AN EVALUATION OF MODEL-BASED TESTING FOR AN INDUSTRIAL TRAIN CONTROL SOFTWARE

Sidorela Suli
ssi17005@student.mdh.se

Examiner: Daniel Sundmark
Mälardalen University, Västerås, Sweden

Supervisor: Eduard Paul Enoiu
Mälardalen University, Västerås, Sweden

Supervisor: Andrea Polini
University of Camerino, Camerino(MC), Italy

Company supervisor: Ola Sellin,
Bombardier Transportation Sweden AB, Östra Ringvägen 2, Västerås, Sweden

June 4, 2018
Abstract

Currently, the increasing complexity of software and the short release cycles are becoming a challenge for testing software in an efficient and effective way. Traditionally, creating tests is done manually by engineers, which are then automatically or manually executed on the actual software. Manually creating test cases is a time-consuming effort. For the last couple of decades, researchers have proposed ways to improve this process by automating parts of the testing steps. One of these approaches that have gained a lot of attention is called Model-Based Testing (MBT). MBT has been suggested as a way of automatically creating tests at a lower cost. Nonetheless, it is not very well studied how MBT is actually applied in industrial contexts and how these tests compare to manually written ones. This is particularly true for industrial control software such as the one found in the train domain, where strict requirements on testing are in place.

In this thesis, we investigate the literature and review the related work on case studies on the MBT use in industry and its evaluation. We perform a case study to evaluate MBT on a train control management system provided by Bombardier Transportation. We use Conformiq Creator MBT Tool to create models for functional requirements of a master controller function and generate test cases. We provide the result of the modeling approach as well as a comparison between automatic test cases created by Conformiq Creator and manual test cases written by industrial engineers at Bombardier Transportation using the following metrics: test coverage and time spent on testing. The results of this comparison suggest that test coverage of MBT is higher and test cases are more detailed than manual testing. Our results are not conclusive in regard to the cost of using MBT, mainly because this depends on different testing scenarios and how testing is performed. We show that MBT is a suitable approach for modeling the functional requirements of a realistic industrial control software function.

In this thesis work, we focus on system-level testing. As future work, applying MBT on lower levels of testing can be a promising way forward for evaluation. In addition, the transformation of these test cases into executable test scripts and the possible problems needs to be investigated further.
# Table of Contents

1 Introduction 7
   1.1 Motivation .............................................. 8
   1.2 Problem Formulation ..................................... 9
   1.3 Objectives ............................................... 9
   1.4 Research Methodology ................................... 10
   1.5 Structure ................................................. 11
   1.6 Contributions ........................................... 11

2 Background 12
   2.1 Definitions ........................................... 12
   2.2 Software Testing ......................................... 13
      2.2.1 Test Process ........................................ 13
      2.2.2 Testing Techniques ................................ 14
      2.2.3 Test Quality Measurements ......................... 14
   2.3 Model-Based Testing .................................... 15
      2.3.1 MBT Fundamentals Tasks ............................ 15
      2.3.2 MBT Taxonomy ...................................... 17
      2.3.3 Perceived Benefits of MBT ......................... 18
      2.3.4 MBT Deployment .................................... 18

3 Case Study Design 19
   3.1 Train Control Management System (TCMS) ............... 19
      3.1.1 Testing of TCMS .................................... 20
      3.1.2 Master Controller - Drive and Brake Control ....... 21
   3.2 Conformiq Creator ...................................... 22
      3.2.1 Modeling Language .................................. 24
   3.3 Research Methodology ................................... 27

4 Case Study Results 30
   4.1 General Description of the Model of SUT ............... 30
   4.2 Automatic Test Case Generation .......................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 Manual Test Cases</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Comparison Between MBT Test Cases and Manual Test Cases</td>
<td>44</td>
</tr>
<tr>
<td>4.5 Research Questions and Answers</td>
<td>49</td>
</tr>
<tr>
<td>5 Threads to Validity</td>
<td>50</td>
</tr>
<tr>
<td>6 Related Work</td>
<td>51</td>
</tr>
<tr>
<td>7 Discussion</td>
<td>52</td>
</tr>
<tr>
<td>8 Conclusions and Future Work</td>
<td>53</td>
</tr>
<tr>
<td>8.1 Future Work</td>
<td>53</td>
</tr>
<tr>
<td>References</td>
<td>55</td>
</tr>
<tr>
<td>Appendix A Functional SUT Requirements</td>
<td>56</td>
</tr>
<tr>
<td>Appendix B Test Cases Generated from Conformiq</td>
<td>56</td>
</tr>
<tr>
<td>Appendix C Manual Test Cases of Master Controller</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Figures

1. Thesis Outline ........................................ 12
2. V Model Test Types ................................... 13
3. Model-Based Testing Process .......................... 16
4. Overview of the taxonomy [1] ......................... 17
5. TCMS units connected to network buses and intelligent units. ... 20
6. The split between development and test of TCMS components testing. ... 21
7. Master Controller ...................................... 22
8. Conformiq Creator Modelling Language. ............... 24
9. Conformiq Creator Test Design Perspective .......... 24
10. Conformiq Creator Modelling Perspective .......... 25
11. Conformiq Creator Activity Diagram Example. ...... 25
12. Conformiq Creator Structure Diagram Example. ... 26
13. Conformiq Test Design Views. ....................... 26
14. Test Targets ........................................... 27
15. Case Study Design ..................................... 28
16. Structure Diagram ..................................... 30
17. Main Activity Diagram ................................. 32
18. Read Master Controller ................................ 33
19. Read Signalling System ................................ 35
20. Check if analog values from two input channels differ - Activity Diagram ... 36
21. Checks which input channel is cutted .................. 37
22. Generate test cases from the model .................. 37
23. Generated Test Cases. ................................ 38
24. Test Target View: Input and Output Values for a Test Case. .......... 38
25. A selected test case in the models .................... 39
26. Test Coverage ......................................... 40
27. Test sub-process Quality Gates. ...................... 41
28. The V Model, illustrating test levels and test types .... 42
29. List of test metrics per test level. ................... 43
| 30 | Test Coverage .......................................................... | 45 |
| 31 | Manual Test Cases of Master Controller .......................... | 57 |
| 32 | Manual Test Cases of Master Controller .......................... | 57 |
List of Tables

1. Test sub-process Quality Gates. ........................................... 41
2. Test basis per test level. ...................................................... 43
3. Requirement representation in Models and Automatic Test Cases ............ 44
4. Number of Coverage Items Covered by Automatic Test Cases .............. 45
5. Time spend on MBT test generation. ....................................... 47
6. Comparison on test case time generation .................................. 48
7. Test sub-process Quality Gates. ........................................... 56
1 Introduction

The use of software in a variety of different domains is rapidly increasing and systems that rely on it become rather complex. Making sure that the software works as expected and in a correct manner becomes hard to assure resulting in software errors that could result in loss of money and loss of human lives. The errors in software can be due to engineers misinterpreting requirements, design violations or careless mistakes done by programmers. Currently, an increasing number of systems are controlled by software and rely on the correct operation of the software. Despite all the various ways to check the quality of a software such as careful analysis and process management, Software Testing is considered as a primary method used for quality assurance [2]. Software testing is the process of finding out that the software works as intended, or the process of executing the software with the intention of finding software bugs. Software testing is a process of validating and verifying that a software meets different requirements. Test cases are using certain combinations and sequences of inputs in order to validate and confirm that the software produces the expected behavior. Since one of the goals of software testing is to uncover software bugs, testing can be used to improve software reliability, usability, and maintainability.

Software testing focuses on validation and verification, and is considered an important activity in the software development process. The validation process our goal is to specify whether the software is built right and to verify if the software specifies business requirements. In the verification process, we check if the software is built correctly according to the design and requirement specifications. In the verification process, different activities are involved, such as design and specification analysis. Another classification of Software Testing is based on the level of abstraction is applied on, where different types of testing are available. For instance, according to the V-Model [2], the below-listed testing types are applicable: Unit Testing, Module Testing, Integration Testing, System Testing and Acceptance Testing. These type of testing approaches are explained in more detail in Section 2.2.2.

Software Testing is also categorized based on the definition of black-box and white-box approaches [2]. The black-box testing approach is known as functional testing, and only inputs and outputs of the systems are known. The system under test (SUT) is referred as a black-box and the internal parts, i.e structure, design, implementation, are not known by the tester. Black-box Testing is applied to different test levels, i.e System Testing, Integration Testing or Acceptance Testing. Meanwhile, in White-box Testing the internal parts of the system being tested are known by the tester. White-box testing is known as structural testing. The software under test is exercised with many inputs, in order to cover all paths of the code and determine possible outputs. In this testing technique, the internal structure of the software is analyzed. White Box Testing method is applied to Unit, Integration or System Testing.

Traditionally, creating tests is done manually by engineers, which are then automatically or manually executed on the software with the purpose of checking the result of this execution. Test results are communicated between developers and testers redundantly, so the software bugs can be detected and localized. Effective and efficient testing of a software system means effort and it is time-consuming. Recently, researchers have proposed ways to improve this process by automating parts of the testing steps.

A popular approach to improving software testing that has been proposed in the literature is called model-based testing (MBT) [3]. Typically MBT implies the use of graphical or textual models to generate test cases using graphical models to generate test cases by most of the engineers. In this way, an engineer will be able to create a model out of which test cases are automatically created based on certain criteria; automatically generate the test cases, test steps, and output checks, without any user involvement, for direct automated test execution.

Model-Based Testing is not a new field of research, starting in the late 1990s. In this technique, test cases are generated from existing high-level and abstract models of the system. Model-Based Testing process is similar to specification-based testing [1]. In Model-Based Testing, the model can
represent a part of the requirements to be tested.

Briefly, the steps of Model-Based Testing involves understanding the system under test (SUT), i.e. the software is tested according to certain requirements. In Model-Based Testing, models are used to generate input and output values for test cases which are executed. MBT promised to be a paradigm shift in how systems are tested based on automating certain parts of the testing process. In addition, the modeling activity in MBT could potentially help by clarifying and validating requirements.

The aim of this thesis is to briefly survey the scientific knowledge of using model-based testing at scale in industrial practice. In the end, a case study within Bombardier Transportation Sweden AB is performed. The case study is conducted on a certain function of the Train Control Management System (TCMS), the Master Controller. To evaluate MBT in this case, we use Conformiq Creator Tool for modeling the high-level requirements and generate test cases. Several diagrams are created using functional and software-specific requirements. The generated test cases from these models are compared to the current way of testing software used in Bombardier Transportation AB.

1.1 Motivation

Nowadays, the increased system complexity is becoming a challenge for properly testing its behavior. There are many companies that are performing testing in a manual fashion from test design, execution, to test management.

Traditionally in the test design phase, test engineers form an understanding of the system using specification and requirements. In essence, they form a mental model of the system itself. In a purely manual test design process, this mental model of a system is turned into test cases in the mind of the test designer. This is an implicit, creative process that is not reproducible and is bound to the ingenuity of the individual engineers. Traditionally test automation has mostly focused on test management or test execution, without approaching the other aspects of testing such as test creation and checking of its results. Focusing on how tests are derived, designed, managed and executed can be seen as an important part of test automation.

Usually, test automation has been mainly focused on automating test management and test execution. Unfortunately, test design often remains a manual activity. Therefore, test design and test execution are two phases of software testing that are kept separated. It is done before executing tests against the system. Even today, automated tests are often created and executed only for regression not to find defects in the new functionality. Traditional manual test design and manual test execution are still the current approaches for testing new functionality.

Testing can be a manual and laborious process which is time-consuming and costly. Hence, using automated and systematic techniques to improve the quality of the software in an efficient manner is of high importance in software development. MBT is a systematic approach, which uses abstraction, test generation, concretization and test automation. One of the main advantages and benefits of this approach is that it uses test automation to allow the generation and execution of test cases in an efficient way. In MBT, a model which represents the desired behavior of a system is used to generate a set of test cases. In addition, properties of the system are checked by systematically executing test cases on the system under test (SUT). In wide terms, MBT involves a model that describes aspects of a system that will be tested in a format and with the precision that allows either completely automatic or semi-automatic generation of test cases.

MBT is sometimes categorized as a black-box testing technique in which from design models created on a high abstraction level [4]. MBT enables the automated creation and execution of test cases, which promises to reduce the cost and time of testing and to increase the quality of the created test cases [5]. Nevertheless, MBT implies the cost of software under test (SUT). But the advantage of this is that an engineer does not have to write new test cases for the features that
have been modified in the model. Having a high abstraction level model is considered an easier way to generate and re-generate test cases compared to manually created test cases. In cases where new features are added, these are also added to the model in order to run in combination with the existing actions in the model [6].

There is a lack of empirical evidence on how the promises of MBT are fulfilled in industrial practice. In this thesis, we focus on evaluating MBT in the railway domain and especially on a system developed by Bombardier Transportation.

1.2 Problem Formulation

The past decades have indicated increasing research efforts on various approaches of model-based testing in different areas [7]. The following steps that traditionally have been performed manually in industry, are promised to be automated by MBT approaches:

MBT approaches are promising to automate the following steps in software testing that traditionally have been performed manually:

- Designing and maintaining test cases
- Optimizing the number of test cases created
- Writing executable test scripts
- Determining the correct test results
- Writing test documentation
- Optimizing regression test suite

However, the key question it seems that have been neglected [8]: how does the MBT approach be used at scale in practice and evaluates in terms of quality and cost compared to the current way of testing software in industry? In reality, there is no conclusive evidence on what benefits and challenges the use of MBT brings in practice. Some of the issues, regarding the use of models for software assurance, that we tackle are also raised by Heimdahl and Pretschner [9] are:

- the inability to adequately validate the created models; if the models are poorly validated, the testing effort will be inefficient and wrong.
- no evidence on how fully automated test case creation techniques based on models and coverage criteria can replace manual test creation performed by experienced engineers.
- cost savings in model-based testing are promised but the cost of creating the models and maybe learning a modeling language are not taken into account.

1.3 Objectives

The objective of this thesis is to describe and evaluate the use of MBT and its applicability in industrial-scale projects. This is fundamental for a successful MBT application in practice. The MBT application field, in this thesis, is the railway domain. Multiple tools will be taken into account including the Conformiq suite of MBT tools.

The main aim of this thesis is to also compare test case creation between manual testing performed at Bombardier Transportation and MBT-based test cases. The manual test cases and the ones generated from models in Conformiq Creator are compared using the following metrics:
1. Test Coverage.

Test coverage metrics measure how much of the coverage items (e.g., requirements, lines of code, model transitions) are tested:

\[
\text{Coverage} = \frac{\text{Number of covered items}}{\text{Total number of test items}} \times 100\% \quad (1)
\]

2. Time Spend on Testing.

We measure the time for creating the models, generating test cases from the models and also the time for generating test cases using manual testing.

The goal of collecting data based on these metrics is to use the data for enhancing the test process measure the quality of the project.

1.4 Research Methodology

In order to address some of the issues, we perform a case study [10], in which the data is collected from Bombardier Transportation Sweden AB, a train manufacturing company, by using an MBT tool to model a train control management system and to assess this representative approach to automated model-based testing in a realistic environment. The company has provided the information needed to perform the thesis.

We have used both quantitative (i.e., measurements) or qualitative (i.e., artifacts and descriptions) type of data. We have defined a case study protocol for detailing the procedures for modeling the SUT as well as the collection and analysis of the raw data. A case study protocol is document that contains an entire set of procedures, also involving the procedure of collecting data. A Case Study Protocol (CSP) is a set of directions that can be used to build and control a case research project [11]. Thus, it blueprints the operations guiding the of researcher(s) throughout a case research project. Further, a case study protocol can be specifically beneficial in research projects involving multiple researchers as it guarantees consistency in data collection and analysis [11]. Yin (1994) [11] proposes a very simple protocol approach for case study giving priority to field procedures, case study questions, and a guide for the final write up. template is to provide a common structure case study. The case study protocol is based on the below steps:

1. Getting started by designing the case study and by defining the research question and research questions.

2. Data gathering involving organizing data gathering, access, and permissions required within Bombardier Transportation; types of data gathering such as observation, interviews with Software testers and documents review are used. Multiple data collection methods are taken into consideration, using qualitative and quantitative data.

3. Creating the models based on functional software requirements using Conformiq Creator Tool and generating test cases.

4. Data analysis of the measures used to evaluate automatically generated test cases from the models and manual test cases actually performed by software testers from Bombardier Transportation.

We investigate the following research questions:

1. RQ1. How can MBT be used for modeling and testing of a train control management system?
2. RQ2. Is MBT applicable in an industrial context for test generation?
3. RQ3. How do MBT tests compare with manually created tests in terms of cost and requirement and model coverage?

The following steps will be followed to answer the research questions (these are written in terms of four main research goals):

1. RG1. Use an MBT approach suitable for modeling a train control management system.
2. RG2. Generate, execute and check the results of MBT-based tests.
3. RG4. Compare MBT with traditional hand-crafted tests in terms of cost and quality.

1.5 Structure

This thesis report is divided into 8 chapters and is structured as follows. Chapter 1 highlights the selected problem domain, the background of the topic, research aims, objectives, expected outcomes and adopted research methodologies, which will be utilized in this study. In Chapter 2 we present the formal definition of coverage criteria in model-based testing, background information on software testing and Model-Based Testing. This chapter illustrates some of the most important definitions as well and also defines the concepts related to MBT and Software Testing. We present the case study design in Chapter 3 and the results of the case study on Chapter 4. We consider the limitations and threads of validity on Chapter 5, and Related Work on Section 6. We make a discussion on Chapter 7, and finally, we conclude with Chapter 8: Conclusions and Future Work.

1.6 Contributions

The thesis is focused on model-based test generation based on the Conformiq Creator modeling language as test models and certain coverage criteria that are applied. Figure 1 provides an abstract overview of automatic model-based test generation. For more details one could check the taxonomy of Utting et al. [12] that includes test execution and requirements. In contrast, we just depict just the elements necessary to give an outline of MBT.

The contributions of this thesis are focused on the combination of different test models, the relations of test models and coverage criteria, and the combination of coverage criteria. This thesis contains three main contributions.

First, we investigate the literature and provide a related work on case studies detailing MBT use in industry and its evaluation. Second, by using a well-known commercial MBT tool (i.e. Conformiq Creator) we create the models based on requirement specifications of the Train Control Management System (TCMS) from Bombardier Transportation. Third, the test cases automatically generated from Conformiq Creator are compared with the actual manual test suites created at Bombardier Transportation. The tests are compared based on: model coverage and time spent on testing in both approaches including the time for learning the tool and the modeling language. Overall, an evaluation detailing the execution of the MBT tool, data collection, data analysis and interpretation of the results, models, and test case generation and a thesis report detailing the case study is included in this thesis.
2 Background

In Section 2.1 we outline several important terms related to this thesis. In Section 2.2 we explain what software testing is, we describe different test processes and test types in Section 2.2.1, and testing techniques in Section 2.2.2. Furthermore, in Section 2.2.3 we describe the measures and present the coverage criteria. Model-Based Testing and its fundamentals are explained in Section 2.3 and 2.3.1 respectively. An overview of the taxonomy of Model-Based Testing is given in Section 2.3.2. In addition, we present the perceived benefits of implementing Model-Based Testing in industry (Section 2.3.3). Finally, in Section 2.3.4 we briefly explain ways on how Model-Based Testing can be deployed.

2.1 Definitions

In this section, we explain some of the concepts that have been used in this thesis.

- **TCMS** Train Control Management System (TCMS) is the Software under Test.
- **ATP** - Automatic Train Protection (ATP) is a component of TCMS that we have also modeled.
- **Conformiq** - Model-Based Testing Tool.
- **Test coverage** - The percentage of testable features of a test object that have been tested.
- **Statement Coverage** Mainly, in software testing, testers can be required to generate test cases that exercise all statements at least once [13]. A test case can be served as input to the SUT and is executed during testing. Statement coverage is typically defined as, the fraction of total number of blocks (or) statements that are executed by test data [13].
- **Branch Coverage** The branch coverage is the fraction of total number of decisions (or) branches that are executed by test data [14]
- **Path Coverage** A test case is executed at least once i.e. all the execution paths of the model from entry to exit are executed during testing.
2.2 Software Testing

Software testing is an essential activity of evaluating and executing software with the intention to find out errors and bugs. The software requirements and components are exercised and evaluated manually or by using different automation tools, to find out whether the software meets the specified requirements, and the difference between actual output and expected output. Software Testing is considered one of the most important activities in the software development process. Software Testing is considered as an important phase and the Software Development Life Cycle (SDLC), and in most of the cases more than 40% of the development time is spent on Software Testing [15, 16].

2.2.1 Test Process

Software testing is considered as a critical phase of the Software Development Life Cycle (SDLC). Software under test (SUT) goes through various phases such as test analysis, test planning, test case preparation, test execution. As shown in Figure 2, in the V-Model the following types of testing are defined:

1. Function/Unit testing. A unit is the smallest part of a software, and in Function/Unit testing individual units are tested.
2. Module testing. In Module testing components of a system are tested.
3. Integration testing. Different units of a software are grouped and tested together. The goal of Integration Testing is to find faults when sub-systems are integrated. The communication between different components is tested, to make sure data is flowing across various components correctly.
4. System testing. A complete software is tested. The overall system is tested to guarantee that its behavior or functioning is as specified in requirement document.
5. Acceptance testing. The software is tested to verify if the operational and business needs are satisfied. Acceptance testing is carried out as alpha and beta testing to ensure that

![Figure 2: V Model Test Types](image-url)
customers can observe the intended functionality. A feedback is received to further continue the improvement of the quality of software.

2.2.2 Testing Techniques

Testing can be conducted under several conditions. Two of the most influential aspects is the knowledge and the observability of the SUT's internal matters. In the following, we present black- and white-box testing.

Black-box Testing

Black Box Testing is used when the code of the module is not available. Black-box testing methods include equivalence partitioning, boundary value analysis, all-pairs testing, model-based testing, use case testing, fuzz testing, state transition tables, decision table testing, exploratory testing and specification-based testing [17]. In Functional Testing, also known as black box testing, the only information used to develop is the specification; that is, the implementation is treated as a black-box. The following steps are used:

- identify the functions the software is expected to perform,
- develop test data to check whether the functions are performed, and
- rely on an oracle to determine the correct response to the test data.

Important advantages to functional testing are that the test cases are useful even if the implementation changes, the test cases can be developed in parallel with the implementation, and the test cases can be used for customer acceptance testing. Disadvantages to functional testing are that many of the test cases may exercise the same code, programmed functions beyond are not tested, and an oracle may not exist.

2.2.3 Test Quality Measurements

here is the need for means of measuring how good test cases; quality measurements and completeness. On such approach is to use coverage criteria.

Coverage Criteria

Test coverage measures what parts of the software are exercised when we run a test suite, and determines whether the test cases are actually covering certain test goal items such as the states of the model created. The output of measuring the coverage can be beneficial in many ways to find a way to know when to finish testing. This information can be used to find gaps in testing, i.e parts that are not covered.

Transition-Based Coverage Criteria

This testing strategy is focused on the checking the transition that is exercised:

- the all transitions- criterion requires the exercise of every transition;
- the all transition pairs - criterion requires the exercise of every transition as well as the consecutive firing of any possible pair of sequential transitions.
- the all transition sequences - criterion requires the exercise of any possible sequence of transitions.
State-based coverage criteria

State-based coverage criteria focuses on reaching of states. The following state-based coverage criteria may be considered:

- the all states-criterion requires to cover every state;
- the all state pairs-criterion requires to cover every state and any possible pair of sequential states;
- the all state sequences- this criterion requires to cover any possible sequence of states.

Logic-Based Coverage Criteria

- Decision Coverage: To satisfy Decision Coverage, a test suite must cover all possible decision outcomes in a model.
- Condition Coverage is satisfied if all atomic boolean conditions of each decision are evaluated too true and false, respectively, at least once.

2.3 Model-Based Testing

Mark Utting and Bruno Legeard [18] describe four main approaches to model-based testing. These four approaches consist of generating input data from a model, generating test cases from an environment(modeling tool), generating test cases and test scripts.

In the first approach, the domain input values and information related to them are provided by the model. These input values are selected carefully and combined in order to generate and produce test input data. Nevertheless, the test design problem is not solved due to the lack of information whether the test case passed or failed.

In the second approach, a model is used to explain the expected environment of the SUT. In this approach, the behavior of the SUT is not modeled, thus it is not possible to use the output values and it is difficult to determine whether a given test passed or failed. The aim of the third approach is to generate executable test cases, which include a variety of mapping information. The test inputs are related to the expected output values and the corresponding operations. The expected behavior of the SUT, i.e., input and output relationships and the behavior of SUT.

The final approach focuses on the usage of an abstract description of a test case. Abstract test cases are transformed into executable test scripts.

2.3.1 MBT Fundamentals Tasks

MBT can be defined as the process of automating the test design where tests are automatically generated from a model of the System under Test (SUT) [19].

MBT can be seen as a kind of from design models, projected from system models at a high abstraction level [18]. One reason for selecting MBT among other testing approaches is its attempt to design of functional test order to reduce cost, generate systematically generating test cases based on the model [18]. In model-based testing, test suites are derived (semi-)automatically from the test model. Coverage criteria are often considered at the model level. Model-based testing can be applied to all levels from unit tests to system testing.

Model-based testing is usually used for functional testing for which the test specification is given as a test model. The test model is derived from the system requirements. In model-based
testing, a large set of possible tests are generated, but only a selected set of tests can be executed within feasible time. Therefore, the key aspect of MBT is how to select the tests that are most likely to expose failures in the system. Test selection criteria define how select test cases out of the number of possible traces. Test designers select certain criteria to limit the number of generated tests for example by selecting the highest-priority tests or ensuring specific coverage of system behaviors.

In short, in working with MBT understandability of the system under test (SUT), i.e. the software is developed according to the right requirements. After that design models are generated from those requirements, and then test cases are generated by using a tool from those designs. Finally, the system, done according to those generated test cases.

MBT can be represented by the following sequence of steps:

**Understanding the SUT.** SUT is analyzed properly with all possible environment behaviors. A model of the SUT is created on the basis of requirements, e.g. the software requirement specification document. The test model represents the expected operational behavior of the system under test (SUT) and its environment. Test designers can use standard modeling languages such as UML or SysML to formalize the points of control and observation of the SUT, the systems expected dynamic behavior and test data for various test configurations. Elements of the SUT such as states, transitions, and decisions to the requirements are modeled, to ensure bidirectional traceability between the requirements and the model and later to the generated test cases and test results.
**Test Selection Criteria and Test Case Generation.** This is one of the most important steps in MBT because the purpose is to define test cases, which should be good enough to detect severe and likely faults. The test selection criteria are developed in order to choose specific behavioral models of the SUT that would require testing, i.e. indirectly it describes a test suite. Models can usually generate an infinite number of tests, so test designers select criteria to limit the number of generated tests (for example, by selecting the highest-priority tests or ensuring specific coverage of system behaviors). A common approach for test selection is based on structural-model coverage (for example, determining the tests coverage of model elements). After having selected the test selection criteria, test case specifications are developed which transforms the test selection criteria to an operational profile, i.e. test cases are generated at this stage.

**Test Case Execution.** Typically, test generation in MBT is fully automated. The generated test cases are sequences of high-level events or actions by the SUT, with input parameters, expected output parameters, and return values for each event or action. This step is mainly done by executing the test case by giving a concrete input to the SUT and recording its output. The input is concretized by a component known as an adaptor. It also takes care of the output.

### 2.3.2 MBT Taxonomy

Model-Based Testing can be interpreted in multiple different ways. In Figure 4 we give an overview of the Utting’s taxonomy. The definition of the process gives rise to six dimensions of MBT approaches. Along with all possible instantiations of each dimension, these are largely independent of each other, but not entirely.

![Figure 4: Overview of the taxonomy](image)

Step 1 (building the model) is reflected by the three dimensions within the model specification category: scope, characteristics, and modeling paradigm. Steps 2 and 3 (choosing test selection criteria and building test case specifications) are reflected by the test selection criteria dimension within the test generation category. Step 4 (generating tests) is reflected by the technology di-
2.3.3 Perceived Benefits of MBT

1. **SUT fault detection.** The aim of testing is to detect faults in the SUT, which are usually exercised by certain interesting combinations of variables. Comparative studies [20][21][22] show that model-based testing works achieve better fault detection than manually designed tests. However, its fault detection power depends on the skill and experience of those writing the model and choosing the test selection criteria [19].

2. **Reduced testing time and cost.** Model-based testing practices promise to lead to less time and effort spent on testing in terms of the time needed to write and maintain the model, as well as the time spent on directing the test generation is less than the cost of manually designing and maintaining a test suite. It might also save time during the failure analysis stage after test execution. Firstly, because failures are reported in a consistent way and secondly because some model-based tools are capable of finding the shortest possible test sequence that causes the failure. Thirdly, both the code and the abstracted test can be inspected, thus they give an overview of the test through the model [19].

3. **Model Coverage.**

   The generated test cases from the model can be evaluated using coverage criterion, defined before the test generation. Coverage can also be expressed for a model, and typically deals with the control flow through the model [19].

2.3.4 MBT Deployment

Several methods have looked at how to deploy and use MBT: Offline Generation of Executable Tests, Offline Generation of Manual Tests, and Online Testing. In the offline generation of executable tests, an MBT tool generates sets of test cases that are computer readable. Later, these computer-readable test cases can be automatically deployed. For instance, these test cases can be set of (Python) classes embody incorporate the logic of generated tests. Offline generation of manually deployed tests refers to generating test cases that are readable by a human and can be later deployed manually.
3 Case Study Design

In this chapter, we introduce the Software under Test, the modeling language used to create the models and more details on the research methodology. We explain in detail how the Train Control Management System (TCMS) works in Section 3.1 and how testing of TCMS is performed in Section 3.1.1. Furthermore, in Section 3.1.2 we describe how the Master Controller (Drive and Brake Control), a function of TCMS, works.

In Section 3.2 we explain Conformiq Creator, the Model-Based Testing Tool we have selected to model the SUT, and in Subsection 3.2.1 we briefly go through its modeling environment and explain the modeling language of Conformiq Creator. Finally, in Section 3.3, we present the research methodology, we investigate 3 research questions followed by 3 research goals.

3.1 Train Control Management System (TCMS)

TCMS is the name of the software-based sub-system of the train software system. TCMS is in charge of much of many critical operations and safety functionalities of the train. Testing of this safety-critical software must, therefore, be rigorous. TCMS is an extremely complex system with multiple types of software and hardware components. Testing must be focused on the most important functionality. The TCMS system tests aim to verify that TCMS is implemented according to the functional and software requirements. These units are usually subject to thorough manual testing, but most of this type of testing is not within the scope of this thesis as it is focusing on debugging and unit-level testing plan. Test automation is currently used for testing TCMS software development in-house. The Test Execution Manager (TEM) tool enables test cases to be scripted semi-automatically.

The Train Control and Management System (TCMS) is a high capacity, infrastructure backbone built on open standard IP-technology that allows easy integration of all control and communication requiring functions onboard the train. The train control system controls all parts of train operation, including propulsion, line voltage, and passenger comfort systems. It is made up of many subsystems which can be grouped into three categories.

- TCMS - The distributed computer control system.
- OTC - All wired logic on the train, including the drivers desk.
- Other train subsystems - All train subsystems excluding TCMS and OTC.

TCMS is the center of the distributed system that controls the train. TCMS is involved in almost all train functions either in a controlling or supervising capacity. Examples of train functions include collecting line voltage, controlling the train engines, opening and closing the train doors, and uploading diagnostic data. The different intelligent units of TCMS are connected to each other and the other intelligent units on the train via different communication links, such as MVB and IP networks. The Central Computing Units (CCUs) contains the control program of the vehicle, which is an application developed specifically for the different vehicle types.

An example of the TCMS network topology is shown in Figure 5. The topology shows the TCMS units connected to communication buses. The system contains all relevant types of intelligent units found on the train and which communication interface each unit is connected to. The TCMS hardware and software developed for each particular train project consists of a substantial amount of configuration of the communication gateways units and application development for the CCUs and HMI (Human Machine Interface).
3.1.1 Testing of TCMS

TCMS is a complex system with operation-critical functionality, and testing TCMS is absolutely necessary in order to be able to achieve a safe and efficient train control system.

TCMS is an extremely complex system with multiple types of software and hardware components. Testing must, therefore, be rigorous and focused on the most important functionality as exhaustive tests are practically unfeasible.

The current test approach in TCMS consists in writing test cases, test oracles, and test scripts manually based on very detailed TCMS SW requirements. The test execution is partly automated using test scripts. The handling of test artifacts in MS Excel/DOORS and a test management tool is done manually. There exists a tool that is running the simulation of the subsystem in order to run integration tests between TCMS and other subsystems. The simulation of the subsystems is written manually. Once tests scripts and the simulation of the subsystems are available, the testing can be run semi-manual (e.g. configuration, start, restart after a failure is still manual). This automatic testing is often used for daily regression tests. The challenges deriving from this approach are diversified. Among others, they consist of:

1. Requirements are usually incomplete and other descriptions such as event lists are needed.
2. All test cases are written for the test on the lowest level, there is no integration testing of complete functions.
3. All test cases are considered to be equally important.
4. Root cause analysis of failures is done manually.
5. There is no validation of the design before the implementation is started.
TCMS components are divided into function groups and all TCMS components are tested on their own on the TCMS component test level. TCMS Component Tests are black box functional tests. Boundary value and equivalence partitioning tests are mandatory. Testing of TCMS components can be done in isolation from the rest of the software in a controlled simulated environment, making the component tests an ideal target for regression testing. Component tests are therefore part of the regression test process. Only components that have been modified during the last development iteration are required to be part of the regression tests but regression testing related components are encouraged. When a new functionality is added, new test cases have to be added to the component test specification. These test cases all become part of the next round of regression tests.

The test basis for component test, shown in Figure 6, is based on the design rather than the software requirements, requiring a different approach to test specification. Tests cases are identified by testers but written, scripted and executed by the developers. The developers are partly responsible for component test specification and fully responsible for test execution. This means that the person developing the code will be the same one as the person who performs the tests.

3.1.2 Master Controller - Drive and Brake Control

Master Controller, shown in Figure 7 is a function of TCMS and the subsystem on which we have performed the case study on. The master controller is used by TCMS to create the traction/brake reference used as input for both the traction and brake.

The train is driven either by a driver using a master control handle or automatically by the signaling system. TCMS receives a drive order as a traction reference from master control handle or signaling system. The traction reference is forward to the motor converters that power the motors. In case of service brake demand from the master controller or signaling system, TCMS will send the demanded brake force reference to the motor converters (for dynamic brake) and to the BCUs (for pneumatic brake). In case of emergency brake, the master controller and/or signaling system de-energizes the emergency brake loop ordering full pneumatic brake (emergency brake) without any dependency of the TCMS.

Traction and brake are, in manual operation, controlled by one single master controller. It
contains both drive and brake in the same controller. The traction/brake reference is analog (-100 to 100%) without any notches for either drive or brake.

The analog value is measured with a potentiometer and the fixed positions (coasting and emergency brake) are indicated with a microswitch. Both the potentiometer and the micro switches are doubled (two channels) to give redundant feed to the control system. The microswitches for emergency brake are connected to the emergency brake loop and the microswitch for coasting is connected to TCMS.

For the case study, we considered 15 functional requirements, on which we have created the models explained in Section 4.1. The requirements are related to Master Controller and ATP (Automatic Train Protection) traction/brake reference calculation. These 15 (fifteen) functional requirements are used to create the high-level models representing Master Controller and ATP System, by using Conformiq Creator MBT Tool. The Conformiq Tool and the modeling language are explained in Section 3.2 and 3.2.1.

3.2 Conformiq Creator

The Conformiq is a company, known for developing an automated test design tool, named Conformiq Creator. Conformiq Creator is a tool for automatic test case design that is driven by designing models. The Conformiq Creator designs tests, i.e. black-box tests, automatically when it is given a design model of the system as an input. As the tests are black box tests, therefore it is not necessary to design a model which reflects exactly the same internal behavior of the SUT. However, the design model should reflect the intended behavior of SUT.

Conformiq Creator is designed for people without programming skills to specify, review and create models for test generation. Domain-specific customization of models and deeper integration into existing toolchains is provided. Creator comes with a new modeling paradigm developed by Conformiq for creating models for test generation. There is also a new frontend for a proven test generation environment, specifically developed for (but not limited to) using Conformiq in system and user acceptance testing.

**Key Features of Conformiq Creator:**

- Support for a subset of standard activity diagram symbols - Mapping to UML activity diagrams
• Support for graphical data flow specification

• Support for modular model construction allowing independent development of isolated functional aspects (enables composition of larger models from supplied parts)

• Support for domain-specific SUT interface specification
  – Fully automatic generation of action keyword repository
  – Support for SUT interface change management
  – Import of interfaces from 3rd party (test execution) tools
  – Allows further automation of test harness implementation for user interface testing.

• Allows to engage and interact with stakeholders other than testers indirectly model creation and review.

• Automates model creation without programming
  – Offers domain specific actions and error checking as building blocks (e.g., login, form entry, sort, search, etc.)
  – Allows reuse and aligning with customer activity diagrams
  – Reuse interface descriptions, e.g., from the existing testing harness

• Enables generation of test harnesses from structure diagrams

Conformiq Creator consists of three main perspectives: Modelling Perspective, Test Design Perspective and Test Review Perspective, and working in Conformiq Creator to generate tests consist of 3 main steps respectively:

1. Model System Operation with Creator
   • Model or import SUT interfaces available for testing using Structure Diagram
   • Model overall SUT operation from highest (business workflows) to lowest level (interface operation) by using standard (informal) Activity diagram(s)
   • Refine your model by adding SUT interactions, data flows, and conditions based on generated Action Keyword Repository to reflect business rules

2. Direct and Review Test Design
   • Validate your model by peer review (Software Testers) and reviewing tests generated in Conformiq Test Design perspective
   • Generate Test cases and execution scripts
   • Validate requirement coverage
   • Validate expected results (Pass/Fail)
   • Enhance visibility of Functional flows

3. Generate Test Scripts and Documentation
   • Render and upload reviewed tests for documentation and/or test execution
3.2.1 Modeling Language

The models, described in Section 4.1, will be modeled using a modeling tool, i.e. Conformiq Creator Modeling Language. The model designed will be used as an input to the automatic test case design tool Conformiq Creator, to generate the test case for the corresponding model. In Conformiq Creator Modeling Language the method of creating tests, that is highly recommended, is to design a model using a graphical notation with the textual notation as an action language.

Definitions and Terminology:

1. **Conformiq Test Design Perspective** - A perspective that lets a user to generate tests, specify test coverage goals, review, export and trace tests to coverage goals.

2. **Conformiq Modeling perspective** - A perspective for creating your model.
3. **Activity Diagram** - A diagram for defining activity flows.

4. **Structure Diagram** - A diagram for defining interfaces available for testing a system or application.
5. **Action** - Either a predefined Creator action or action keyword automatically generated when saving a Structure Diagram. Actions are used in Activity Diagrams in order to express the operation of the system or application under test.

6. **[Test]Design Configuration** - Test Design Configuration is an entity in a Conformiq Creator project that stores test coverage goals as well as scripting backends for exporting the tests generated based on these goals. A Conformiq Creator project can have multiple test design configurations. The test coverage goals can be modified in the Conformiq Test Design view.

7. **Test Targets view** - An editor in Conformiq Test Design perspective that lets a user to
select coverage targets for test generation.

8. **Scripting backend** - A plugin that allows exporting the tests generated by Conformiq to a specific output format (for example Excel scripting backend exports generated tests into excel document as human-readable tests).

### 3.3 Research Methodology

Firstly, we have conducted an informal state-of-the-art and literature review study on Model-based Testing, where we have reviewed several papers and their results. Those papers were found on different databases like IEEE, Springer and Google Scholar. The papers we selected to review are mostly case studies on using MBT tools or evaluating MBT application in industry. Results of the literature review are shown in the Related Work Section.

Secondly, we perform a case study where we investigate three research questions. A case study is a research method that supports detailed investigations, in order to understand and describe clearly the phenomenon or test theories, by using qualitative and qualitative analysis. The case study design is shown in Figure 15. The model is based on Software Functional Requirements, in our case SRS TCMS Requirements. A model is an object that represents a detailed abstraction of a system from a specific perspective. The models we have created are behavior models, that describe how software systems react when provided with certain inputs.

MBT consists of three main steps. The first step is to create an abstract and simplified behavioral model of the system being tested. The model should focus on the key aspects to be tested, based on the specified requirements [18]. Next step is generating abstract test cases from that behavioral model. The number of possible tests may be infinite, so we need to specify test selection criterion to generate the test cases.

After the test cases are automatically generated they need to be transformed into executable concrete tests. This step is done in diverse approaches like using a separate transformation tool or writing some adaptor code, by solving a big gap between abstract test cases with the concrete SUT [19].
On the other side, in Manual Testing at Bombardier Transportation, creating test cases goes through three main steps: test planning, test preparation, and test specification. Test planning starts at any time during the project start-up phase. Writing the test plan is performed alongside the other planning activities in the project. Before the test plan can be approved, many other planning deliverables have to be approved first:

- The test basis is up-to-date and has been approved.
- The project test plan has been approved.
- Test environment has been evaluated.
- Test data has been gathered and test metrics updated.

![Diagram of Test Plan](image.png)

**Figure 15: Case Study Design.**

As the last step of our case study, we perform an initial comparison between test cases generated automatically from the models of Conformiq Creator, and test cases written manually by test developers. Test cases are compared using the following metrics:

- Test Coverage
  - Model coverage
    * Branch Coverage - Branch coverage is a testing method, which objective is to guarantee that all the branches from each decision node are executed at least once, and ensure that all accessible code is executed.
    * Statement Coverage - Statement coverage is a white box testing technique, which goal is to execute all the statements at least once in the model. Statement coverage can be used as a metric as well, which calculate and measure the number of statements in the model that have been executed and are covered in a test case.
    * Path Coverage - Path coverage is a testing method which intention is to design test cases in order that all paths in the application are executed at least once.
In the following, we explain the research questions and the research goals.

1. **Research Question 1**

   *How can MBT be used for modeling and testing of a train control management system?*

   In this research Question 1 our goal is to survey the current research work for model-based testing for industrial control software train control, the current solutions, and the research challenges.

2. **Research Question 2**

   *Is MBT applicable in an industrial context for test automation?*

   In Research Question 2 we aim to provide an example of the application of MBT in industry, in a Train Control Management System at Bombardier Transportation.

3. **Research Question 3**

   *How do MBT tests compare with manually created tests in terms of cost and requirement and model coverage?*

   In Research Question 3 we aim to compare the automatic test cases generated from models in Conformiq Creator with manual test cases written manually by test developers at Bombardier Transportation. The test cases will be compared based on the following metrics: model/test coverage and costs. To measure the cost we take into consideration: time for creating the models in Conformiq Creator, time for automatically generating test cases from the models, a number of test cases generated from Conformiq, time to write manual test cases, Test Developer experience, the complexity of requirement.

In the following section, we present the results of the case study.
4 Case Study Results

In this chapter, we introduce the results of this thesis work, i.e models of the SUT on Section 4.1, test cases generated from models on Section 4.2, and a comparison between manual test cases (4.3) and automatically generated test cases on Section 4.4.

4.1 General Description of the Model of SUT

In this section, we describe the models that we have created in Conformiq Creator for the SUT Master Controller. We have modeled the SUT with five Activity Diagrams that defines the activity flows and the behavior of the SUT; and one Structure Diagram that contains the input and output messages and the interfaces of the SUT.

![Structure Diagram](image)

The structure diagram is intended to be used for defining interfaces available for testing the
given system or application. From all of the structure diagrams in a project, Conformiq Creator automatically generates a so-called action keyword repository that is used to specify the operation of the system or application under test in activity diagrams. The Structure Diagram we have modeled for the SUT consists of messages and interfaces. Each of the messages belongs to a specific interface and corresponds to an input/output signal or value in the SUT. One message can contain different types of attributes such as Primitive Field, Structured Field, and Enumeration Field. In the model, we have used only the Primitive field type of attribute. Each attribute can have different data types such as Boolean, Number, String, and Date. In this model we present 4(four) interfaces:

- **InputSignals - Input**
  This is the name of the first interface, which contains twelve input messages and contains the input signals or values. For the messages of this interface, we have used messages with different data types, for example Boolean for CabineREady, SStraction, SStraking, DrivingMode, AutomaticDriving, CabineActivated. We have used Number Data Type for AVmicroswitch, AVpotentiometer, Microswitch1, Microswitch2, AX.

- **Output - Output**
  Output is the second interface, that contains 8 messages with different data type; and corresponds to the output values/signals/messages that will be shown to the driver or the system after processing the input signals. In this interface, we present messages with Number data Type like BrakeForce, TractionForce, Coasting, EmergencyBrake, EBposition.

- **IDU - Output**
  On the third interface, we present the messages that will be shown to IDU (Information Display Unit - Drivers Display). For this interface, we use two data types: Number for CoastingPos, EmerBrPos, PowBrREF messages and String for Microswitch Cutted, Potentiometer Cutted messages.

- **Signaling System - Output**
  On the fourth interface, Signalling System - Output, we include only one message: CoasST, Number data type, that corresponds to the coasting status that will be sent to the Signalling System

Regarding the activity diagrams we have modeled five which are listed below:

1. **Main Activity Diagram** - shown in Figure 17, Diagram 1.
2. **Read Master Controller Activity Diagram** - shown in Figure 18, Diagram 2.
3. **Read Signalling System Activity Diagram** - shown in Figure 19, Diagram 2.
4. **Check if input channels differ Activity Diagram** - shown in Figure 20, Diagram 4.
5. **Check which input channel is cutted Activity Diagram** - shown in Figure 21, Diagram 5.

The following paragraph explains in detail each of the Activity diagram models.

1. In Figure 17, we introduce the main Activity Diagram, based on which the model is loaded in Conformiq Creator and the test cases are generated. The diagram contains 7 nodes that are connected to each other in a control flow.

   This Activity Diagram covers 2 requirements listed below:

   (a) **REQ-108** Only the input from a master controller in a cab that is ready to run shall be used.
(b) **REQ-110** Propulsion/braking effort request from ATP system shall be used when the train is in UTO mode or STO mode.

![Main Activity Diagram](image)

The diagram starts with an *Initial Node*. Initial node is a control node at which flow starts when the activity is invoked and a control token is placed at the initial node when the activity starts. The flow continues to an *Activity Node* named *CabineReady*, which contains three *actions*, values of which are stored in variables respectively:

(a) CabineReady Message  
(b) DrivingMode Message  
(c) AutomaticDriving Message

The actions we have used in *CabineReady Activity Node* are all message type action, and they correspond to the input values that will be taken from the system. For example in this activity node, we take the inputs if the cabin is ready to run, input for the way of driving the train: manual driving or automatic driving. The flow continues with an *Decision Node* named *CabineReady?*. The decision node is a control node that accepts tokens on one or two incoming edges and selects one outgoing edge from one or more outgoing flows. Each token arriving at a decision node can traverse only one outgoing edge. Tokens are not duplicated. Each token offered by the incoming edge is offered to the outgoing edges. The notation for a decision node is a diamond-shaped symbol. The decision logic is specified in the properties of the Decision Node. In *CabineReady?* Decision Node we use the input message CabineReady, which is a boolean data type input message. This Decision node checks if the cabin is ready to run. The decision can have decision input behavior specified. An output is provided, based on the requirement set; in this decision node the requirement is if \(\text{CabineReady} == \text{True}\). If \(\text{CabineReady} == \text{True}\) then the control flow (true) continues to another Decision Node, otherwise the flow goes to the CabineReady Activity Node in order to receive input signals redundantly.

If \(\text{CabineReady} == \text{True}\) the flow will continue to If *Manual Driving?* Decision Node, which gets as an input DrivingMode Message and check if the train will be driven in manual driving. In this case, the train will get the input signals from the Master Controller, so the flow continues to the ReadMC Activity Node. If the train will not be driving in manual mode, throughout an *else control flow* we go to another Decision Node, *ifAutomaticDriving*, and check if the train will be driven in automatic mode, so the input signals will be received...
from Signalling System. If it is true, the control flow continues to Read Signalling System Activity Diagram; otherwise, the flow is sent to CabineReady Activity Node.

Both ReadMC Activity Node and Read Signalling System Activity Node are activity nodes, where an Call Action is used. A call action sends the control flow to another activity diagram specified in the properties of the call action. In our case, if the train will be driven in manual driving the flow will be sent to ReadMasterController Activity Diagram; and, if the train will be driven in automatic mode, the flow is sent to Read Signalling System Activity Diagram.

2. In Figure 18, we have modeled a considerable part of how Master Controller works. The diagram contains 12 nodes and 2 Data Objects; and are connected to each other with Control Flow and Data Flow Connectors. This model covers three requirements that are listed below:

(a) **REQ-102** The two input channels of the master controller shall be converted from voltage to percent according to a given table.

(b) **REQ-103** The converted value (-100% to 100%, Emergency brake position) from each input channel and coasting position inputs shall be presented on the IDU.

(c) **REQ-104** The lowest value of the two input channels shall be used

![Figure 18: Read Master Controller](image)

The flow starts with an Initial Node, and continues with GetAnalogValue Activity Node. GetAnalogValue Activity Node contains two Message Actions: Avmicroswitch Message and AVpotentiometer Message. This Activity Node gets the analog input values from the Master controller. Both Messages Avmicroswitch (analog value of microswitch) and AVpotentiometer (analog value of potentiometer) are stored in variables. Microswitch and Potentiometer are input channels of Master controller that serve to get the analog values (in volt) and convert them into digital one or in power/brake/coasting reference.

After receiving the analog values from the Master Controller channel, microswitch, and potentiometer, the flow is sent to two Call Action Activity Nodes: test and CheckInputChannels, in order to check if the analog signals received differ more than 0.4 V and to compare the analog values in order to use the lowest one.

When the flow will arrive at a Return Node in both Call Action Activity Nodes, the flow will be returned to this Activity Diagram again, and will the flow will continue to Decision Node named Use Lowest Value. This Decision Node we compare which analog value received
from the Master Controller is greater than the other, in order to use the lowest of them. Based on the input analog values, we send the flow continually to one of these four Decisions Nodes: Emergency Brake, Coasting, Brake, and Power. Each of these Decision Activities gets as a parameter the corresponding analog value from an input channel and checks if in the train should be applied emergency brake, brake, coasting or traction (power); and the corresponding reference (in %) calculated from a diagram given in the requirements.

Next, the flow is sent to one of these Activity Nodes correspondingly to Decision Node and its output: EmBrake, Coasting, Brake, Power Activity Nodes. Each of these Activity Nodes contains different output messages corresponding to different interfaces.

**EmBrake** Activity Node contains:

(a) *Emergency Brake Message* corresponding to Output Interface  
(b) *EBposition Message* corresponding to Output Interface  
(c) *Potentiometer Cutted Message* corresponding to IDU Interface, means that this message will be shown on the drivers display IDU.

**Coasting** Activity node contains the following messages:

(a) *Coasting Message* corresponding to Output Interface  
(b) *CoastingPos Message* corresponding to IDU Interface, means that this message will be shown on the drivers display IDU.  
(c) *CoastingSt Message* corresponding to Signallingsystem Interface, means that this message will be sent to the Signalling system as an Output.  
(d) *Potentiometer Cutted Message* corresponding to IDU Interface, means that this message will be shown on the drivers’ display IDU.

**Brake** Activity Node contained the following messages:

(a) *BrakeForce Message* corresponding to Output Interface.  
(b) *PoeBrREF Message* corresponding to IDU Interface.  
(c) *Microswitch Cutted Message* corresponding to IDU Interface.

**Power** Activity Node contained the following messages:

(a) *Tractionforce Message* corresponding to Output Interface.  
(b) *PoeBrREF Message* corresponding to IDU Interface.  
(c) *Microswitch Cutted Message* corresponding to IDU Interface.

The diagram also contains an **Flow Final Node** named *Exit*. It destroys all tokens that arrive at it but has no effect on other flows in the activity. In order to read the input signals from the Master Controller in a continuously way, the flow connector from the Activity Nodes Emergency Brake, Coasting, Brake, Power is sent again at *GetAnalogValue Activity Node*.

3. Figure 19 represents the model of the Signalling System, in the case, the train will be driven in automatic mode and the flow at the main activity diagram is sent to the Call Action Activity Node, *ReadSignallingSystem*. This diagram contains 15 Activity Nodes and 3 Data Objects, and corresponds to 4 requirements listed below:

(a) **REQ-109** The Propulsion/braking effort request from ATP system (0-100%) shall be considered as a traction reference when Traction request = TRUE and Braking request = FALSE.

(b) **REQ-1778** The Propulsion/braking effort request from ATP system (0-100%) shall be considered as a brake reference when Traction request = FALSE and Braking request = TRUE.
(c) **REQ-1779** If the traction/brake requests from ATP system are inconsistent then traction shall be removed after 150ms of inconsistency and full-service brake shall be applied after 500ms of inconsistency. Inconsistent input values are: Traction request = FALSE AND Braking request = FALSE OR Traction request = TRUE AND Braking request = TRUE

(d) **REQ-1780** If the traction/brake requests from ATP system are inconsistent for 150 ms or more then an event shall be set.

![Read Signalling System](image)

**Figure 19: Read Signalling System**

The flow starts with an Initial Node and continues to *GetRequest* Activity Node which contains 3 input messages: SStraction (boolean data type), SSbraking (boolean data type), and RecValfromSS (number data type). The flow continues at a Decision Node names *Power Reference*, which checks if the input signals set the trains into a Power Reference. If this is true the flow continues to *Power Activity* Node which contains two output messages, and after to a return node, that stops all flows in the activity.

Otherwise, the input signals from the Signalling System are compared with values received from the software requirements, if it is a Brake Reference. If yes, the flow is sent to a *Brake Activity* Node which contains a *BrakeForce* Message corresponding to the Output Interface.

The flow also goes to *Inconsistent Value* and checks if both Traction and Brake are equal to False or True. In this case, an event is sent to the driver and a timer is set. If the values are inconsistent for more than 150 ms then the Traction is removed and an event is set; if the values are inconsistent for more than 500ms Full Service Brake is applied, and the flow terminates to a Return Node.

4. The model on Figure 20 checks if the input values received from the Master controller, from two input channels: microswitch and potentiometer, differ more than 0.4 V. The model contains 7 nodes and 3 Data Objects. In this model 2 requirements are modeled:

(a) **REQ-1770** To be evaluated for each MCH in the train: An event shall be set if analog values of the input channels differ from each other with more than 1280 (corresponds to 0.4 V) for longer than 0.5 seconds.
(b) **REQ-1771** To be evaluated for each MCH in the train: If the micro switches for coasting position of the MCH have different status for more than 2 seconds then an event shall be set.

The flow arrives in this Activity Diagram when the *test Call Action Activity Node* in *Read Master Controller Activity Diagram* is reached. The flow starts with an Initial Node and continues to *GetAnalogValue Activity Node*, which contains 2 input messages:

(a) **AVmicroswitch Message** corresponds to the analog values received from the microswitch channel.

(b) **AVpotentiometer Message** corresponds to the analog values received from the potentiometer channel.

A timer is initialized, and the input values are compared if they differ. If yes, an event is set and the flow ends to a Return Node; otherwise, the timer is increased and the values are checked again. When the Return Node is reached, the flow continues to the *Read Master Controller Activity Diagram* on *CheckInputChannels Call Action Activity Node*.

5. In Figure 21, we have modeled 3 requirements listed below:

(a) **REQ-105** If one input channel is cut out or the corresponding AX unit is faulty then the other shall be used for calculating the master controller reference.

(b) **REQ-106** It shall be possible to cut out/in one of the master’s controller input channel at a time from the IDU. The IDU shall indicate which input channel that is cut out.

(c) **REQ-1905** Whenever a Cab is activated and a Master controller channel is a cut-out in that cab an event shall be set and displayed for the driver.

The model consists of 10 nodes and 4 Data Objects connected with Control Flows and Data Flows.

The flow starts with an Initial Node when the *CheckInputChannels Call Action Activity Node* is reached at *Read Master Controller Activity Diagram*.

First, the messages Cabine Activated is received, and in a Decision Node *If Cabine Activated == True* is checked if Cabine Activated == True. *Cabine Activated == False*, the flow terminates in a Return Node, otherwise if *Cabine Activated == True* the flow continues to *GetInputChannel Activity Node* which contains 3 input messages:
(a) AX Message
(b) Microswitch1 Message
(c) Microswitch2 Message

All these three messages are Number Data Type and contain the values of the corresponding microswitch. The input values of these three messages are stored in Variable Data Objects.

Then, in the model, with Decision Node 1 and 2 is checked if one of the channels is cut; and a message is sent to the IDU Interface correspondingly. If both channels are cut, the flow terminates at Return Node 1.

4.2 Automatic Test Case Generation

In Conformiq Creator, test case generation is performed by using the Progress Panel, after the model is loaded as shown in Figure 22. In order to generate test cases from the model, the model should be complete and should contain no errors. If there are modeling errors present, they will be shown in Problems View in Conformiq. Together with the error, a description of it and a location where the error is provided.

The tests will be generated and shown in the Test Cases view. Test case names are constructed from the Requirement Actions we have added to the model.

The generated tests can be exported into various formats for manual or automated test execution. The generated test suit can also be uploaded to an ALM/test management system.
In total, 19 valid test cases are generated from the model, shown in Figure 23. The test cases have an ID or test number, Name, and Targets covered Properties. The Name of the test case can be modified.

![Figure 23: Generated Test Cases.](image)

One can choose a different representation view as shown in Figure 24 for one test case: input and output messages are shown.

![Figure 24: Test Target View: Input and Output Values for a Test Case.](image)

In the test Target view, we can select a test case and on the Model Browser view, we can analyze how the test exercises the model, as shown in Figure 25 for a selected test case. For each
of the test cases, we also can see different test coverages, as show in Figure 26.

We have analyzed different type of achieved coverage for each test case and the results are presented in a table. In the table for each requirement, we present the diagram where we have modeled that requirement, the test case it belongs (automatic test cases generated from the model in Conformiq Creator), and the Branch, Statement and Path Coverage of that test case.
4.3 Manual Test Cases

Train Control Management System (TCMS) is a software-based component of the train control system. TCMS is a highly complex system which consists of many hardware and software and is in charge of many critical and safe operations. Thus, testing must, therefore, be focused on the most important function as exhaustive tests are practically unfeasible. Part of this complex software is third parties as well. Basically, a third party system is a train subsystem that has a control unit which includes software that is developed by a subcontractor of Bombardier Transportation.

There are two test risk issues that are considered during testing of TCMS, and can be divided into two categories: Test object risks and Test process risks. In testing of TCMS features covered are functionality, safety, performance, availability. Regarding our SUT, TCMS, the following test objects are tested: TCMS system, TCMS application, and TCMS components.

The test process can be divided into a number of test sub-processes as shown in Figure 27 and explained in the Table 1.
In the following, the different test levels of our System under Test, TCMS, are explained in details. Once TCMS testing is complete on the system level, the released TCMS will be handed over for integration with OTC and test of the complete train control system. TCMS concerns the following test levels, shown in Figure 28 respectively to V-Model:

1. **TCMS system test**

   The TCMS system tests aim to verify that TCMS is implemented according to the input requirements. TCMS system tests are black-box tests of functional and non-functional
requirements. Included in the TCMS system tests are also operational tests, focusing on the
download and upload of software to and from TCMS. TCMS system tests are tests of TCMS
system requirements. Testing a system level requirement shall only be performed after all
software requirements fulfilling that system requirement has been verified through the TCMS
Software/Hardware test.

The parts that are tested on the system level are all components of TCMS system. The
features that are tested include Functionality, Safety/Reliability, and Performance. The
goals of the TCMS system tests is to verify that the system is implemented according to
the system-level requirements, thereby demonstrating that the TCMS project has fulfilled
its objective.

The test coverage for the TCMS system-level tests consists of:

- All safety-related functional requirements covered
- A subset of performance and non-safety-related functional requirements will be tested
during the TCMS system tests. The TCMS test leader will have to decide how much
more tests will be run on the system level.

Figure 28: The V Model, illustrating test levels and test types

2. TCMS software/hardware (Sw/Hw) integration test

Different parts of TCMS software and hardware will be tested separately and then brought
together for further tests. The TCMS Sw/Hw integration tests have the same test scope, i.e.
include the same software and hardware, as the TCMS system tests.

3. TCMS software integration test

Integration between software components is a crucial step in the development process. Individual components may be implemented and tested according to design, but when brought together faults can be detected. These faults are usually a result of incorrect or incomplete
software design. The TCMS software integration test will find these errors at an early stage during development. Parts that will be tested are TCMS application software, consisting of TCMS components and HMI components, and TCMS configuration.

4. **TCMS component test**

All TCMS components will be tested on their own on the TCMS component test level. The entire component will be tested. The feature that is tested is functionality. The goals of the TCMS component tests are: finding faults in TCMS components and minimize the number of faults passed on to the next test level. TCMS Component Tests are black box functional tests. Testing of TCMS components can be done in isolation from the rest of the software in a controlled simulated environment, making the component tests an ideal target for regression testing. Component tests are therefore part of the regression test process.

In Figure 29 are shown a list of mandatory test metrics per test level.

![Test metrics per test level](image)

Table 2 below shows the test basis per test level:

<table>
<thead>
<tr>
<th>Test Level</th>
<th>Test Basis</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCMS system test</td>
<td>TCMS Technical Requirement Specification</td>
</tr>
<tr>
<td></td>
<td>General Requirements Document</td>
</tr>
<tr>
<td></td>
<td>TCMS download instructions</td>
</tr>
<tr>
<td>TCMS Sw/Hw test</td>
<td>TCMS Software Requirements Specification</td>
</tr>
<tr>
<td>TCMS software integration test</td>
<td>TCMS Software Requirements Specification</td>
</tr>
<tr>
<td>TCMS component test</td>
<td>TCMS component design specification</td>
</tr>
</tbody>
</table>

Table 2: Test basis per test level.

The test process is divided into the following test sub-processes:

- Test planning - The process of defining test scope, laying out the test strategy and planning the test environment.
• Test preparation - During the test preparation sub-process the test basis is reviewed and test conditions are identified.

• Test specification - Apart from further defining the test cases, this step also involves preparations for the test execution phase.

• Test execution - The part of the test process when test cases are executed, faults reported and test data gathered.

• Test reporting - Writing down the results of tests and communicating the results to project management.

4.4 Comparison Between MBT Test Cases and Manual Test Cases

We compare automatic generated test cases with manual test cases based on the metrics mentioned on section 3.3, and we have summarized the results based on these Software Testing metrics: Test Coverage, Requirement Traceability, and Costs and Time. To measure test coverage for MBT we consider Branch Coverage, Statement Coverage and Path Coverage of the model created; and overall coverage of manual test cases. We analyze and compare the Requirement Traceability on both testing approaches. To measure costs and time we consider: the number of test cases, understandability of requirements, and test case generation time. For each of the metrics, i.e Test Coverage, Requirement Traceability, and Costs and Time, we sum up the results on the last paragraph of the subsections respectively.

• Test Coverage

For each of the 15 requirements that we have covered in the model, in Table 3 we show in which diagram and test case the requirements are modeled in.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Diagram Number</th>
<th>Test Case Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-102</td>
<td>2</td>
<td>52, 61, 65, 66, 68, 72, 73</td>
</tr>
<tr>
<td>REQ-103</td>
<td>2</td>
<td>65, 66, 68</td>
</tr>
<tr>
<td>REQ-104</td>
<td>2, 4</td>
<td>61, 65, 66, 68, 72, 73</td>
</tr>
<tr>
<td>REQ-105</td>
<td>5</td>
<td>64, 69, 70, 71, 72, 73</td>
</tr>
<tr>
<td>REQ-106</td>
<td>5</td>
<td>64</td>
</tr>
<tr>
<td>REQ-1905</td>
<td>5</td>
<td>61, 69, 70, 71, 72, 73</td>
</tr>
<tr>
<td>REQ-107</td>
<td>5</td>
<td>61, 69, 70, 71, 72, 73</td>
</tr>
<tr>
<td>REQ-1770</td>
<td>3</td>
<td>52, 53, 54, 61, 64, 65, 66, 68, 69, 70, 71, 72, 73</td>
</tr>
<tr>
<td>REQ-1771</td>
<td>4</td>
<td>28, 33</td>
</tr>
<tr>
<td>REQ-108</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>REQ-109</td>
<td>3</td>
<td>41</td>
</tr>
<tr>
<td>REQ-1778</td>
<td>3</td>
<td>41</td>
</tr>
<tr>
<td>REQ-1779</td>
<td>3</td>
<td>59</td>
</tr>
<tr>
<td>REQ-1780</td>
<td>3</td>
<td>59</td>
</tr>
<tr>
<td>REQ-110</td>
<td>1</td>
<td>33, 41</td>
</tr>
</tbody>
</table>

Table 3: Requirement representation in Models and Automatic Test Cases

From the test cases generated from models in Conformiq Creator we have analyzed different models coverages like: Branch Coverage, Statement Coverage and Path Coverage, which results are shown in Table 4. These results were collected from test case analysis in Conformiq, provided after the test case generation.

In the Conformiq Test Design Perspective, the below testing goals are provided to the user:

– Activity Diagrams
* Nodes
* Control Flows
* Action Invocations
* Data Objects

- Conditional Branching
  * Decision Coverage
  * Compound Decision Coverage

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Branch Coverage</th>
<th>Statement Coverage</th>
<th>Path Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>41</td>
<td>4</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>53</td>
<td>4</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>54</td>
<td>4</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>59</td>
<td>8</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>13</td>
<td>29</td>
</tr>
<tr>
<td>61</td>
<td>8</td>
<td>14</td>
<td>29</td>
</tr>
<tr>
<td>64</td>
<td>7</td>
<td>18</td>
<td>35</td>
</tr>
<tr>
<td>65</td>
<td>7</td>
<td>17</td>
<td>29</td>
</tr>
<tr>
<td>66</td>
<td>9</td>
<td>17</td>
<td>31</td>
</tr>
<tr>
<td>69</td>
<td>8</td>
<td>17</td>
<td>35</td>
</tr>
<tr>
<td>70</td>
<td>6</td>
<td>18</td>
<td>35</td>
</tr>
<tr>
<td>71</td>
<td>8</td>
<td>17</td>
<td>35</td>
</tr>
<tr>
<td>72</td>
<td>13</td>
<td>17</td>
<td>41</td>
</tr>
<tr>
<td>73</td>
<td>10</td>
<td>17</td>
<td>32</td>
</tr>
<tr>
<td>Total</td>
<td>122</td>
<td>248</td>
<td>476</td>
</tr>
</tbody>
</table>

Table 4: Number of Coverage Items Covered by Automatic Test Cases

In Figure 30, we show the total Coverage achieved by each test goal (and its components) and the test case number that covers the current test case goal; and the data regarding the coverage of the automatically generated test cases is summed up in Table 3 and 4.
For all the test cases 100% Coverage is achieved. For each test goal, each diagram and its components (like nodes, decisions, data objects) the coverage percentage is accompanied by the total number and the number of the items covered by the test cases. For example, as shown in the Figure 30, for the selected test case we have 100% coverage, and 13 Control Flows covered out of 13 total Control flows in that Diagram (100% 13/13). On the right side of the figure, we have the test case numbers (52, 53, 59, ..., 72) and by an X is shown the decision/nodes/control flow/data object/etc... that are covered by this test case. For example, the Else Control Flow that connects Decision Node 1 with Decision Node 2 is covered by test cases number 64, 69, 71, 72. This data is provided for all diagrams and their modeling elements including: initial node, activity nodes, decision nodes, control flows, data flows, data objects and final nodes.

Regarding Manual Testing, we used test suites created by industrial engineers in Bombardier Transportation from a TCMS project. Manual test suites were collected using IBM Rational DOORS, a system for requirement management. Through DOORS, it is possible to trace a test case to the requirement it is supposed to fulfill. It is possible to observe which test cases were run on each version of the test object and thereby establish the test coverage of a certain model version. According to interviews performed with software tester Engineers and requirement engineers at Bombardier Transportation, the chosen coverage usually depends on the experience of the tester who is writing the tests. Test cases are written by the experienced testers and after they are reviewed and approved by experienced testers or an independent person. For the majority of programs considered, manually created test suites achieve 100% decision coverage. The coverage achieved by manually created test suites is ranging between 63% and 100%. The manual testing at Bombardier Transportation for system testing is semi-automatic; no direct tool support is used for measuring the test coverage at this level, and this is still an open issue and there are several discussions regarding the application of such a concept. The aim when software testers write the test case is to test all inputs and all outputs, and to test the functionalities of the software. Software testers do not consider the option of combining all inputs to test the software. Furthermore, in manual testing statement coverage is not full, since they focus on testing functionalities and testing inputs and outputs.

To sum up, Manual Testing achieves high coverage, but test cases are not very detailed. MBT achieves higher test coverage compared to manual testing with the automatically generated test cases being more detailed than manual ones. The coverage of the automatic test cases depends on the understanding of requirements, and the quality of the test cases. In MBT, test cases are generated by considering the test coverage as a test goal, thus they are naturally better at this than manual testing.

- **Requirement Traceability**

Requirement traceability is the ability to describe and follow a requirement in both forward and backward direction, by defining and maintaining relationships to related development artifacts such as code, configuration files and test cases. Testing is a significant component in the software development life-cycle. Traceability is a term broadly used in the software testing and is an active research area in software engineering [23]. A large number of test cases leads to effort and cost spent, that is why many testers give a high importance to traceability. Having many test cases leads to an increase in effort and cost spent on testing, thus many industrial developers, testers and managers give a lot of importance to traceability. It is more convenient and important to create, maintain and find the links of traceability in testing through an automated process as requirement traceability links are outdated when a software evolves.

Traceability in model-based testing helps to trace back to the respective requirements and design models when a test fails. A large number of problems are identified in the designing phase.

For example, calculation of a Branch Coverage, Statement Coverage and Path Coverage as shown in Table 4. From Table 3, we can see which requirements are part of a diagram and if they are represented in two different diagrams.
Industries usually find it difficult to track the results back to the system requirements in the MBT approach. Hence, requirement traceability is a challenge in MBT. Recently, many studies [24] [25] have come up with a better way to make measure traceability in the MBT approach.

In Conformiq Creator the traceability can be observed by using the Render Test Cases functionality. All tests generated at this point will be converted by all attached scripting backends according to their property settings. Test case rendering is presented in Appendix B.

For the manual test case created for the Master Controller, the traceability can be tracked through DOORS [26], and there are several ways to make the requirements traceable through the test cases. The requirements are stored in DOORS application, and are linked to other requirements and to test cases. Traceability is through an excel document generated from DOORS. Regarding Component Testing, traceability cannot be tracked since the requirements are not in DOORS. In this case, the traceability is recorded manually by using a specific tool.

By comparing and trying to track the requirement traceability of both approaches, we conclude that the requirement traceability is lower in MBT compared to the use of Manual Testing and Bombardier Transportation’s current way of working.

- **Costs and time**

To measure costs and time, we have examined the number of test case, understandability of requirements, and test case generation time.

- **Number of Test Cases**
  
  For automatic generated test cases in Conformiq Creator, 19 valid test cases are generated from the model of SUT, TCMS and ATP.

  Meanwhile, for the same requirements, 2 test suites are written by Software Tester Developer. Each of the test suite contains 10-15 test cases. The test cases from Manual Testing are listed in the Figure 32, in Appendix Section.

- **Understandibility of requirements**

  This metric depends on the experience of the software tester. Usually, every tester writes the test cases depending on his own knowledge and understanding of the system, and this is considered a major challenge in Manual Testing.

  In MBT, the automated test cases are not fully understandable by humans, but this always depends on the tool one is using. For example, Conformiq Creator adds reasonable details on the test cases, so that humans can understand what are the details and what is to be tested.

- **Test Generation Time**

  In order to analyze the time spent on tests we gathered data throughout the thesis work period. Table 5 presents a summary of the time spent of creating the models and generating the test cases. A comparison regarding time of generating automatic test cases, and time of writing the manual test cases of Master Controller is shown in Table 5.

<table>
<thead>
<tr>
<th>Time spend on MBT test generation</th>
<th>Structure Diagram 1</th>
<th>Diagram 2</th>
<th>Diagram 3</th>
<th>Diagram 4</th>
<th>Diagram 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating the model (hour)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Generate tests</td>
<td></td>
<td>2 minutes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total (hours)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Table 5: Time spend on MBT test generation.
As shown from the table, we spent 7 hours to create the model, and 2 minutes to automatically generate the test cases, by just clicking one button. We cannot generate test cases if there are errors in the model. If there are paths, statements or nodes that might not be reached in all the combination of input, not full coverage will be achieved, and the test case generation will take more than 4 minutes, this depending on how much the model is not complete. The test generation time depends on the complexity of the model, the test design configuration and the Conformiq Creator algorithmic settings. In the time spend on creating automatic test cases, we also consider the upfront time for learning the modeling tool and modeling language. To conduct our case study, we spend 5 weeks on learning the modeling language.

From data gathered from interviews with Software testers at Bombardier Transportation, they do not keep record of time spend on manual test cases. The test cases are written 2-3 years ago, and engineers are working on maintaining them. Approximately, for the Master Controller 3-4 hours are spend to write the test cases, this counting on the complexity of the requirement and the number of test cases. The Master Controller requirements are covered by 2 test suites, and roughly 8 hours are spend on writing these test cases. In Table 6 we show the comparison between manually and automatically test cases for the Master Controller, considering the ‘time’ metric.

<table>
<thead>
<tr>
<th></th>
<th>MBT</th>
<th>Manual Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate automatic test cases</td>
<td>7 hours</td>
<td>-</td>
</tr>
<tr>
<td>Write manual test cases</td>
<td>-</td>
<td>8 hours</td>
</tr>
</tbody>
</table>

Table 6: Comparison on test case time generation

From this comparison we suggest that generating automatic test cases from the models we used in this study takes less upfront time compared to Manual Testing.

Cost and time are one the most important attributes of any approach. In this thesis scope, we were not able to define which of the testing approaches is more costly, Manual Testing or MBT as this depends on the different scenarios, the application to be tested, the method of testing and conditions of testing.

- **Advantages and disadvantages of MBT and Manual Testing**

  In manual testing developers or software testers run test cases manually, comparing program requirements and actual outcomes in order to find code defects or bugs. Manual testing is a good fit for smaller projects as well as the projects which have not stable products specification or changes continuous basis in terms of design, functionalities and financial resources. Manual testing is not accurate at all times as test cases are executed by humans, so the chance of human errors increases and it is less reliable; It is time-consuming as well as it requires an upfront investment. Manual testing is only practical when the test cases are run once or twice, as well with systems where designs, functions are changing frequently. Software automation tools are expensive. Short-term cost is much lower with manual testing. Here directly involved human observation which may more useful if the goal of product is user-friendly and improved customer experience. Automated tests some of the issues the end user might encounter might be overlooked in an automated test. A human user might handle the program in a way which might give rise to errors that are more likely to be skipped in an automated test but caught with manual testing. It’s flexible than automated tools where automated tools run according to a set of rules. Test cases are set up and programmed into the automated tool, and then the tests are run. When a change occurs in the project the whole process might have to be repeated. But with manual testing this can be easily incorporated into testing routine. In terms of result or quick response upon priority issues or bugs manual testing more effective for reporting and fixing compare to automated testing. Certain tests are tough to be done manually, e.g. low level interface regression testing is extremely tough to be performed manually. As a result, it is prone to mistakes and overseen
when done by manually. A major consideration for will be repetitive. As a result, several testers had a tough time staying engaged during this process.

Automated testing runs tests quickly and effectively, could increase your test coverage, scope and depth of test cases and helps to improve software quality.

Automated testing is more reliable, as it is performed by tools and/or developed scripts. Its a practical option when the test cases are run repeatedly over a long period of time period. The one time cost of automation tools can be expensive. It is important to only use the ones that will give you full, or high coverage, as you can find. When scripts or tools not able to run in such as situation testing could not be performed, few changes at UI level also need to be change in scripts against manual testing. Few scenario may be change takes time to develop its automation script. A considerable amount of time goes into developing the automated tests and letting them run. Different tools have different limitations and they execute with respective environment and configurations.

4.5 Research Questions and Answers

In this subsection, we provide the answers to the research questions we raised at the beginning of the design of the Case Study.

1. **How can MBT be used for modeling and testing of a train control management system?**

   In Research Question 1 we aim to make the current research directions explicit for model-based testing for a train control management system. Related to this, we will identify the motivation for adopting model-based testing in a train management system that is explained in Section 1.1 where we have presented the motivations to implement MBT in the industry; the current solutions and the research challenges - which answers are discussed in Section Related Work. Also, by the results of our Case Study, we provide an example on how MBT can be used in industry, this answer of this part of the research question is showed on Section 4.1 where we describe the models of the SUT (TCMS and ATP)

2. **Is MBT applicable in an industrial context for test automation?**

   In Research Question 2 we aim to identify the different application domains in which model-based testing has been applied. This will highlight the current scope and applicability of model-based testing in an industrial context. This research question is answered in Section 4.1, where we present the models of SUT designed in Conformiq Creator, that show an example that MBT is applicable in industry.

3. **Are MBT-based tests less costly to create, and check their results than manual test cases created by industrial engineers?** This research question is answered on the basis of comparison between the Test Cases generated from the models in Conformiq Creator and Manual Test Cases. The test cases are compared based on the metrics explained in Section Study Case Design. As mentioned in the previous sections, the automatic test cases were generated from the models in Conformiq Creator. The results of the comparison and the answer to this research question are explained in Section 4.4.
5 Threads to Validity

Model-based testing offers a large number of benefits, that we have explained previously in Section 2.3.3, but it has some limitations as well. In this chapter, different validity threats related to Case Study are addressed.

Since nowadays software is complex, some parts of it might be difficult to model, by the reason of it is not necessary that all the areas of a system might be suitable for the use of model-based testing. This leading to the fact that in these parts manual testing could be used. One disadvantage is that it might take a lot of time finding out which parts and aspects of the software should be modeled and which ones should be tested manually [19].

Furthermore, software projects progress and in some cases the requirements become out of date. If this would happen to a software where Model-Based Testing is applied, the wrong model will be created and there will be a significant amount of errors in the SUT [19].

In the test design phase of manual testing, there is some test case designed with the intention to measure how testing is progressing. These measurements are beneficial and useful in Model-Based Testing since in the latter can be generated a huge number of test cases. Test progress is measured by considering other metrics, such as model and requirements coverage metrics [19].

Another practical concerning limitation of Model-Based Testing that different skills are needed to implement it comparing to manual testing. The model designers should create high-level models, and able to abstract them to be proficient in this area. This requires a lot of training and leads to increased testing costs [19].

In the cases where we have failed a test when applying Model-Based Testing, we should be able to determine the reason of causing the failure, i.e the SUT, an error in the model, or the adaptor code. In manual testing as well, we have to decide the reason for the failure, if the failure was due to a fault in the test script or in the SUT. However in Model-Based testing complex and less intuitive test sequences are generated, comparing to manually designed test sequences. Hence, to find the cause of the failed test might be more challenging and time-consuming [19].

Another limitation of the selected tool, Conformiq Creator, is that the data types for the input signals are only number, string, boolean or date. This tool does not offer float or decimal data type, so for all input and output signals of Master Controller, we have approximated the values.
6 Related Work

MBT is a promising approach for generating test cases from models of the system under test (SUT). These models are the description of system behavior.

Model-Based Testing is not a new concept in the software development and testing domain. Research on MBT started in the late 1990s, and with time model-based testing became an important term in the software development domain. Case study [18, 22], test cases consisting of abstract sequences of input and expected outputs were generated from models. The test cases were transformed into executable ones. Their work concluded that the test cases generated from the model are able to detect 11% more requirement faults comparing to test cases generated directly from requirements.

Our work is related to previous research on the efficiency of MBT, especially the investigation of costs and the empirical evidence for the use of MBT in an industrial context. In addition, it is also related to the return on investment of testing methods and their automation in general. Many model-based testing approaches are available [27] and a taxonomy to classify MBT approaches has been defined [1] on the basis of the three dimensions model specification, test generation and test execution which has later been complemented by the dimension test evaluation. Although several model-based testing approaches have been developed [27] and MBT has been used in practice [18][28], compelling evidence on its effectiveness as well as its cost reduction benefits are rarely available [27].

One of the first MBT studies on a large scale system was conducted by Dalal and Karunanithi [21]. In their work, they compare MBT with manual testing in four systems. For the comparison, they consider modeling difficulties, testing coverage, and test case implementation. Our work considered such factors using similar metrics for the SUT model and manual test cases. Therefore, we empirically observed how MBT can be applied to a Train Control Software, and how these factors influence other measured variables as the time spent, understandability of requirements, number of test cases and experience of software testers.

IBM reported significant cost reductions in system testing using the MBT generator called GOTCHA-TCBeans. They compared manually developed tests with MBT automatically generated. Their conclusions are derived mostly from gathered experience during modeling activities, but according to the discussed case studies, we could not reason the acknowledged percentages. In our study, we infer our conclusions from a case study, in which we defined metrics according to the literature, controlling different factors and analyzing possible threats to its validity. Microsoft reported the usage of MBT tools and methodologies in order to assure high-quality protocol documentation. Despite being a case study, they statistically compared test suites developed using MBT with remaining test suites for a sample of requirements tested in one testing site. At IBM, engineers and researchers found significant cost reductions and good fault finding effectiveness when performing two studies where they used a model-based test generator GOTCHA-TCBeans [18]. In the first case study [18], they tested the locking feature of the Portable Operating System Interface (POSIX) which was already tested by a manually developed test suit. The effort done for manual testing was estimated to be 12 person-months and the number of defects found was 18. MBT took 10 person months and detected two additional defects. Also, the overall cost for MBT was reduced by 17%.

Pretschner and colleagues [22] reported a network MBT comparison in the automotive industry. Their purpose was to compare automatically generated test suites with handcrafted ones regarding system testing and to the best of our knowledge, this work is the only empirical study measuring the fault detection for both approaches. Despite their conclusions are very similar to ours on the fault detection overview (which states that the same amount of defects were detected by manual and model-based tests), the conducted experiments differ in significant ways.

In the second case study by IBM [18], testing was performed in order to validate a part of a
Java garbage collector. The SUT used in this case study contained 6914 LOC (lines of code). The SUT was initially tested by using specJVM and JCK1.3, through which they found three design errors and 13 implementation errors with statement coverage of 78%. In addition, the authors used MBT with FSM (Finite State Machines) models and ran the tests generated from the MBT. They found additional two implementation faults. Also, the statement coverage achieved by MBT test cases was better than the one achieved by manual tests (from 78% to 83%). Finally, they mentioned that the and testing was around 3-4 person months and they claimed it as half the time they had spent previously on testing similar systems.

7 Discussion

The above results show the prospective strengths of MBT. We also discussed the related work in the previous section. There are, however, further interesting points to discuss. We still need humans to formulate test case specifications; that structural criteria alone do not suffice as basis for test case generation is widely undisputed. MBT is not entirely a pushbutton technology yet, the models are created manually. It is possible to conceive and hand-craft 100 tests in a few hours, but this becomes more complicated for 1,000 tests. However, the number of test cases must be restricted to a minimum because they not only have to be applied but also to be evaluated: if there is a deviation in behaviors, then the test run must be manually inspected. As mentioned above, it is, in general, likely that the benefits of automation are greater if significantly more tests could be run. This is not always the case for embedded systems. We are also aware that we used one specific modeling language, and tested an implementation at a certain stage of development. We do not know if our findings generalize for implementations in a more mature state. Furthermore, we cannot judge whether or not different coverage criteria exhibit the same characteristics. In terms of test case generation technology, we do not think that our approach is fundamentally different from others.


8 Conclusions and Future Work

In model-based testing, the important task is to build a behavioral model of the system that has to be tested. It is those parts of the system and its requirements which are important to test. The parts that need to be deeply tested must have a lot of details test generation tool a to find all possibilities. However, if the model is too complex then it will take a long time to build, debug and to generate the test cases.

In this thesis, we have conducted a literature review and a case study. Our goal was to make an evaluation of Model-Based Testing for an Industrial Train Control Software. We compare automatically and manually test cases in different metrics, i.e. test coverage (i.e. branch, path and statement), requirement traceability, cost and time. We wanted to point out the potential difference between there test cases based on the metrics mentioned.

The modeling of the SUT is performed using Conformiq Creator and its modeling language, based on Software Requirements of TCMS, Master Controller. One of the advantages of Conformiq Creator is that it only generates new test cases if the model is modified, and reuses previously generated valid test cases. However, in that case, a new test suite will be generated and sometimes it becomes difficult to figure out which test case is new and which is old. Sometimes it happens that a behavioral model contains error. Therefore, Conformiq provides excellent debugging information, e.g. reporting when a possible deadlock can occur in the model. However, sometimes the information that is given from Conformiq is really weak or none at all and it only reports that it was unable to cover some parts of the model without any defect notification.

After gathering the data from both testing approaches, MBT and Manual Testing, the results of the Case Study showed that in MBT generated test cases it is difficult to make the test cases traceable to the requirements. Requirement traceability varies from different approaches to use MBT i.e. we used Conformiq Creator MBT tool for generated automated test cases in MBT but that can be done by using Microsoft Spec Explorer, Conformiq Qtronic or other tools, the test cases can be linked back to requirements. MBT has higher test coverage and more detailed test cases if compared to the Manual Testing approach. Moreover, generating automatic test cases from the models takes less time comparing to Manual Testing.

8.1 Future Work

Based on the work and results of the thesis, we demonstrate that Conformiq Creator is beneficial for testing a component with an incremental design.

Nevertheless, in order to evaluate MBT in more depth, we propose and recommend the application of MBT in a larger SUT. Since our work was focused in component testing, we suggest the application of this testing approach on a wide scale by evaluating other types of testing as well, i.e unit testing, integration testing and finally system testing.

Also, a possible continuation of this work is to execute the automatic test cases in the Train Control Management System.
References


# A Functional SUT Requirements

On Table 7 are shown the requirements we have modeled.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-102</td>
<td>The two input channels of the master controller shall be converted from voltage to percent according to the following table.</td>
</tr>
<tr>
<td>REQ-103</td>
<td>The converted value (-100% to 100%, Emergency brake position) from each input channel and coasting position inputs shall be presented on the IDU.</td>
</tr>
<tr>
<td>REQ-104</td>
<td>The lowest value of the two input channels shall be used.</td>
</tr>
<tr>
<td>REQ-105</td>
<td>If one input channel is cut out or the corresponding AX unit is faulty then the other shall be used for calculating the master controller reference.</td>
</tr>
<tr>
<td>REQ-106</td>
<td>It shall be possible to cut out/in one of the master controller input channel at a time from the IDU. The IDU shall indicate which input channel that is cut out.</td>
</tr>
<tr>
<td>REQ-1905</td>
<td>Whenever a Cab is activated and a Master controller channel is cut-out in that cab an event shall be set and displayed for the driver.</td>
</tr>
<tr>
<td>REQ-107</td>
<td>The Master controller reference shall be calculated according to table below:</td>
</tr>
<tr>
<td>REQ-1770</td>
<td>An event shall be set if analog values of the input channels differs from each other with more than 1280 (corresponds to 0.4 V) for longer than 0.4 seconds.</td>
</tr>
<tr>
<td>REQ-1771</td>
<td>If the micro switches for coasting position of the MCH have different status for more than 2 seconds then an event shall be set.</td>
</tr>
<tr>
<td>REQ-108</td>
<td>Only the input from a master controller in a cab that is ready to run (Cab A1 is ready to run OR Cab A2 is ready to run is set) shall be used.</td>
</tr>
<tr>
<td>REQ-109</td>
<td>The Propulsion/braking effort request from ATP system (0-100%) shall be considered as a traction reference when Traction request = TRUE and Braking request = FALSE.</td>
</tr>
<tr>
<td>REQ-1778</td>
<td>The Propulsion/braking effort request from ATP system (0-100%) shall be considered as a brake reference when Traction request = FALSE and Braking request = TRUE.</td>
</tr>
<tr>
<td>REQ-1779</td>
<td>If the traction/brake requests from ATP system are inconsistent then traction shall be removed after 150ms of inconsistency and full service brake shall be applied after 500ms of inconsistency. Inconsistent input values are: Traction request = FALSE AND Braking request = FALSE OR Traction request = TRUE AND Braking request = TRUE</td>
</tr>
<tr>
<td>REQ-1780</td>
<td>If the traction/brake requests from ATP system are inconsistent for 150 ms or more then an event shall be set.</td>
</tr>
<tr>
<td>REQ-110</td>
<td>Propulsion/braking effort request from ATP system shall be used when train is in UTO mode or STO mode.</td>
</tr>
</tbody>
</table>

Table 7: Test sub-process Quality Gates.

# B Test Cases Generated from Conformiq

In Figure 31 are shown the exported test cases in Excel document.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformiq Test Specification</td>
<td>Thesis_Model</td>
</tr>
<tr>
<td>Project</td>
<td></td>
</tr>
<tr>
<td>Generation Date</td>
<td>5/16/18 4:35 PM</td>
</tr>
<tr>
<td>Number of Tests Exported</td>
<td>17</td>
</tr>
<tr>
<td>Conformiq Options</td>
<td></td>
</tr>
<tr>
<td>Automatic Test Case Naming</td>
<td>true</td>
</tr>
<tr>
<td>Enable Perturbation Support</td>
<td>false</td>
</tr>
<tr>
<td>Lookahead Depth</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Communication Delay</td>
<td>0</td>
</tr>
<tr>
<td>OSI Methodology Support</td>
<td>false</td>
</tr>
<tr>
<td>Only Finalized Runs</td>
<td>disabled</td>
</tr>
<tr>
<td>Require Conversion for Interoperability Testing</td>
<td>false</td>
</tr>
<tr>
<td><strong>Coverage Summary</strong></td>
<td></td>
</tr>
<tr>
<td>Activity Diagrams</td>
<td>100% (197/197)</td>
</tr>
<tr>
<td>Conditional Branching</td>
<td>100% (36/36)</td>
</tr>
</tbody>
</table>

![Figure 31: Manual Test Cases of Master Controller](image)

## C Manual Test Cases of Master Controller

![Traction/brake reference](image)

**Figure 32: Manual Test Cases of Master Controller**