the evaluation result could be visualized on the model easily.
Figure 36. Implemented tool for evaluation of test suite based on a coverage criterion method.
Chapter 6

Conclusion

Building safety-critical systems out of PLCs does not ensure that the resulted system is well tested. Testing the functional behavior of FBD-based PLC programs is fundamental during the development process of industrial safety-critical systems. For such systems to be used in practice, compliance to standard quality requirements (R1) including component testing and coverage assessment is needed. In particular, the experience of using structural testing is relatively new in the safety-critical software industry and therefore it lacks applicability on PLC software expressed using FBDs (R2-R3). We claim that in this context, strengthening this experience and providing means of measuring the test effort of using structural testing in an industrial setting is essential.

In the conducted literature study we have studied the traditional structural coverage criteria. We have studied several research papers and industrial reports, which are describing the experimentation of structural testing. Based on a real industrial case study we have performed a study on the effects of structural testing on the software development lifecycle, software development method, and tools used during the testing process. A commercial code coverage tool called “TestCocoon” was briefly evaluated. We also show that the tool is missing important features and some of the required specifications that could not be found in any commercial tool since it is not focused and implemented specifically for our specific industrial requirements.

FBD software development guides us to narrow the structural testing domain only for measuring structural coverage by functionality-based testing, since the FBD program code generated from the FBD model is supposed to be safe and bug-free. Bombardier Transportation AB follows the FBD software development, which gives us the opportunity of study the application of structural testing on both the FBD model and the FBD program code generated from the model. We have studied the FBD model to FBD program code transformation in order to identify the relations between the model and the code as well as gaining deeper knowledge for analyzing the FBD program code. We have analyzed the effectiveness of structural testing on the FBD program code based on three coverage criteria. In general, applying structural testing on the FBD program code generated from an FBD model is not effective since the code structure is optimized during the model to code transformation. The optimization that raises issues in structural testing is mainly the transformation of some function blocks to functions and function calls. Also, the FBD program code is composed of almost exclusively assignments with Boolean expressions instead of code segments with explicit control statements. These assignments are not detected as branches or conditions in FBD code coverage analysis. Therefore, measuring structural coverage by functionality based testing is infeasible. We also showed that it could be the case of achieving good results in measuring the structural coverage based on the FBD model by using functionality-based testing. With respect to the requirements and limitations identified during the study, one manual and one automated methods are proposed. The manual method is
applied after testing completion in order to measure the thoroughness of the test suite. Since it is manual, it introduces extra effort and it is error-prone. The automated method is applied before starting the execution of the test suite. It evaluates the test suite based on a coverage criterion on the FBD model.
Chapter 7

Future Work

There exist several structural coverage criteria that are not studied in this thesis. Some of these criteria might suit an FBD program code or in general software implementations with specific code structure. Also, there might be the possibility of defining a genuine or an adapted coverage criterion for these types of system-under-test.

Developing further the proposed method and the implemented tool is possible. The tool is developed as a proof of concept and it requires more effort to be used for a more general purpose.

Furthermore there is an opportunity for research in generating test cases for the parts which are identified as lacking testing by our proposed method. The test case identification must be based on functional specification and not only the model structure.

The other extension to this work could be to study the application of structural testing for other levels of PLC development.
References


and analysis. 2002.


43. ZIBRAN, M. F. Evaluating Test Quality. The University of Saskatchewan Department of Computer Science. 2012.