

Dynamic test script generation for test and verification in continuous integration

ISAK DAHLSTRÖM

BACHELOR'S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY



Dynamic test script generation for test and verification in continuous integration

Isak Dahlström
`hej@isakdahls.se`

Department of Electrical and Information Technology
Lund University

Supervisor: Christin Swahn

Examiner: Christian Nyberg

June 7, 2026

© 2026 Isak Dahlström

LTH Faculty of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Abstract

Complex software must be tested carefully. This thesis looks at how automated testing can be improved for Alstom's Radio Block Centre, a system that supervises trains and sends movement authorities.

The work focuses on three practical ways to improve an existing automated test setup. First, it studies how the test suite can be made more dynamic by introducing randomness into current test scripts. This is done by changing train length, which is normally fixed in the existing tests. Second, it develops a template-based method for generating movement authority tests from site data. Third, it defines a workflow for checking whether documented minimum and maximum values for user-configurable parameters are included in the existing test scripts.

The results show that random variation of train length can expose hidden assumptions in test scripts. They also show that route-specific tests can be generated from site data and that missing boundary-value tests can be found in a structured way. These methods show that the test suite can be made more dynamic, while also providing Alstom with ways to improve coverage in the existing test environment.

Keywords: ERTMS, Radio Block Centre, Automated testing, Test generation, Movement authority, Template-based method, Boundary value analysis, Train length

Sammanfattning

Komplex mjukvara kräver testning på flera nivåer. Detta examensarbete undersöker automatiserad testning av Radio Block Centre (RBC). En RBC bestämmer hur långt ett tåg får köra och i vilken hastighet.

Examensarbetet har tre fokusområden. Det första området undersöker hur nuvarande testscript kan göras mer dynamiska genom att införa slumpmässighet. Slumpmässighet införs genom att variera tåglängd. Det andra området fokuserar på hur en mallbaserad metod kan användas för att generera testscript utifrån site data. Det tredje området letar efter en systematisk metod för att kontrollera om gränsvärden för projekterbara parametrar testas på subsystemnivå.

Resultatet visar att testmiljön kan bli mer dynamisk genom att använda slumpmässighet. Resultatet visar även att det är fullt möjligt att förbättra täckningen av testerna genom att använda en mallbaserad metod för att generera testscript utifrån site data. Även om ett fåtal av de genererade testscripten misslyckades, så kan avvikelserna användas för att vidareutveckla mallen. Avslutningsvis presenteras en metod som Alstom kan använda för att analysera täckning av projekterbara parametrars gränsvärden på subsystemtestnivå.

Nyckelord: ERTMS, Radio Block Centre, Testautomatisering, Testgenerering, Movement authority, Mallbaserad metod, Gränsvärdesanalys, Tåglängd

Contents

1	Introduction	5
1.1	Background	5
1.2	Purpose	6
1.3	Goal	6
1.4	Research questions	7
1.5	Motivation	7
1.6	Delimitations	7
2	Technical background	9
2.1	System context	9
2.2	ERTMS specifications	10
2.3	Types of testing	10
2.4	Eurobalises	11
2.5	Radio block centre	12
2.6	Test environment	13
3	Methodology	14
3.1	Work phases	14
3.2	Information gathering	15
3.3	Document studies	16
3.4	Test simulation	17
3.5	Data processing	18
3.6	Train length variations	19
3.7	Template-based test generation	20
3.8	Boundary value coverage	23
3.9	Large language models	26
3.10	Evaluation of sources	28
4	Analysis	30
4.1	Test log files	30
4.2	Train length variations	30
4.3	Template-based test generation	34
4.4	Boundary value coverage	37

4.5	Problems and solutions	39
5	Results	42
5.1	Train length variations	42
5.2	Template-based test generation	46
5.3	Boundary value coverage	48
6	Discussion	51
6.1	Interpretation of results	51
6.2	Common patterns	54
6.3	Limitations	55
6.4	Implications for test automation	56
7	Conclusion	57
7.1	Summary	57
7.2	Ethical aspects	58
7.3	Research questions	58
7.4	Contributions	59
7.5	Practical application	60
7.6	Future work	60
8	References	62
A	Code	65
A.1	Template-based test generation	65
A.2	Boundary value coverage	65
A.3	Data processing	66
A.4	Execution instructions	66

Figures

2.1	ETCS level 2 components.	12
2.2	RBC isolated by using simulators.	13
3.1	Test script execution flow.	17
3.2	Data processing from raw logs to plotted data.	18
3.3	Test generation flow.	21
3.4	Route from signal HST455@SME to HST453@SME.	22
3.5	First relevant balise identified along the route.	23
4.1	Effect of short train length on l_endsection.	33
4.2	Effect of long train length on l_endsection.	34
5.1	Outcome overview and error categories for minimum train length. . .	43
5.2	Affected elements and error categories for minimum train length. . .	43
5.3	Outcome overview and error categories for maximum train length. . .	44
5.4	Affected elements and error categories for maximum train length. . .	45
5.5	Outcome overview and error categories for randomised train length. .	45
5.6	Affected elements and error categories for randomised train length. .	46
5.7	Test outcomes for the Jenkins runs.	47
5.8	Failures and error categories for the third run.	48
5.9	Effect of the user-source filter on the analysed parameter set.	49
5.10	Boundary value coverage for analysed parameters.	50

Tables

3.1	Extracted CSV fields and descriptions.	19
3.2	Experiment configurations for changing l_train.	19
3.3	Template parametrisation examples.	21
3.4	Boundary coverage categories used for RQ3.	26
3.5	Use of large language models during this thesis.	27
4.1	Test outcomes for different train length configurations.	31
4.2	Value mismatches for minimum train length.	31
4.3	Value mismatches for maximum train length.	32
4.4	Value mismatches for randomised train length.	32
4.5	Failure categories in the final RQ2 Jenkins run.	36
4.6	Parameter counts after applying the analysis limits.	37
4.7	Examples of boundary values found in test scripts.	38
4.8	Problems covered in this section.	39
5.1	Outcome counts for changed train length configurations.	42
5.2	Progression of RQ2 Jenkins runs.	47
5.3	Overview of parameter coverage and test counts.	50

Acknowledgements

I would like to thank everyone who supported me during this thesis. First, thanks to Göran for welcoming me to his team and giving me the opportunity to develop my thesis at Alstom. A special thanks goes to Mattias, who took me under his wing, helped me understand the test environment and patiently explained many concepts that were new to me. I also want to thank Oscar and Anders, for answering my questions and helping me understand both the technical details and the railway domain. Thank you to the rest of the team at Alstom and to all co-workers in Hässleholm for making me feel welcome during the thesis work. I also want to thank Martin for putting me in touch with Alstom. Thank you to Christin, my university supervisor, for her support and guidance during the thesis. Thank you to Ulla, for helping me with useful writing advice. Finally, I want to thank Christina from secondary school. I hope she knows how much her support meant to me.

Terminology

AdSim

Test component used for simulating a country specific adaptation (GA).

Balise

Electronic component on the track that transmits data to a passing train, allowing the train to determine its position.

CI

Continuous Integration.

CSV

Comma Separated Values.

Docker

Software for building and containerising applications [1].

Doxygen

Software that generates documentation from annotated source code.

ERTMS

European Rail Traffic Management System.

ETCS

European Train Control System.

European Rail Traffic Management System

International standard for railway signalling and control. Two main components: ETCS for train control and GSM-R for communication.

European Train Control System

Signalling and control component of ERTMS that monitors train speed and ensures trains do not exceed their authorized limits.

GSM-R

Global System for Mobile Communications - Railway.

Jenkins

Automation server used to execute tests in the continuous integration environment.

L_TRAIN

Train length, expressed in meters, as defined in SUBSET-026 Chapter 7.

LRBG

Last Relevant Balise Group.

MA

Movement Authority.

Movement Authority

Digital permission sent from the RBC to a train, defining the distance and speed a train is allowed to travel.

Onvia Control L2-B

Alstom product name for the Radio Block Centre system studied in this thesis.

Radio Block Centre (RBC)

Central safety system that supervises train speed based on track position.

RBC

Radio Block Centre.

RQ1

Research question 1.

RQ2

Research question 2.

RQ3

Research question 3.

Struct

A data structure that groups related values under one name.

SUBSET-023

ERTMS Glossary of Terms and Abbreviations.

SUBSET-026

ERTMS System Requirements Specification.

Tcl

Tool Command Language.

TclExpect

Extension to Tcl that automates interactions with software exposing a text terminal interface.

VCS

Vital Computer System.

YATE

Alstom's test engine used to run test scripts and manage communication between the RBC software and the simulators.

Introduction

This chapter provides the background needed to understand the thesis. It explains automated testing, introduces railway signalling, presents the purpose, motivation and research questions.

1.1 Background

Modern software is often too complex to verify manually. Automated testing helps in three main ways.

- It runs the same test steps in the same way every time.
- It helps detect when new changes break existing logic.
- It makes it possible to test rare cases in a simulated environment.

Alstom develops systems for railway traffic. Trains move on physical tracks, but software controls when a train may move and how far it may go. These decisions are handled by signalling systems. In modern systems, the signalling software can supervise trains remotely and communicate with them by radio via mobile network (GSM-R).

The system described in this thesis is the Onvia Control L2-B, which is Alstom's Radio Block Centre (RBC). The main job of the RBC is to supervise trains and send them digital permissions to move, called Movement Authority (MA). These are sent via radio according to the international European Rail Traffic Management System (ERTMS) standard. The purpose of ERTMS is to ensure interoperability between national railway systems in Europe.

1.1.1 Related work

Previous studies have covered railway signalling, ERTMS verification, RBC testing and automatic test generation. The closest study by topic is about model-based test case generation for an RBC function [2]. Although both that study and this thesis

deal with automatic test generation for an RBC, the generation approach differs. That paper used model-based generation, while this thesis uses site data and a template-based test script. Automated RBC testing at Bombardier in Hässleholm was also studied in [3]. That study examined the same Hässleholm test environment, but focused on test execution time rather than generation of new test cases.

Two other interesting studies also focus on railway signalling, though they address different problems. Functional testing of the ERTMS/ETCS trackside system was studied in [4], while artificial intelligence for railway signalling was explored in [5]. No identified study combines train-length variation, template-based MA test generation from site data, and boundary-value presence analysis in the same RBC test environment.

1.2 Purpose

The purpose is to find ways to improve testing of signalling software. The research focuses on three areas: introducing variation into test scripts, generating tests from site data, and defining a method for checking boundary values for site-data parameters in existing tests.

1.3 Goal

The goal is to improve testing of railway signalling software by analysing and extending an existing automated test setup.

The testing is performed in an isolated environment where the RBC is executed together with simulators. The surrounding systems are replaced by simulators. Test cases are implemented as TclExpect scripts executed through YATE. These scripts define system inputs and verify the RBC responses based on expected behaviour.

The work focuses on improving how the simulation-based test setup is used. The thesis studies how test inputs are defined, how parameter variations affect test outcomes and how coverage of configuration data can be assessed.

Research question 1 (RQ1) analyses how variations in train length affect the outcome of the automated test suite. Train length is configured in a setup script executed before each test. Boundary values are tested explicitly and additional values are generated automatically using a randomisation function. The goal is to identify tests that pass but rely on implicit assumptions in the test design.

Research question 2 (RQ2) focuses on developing and evaluating a method for generating test scripts from site data. The site data includes track layout, balise positions and route definitions. The method uses this data to follow routes and extract balise positions. These positions are then used to generate test scripts automatically. The goal is to evaluate whether this approach is feasible.

Research question 3 (RQ3) defines and demonstrates a method for checking whether minimum and maximum configured parameter values are represented in the existing test suite. Coverage is defined as whether each parameter value appears in at least

one test script. Python scripts are used to analyse test scripts and identify missing boundary values.

1.4 Research questions

The following questions are answered through this thesis:

1. How can the static test suite adapt to a more dynamic (randomised) behaviour to be used in continuous integration?
2. How could Alstom generate test scripts covering the full test site for at least one selected signal type based on a template test case?
3. How can Alstom ensure that site data configuration used for subsystem test covers all min/max values of all the configurable parameters?

1.5 Motivation

This thesis provided an opportunity to broaden knowledge in automated testing and script generation, which are increasingly critical disciplines within software engineering. For engineers, automated testing is valuable because it establishes a repeatable process that reduces reliance on manual checks, ensuring that system logic remains intact as software evolves. Furthermore, as industries shift towards automated and data-driven systems, expertise in test automation represents a highly valuable skill set for modern software engineering careers.

This research focuses on railway signalling, an area where software is a main component. In such systems, testing is required at many levels. A failure can have operational consequences, which increases the importance of reliable and systematic testing.

By establishing a methodology for test generation, this thesis aimed to investigate how Alstom can extend their test suite coverage. The development of a prototype sought to provide a proof of concept for future development.

1.6 Delimitations

These delimitations specify what was not included:

- The thesis did not involve making changes to the underlying signalling logic.
- The method for automated script generation focused on one selected signal type as a proof of concept.
- The thesis focused on subsystem and integration testing of railway signalling software.
- The thesis developed methods for script generation, not production-ready software.

- The thesis did not address bugs or performance issues within external third-party software or libraries used in the test environment.
- Testing was limited to Alstom's environment in Hässleholm, Sweden.

Technical background

This chapter presents the technical background of the thesis. It describes the system context of the railway signalling software, introduces testing concepts, explains the architecture of the Radio Block Centre and the simulated test environment.

2.1 System context

When testing software, a common approach is to create a digital version of the real world to observe system behaviour. In Hässleholm, Alstom develops railway signalling software for their Onvia Control L2-B system. This system runs on a hardware platform called Vital Computer System (VCS). To minimise the risk of failure, the VCS uses a redundant architecture. The signalling logic runs in parallel on two computers. The outputs from these computers are constantly compared. If the outputs are not identical, the system detects the inconsistency and immediately enforces safe signalling behaviour [6].

In the 1990s, the European railway industry and European institutions initiated ERTMS to replace many incompatible national train protection systems with one common system. The goal was technical interoperability and harmonisation across Europe. The ERTMS standard defines the European Train Control System (ETCS) levels, including level 1 and level 2 [7]. These levels describe how trains interact with trackside systems.

Level 1 keeps conventional lineside signals and uses balises for data transmission. Balises are installed in close proximity to the signals. If a train stops at a red signal, the train waits for an updated movement authority from the balise. Level 2 removes the need for lineside signals and uses radio communication (GSM-R) between the train and the RBC. The RBC generates movement authorities and transmits them directly to the train. In level 1 and level 2, the railway is divided into fixed blocks. Only one train is allowed in each block at a time and movement authorities are based on these blocks [7], [8].

2.2 ERTMS specifications

Modern railways use many different systems that must work together, even when they are developed by different organisations and used in different countries. To make this possible, common rules must describe how trains and trackside systems exchange and interpret information [7]. These shared rules ensure that different systems still behave in a compatible way when they interact.

The ERTMS specifications provide these common rules through a large set of documents, known as subsets, which cover system requirements, interfaces, communication protocols, test specifications and engineering rules [9]. Within SUBSET-026, Chapter 7 defines the variables used in the data flow, including their meaning, constraints and valid ranges, while Chapter 8 details the structure of the messages and packets in which these variables are transmitted [8].

2.3 Types of testing

Software verification and software testing are not the same. Verification focuses on requirements and evaluates whether work from each development step meets requirements for completeness, correctness, and consistency [10]. Testing has a broader scope and evaluates software behaviour within the chosen test coverage [11].

Software testing is used to check that a system behaves as expected. Some tests check small parts of the software. Other tests check how larger parts work together. Common test levels include unit testing, integration testing, subsystem testing and system testing. Each level finds different types of problems [12].

When testing moves from small units to larger systems, the focus changes. The question is no longer only whether one function works. At higher test levels, the focus shifts from single functions to system behaviour. The tests check whether connected parts exchange the right data, react correctly to input, and produce the expected outputs.

In complex systems, higher-level testing plays a central role in identifying faults that do not appear at lower test levels. These systems often rely on complex configurations, external inputs, and interactions between independent subsystems. Such behaviour is better assessed in an integrated test environment, where the relevant system interactions are present.

Railway signalling systems are tested at several levels, from small unit tests to larger integrated environments. In this thesis, the focus was subsystem testing and integration testing around the RBC. Integration testing evaluates how different systems interact to ensure that data flows correctly between their interfaces. Subsystem testing is similar to integration testing but has a broader scope, examining how the complete software fulfils both its functional and non-functional requirements [13]. In subsystem testing and integration testing, the RBC is simulated with inputs such as train messages, route data, simulated infrastructure events, and the outputs are checked against expected behaviour.

2.4 Eurobalises

Located along the track between the rails, the balise is a transmission device used in ERTMS. A passing train reads data from the balise, which sends information in the form of telegrams. In ETCS level 2, the balise is a passive component powered by the train antenna when the train passes above it. Fixed telegrams are used in level 2, while variable telegrams are used in level 1 [6].

Two types of telegrams exist:

- fixed telegrams, containing predefined static information,
- variable telegrams, updated dynamically via a lineside electronic unit.

Balises are installed in groups called balise groups. Each balise in the group transmits a telegram. The train combines these telegrams into one message.

Balise groups provide:

- position reference for the train,
- track and signalling information,
- direction information.

In level 1, balises transmit the movement authorities directly to the train. In level 2, they serve as position reference points. Since the radio network cannot determine the precise location of the train, the system relies on the train passing a balise to reset its positioning data. As the train travels further away from a balise, the positioning uncertainty increases. A benefit of level 2 is that new movement authorities can be updated instantly via GSM-R at any time, meaning the train does not have to wait until it passes a physical balise on the track [8].

2.5 Radio block centre

The RBC is a centralised safety unit that receives train position information via radio and sends movement authorities to trains [14]. The RBC processes information from trains and trackside systems (such as interlocking) and generates movement authorities and other information based on this input [8].

Communication between the RBC and trains is bidirectional and uses GSM-R. The train reports its position and status to the RBC. The RBC responds with movement authorities and other constraints based on this information [8].

The interlocking is one of the components responsible for controlling signals and switches to prevent unsafe conditions [14]. The interlocking provides information such as track occupancy, route settings and switch positions. The RBC uses this information together with train reports to determine safe movement authorities.

The RBC interacts with trains, interlocking systems and operator control systems, as shown in Figure 2.1. The resulting response depends on the timing, order, and combination of these inputs. A problem can still appear even when each component works correctly on its own. For this reason, the RBC must be tested at several levels, from unit tests to full system tests.

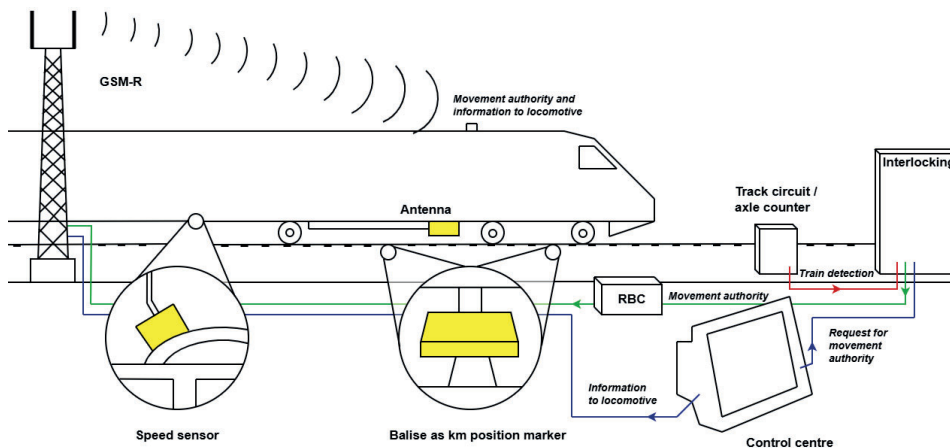


Figure 2.1: ETCS level 2 components.

Onvia Control L2-B is Alstom's RBC implementation for ETCS level 2. The RBC makes decisions based on train reports, infrastructure states, and site-specific configuration data. The behaviour of the RBC depends on both the executable logic and its input data.

2.6 Test environment

The RBC is tested at subsystem test level by running scripts written in TclExpect. TclExpect is an extension of Tcl [15] and is used for automated testing. These scripts interact with YATE to verify the system logic. YATE is a command-line tool used to run tests on the RBC software. The RBC software is isolated by simulating the surrounding components (Figure 2.2). The test scripts communicate with YATE, which communicates via a Test Control Hub to interact with three simulators.

- Train simulator that simulates train movement.
- Interlocking simulator that simulates interlocking switches.
- Centralised Traffic Control simulator that simulates operator commands from Traffic Management System.

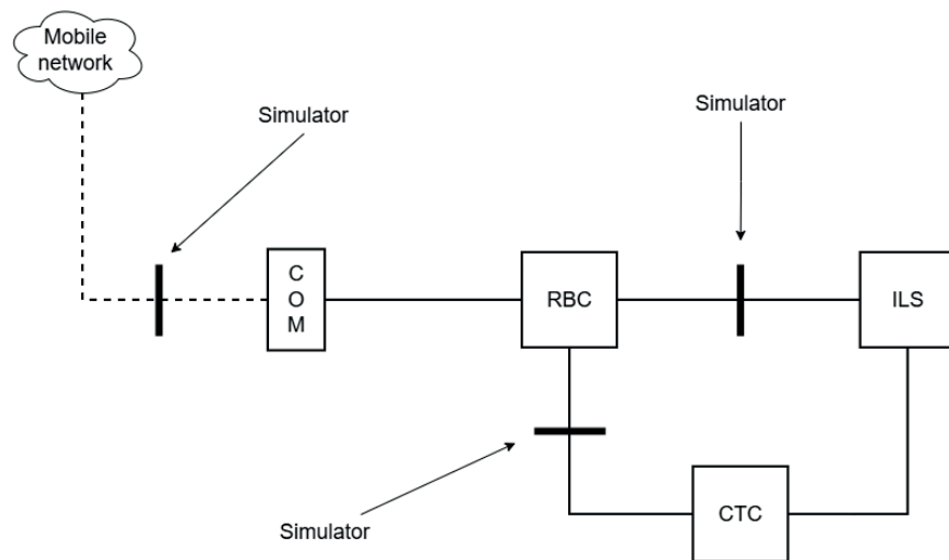


Figure 2.2: RBC isolated by using simulators.

The RBC is not tested with a full real railway installation. It is tested in a simulated environment, as shown in Figure 2.2. The surrounding systems are replaced by simulators. This makes it possible to send controlled events, observe the response and repeat the same scenarios many times.

Methodology

This chapter describes how information was gathered, how test data was processed and how the three methods were developed. The chapter also explains the main methodological limitations.

3.1 Work phases

The thesis work was carried out in four overlapping phases. Notes, scripts and analysis were updated throughout the project. The phases show how the work moved from learning the test environment, to changing train length, to generating test scripts, and finally to checking boundary values.

Phase 1: Getting started and changing train lengths

The first phase focused on learning about railway signalling and implementing the train-length variation used for RQ1.

- Gathered initial knowledge through meetings and training.
- Completed courses on railway signalling and ERTMS using Alstom University.
- Added a randomisation function to the train setup files.
- Ran the modified test scripts and collected the logs for minimum and maximum train length.
- Created log cleaning and plotting scripts to turn raw logs into CSV data and initial charts.

Phase 2: Generating automated test scripts

The second phase developed the method for generating new test scripts from route data.

- Studied the existing scripts to understand how train locations and train variables were represented.
- Wrote a test script generator in Python that parsed route data and used a template file to create new test scripts.
- Executed the generated test scripts, collected the results and fed them into the CSV and chart workflow.

Phase 3: Finding boundary values

The third phase required the most time and effort. It focused on processing existing test files to extract documented boundary values.

- Spent most of the development time building the boundary-value analysis script in Python.
- Built the script using test-driven development. The code was first tested against a single file, then updated to handle two files and finally scaled up to process all files.
- Used the script to extract boundary values from struct definitions and scan test files to check whether those values appeared with the corresponding parameter names.
- Ran the extracted boundary datasets through the data processing workflow to see which limits were missing from the current test coverage.

Phase 4: Final analysis and source review

The final phase refined the analysis scripts, reviewed the quality of the data sources, and completed the thesis text.

- Refined the plotting scripts.
- Evaluated the reliability and limitations of all documents.
- Revisited early Markdown notes and Alstom University modules during the final writing.

3.2 Information gathering

At the start of the thesis, information was gathered through meetings, internal onboarding material and example test scripts. This was necessary to understand how the RBC interacts with trains, routes, balises and surrounding systems before working with the existing tests.

The next step was to understand how the automated tests worked in practice. This included learning how TclExpect scripts were structured, how the simulators were started, how train messages were sent, received, and how results were stored in log files. This knowledge formed the basis for all later work in the thesis.

3.2.1 Informal presentations

Alstom hosted one-on-one sessions. During these sessions the supervisor from Alstom explained topics by drawing on a whiteboard. Notes from these sessions were saved.

3.2.2 Alstom University

Alstom has an internal learning platform called Alstom University [16]. This platform hosts educational material such as videos, webinars and quizzes. The first two weeks were spent learning by watching videos, taking notes and answering quizzes. Some quizzes were required to gain access to internal tools.

The following courses were completed.

- Git Basics training.
- Railway Signalling Overview.
- ERTMS-ETCS Introduction (Basic).
- Alstom Railway Safety Induction.

3.2.3 Markdown notes

A Markdown file was created each day. In this file, all information gathered during the day was saved. For example, notes from one-on-one sessions were saved as Markdown and referenced regularly during the research.

3.3 Document studies

Documents were an important source in the study.

- Standards and requirement documents were used to understand message and packet meanings in ERTMS communication.
- Internal test documentation and example scripts were used to understand the existing testing setup.
- Site data and route-related files were used for test generation.
- Source code comments, documentation and parameter descriptions were used for the boundary-value research.

These documents supported the work throughout the thesis. When a packet field, a route object, or a parameter range was unclear, the documentation was used to interpret it.

3.3.1 ERTMS specifications

In RQ1, the specifications were used to interpret the train length variable and its components. In RQ2, SUBSET-026 Chapter 8 was used to interpret message and packet numbers expected by the test scripts. For example, some generated tests failed due to expecting `nid_message = 33` and `nid_packet = 15` and not receiving them. The specification was used to interpret these values. In RQ3, minimum and maximum parameter values were obtained from internal Alstom documentation, which is based on SUBSET-026 Chapter 7.

3.4 Test simulation

Before modifying or generating test scripts, the RBC testing process was analysed. In the existing environment, the RBC was tested by sending controlled events from simulators and verifying that the returned messages matched expected behaviour. Each test run started by preparing the test environment, selecting site data and initialising one or more simulated trains. The trains then connected to the RBC and established communication sessions. The initialisation sequence is shown in Figure 3.1.

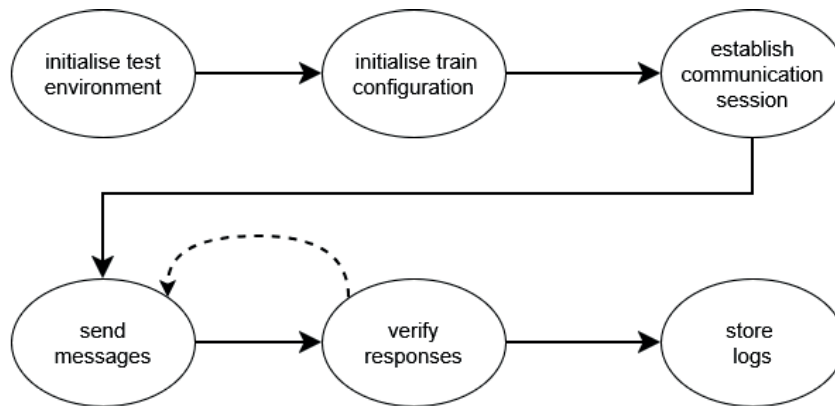


Figure 3.1: Test script execution flow.

After initialisation, the test script sent a predefined sequence of train messages to the RBC and waited for specific responses. Depending on the test scenario, the responses included system version information, acknowledgements, and movement authority messages. The script verified both message order and packet values. In scripts with conditional steps, the verified response could lead to further messages. If all expected conditions were satisfied, the script reported pass. If a message or value did not match the expected result, the script reported fail or unresolved.

This execution structure formed the basis for RQ1 and RQ2. RQ1 modified a train parameter by updating an initialisation configuration file used by all tests. RQ3 used existing test script files to extract and compare parameter values, rather than relying on test execution logs.

3.5 Data processing

After each test run, the raw logs were collected. The logs were too large to inspect manually. They contained the test result, but also a large amount of other information that was not needed for the thesis. A Python script (Appendix A.3.1) was used to extract test outcome, failure step, variables, function calls and error messages. The data processing workflow (Appendix A.3) is illustrated in Figure 3.2.

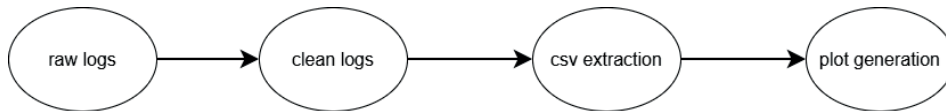


Figure 3.2: Data processing from raw logs to plotted data.

When redundant information had been removed, the cleaned logs were converted into Comma Separated Values (CSV) format. This made it possible to compare runs, count failures, group error categories and find parameters that failed often. The CSV files were used as input for plots and summary calculations.

This Python workflow turned a large amount of raw logs into smaller data files that could be counted, grouped and plotted. All analyses in RQ1 and RQ2 were based on the processed CSV data rather than on raw log data.

3.5.1 Extracting relevant information from log files

Each log file is created when a test run starts and is updated throughout the execution. It records the test configuration, start time, runtime events, and completion time. A Python script was used to extract structured information from the log files. The script extracts the test outcome, which is reported as pass, unresolved, or fail. The script also extracts associated parameters, the log entries that triggered the outcome, and the corresponding error descriptions. The log extraction script is provided in Appendix A.3.1.

The extracted information was then organised into the fields shown in Table 3.1. These fields include the error category, the parameter linked to the failure, and the expected and actual values.

3.5.2 Plotting data from log files

A Python script was used to generate plots from the CSV data using Matplotlib [17]. The script produced visualisations of test outcomes, error categories and parameter-related failures. This script was used for all research questions, with case-specific logic to handle differences in data structure. The plotting script is provided in Appendix A.3.3.

Table 3.1: Extracted CSV fields and descriptions.

Field	Description
test_name	Test script traceability
status	Test outcome
error_category	Type of failure
failed_step	Execution step of failure
parameter	Associated failing parameter
expected_value	Scripted target value
actual_value	Observed run value
log_file	Test file traceability

3.6 Train length variations

Alstom’s simulators support all variables defined in SUBSET-026 Chapter 7 [8]. The train-length experiments used the same basic setup, with specific test parameters outlined in Table 3.2.

- An internal configuration file set predefined default values.
- The configuration file is not included because it is confidential.
- The default value of `l_train` was changed for all tests.
- All other variables stayed unchanged.

Table 3.2: Experiment configurations for changing `l_train`.

Run type	<code>l_train</code>	Type	Purpose
Minimum	10	Static	Evaluate low boundary
Maximum	1000	Static	Evaluate high boundary
Randomised	10–1000	Random	Evaluate varying values

3.6.1 Minimum value for train length

The first RQ1 experiment changed the default train length to a low boundary value. In the train setup used by the existing test suite, `l_train` was normally initialised to 70. In this experiment, that value was changed to 10, while the test scripts were otherwise left unchanged. The purpose was to test the lower boundary before randomisation.

After introducing the change, the test suite was executed and the resulting logs were collected for processing. The workflow shown earlier in Figure 3.2 was used to process the results so they could be compared with other train length configurations.

3.6.2 Maximum value for train length

The ERTMS specification defines a maximum train length of 4095 m in SUBSET-026 Chapter 7 [8]. This value was not used because it is much higher than real train lengths used in Sweden.

- Most trains are limited to 750 m.
- Some cases allow up to 900 m.
- Planned infrastructure supports up to 1050 m [18], [19].

The selected value of 1000 m is within the realistic range described above. This value was used to test the upper boundary before randomisation. In this run, the default value of `l_train` was changed from 70 to 1000. As in the minimum value experiment, the rest of the test suite was left unchanged.

The tests were executed with the modified configuration file, and the logs were collected using the same steps as in the minimum value run. Using the same execution and processing steps for both cases allowed the results to be compared directly.

3.6.3 Randomised value for train length

The third RQ1 experiment used a different `l_train` value for each run. The value was randomised between 10 and 1000. The purpose was to check whether the test suite could handle more than one fixed train length.

Instead of using a fixed value for the `l_train` variable during initialisation, a Tcl function was created to randomise the value in the range 10 to 1000. During each setup, the function was called once and the returned value was assigned to `l_train`. This means that each test script used one random train length. The train length was not changed while the script was running.

```
# random integer number in range [min, max]
proc RandomIntegerInRange {min max} {
    return [expr {int(rand()*($max-$min+1)+$min)}]
}
```

Only `l_train` was changed in the configuration file. All test scripts were left unchanged. The logs were processed using the same workflow as in the minimum and maximum value runs.

3.7 Template-based test generation

RQ2 used a template-based approach to generate movement authority test scripts. Scripts were generated for all routes in the existing site data. Existing tests only covered selected positions on the track. Instead of writing new scripts for each route manually, one existing movement authority test was selected as a starting point

and all site-dependent elements were extracted so that values could be replaced automatically.

The test generation process in Figure 3.3 created route-specific scripts from site data, ran them and analysed the outcomes.

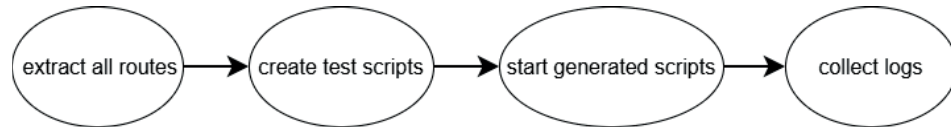


Figure 3.3: Test generation flow.

Template-based test generation was chosen as the existing test suite already contained validated logic for train setup, movement authority request, and message verification.

3.7.1 Template test script

An existing movement authority test script was first simplified to focus only on movement authority generation and route locking. The simplified script used one route instead of three, which made local runs faster during development. The simplified movement authority script was used as a template for generation. The template kept the main movement authority test steps, including route setup, train initialisation, startup, movement authority request and message verification.

The original test script used site-specific values such as the route start and stop signals, the Last Relevant Balise Group (LRBG) and other route-dependent identifiers. These values were replaced with placeholders to enable reuse of the same test logic across multiple routes (Table 3.3).

Table 3.3: Template parametrisation examples.

Original value	Placeholder
HST352@SME HST354@SME	@ROUTE_START_STOP_SIGNAL@
MA_for_FS_OS_LS.exp	@TEST_FILE_NAME@

During test generation, the route start and stop signals were populated from route data, while file names were generated by combining the selected start and stop signals. In this way, the test script generator (Appendix A.1) only modified site-specific input data while preserving the simplified test structure.

3.7.2 Fetching route data

Route data was extracted from configuration files in the test environment. These files listed the routes that could be used for test generation. A shortened excerpt

of the route data format is shown below.

```
set RouteData(SA,ILOCK2,HST122@HOG,HST22@HOG) 1
set RouteData(GA,ILOCK2,HSTL1@ARA) 0
```

The first row contains a route with start signal HST122@HOG and stop signal HST22@HOG. The second row contains only one signal. Only rows with both a start and stop signal were extracted for test generation.

The generator script used these signal pairs as input and generated one test script per route. This allowed the method to scale with the site data. When new routes were added to the configuration file, test scripts for those routes could be generated by rerunning the script.

3.7.3 Finding nearest balise

The generated test required identification of the nearest balise, defined as the first balise encountered when traversing the route from the start signal to the stop signal. The output included three values.

- The balise identifier.
- Its distance from the start signal.
- Its direction.

A route is defined by a start signal and a stop signal. The route is traversed from the start signal to the stop signal. Multiple object types may exist along the route, including balises, level crossings and other infrastructure elements. The traversal iterates over all objects along the route. The logic was adapted in two steps.

- 1) Return only objects of type balise.
- 2) Extract the identifier and position of each balise.

The traversal keeps the route order, which allows the first balise along the route to be selected. Multiple balises may exist along the route. The selection depended on route direction, switch configuration and RBC border constraints. From the filtered balises, the first entry in traversal order was selected, provided it matched the direction of travel and supported route locking.

The route (HST455@SME, HST453@SME) represents movement from HST455@SME to HST453@SME. In Figure 3.4, this corresponds to train movement from right to left.



Figure 3.4: Route from signal HST455@SME to HST453@SME.

For this route, the first selected balise is 488@SME, as shown in Figure 3.5.



Figure 3.5: First relevant balise identified along the route.

3.7.4 Test script generation

After the required route and balise data were extracted, the generator script combined these values with the template to generate route-specific Tcl test scripts. The output was a set of executable movement authority test scripts, each associated with a specific route in the site data. The generated scripts were written directly to the correct location in the test suite and executed both locally and in Continuous Integration (CI) through Jenkins [20].

3.8 Boundary value coverage

RQ3 examined whether documented boundary values for site-data parameters were represented in the existing automated tests. The analysis focused on minimum and maximum values documented in site-data struct definitions. The method was designed as a repeatable analysis workflow. No tests were executed for this part of the thesis.

The workflow consisted of three main steps.

- 1) Extract documented minimum and maximum values from site-data header files.
- 2) Extract explicit parameter values from existing automated test scripts.
- 3) Compare the test values with the documented minimum and maximum values.

The full workflow was executed by `main.py` (Appendix A.2.3). The script orchestrated file selection, range extraction, test-value extraction and boundary comparison.

3.8.1 Method scope

Boundary value coverage was defined as the presence of documented minimum and maximum parameter values in automated test scripts. A parameter was considered to cover a boundary if the documented boundary value was found together with the corresponding parameter name in a test script, for example as `l_train 10` or `l_train 1000`.

The analysis was limited to documented site-data parameters. Only documented minimum and maximum values were used. The analysis did not evaluate whether the tested system accepted, rejected, or handled the boundary value correctly. The method produced observations about test input coverage, not behavioural correctness.

This scope was chosen to keep the method repeatable and traceable. Each reported coverage result could be traced to a documented range in a header file and to an explicit value found in a test script.

3.8.2 Input sources

The method used two input sources.

- 1) Site-data header files with struct definitions, parameter ranges and documentation tags.
- 2) Automated test scripts from the production environment.

The analysis compared parameter ranges from the struct definitions with values found in the test scripts. Documented ranges served as reference boundaries, while script values served as observed values. Runtime data, execution logs and system responses were excluded.

3.8.3 Extraction of documented parameter ranges

Documented boundary values were extracted from site-data struct definitions. The script `parse_struct.py` (Appendix A.2.1) scanned header files for `typedef struct` declarations with Doxygen-style `\range` annotations. These annotations described the allowed values for each parameter.

A simplified example of a supported documented range is shown below.

```
typedef struct {  
    /** \brief Banana timing value.  
     * \range 30 - 300  
     * \default 30  
     * \tool_source User */  
    uint32_t espresso;  
} Banana;
```

From this example, the script extracted `Banana` as the struct name, `espresso` as the parameter name, 30 as the minimum value, 300 as the maximum value, and `numeric` as the range type.

The extraction supported two range types.

- 1) Numeric ranges, such as 0-100.
- 2) Boolean ranges, where the boundary values were `FALSE` and `TRUE`.

Other range formats were excluded from the proof of concept. Parameters without explicit `\range` documentation were excluded from the analysis, even if they appeared in automated tests. For each supported range, the script extracted five fields.

- 1) Struct name.
- 2) Parameter name.
- 3) Minimum value.
- 4) Maximum value.
- 5) Range type.

The extraction process ensured that every analysed boundary value came from documented parameter information. Using the same header files and parsing rules made the analysis reproducible.

3.8.4 Selection of user-configurable parameters

The analysis was limited to parameters marked with the documentation tag `\tool_source User`. During parsing, the presence of this tag was stored with each parameter as the boolean field `source_user`.

Only parameters where the `source_user` field was set to true were included in the boundary-value comparison. Parameters without the tag were excluded, even if they had documented ranges or appeared in automated test scripts.

A parameter could have a documented range and still be excluded if it was not marked as user-configurable. The example below has a valid boolean range, but no `\tool_source User` tag.

```
typedef struct {  
    /** \brief Latte flag.  
     * \range Boolean  
     * \default FALSE */  
    bool_t latte;  
} Pear;
```

In this case, `latte` was excluded from the boundary-value comparison because the parameter was not marked with `\tool_source User`.

3.8.5 Extraction of test values

Observed parameter values were extracted from existing `.exp` test scripts. The script `find_tests.py` (Appendix A.2.2) searched the scripts for assignments involving the selected parameters and collected explicit values assigned to those parameters.

3.8.6 Boundary comparison

Boundary value coverage was determined by comparing documented boundary values with observed test values. For each parameter, the documented minimum and maximum values were compared with the values extracted from the test scripts. Each boundary was evaluated separately. As shown in Table 3.4, a parameter could be classified into one of four categories.

Table 3.4: Boundary coverage categories used for RQ3.

Category	Meaning
No boundary	Neither minimum nor maximum found
Minimum only	Only the minimum value found
Maximum only	Only the maximum value found
Both boundaries	Minimum and maximum values found

This classification distinguished parameters with no detected boundary values from parameters where one or both documented boundaries were found. The comparison was implemented in `find_tests.py` (Appendix A.2.2).

3.8.7 Implementation via test-driven development

Unit tests were used to check the extraction, file-selection, comparison and workflow logic. The tests specified expected behaviour for struct parsing, range extraction, file discovery, parameter search, value extraction and execution of the complete RQ3 workflow.

The unit tests are provided in the appendix.

- Struct parser tests: Appendix A.2.4.
- Test script finder tests: Appendix A.2.5.
- Boundary-value workflow tests: Appendix A.2.6.

This testing approach reduced the risk of errors in the analysis scripts. The tests helped verify that the scripts selected the intended files, extracted documented ranges correctly and matched boundary values against the corresponding parameters in the test scripts.

3.9 Large language models

The author takes full responsibility for all research results, analysis and conclusions. Large language models were used only as support tools. They helped with writing, syntax explanations and variable naming. They were not used as technical sources.

Table 3.5: Use of large language models during this thesis.

Purpose	Tool	Phase
Initial approval draft	Gemini	Pre-approval
Explaining syntax	M365 Copilot	Research
Variable naming	M365 Copilot	Research
Text revision	M365 Copilot	Writing

To ensure transparency regarding the use of artificial intelligence tools, the large language models used during this research are summarised in Table 3.5.

No large language model was used as a technical source or as a replacement for standards, internal documentation, test results, or independent reasoning.

3.9.1 Initial approval draft

Gemini [21] was used to assist in drafting an initial description of the thesis. The initial description was provided to Alstom prior to formal approval. It was mainly used to explain previously unknown train-related terms.

3.9.2 Explaining syntax

M365 Copilot [22] was used to explain Tcl syntax in existing automated test scripts. This support improved understanding of run flow, parameter usage and script structure in the existing environment. The model supported RQ2 by assisting in the creation of `GetFirstObjectOfType` from the existing function `PrintRouteObjects`.

3.9.3 Variable naming

M365 Copilot was used to suggest more descriptive variable names during Python development. The purpose was to improve code readability without changing functionality or program logic.

In most cases, variable names were defined independently. When an appropriate and descriptive name was difficult to determine, M365 Copilot suggested alternatives. Only names that accurately reflected the role and content of the variable were adopted.

Before: `blocks`

After: `struct_block_list`

This clarified that the list contained declared `struct` instances, which was more specific than an unspecified collection of blocks.

3.9.4 Text revision

The starting point was always existing text. Large language models never wrote text independently. They were used only for minor language improvements and did not introduce new ideas, arguments, or interpretations.

M365 Copilot was used during the writing phase to clarify language, perform sentence-level revisions, improve sentence structure and refine wording for precision. It was not used to generate scientific arguments, interpret results, or draw conclusions.

An example of this type of revision is shown below.

Before: It also made the analysis reproducible because the same header files and parsing rules could be applied again.

After: Using the same header files and parsing rules made the analysis reproducible.

3.10 Evaluation of sources

Formal standards and specification documents [8], [9], [14], [23] were the most reliable sources. They were used for message definitions, parameter meanings, valid ranges, and system terminology. These documents are reliable because they are maintained by official organisations and used across the railway industry. They do not describe Alstom’s implementation in detail, since each supplier builds its own system based on the specifications.

The railway software standard EN 50716 [10], [11] was used to define verification and testing in the context of railway software development. The standard was treated as reliable because it is a formal European standard for railway applications. It was used only for general definitions.

Reports and articles [18], [19] were used to provide contextual information about railway systems and train configurations. These sources are written for broader audiences and were used to support background understanding. They were not used as primary technical evidence for the implemented methods.

Related work [2], [3], [4], [5] was used to provide background and show that similar topics have been studied before. These sources were not used as evidence for the tested system or as a basis for the implemented methods.

The textbooks [12], [13] were used as academic sources for software testing concepts. They supported definitions and explanations of testing levels and methodology. These sources were treated as reliable because they are established textbooks, but they were used only for general concepts rather than system-specific claims.

Technical documentation for Tcl, Docker, Jenkins, and Matplotlib [1], [15], [17], [20] was used to understand the tools used in the thesis. Tcl supported the test scripts, Docker supported the container environment, Jenkins supported final test execution, and Matplotlib was used for plotting. These sources explained tool

usage, but they were not evidence for the correctness of the railway signalling behaviour.

Internal training material from Alstom [6], [7], [16] was used to understand system concepts, terminology, and railway signalling background. The material reflects established knowledge within the company, but it does not always reference external sources. The platform was also difficult to navigate, which made some information harder to find.

Internal documentation, test scripts, and site-data files were used to understand the test environment, test structure, parameter definitions, and system configuration. Test scripts and site-data files were treated as reliable because they came from the validated Alstom test setup. Developer comments and wiki pages were used only as supporting material because they may contain missing or outdated information.

Execution logs from local runs and Jenkins builds were used as evidence for RQ1 and RQ2. They came from the validated test environment, but they required filtering and domain knowledge to interpret correctly.

Large language models [21], [22] were not treated as sources because they were used only as support tools during writing and development.

Informal sources, including discussions and observed tool behaviour, were used to understand workflows and implementation details. These sources were not reproducible and were not used as independent evidence. They were used only to guide interpretation and implementation decisions.

Analysis

This chapter analyses test runs, generated test scripts, parameter usage and problems identified during the study.

4.1 Test log files

In RQ1 and RQ2, the tests were executed with the YATE test engine. The test engine reported each test outcome as pass, fail, or unresolved.

- A pass outcome means that the test completed successfully.
- A fail outcome means that the test executed but did not satisfy its expected result.
- An unresolved outcome means that the test did not complete in a way that allowed the expected behaviour to be evaluated.

In this thesis, unresolved outcomes were treated as test execution problems rather than as failed system behaviour. All fail outcomes were analysed to identify recurring failure mechanisms. The unresolved outcomes were included in the outcome counts, but they were not used to derive failure mechanisms.

4.2 Train length variations

The production baseline used `l_train` = 70 m. In that baseline, the same test set was executed with no modifications to the configuration file. All 2738 tests passed.

Three train length configurations were analysed. The minimum configuration used `l_train` = 10 m. The maximum configuration used `l_train` = 1000 m. The randomised configuration used one value between 10 m and 1000 m for each test script. Table 4.1 summarises the outcomes.

Table 4.1: Test outcomes for different train length configurations.

Configuration	Length (m)	Fail	Unresolved
Minimum	10	103	4
Maximum	1000	149	8
Randomised	10–1000	120	5

Changing `l_train` introduced fail and unresolved outcomes in the same test set that passed with the baseline configuration. The maximum train length produced the most failed tests. The randomised configuration produced more failed tests than the minimum configuration, but fewer than the maximum.

4.2.1 Minimum train length

The minimum train length configuration produced 103 failed tests and 4 unresolved tests (Table 4.1). A shorter train length was sufficient to expose fixed assumptions in the existing test suite.

Several failures with the shorter train were linked to fixed expected values in the test scripts. The clearest pattern appeared when the difference between the expected and actual values was 60 m. This is the difference between the fixed train length of 70 m and the minimum train length of 10 m. Table 4.2 shows examples where the actual value was 60 m lower than the expected value.

Table 4.2: Value mismatches for minimum train length.

Parameter	Expected	Actual	Difference
<code>l_endsection</code>	3637	3577	-60
<code>d_emergencystop</code>	606	546	-60
<code>d_ref</code>	375	315	-60
<code>d_tafdisplay</code>	424	364	-60

In all four examples, the actual value was 60 m lower than the expected value. These mismatches indicate that the expected values were based on the fixed baseline length. This does not mean that all these failures had the same cause. Some failures involved message timeouts or other waiting conditions. These failures indicate that shorter trains exposed assumptions about route geometry, train position, message timing, and expected packet content.

4.2.2 Maximum train length

The maximum train length configuration produced 149 failed tests and 8 unresolved tests (Table 4.1). This was the highest number of failed tests among the three configurations.

A similar pattern of expected values emerged here, but with inverted results. The most apparent mismatches occurred when the expected and actual values differed by 930 m, the exact difference between the fixed train length of 70 m and the maximum train length of 1000 m. As shown in Table 4.3, the actual values in these cases were consistently 930 m larger than expected, reinforcing the observation that the test scripts assumed a fixed 70 m train.

Table 4.3: Value mismatches for maximum train length.

Element	Expected	Actual	Difference
<code>l_endsection</code>	378	1308	930
<code>d_emergencystop</code>	606	1536	930
<code>d_ref</code>	274	1204	930

Similar to the minimum configuration, the increased length also triggered other issues. Timeouts, missing actual values and altered packet fields indicate that longer trains exposed wider operational assumptions regarding message timing and route geometry.

4.2.3 Randomised train length

The randomised train length configuration produced 120 failed tests and 5 unresolved tests (Table 4.1). Each test script received one random `l_train` value between 10 m and 1000 m. That value stayed fixed during the execution of the test script.

Unlike the minimum and maximum configurations, the randomised configuration varied the train length between test scripts instead of using one fixed value for all tests.

Some failures in the randomised configuration were linked to fixed expected values in the test scripts. The most apparent pattern appeared when the value difference matched the difference between the randomised train length and the fixed train length. Table 4.4 shows examples of this pattern.

Table 4.4: Value mismatches for randomised train length.

Element	Length (m)	Expected	Actual	Difference
<code>l_endsection</code>	105	2125	2160	35
<code>d_ref</code>	997	274	1201	927
<code>d_emergencystop</code>	447	606	983	377

In all three examples, the value difference matched the difference between the randomised train length and the fixed train length of 70 m. For example, the `D_REF`

row used a randomised train length of 997 m. The difference from the fixed was therefore 927 m, which matched the observed value difference. This indicates that the expected values were based on the fixed train length rather than on the train length used during the test script.

The randomised failures also included message timeouts. These failures occurred in test elements such as `waitTrainMessage`. In those cases, the test waited for a specific message or packet sequence. When the changed train length affected the progression of the scenario, the expected message did not appear at the point where the test expected it.

The randomised configuration showed that the failures were not only caused by the two boundary values. Failures also appeared for train lengths inside the selected interval. This indicates that parts of the existing test suite depended on the baseline 70 m train length, either through fixed expected values or through expected message timing and ordering.

4.2.4 Failure mechanisms

The analysed failures showed three recurring mechanisms.

- 1) A value mismatch in the expected movement authority data.
- 2) A timeout caused by fixed message-order expectations.
- 3) An unexpected message caused by timing constraints in the test script.

As illustrated in Figure 4.1 and Figure 4.2, `L_ENDSECTION` defines the total distance from the estimated rear end of the train to the upcoming signal representing the end of the Movement Authority (MA). `L_ENDSECTION` is the sum of the rear-end position uncertainty (`L_DOUBTOVER`), the train length (`L_TRAIN`), the distance to the next balise group (`D_LRBG`), and the remaining distance to the upcoming signal.

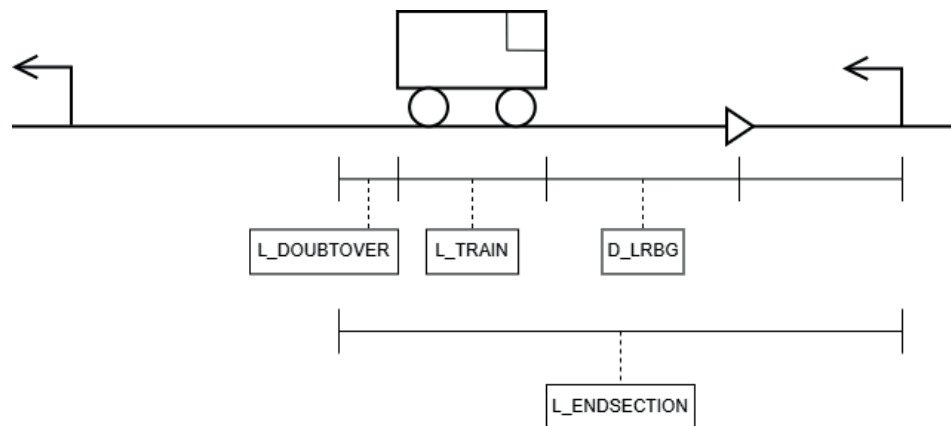


Figure 4.1: Effect of short train length on `L_endsection`.

When `l_train` changes, the total MA distance changes with it, directly affecting the calculated `l_endsection` value. Figure 4.1 demonstrates how a shorter train reduces the calculated section length, while Figure 4.2 shows the corresponding increase for a longer train configuration.

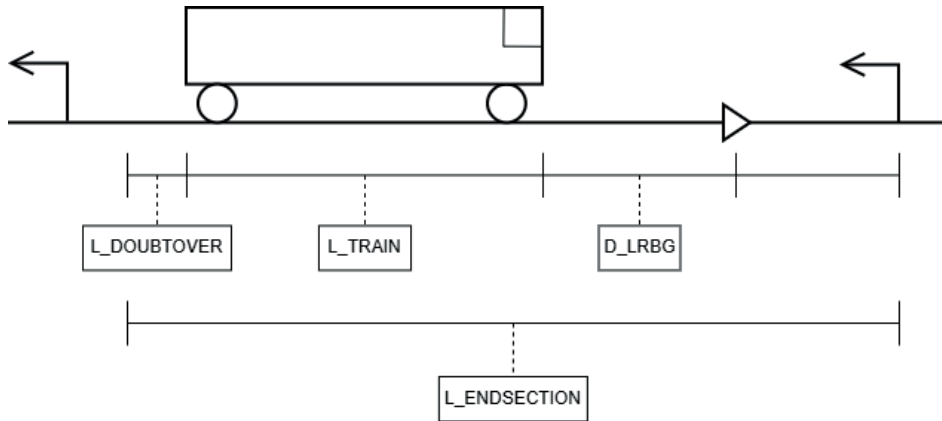


Figure 4.2: Effect of long train length on `l_endsection`.

These value mismatches explain some of the failed tests, as they expected a fixed `l_endsection` value but the actual value changed with the train length.

The second failure pattern appeared in some tests using `waitTrainMessage`. Instead of just checking if a message arrived, these tests waited for a specific packet at an exact point in the script. When the modified train length altered the scenario progression, it shifted the timing of later messages, pushing them outside the expected window and triggering a timeout.

A third mechanism affected some tests using `prepareWait`. These test scripts used preconfigured waiting patterns that assumed a strict message sequence. Because the changed train length shifted the timing and order of subsequent messages, the tests often encountered valid but unexpected messages, causing them to fail.

The failed test scripts relied on fixed values, messages and waiting patterns. Because these scripts were hardcoded around the original train length, they do not prove bugs in the RBC logic.

4.3 Template-based test generation

As described in Section 3.7, RQ2 used a template-based method to generate test scripts from site data. This section analyses the main implementation decisions, test runs, failure categories and remaining template limitations.

4.3.1 Generation approach

The method used template-based generation instead of generating complete test scripts from scratch. This was a suitable choice because the existing movement authority test already contained the required train setup, movement authority request and message verification sequence. This meant that the generator only had to insert route-specific values into a known test structure. Table 3.3 in Section 3.7.1 shows examples of placeholder replacements.

The main implementation decisions were:

- a template-based approach after supervisor guidance,
- Python because it is simple and powerful for file processing,
- Docker and Jenkins paths as the generated scripts had to run in Alstom's existing test environment.

These decisions kept the majority of the test logic in the template. This was useful because route-specific failures could be addressed by improving the template, rather than by changing the generator alone.

4.3.2 Evaluation environment

The final evaluation used test runs from Jenkins, a CI platform used by Alstom for automated test execution. Jenkins executed the generated scripts in the production test environment. The log files were processed to extract outcome data for the generated scripts. The logs were processed through the log-cleaning and CSV pipeline described in Section 3.5. The extracted CSV fields are listed in Table 3.1. The evaluation was limited to AdSim 2y, which was the simulator configuration used for the studied test run.

4.3.3 Progression of generated test runs

The generated tests were executed in three Jenkins runs. The progression is relevant because the template was changed between runs. The outcome counts for these runs are presented in the next chapter (Table 5.2).

- The first Jenkins run showed that the generated scripts could execute in the automated test environment.
- The second Jenkins run produced more failed outcomes. This suggests that a template change can improve one route while making other routes fail.
- The third Jenkins run had the highest number of successful outcomes and the lowest number of failed outcomes among the three runs.

The third Jenkins run was therefore selected as the frozen RQ2 outcome. Compared with the first Jenkins run, the third Jenkins run reduced the number of failed outcomes from 109 to 104. The progression supports one limited claim. The prototype can generate and execute a large set of route-specific movement authority

tests. It does not show that the template developed in this thesis can handle all routes.

4.3.4 Failure categories in the final run

The final Jenkins run had 104 failed generated tests and 0 unresolved tests. The failed tests were concentrated in two categories, message timeouts and missing balises. Table 4.5 shows the failure distribution for the final run.

Table 4.5: Failure categories in the final RQ2 Jenkins run.

Category	Element	Count
message_timeout	getAndAckMa	70
message_timeout	completeTrainStartup	18
missing_balise	GEO_BALISE_A	16

The table shows that most remaining failures were connected to message timing in the generated test flow. The largest group occurred in `getAndAckMa`. These failures indicate that the generated script did not receive the expected movement authority message in the expected step. A smaller timeout group occurred in `completeTrainStartup`. These failures indicate that the generated train setup did not reach the expected message state. The missing-balise failures had a different cause. In these cases, the generated route test could not find a usable `GEO_BALISE_A`. This points to a limitation in the generic route-to-balise selection logic. It does not, by itself, prove faults in the RBC logic.

4.3.5 Template limitations and route-specific cases

The final template included changes that solved some of the problems found during development.

- Route expansion was added before balise lookup. This reduced the number of failed generated tests from 109 to 104.
- Direction-dependent values were calculated from the selected route object. This reduced the need for separate templates for nominal and reverse directions.
- Missing `GEO_BALISE_A` cases were reported as `FAIL`. This represented the outcome better than leaving the tests as unresolved.

These changes improved the prototype, but 104 generated tests still failed. The remaining failures indicate that the template did not correctly handle all cases, such as switches, borders and other track-layout conditions. The generated tests handled many routes, but not all of them. Full pass coverage would require more template improvements, more time and specific expertise in the test environment.

4.4 Boundary value coverage

This section analyses the boundary value coverage output for RQ3 by explaining what the generated CSV file shows and what limits its interpretation.

4.4.1 Analysed parameter set

A parameter was included in the analysis only if it matched the thesis scope and could be compared with test-script values. This does not mean that other parameters were invalid in the system. It only means that they were outside the limits of this analysis.

- The parameter was marked with `\tool_source User`.
- The parameter came from a header file whose name started with `psr` or `oma`.
- The parameter had a documented minimum and maximum value.
- The documented values were numeric or boolean.

Table 4.6 displays the number of parameters included in the analysis for each struct. Only parameters that matched the limits above were included. The table does not count all parameters in each struct.

Table 4.6: Parameter counts after applying the analysis limits.

Struct	Count
LevelCrossingInputA	12
TrainLimitsA	8
CBR2DataA	4
TextMsgDataA	4
RbcHandoverA	4
OdometrySupervisionA	3
EntranceTimerA	2
GeoRbcBorderA	2
AxleLoadStepA	1
Packet41DataA	1
TsrLimitsA	1
TsrTraitsA	1

The table shows that the analysed parameter set was concentrated in a few structs. `LevelCrossingInputA` contributed the largest number of included parameters.

4.4.2 Output structure

While the extraction workflow is described in Section 3.8, this section focuses on how that output was interpreted. The CSV file contained one row for each analysed (`struct_name`, `parameter`) pair, storing key fields such as the documented minimum and maximum values, whether those values were found in the test scripts, whether the parameter name was unique, and how often the parameter name appeared. This distinction was important because a parameter could appear multiple times without its documented minimum or maximum value being found.

4.4.3 Meaning of covered boundaries

A boundary marked as covered means that the documented minimum or maximum value was found in at least one analysed test script. It does not, by itself, prove that the behaviour at that boundary was verified. This distinction matters when reading the result. For example, if a script sets `maximum_travelling_time` to 300, the method can report that the maximum value was found. It cannot show that the test checks the expected behaviour for that value.

The same caution applies when a boundary value was not found. This means that it was not detected in the analysed scripts. This does not prove that the value can never occur during execution. The value may be assigned through a variable, calculated at runtime, or passed through a helper function, which means that the analysis script may miss it.

4.4.4 Parameter usage and boundary coverage

The CSV file showed that some parameters appeared in many test scripts without their documented minimum or maximum value being found. Table 4.7 lists how many test scripts contained each parameter name and whether the documented minimum and maximum values were found.

Table 4.7: Examples of boundary values found in test scripts.

Parameter	Tests	Minimum	Maximum
<code>maximum_travelling_time</code>	63	No	Yes
<code>start_offset</code>	101	Yes	Yes
<code>minimum_time_before_passage</code>	120	No	No
<code>lx_speed</code>	140	No	Yes
<code>nLength</code>	221	Yes	No

These examples show that the number of tests containing a parameter was not always related to boundary coverage. The `minimum_time_before_passage` parameter appeared in 120 test scripts without either documented boundary value being found. The `nLength` parameter appeared in 221 test scripts, but only the

documented minimum value was found. This shows why boundary coverage had to be checked using both the parameter name and the boundary value.

4.4.5 Limits of the boundary value analysis

The boundary value analysis had three main limitations.

- The analysis searched the test scripts as text files. It did not execute the scripts. A missing boundary value means that the value was not found in the analysed scripts. It does not prove that the value is impossible to reach.
- The analysis required each parameter name to be unique in the selected scope. The test scripts usually contained parameter names, but not the struct names from the header files. Without this filter, 31 of 101 parameter rows had names that also appeared in other structs.
- Special and reserved values are relevant to boundary testing, but they were outside the scope of this work. Minimum and maximum values could be extracted reliably from the header files, whereas special and reserved values required parsing of SUBSET-026 Chapter 7 [8].

These limitations restrict what RQ3 can support. The analysis can show whether documented minimum and maximum values were found in the selected test scripts. It cannot prove that the tests checked all behaviour at those values.

4.5 Problems and solutions

This section describes problems encountered during the analysis and how they were handled. Table 4.8 gives an overview of the problems covered in this section.

Table 4.8: Problems covered in this section.

Problem	Research question
Processing large test outputs	RQ1, RQ2
Test assumptions and placement	RQ1, RQ2
Managing generated test scripts	RQ2
Coverage data extraction	RQ3

4.5.1 Processing large test outputs

Large test runs were needed to evaluate RQ1 and RQ2. The main problem was not running the tests, but handling the large result sets. Manual inspection of each failed test became impractical when the output grew. Log cleaning, normalisation and grouping were needed to make the test outcomes comparable.

4.5.2 Test assumptions and placement

Randomising `l_train` showed that some existing tests depended on the default train length of 70 m. In those tests, expected packet content, distances, or message sequences were based on the baseline train length rather than calculated from the randomised train length. The affected failures were treated as assumptions in the existing test scripts, not as errors in the randomisation method.

For each generated MA test, the train had to start at the correct place on the route. It also had to start in the correct direction and at a valid distance from the relevant signal or balise. In more complex routes, such as routes with many switches or end-of-line sections, the train start position needed more specific handling. This made automatic placement harder than generating a route and using the first available object.

A single template could still be used, but it needed route-specific handling to pass more test scripts. A route could be valid in the site data, but still need a specific start position in the test environment. The template was updated to handle some edge cases, including cases where a balise was located at the same position as an RBC border and the border had to be expanded. Other route-specific cases were left unresolved.

4.5.3 Managing generated test scripts

The generated scripts had to be managed without adding all generated files to version control. Only the script used to generate the test scripts was checked in. The generated test scripts were then created at runtime and executed by the test environment.

4.5.4 Coverage data extraction

The test scripts often included explicit parameter names, but almost never named the structs that contained them. This made it difficult to connect a value found in a test script to the correct site-data struct. The problem affected the boundary analysis as minimum and maximum values were collected from structs in site-data header files, while the test scripts usually only contained parameter names.

The analysis relied on the assumption that each included parameter name was unique in the selected scope. This assumption made the analysis possible, but it could assign a value to the wrong struct if the same parameter name was reused. The `\tool_source User` filter was used because Alstom requested this scope. Within this filtered scope, all included parameter names were unique.

Special and reserved values were not included in the main RQ3 result. The result only used documented minimum and maximum values from site-data header files. Special and reserved values were harder to use because they were defined in SUBSET-026, Chapter 7 [8], not in the same format as the analysed site-data structs.

An extraction of special and reserved values was attempted, but reliable extraction

was more difficult than parsing the site-data header files. Chapter 7 of SUBSET-026 is a 91-page PDF with many tables, where the meaning of a value often depends on surrounding table information [8]. A complete extraction would require more robust document parsing than was possible within the thesis time frame.

Special and reserved values were excluded from the main boundary value coverage result. The main result was limited to minimum and maximum values because these values were available in a form that could be parsed consistently from the analysed header files.

Results

This chapter reports how the existing test suite reacted when `l_train` was changed, the results from the generated MA tests, and how many documented boundary values were found in the analysed tests.

5.1 Train length variations

The results show that most tests continued to pass when `l_train` was changed from the 70 m baseline, but each changed configuration also produced failed and unresolved outcomes. In the production baseline, all 2738 tests passed. Table 5.1 provides an overview of the three changed configurations.

Table 5.1: Outcome counts for changed train length configurations.

Configuration	Length (m)	Passed	Failed	Unresolved
Minimum	10	2631	103	4
Maximum	1000	2581	149	8
Randomised	10–1000	2613	120	5

Table 5.1 shows that every changed train length produced non-passing outcomes. The maximum train length produced the highest number of failures, with 149 failed tests and 8 unresolved tests. The minimum train length produced the lowest number of failures, with 103 failed tests and 4 unresolved tests. The randomised configuration was between the two fixed configurations, with 120 failed tests and 5 unresolved tests.

5.1.1 Minimum train length

For the minimum train length configuration, 2631 tests passed, 103 failed and 4 were unresolved. This configuration used `l_train` = 10 m. The result shows that

reducing the train length was enough to change the outcome in a production test environment where all tests passed with the baseline value. Figure 5.1 presents the outcome overview and error categories for the minimum train length configuration.

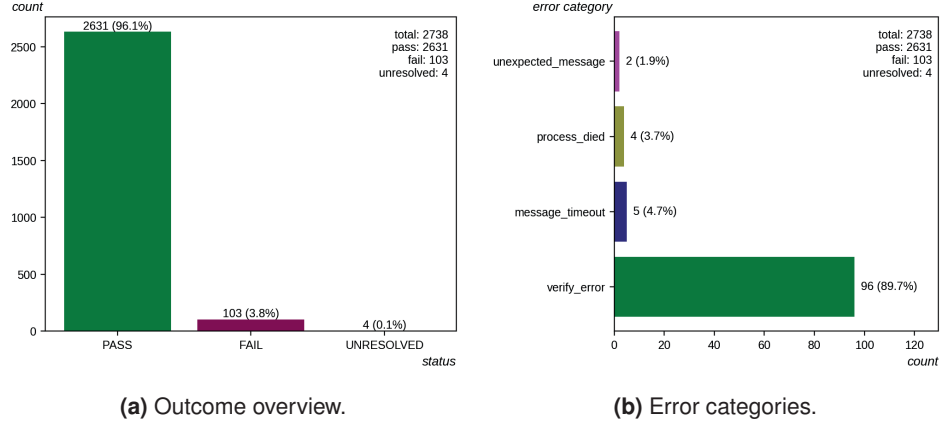


Figure 5.1: Outcome overview and error categories for minimum train length.

Figure 5.1 (a) shows that the majority of tests passed, while 103 failed and 4 were unresolved. In (b), the failed outcomes are grouped by error category. The largest category was `verify_error`, with 96 failures. This accounted for 89.7% of the non-passing outcomes in this run. The failed outcomes were also grouped by affected element. Figure 5.2 displays the affected elements and the same elements separated by error category.

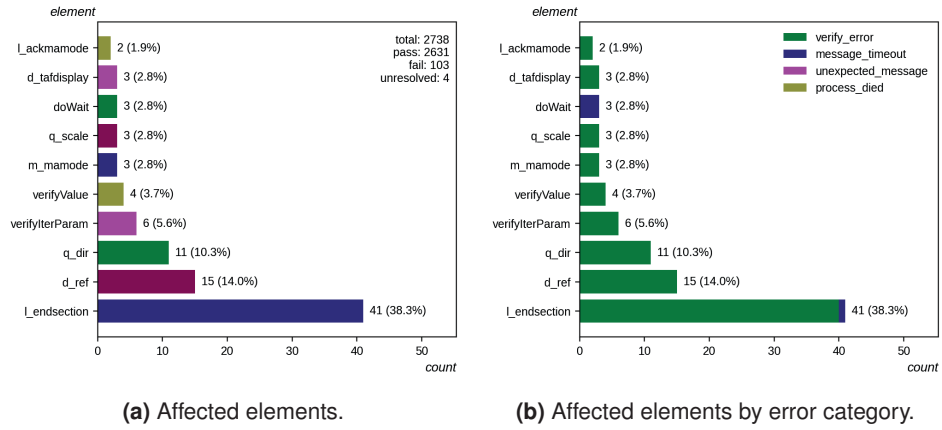


Figure 5.2: Affected elements and error categories for minimum train length.

Figure 5.2 (a) shows that the parameter `l_endsection` appeared most frequently in the failed outcomes. The next largest groups were `d_ref` with 15 occurrences

and `q_dir` with 11 occurrences. In (b), the same elements are separated by error category. The visible failures in this configuration were concentrated in a small number of elements. Most of these listed elements were connected to `verify_error`.

5.1.2 Maximum train length

The maximum train length configuration used `l_train = 1000 m`. This run produced the largest number of non-passing outcomes. In total, 2581 tests passed, 149 failed and 8 were unresolved. Figure 5.3 presents the test run outcomes and the error categories related to the failures.

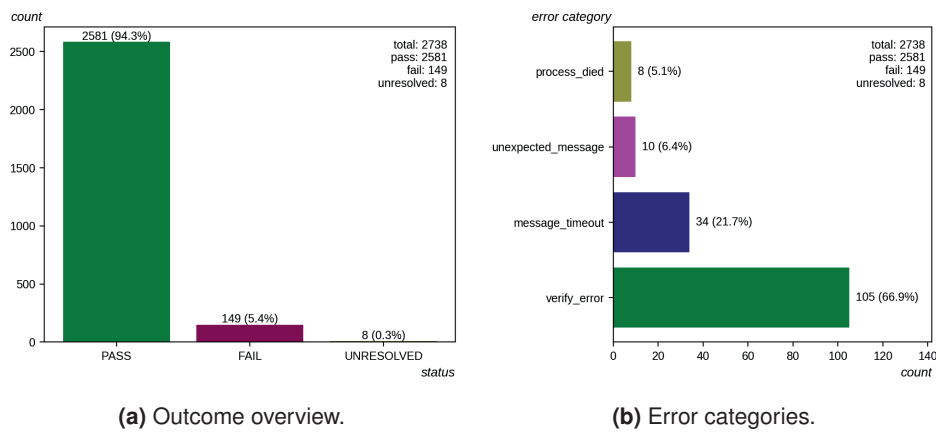


Figure 5.3: Outcome overview and error categories for maximum train length.

The category with the most failures was again `verify_error`, with 105 outcomes. Compared with the minimum train length run, the maximum train length run had more failures in every error category. The largest increase was `message_timeout`, which increased from 5 to 34 outcomes. Figure 5.4 presents the affected elements.

The element with the most failures was `l_endsection`, with 39 occurrences, followed by `d_ref` with 17 occurrences, `waitTrainMessage` with 16 occurrences and `doWait` with 15 occurrences. These elements show both value-related failures and waiting-related failures in this configuration.

5.1.3 Randomised train length

The randomised train length configuration used one `l_train` value between 10 m and 1000 m for each test script. This run produced 2613 passing tests, 120 failed and 5 unresolved. Figure 5.5 presents the outcome overview and error categories for the randomised run.

The randomised run produced fewer failures than the maximum train length run, but more than the minimum train length run. The error categories followed the same order as in the minimum and maximum train length runs. The largest

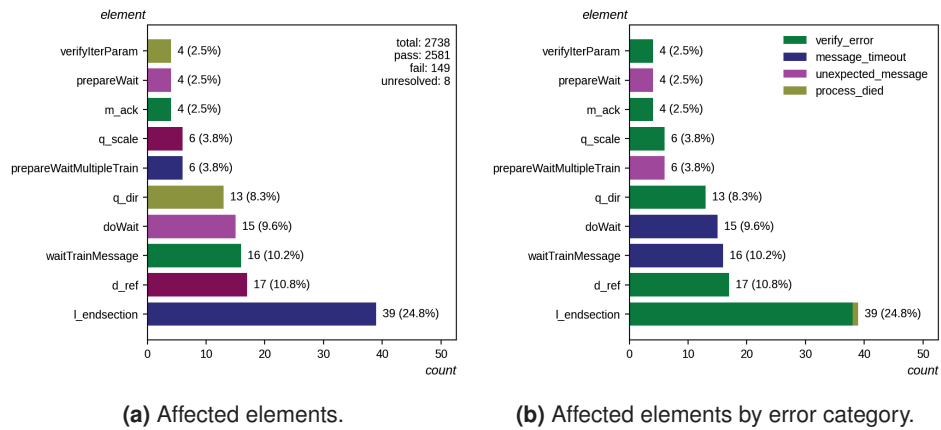


Figure 5.4: Affected elements and error categories for maximum train length.

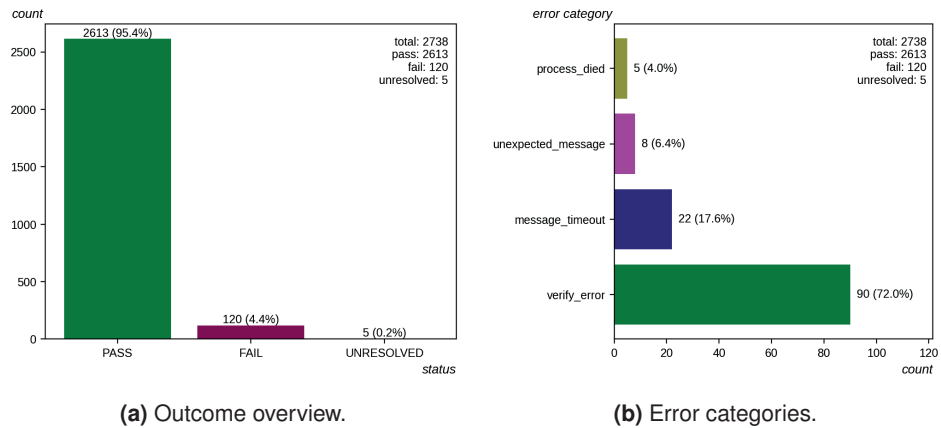


Figure 5.5: Outcome overview and error categories for randomised train length.

category was `verify_error`, with 90 outcomes, followed by `message_timeout`, with 22 outcomes. Figure 5.6 presents the affected elements.

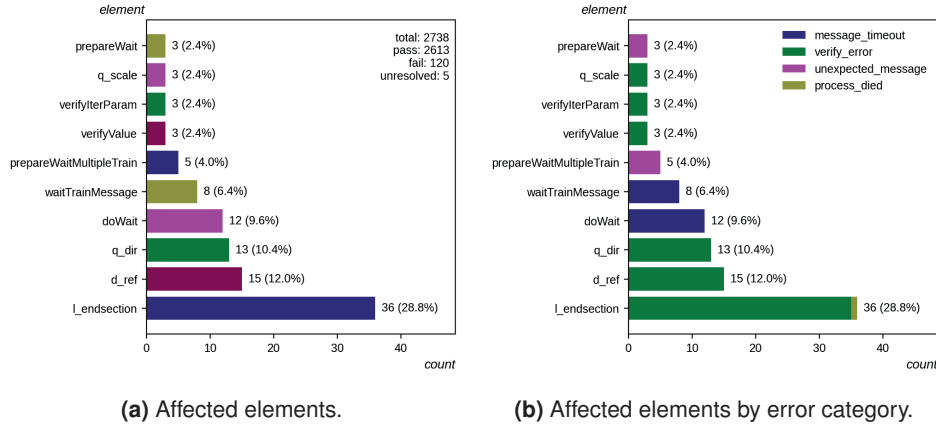


Figure 5.6: Affected elements and error categories for randomised train length.

The most frequent affected element was `l_endsection`, with 36 occurrences. The next largest groups were `d_ref` with 15 occurrences, `q_dir` with 13 occurrences and `doWait` with 12 occurrences. All three results show that the test suite reacted to all three changed configurations.

5.2 Template-based test generation

For RQ2, the main result was that the generator produced executable MA tests that could run in Jenkins. The generated scripts were executed in the same type of automated test environment as the existing production scripts.

The generated tests used one template and route-specific values from site data. This made it possible to create many route-specific tests without writing each test manually. The final outcome was based on the third Jenkins run, where 426 of 530 generated tests passed.

Not all generated tests passed, but the result shows that the template-based method worked for many routes. Even if the remaining tests failed, they were still useful because they showed where the template did not correctly handle all route-specific conditions.

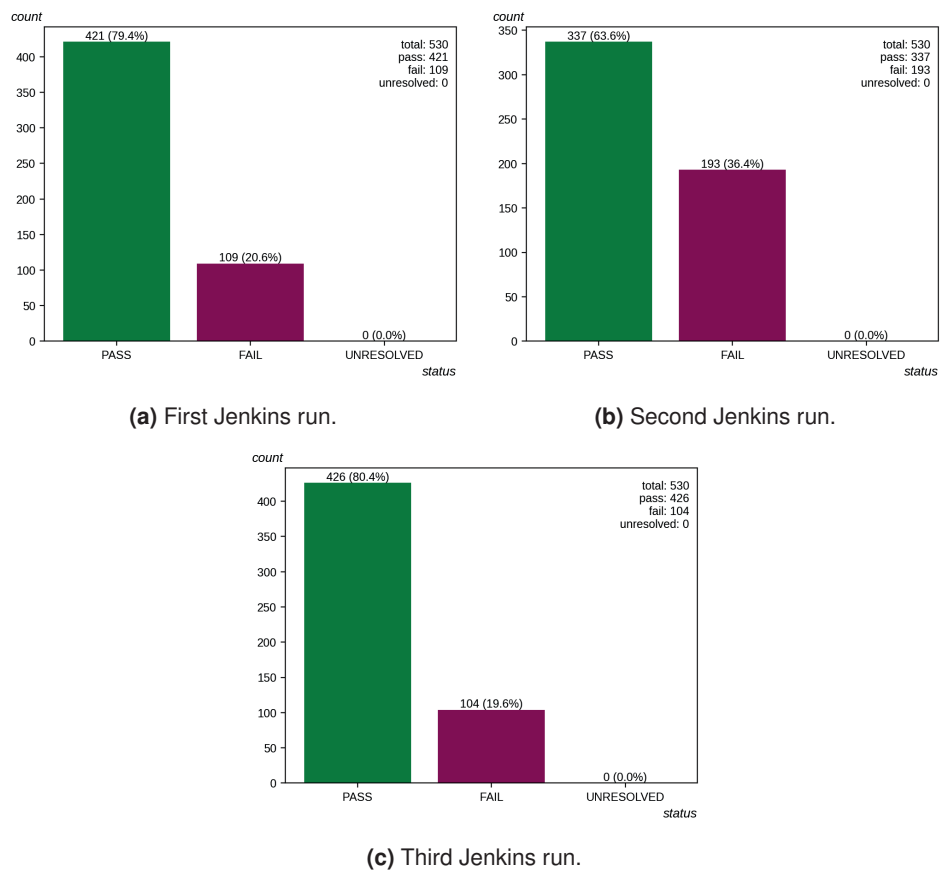
5.2.1 Generated test outcomes

The results from the three Jenkins runs, each consisting of 530 generated tests, are shown in Table 5.2.

Table 5.2: Progression of RQ2 Jenkins runs.

Run	Passed	Failed	Unresolved
Jenkins 1	421	109	0
Jenkins 2	337	193	0
Jenkins 3	426	104	0

In the first run, 421 tests passed and 109 failed. The second run had the weakest result, with 337 passed and 193 failed. The third run had the best result, with 426 passed and 104 failed. The same progression is shown visually in Figure 5.7.

**Figure 5.7:** Test outcomes for the Jenkins runs.

The third run was used as the final RQ2 result. The results show that the generator worked for most routes after route-specific additions had been made to the template. In the final run, 426 of 530 generated tests passed and 104 failed.

The 19.6% failed tests show that the generator did not handle all routes, but the passing majority shows that the approach worked for most routes. All generated tests reached execution, which means that they were not just files from a script. They became test scripts that could be run, counted and compared. The failures also helped identify routes that needed more analysis.

5.2.2 Failure categories in the final Jenkins run

A closer look at the generated tests shows that the failures were concentrated in two categories. Figure 5.8 shows the failures and their error categories for the selected third run.

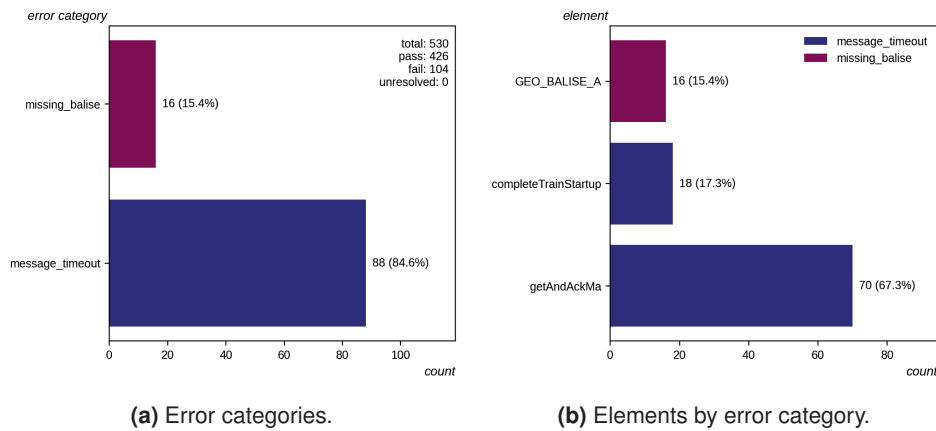


Figure 5.8: Failures and error categories for the third run.

Figure 5.8 (a) shows that there were only two failure categories. The largest category was `message_timeout`, with 88 occurrences. The remaining 16 failures were `missing_balise`. In (b), the same failures are grouped by affected element. `getAndAckMa` accounted for 70 failures. `completeTrainStartup` accounted for 18 failures. `GEO_BALISE_A` accounted for 16 failures. Most generated tests passed, and the failed tests were mainly connected to a small number of recurring failure groups.

5.3 Boundary value coverage

The remaining results focus on boundary values. The analysis script searched for documented minimum and maximum values in the test files. A boundary value reported as covered indicates that the value was identified in the data, but does not mean that the system behaviour at that value was verified.

5.3.1 User-source filtering

The final RQ3 result used the dataset with the user-source filter enabled, which limited the analysis to user-configurable parameters. Figure 5.9 shows that the

user-source filter reduced the analysed dataset from 101 parameters to 43. With the filter enabled, all included parameter names were unique. Without the filter, 31 parameter names were not unique.

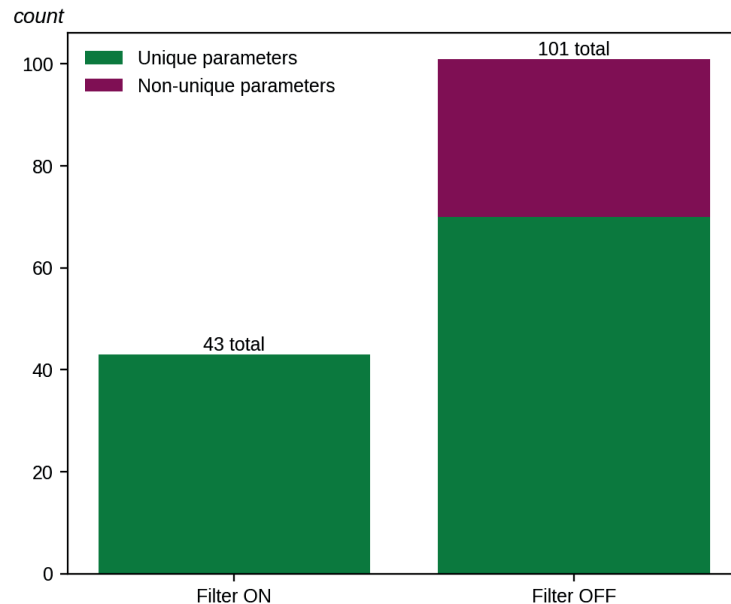


Figure 5.9: Effect of the user-source filter on the analysed parameter set.

5.3.2 Minimum and maximum boundary coverage

Figure 5.10 presents the main boundary value coverage result. The plot groups parameters into four categories: neither boundary found, only minimum found, only maximum found, and both boundaries found.

The boundary value coverage analysis shows that out of 43 parameters, 29 (67.4%) had neither minimum nor maximum values observed in the analysed tests. In contrast, 7 parameters (16.3%) had both boundary values detected, 3 (7.0%) had only minimum coverage, and 4 (9.3%) had only maximum coverage. Complete boundary value coverage at the subsystem level was achieved for less than one-fifth of the investigated parameters.

5.3.3 Parameter coverage examples

Table 5.3 shows examples from the boundary value coverage output. The table connects each parameter to its struct, minimum and maximum coverage status and the number of tests where the parameter name was found.

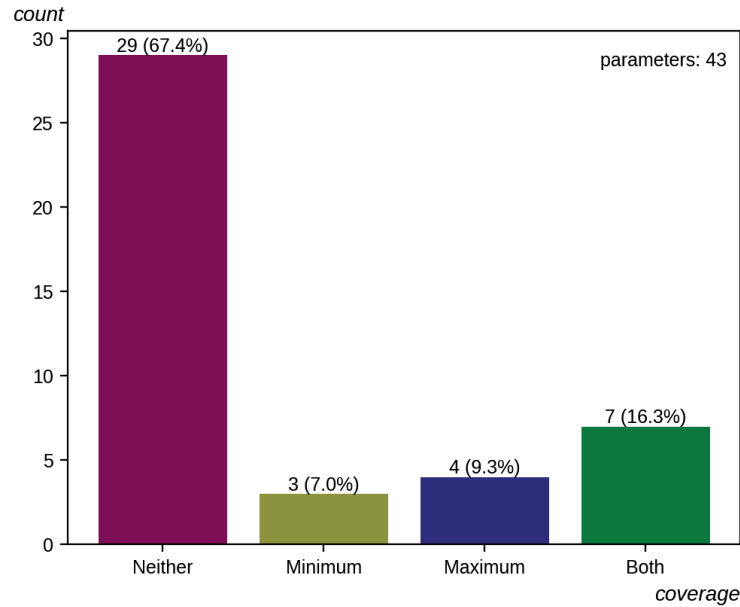


Figure 5.10: Boundary value coverage for analysed parameters.

Table 5.3: Overview of parameter coverage and test counts.

Struct	Parameter	Minimum	Maximum	Tests
Packet41DataA	nD_LEVELTR	True	True	90
LevelCrossingInputA	lx_start_location	False	True	136
LevelCrossingInputA	start_offset	True	True	101
LevelCrossingInputA	end_offset	True	True	70
LevelCrossingInputA	lx_speed	False	True	140

A parameter can occur in many tests without both boundaries being found. For example, `lx_speed` was found in 140 tests, but only the maximum value was reported as covered. In contrast, `start_offset` and `end_offset` had both boundary values reported as covered.

5.3.4 Special and reserved values

The main RQ3 evaluation relied on the documented minimum and maximum ranges found in the site-data header files. Special and reserved values were excluded from the boundary value analysis, as they are specified in SUBSET-026, Chapter 7 [8] rather than directly defined in the site-data structs. An experimental extraction of special and reserved values was attempted, but reliable extraction of these values was outside the scope of this thesis.

Discussion

This chapter explores what the results suggest about the current test environment and the methods developed in this thesis.

6.1 Interpretation of results

The main finding is that the existing test setup could be made more dynamic with relatively small changes. Each research question showed this in a different way. RQ1 showed that changing `l_train` caused failures in tests that passed with the baseline value. RQ2 showed that route-specific MA tests could be generated from site data, but also that a single template did not handle every route. RQ3 showed that boundary value presence could be checked systematically, but not that boundary behaviour was verified.

These findings should be interpreted with caution. The thesis did not perform a full root-cause analysis of every failed test. It also did not prove that the RBC logic was wrong. The results mainly show how the test environment reacts when assumptions about train length, route position and parameter values are challenged. Together, the results suggest that input variation, template-based generation and boundary-value checks can make the existing test suite more dynamic.

6.1.1 Train length variations

RQ1 explored how changing train length affected an existing automated test suite. All changed configurations produced failed and unresolved outcomes, while the same test set passed with the fixed value of `l_train = 70` m. This shows that the existing test suite was sensitive to train-length variation.

The most important interpretation is that train-length variation was relatively easy to introduce, and most tests still passed when `l_train` changed from the 70 m baseline. The failed tests showed where fixed assumptions about train length exist, and these findings can be used to further develop the test suite. One of the best examples of this was `l_endsection`. When a test still expected a value based on the baseline train length, the changed train length produced a value mismatch.

The concentration of failures supports the same interpretation. In the minimum, maximum and randomised configurations, `l_endsection` appeared most frequently. The failures were not spread randomly across all elements. They were concentrated in a smaller set of elements and error categories.

The repeated concentration of failures suggests that many failures came from assumptions in the existing test scripts, not from unrelated system behaviour. The result does not prove that all `l_endsection` failures had the same cause. It also does not prove that every failed test could be fixed by changing one calculation. However, it provides strong evidence that fixed expected values were a weakness in Alstom's test environment.

The maximum train length produced more failed outcomes than the minimum train length, as shown in Table 5.1. A likely explanation is that a long train is harder to place in a route scenario. A train with `l_train = 1000` m can extend across several track areas, and it can therefore interact with route boundaries, signals, level crossings, or waiting positions in ways that a shorter train cannot. This may explain why the maximum train length produced more failures, although individual failures were not analysed in detail.

The randomised train length result is also important because it showed that randomisation could be added without breaking most of the existing test suite. Sensitivity was not limited to the minimum and maximum values. Failures also appeared for values inside the interval. This shows that making the test suite more dynamic through randomisation is possible. The failed tests also identify where the existing test suite needs further development.

A next step would be to calculate more expected values from the train length used in each test. For example, expected values related to `l_endsection` could be calculated from train length and route geometry instead of being fixed for `l_train = 70` m. This would make it easier to use randomised train lengths as part of the existing test suite.

6.1.2 Generated movement authority tests

RQ2 explored how test scripts could be generated from site data using a template. The results show that a template-based generator can create executable test scripts from site data. In the final Jenkins run, 426 of 530 generated test scripts passed. This shows that route-specific test scripts could be generated with limited manual work by reusing existing test logic.

The result should not be interpreted as a complete solution for all routes. The final run still had 104 failed test scripts. The failures were mainly connected to `message_timeout` and `missing_balise`. This means that the generator worked for many routes, but more route-aware logic is needed before the method can handle all routes.

The template-based approach was a practical choice because existing test logic could be reused. One alternative would have been to parse a large visual map of the track layout. That approach would probably have required more work before

any tests could be generated. The chosen method instead used a simplified version of an existing MA test as a template and inserted site data to create route-specific scripts. This reduced how much new test logic had to be created and made the method practical for improving test coverage with limited effort.

This choice also created a limitation. A template can only handle the routes that its logic supports. Simple routes with a start signal, a stop signal, and a suitable balise were easier to handle. Some routes were harder, such as routes near RBC borders, routes with difficult balise placement and routes where train position depended on direction or other track objects.

The test scripts that failed were still useful because they showed which routes the template could not handle yet and where the generator needs further development. This is valuable because the failures were not random, but followed recurring categories that can guide further development. A few repeated failure types are easier to investigate than many unrelated failures.

The progression between Jenkins runs also shows why generated test logic needs to be evaluated on many routes. Changes to the template affected the pass rate, as the second run performed worse than the first run, while the third run performed best. This suggests that template changes can improve some routes while causing failures in others, so generated test logic should be evaluated on a broad set of routes, not only on one example route.

A next step would be to add more route-aware logic to the template. The generator could analyse more route objects before selecting a train start position and a balise. Examples of relevant objects include `GEO_SIGNAL_A`, `GEO_TD_A` and `GEO_BALISE_A`. This could make the generator more robust, but it would also make the template more complex because it would need to handle more route-specific cases. Handling these cases would require domain knowledge about how different route objects affect the expected MA behaviour.

6.1.3 Boundary value coverage

RQ3 explored how minimum and maximum configured values could be checked against existing tests. The method provided a systematic way to compare documented parameter ranges with values found in test scripts.

The most important interpretation is that the method checks value presence, not verified behaviour. If a boundary value was found in a test script, the method marked that boundary as covered. This does not prove that the test checked the correct system behaviour for that value. It only proves that the value was found in the analysed test material.

The method is valuable because it provides Alstom with a systematic way to check whether documented boundary values are present at subsystem test level. Since many analysed parameters had neither the minimum nor the maximum value detected, the result suggests that documented minimum and maximum values were not tested enough at subsystem test level. This also shows why direct boundary

checking is useful, as a parameter can appear in many tests without its documented minimum or maximum value being present.

The limitation that affected RQ3 the most was the mapping between parameter names and structs. The method extracted documented ranges as (`struct`, `parameter`) pairs, while the test scripts usually contained only parameter names. This meant that the coverage search matched values by parameter name, not by the full (`struct`, `parameter`) pair.

The user-source filter was used to focus the analysis on user-configurable parameters. This reduced the analysed input dataset, but it was not intended to solve the mapping problem. A parameter could still appear in more than one struct, and if the test script only contains the parameter name, the method cannot prove which struct the value belongs to.

This means that some reported coverage may be linked to the wrong struct. This was the main consequence of the mapping limitation in RQ3. The method can show that a value exists in a test file. It cannot always show that the value belongs to the struct that was meant to be checked. The result should therefore be treated as possible boundary-value presence, not confirmed struct-specific coverage.

When checking if a parameter is unique, the search should not only use the filtered input data. A better implementation would mark a parameter as not unique if the same name exists outside the filtered set and the test-script match cannot be tied to a specific struct. Under that stricter interpretation, `is_unique = false` would mean that the method cannot confidently connect the found value to one documented struct.

Special and reserved values were not included in the main result because they would have required more work than was possible within the thesis. Minimum and maximum values could be parsed from the analysed header files, while special and reserved values were defined in tables in SUBSET-026 Chapter 7. These tables were less structured, and connecting each value to the correct parameter would have required more manual work. With more time, special and reserved values could be added through manual extraction or a more structured source.

An alternative implementation would be to connect parameter use in tests more clearly to the struct where the parameter was defined. This could reduce the need to rely on uniqueness. It would require deeper static analysis of the test scripts and helper functions. It could also require more knowledge about how test functions map parameter names to site-data structs.

6.2 Common patterns

The three research questions revealed a common pattern where the methods worked in many common cases, but limitations appeared when existing assumptions were challenged.

- In RQ1, the existing tests often assumed that train length was fixed. When

`l_train` changed, expected values and messages failed.

- In RQ2, the template assumed that the same test logic could handle many routes. This worked for most generated test scripts, but not for all routes.
- In RQ3, the method assumed that finding a parameter value in a test script was enough to reason about boundary-value presence. This was useful, but it did not prove boundary behaviour.

The repeated failure categories were also useful because they made the problems easier to investigate. In RQ1, recurring affected elements included `l_endsection`, `d_ref` and `q_dir`. In RQ2, recurring failures appeared in `getAndAckMa`, `completeTrainStartup` and `GEO_BALISE_A`. These repeated categories suggest that the problems are not random and can be investigated systematically.

This matters for future improvement because repeated failure mechanisms are easier to investigate than many unrelated failures. Since the results point to a limited number of recurring mechanisms, the test suite can be improved step by step.

6.3 Limitations

The first limitation is that RQ1 and RQ2 relied on processed logs. The raw logs were too large to inspect manually, so the workflow reduced them to information that could be counted and plotted. This made the analysis possible, but if relevant context was removed during cleaning, some failure explanations may have been incomplete.

The log-processing method also changed during the thesis. Early versions kept too much unrelated information, while later versions focused on the values and context needed for the analysis. This improved the analysis, but it also means that log interpretation was partly shaped by what became important during the study. A professional parsing framework could have reduced the risk of missing useful information.

The second limitation is that the thesis did not perform full root-cause analysis for every failed test. This was most visible in RQ1 and RQ2, where the analysis focused on recurring mechanisms rather than explaining every failed script. This was a reasonable scope decision, but it means that the results show patterns, not the exact cause of every failure.

The third limitation is related to the unit tests used for RQ3. The unit tests checked parsing, file selection, value extraction and workflow behaviour, which improved confidence in the scripts. However, they only checked expected cases and known edge cases, so they did not prove that every possible input format was handled.

The fourth limitation is that the RQ3 tests verified that the script could find parameter names and boundary values, but not whether each found value belonged to the intended (`struct`, `parameter`) pair in every test context.

The fifth limitation is that the selected train-length interval did not cover all possible train lengths. The thesis used 10–1000 m for the test runs because this was more realistic for the studied environment, even though the ERTMS specification allows larger values. However, the purpose was not to test every possible train length, but to make the existing tests more dynamic through randomisation.

The sixth limitation is that RQ3 excluded special and reserved values. This was necessary within the thesis scope, but it leaves some boundary-related values outside the main result. Minimum and maximum values are not the only values that matter in signalling data, as some special values can have protocol-specific meaning.

6.4 Implications for test automation

The results suggest that test automation should support engineering judgement, not replace it. Randomised values, generated test scripts and boundary-value checks can make the test suite more dynamic, but the results still need expert interpretation.

- RQ1 indicates that randomisation can be introduced with limited changes because most tests still worked when `l_train` was randomised. The failed tests are also useful because they show where the existing test suite depends on fixed assumptions. The goal is not to keep randomisation producing failures, but to use those failures to improve the tests.
- RQ2 demonstrates that template-based generation can improve test coverage with limited effort by reusing existing test logic. A generator can apply a similar scenario to many routes and create route-specific scripts from site data. This can reveal route-specific cases that would otherwise remain untested. The method is most useful when generated failures are reviewed and used to improve the template.
- RQ3 provides a systematic way to check whether documented boundary values are present at subsystem test level. This is useful because it changes boundary checking from a manual question into a repeatable workflow. A next step could be to generate or recommend minimal tests for missing values, which would turn coverage analysis into active test improvement.

Automation can reduce manual search work, expose weak assumptions and produce repeatable evidence, but it still depends on engineers to interpret the results and decide how the test suite should be improved.

Conclusion

This chapter summarises the most important results of the thesis. It also answers the research questions, explains how the results contribute to the purpose of the thesis and describes how the results can be used. The chapter ends with future work.

7.1 Summary

This thesis investigated three ways to improve testing work in an automated railway signalling test environment. The first was to make the existing test suite more dynamic by introducing randomisation. This was done by randomising train length. The second was to use templates and site data to generate route-specific test scripts and increase test coverage. The third was to find a systematic way to check coverage of user-configurable parameters at subsystem test level.

- RQ1 demonstrated that the existing test suite could be made more dynamic by randomising `l_train`. Most tests still worked when the train length was changed, which indicates that randomisation can increase test variation with limited changes to the existing tests. The failed and unresolved tests were also useful, as they showed where fixed assumptions about train length exist. These assumptions were mainly linked to fixed expected values, fixed message expectations and fixed waiting patterns.
- RQ2 showed that route-specific MA test scripts could be generated from site data by reusing an existing test script as a template. In the final Jenkins run, 426 of 530 generated scripts passed. This indicates that template-based generation can increase route coverage with limited manual effort. The remaining failures showed that more route-specific handling was still needed.
- RQ3 provided a systematic way to check coverage of user-configurable parameters at subsystem test level by comparing documented minimum and maximum values with values found in existing test scripts. The result suggests that documented boundary values were not tested enough in the analysed

scope. However, the method shows possible boundary-value presence, not verified boundary behaviour or confirmed struct-specific coverage.

7.2 Ethical aspects

Since railway signalling is a complex field where systems must work correctly, the results had to be interpreted and presented with care. Failed tests were treated as evidence about the test environment, such as fixed assumptions, missing route-specific handling or limitations in the test scripts.

7.2.1 Confidential information

The thesis was carried out with Alstom and used internal test setups, internal site data, internal software and implementation details. Data security was therefore important throughout the work. Internal material was handled in Alstom's environment, and source code or detailed implementation information was not shared through public tools or external services. Some details were also abstracted or omitted from the thesis. This included exact file paths, product-specific implementation details and information that was not needed to understand the academic argument.

The results were presented through methods, result categories, counts and general failure mechanisms. This made it possible to explain the work without exposing unnecessary internal details. The thesis therefore focused on what the methods showed, not on confidential implementation information.

7.2.2 Public benefit

Railway signalling software is part of the infrastructure needed for reliable and sustainable railway transport. As railway systems continue to develop and expand, improved testing methods can support the long-term dependability of such systems. The public benefit of this thesis is indirect. By helping engineers find hidden assumptions, route-specific weaknesses and boundary-value gaps, the methods can support better and more reliable railway software.

7.3 Research questions

This section answers the three research questions stated in the introduction.

7.3.1 Research question 1

RQ1 showed that the existing automated test suite could be made more dynamic by randomising train length.

Most tests still worked when `l_train` was randomised, which indicates that randomisation can improve test variation with limited changes to the existing test

suite. The failed and unresolved tests were also useful because they showed where the test suite needs further development.

These outcomes showed that several test scripts relied on expected values or message sequences that were only valid for the baseline configuration. `l_endsection` is one of the best examples, because the expected value depends on the train length and route geometry.

The result does not indicate that the RBC logic was incorrect. It shows that parts of the test suite depend on fixed assumptions. This is useful because those assumptions can be reviewed and addressed to make the test suite more dynamic.

7.3.2 Research question 2

RQ2 showed that route-specific MA test scripts can be generated from site data by reusing an existing test script as a template.

The generated scripts used route, signal and balise information from site data and were written directly to the correct location in the test suite.

In the final Jenkins run, 426 of 530 generated scripts passed. This indicates that template-based generation can increase route coverage with limited manual effort.

However, the method was not complete. The failed tests showed that a single template does not cover all routes and that more route-specific handling is still needed. The method should be treated as a way to support test generation, not as a replacement for manually maintained tests.

7.3.3 Research question 3

RQ3 provided a systematic way to check whether user-configurable parameters were represented at subsystem test level. The method compared documented minimum and maximum values with values found in existing test scripts, which created a repeatable analysis workflow for checking possible boundary-value presence.

The results suggest that documented minimum and maximum values were not tested enough at subsystem test level. However, the method shows possible boundary-value presence, not verified boundary behaviour or confirmed struct-specific coverage.

7.4 Contributions

The purpose of the thesis was to investigate practical ways to improve testing of signalling software through randomisation, template-based test script generation and systematic boundary-value checks. The results contribute to this purpose in three main ways.

- The train-length test runs showed that randomisation can make an existing test suite more dynamic with limited changes. They also showed where fixed train-length assumptions exist in tests that otherwise pass.

- The generated MA test scripts demonstrated that existing scripts can be reused as templates and combined with site data to create route-specific scripts. This can increase test coverage with limited manual effort and make route-based testing more systematic.
- The boundary-value analysis provided a systematic way to check whether user-configurable parameters were represented at subsystem test level. The method found that documented minimum and maximum values were not tested enough at subsystem test level, while also providing a repeatable way to find possible coverage gaps.

Together, these results provide Alstom with ways to improve testing of signalling software through randomisation, template-based generation and boundary-value checks.

7.5 Practical application

The results provide Alstom with input for improving the existing test suite.

- RQ1 can help Alstom identify test scripts that depend on fixed train-length assumptions and use failed tests as input for further test-suite development.
- RQ2 presented a way for Alstom to generate route-specific test scripts from templates and site data, which can increase coverage with limited manual effort.
- RQ3 created a systematic method for checking whether user-configurable parameters are represented at subsystem test level.

These results should be used as input for review and future test improvement, not as final proof that the current tests are complete.

Recurring failure categories, affected elements and missing boundary values give concrete starting points for deciding which tests should be reviewed or improved first.

7.6 Future work

For RQ1, the main improvement is to reduce fixed expected values in the existing test suite. Train-length-dependent fields, such as `l_endsection`, should be calculated from route geometry and `l_train` instead of depending on one fixed train length. This would make randomised train-length testing more useful and help turn failed tests into input for further test-suite development. The maximum-length failures should also be studied in more detail, because long trains can occupy track areas in ways that shorter trains do not.

For RQ2, the generator should be improved with more route-aware logic. The main functions to improve are balise selection, train positioning and route-specific

handling. These improvements could increase the pass rate and make the generated scripts useful for more routes.

For RQ3, the most important improvement is stronger mapping between parameter names and structs. Future work should analyse helper functions, setup files, call chains or runtime data to determine which struct a test value belongs to. Until this is addressed, the result should remain limited to possible boundary-value presence. RQ3 should also be expanded to include special and reserved values, since these values are important for complete boundary-related testing. Future work could obtain these values in a structured format or extract them manually from SUBSET-026, Chapter 7.

Another future direction is to generate new test scripts for missing boundary values. This would turn the method from a detection tool into a method that also supports test-suite improvement. A missing boundary value could be reported, reviewed and then used to create a minimal test script.

The final future direction is further integration into CI. Randomised train-length testing can be added when the fixed expected values have been addressed. Generated test scripts and boundary-value checks should also be integrated carefully, so that failing tests give useful signals.

References

- [1] Docker. “Docker Engine | Docker Docs.” [Online]. Available: <https://docs.docker.com/engine/>. Accessed: 20.03.2026.
- [2] D. Schwencke, “Model-based test case generation using the example of an RBC function,” *Signal + Draht*, 2020. [Online]. Available: <https://elib.dlr.de/131267/>, Accessed: 17.05.2026.
- [3] T. Yadavalli, “An Industrial Case Study to Improve Test Case Execution Time,” M.S. thesis, Blekinge Institute of Technology, 2020. [Online]. Available: <https://bth.diva-portal.org/smash/record.jsf?pid=diva2%3A1514587>, Accessed: 17.05.2026.
- [4] P. di Tommaso, A. Fantechi, S. Gnesi, F. Mazzanti, A. Ferrari, and G. O. Spagnolo, “The Simulation of Anomalies in the Functional Testing of the ERTMS/ETCS Trackside System,” in *Proceedings of the 9th IEEE International Symposium on High-Assurance Systems Engineering*, 2005, pp. 131–138. DOI: 10.1109/HASE.2005.12. [Online]. Available: <https://lnu.diva-portal.org/smash/record.jsf?pid=diva2%3A1213844>, Accessed: 17.05.2026.
- [5] B. Smakic, “Artificial Intelligence Applications for Railway Signalling,” M.S. thesis, KTH Royal Institute of Technology, 2021. [Online]. Available: <https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1621923>, Accessed: 17.05.2026.
- [6] Alstom University, *ERTMS-ETCS Introduction (Basic): 2.02 Subsystems and Components*, Web-based internal training platform, Alstom, Accessed: 13.05.2026.
- [7] Alstom University, *Railway Signalling Overview: 6.9 ERTMS*, Web-based internal training platform, Alstom, Accessed: 13.05.2026.
- [8] European Union Agency for Railways. “SUBSET-026: System Requirements Specification.” version 4.0.0. Index 004 in CCS TSI Appendix A – Mandatory specifications. [Online]. Available: <https://>

- www.era.europa.eu/system/files/2023-09/index004_-_SUBSET-026_v400.zip. Accessed: 13.05.2026.
- [9] E. U. A. for Railways. “ERTMS Specifications.” [Online]. Available: <https://www.era.europa.eu/era-folder/1-ccs-tsi-appendix-mandatory-specifications-etcs-b4-r1-rmr-gsm-r-b1-mr1-frmcs-b0-ato-b1>. Accessed: 15.05.2026.
- [10] European Committee for Electrotechnical Standardization, “Railway Applications – Requirements for Software Development,” CENELEC, EN Standard 50716, Nov. 2023, p. 25.
- [11] European Committee for Electrotechnical Standardization, “Railway Applications – Requirements for Software Development,” CENELEC, EN Standard 50716, Nov. 2023, p. 24.
- [12] P. Ammann and J. Offutt, “Introduction to Software Testing,” in New York, NY: Cambridge University Press, 2008, pp. 5–8.
- [13] A. Dennis, B. H. Wixom, and D. Tegarden, “System Analysis & Design An Object-Oriented Approach with UML,” in Hoboken, NJ: John Wiley & Sons, Inc., 2015, pp. 475–476.
- [14] European Union Agency for Railways and UNISIG and EEIG ERTMS Users Group. “ERTMS/ETCS Glossary of Terms and Abbreviations (Subset-023), Version 4.0.0.” [Online]. Available: https://www.era.europa.eu/system/files/2023-09/index003_-_SUBSET-023_v400.pdf. Accessed: 13.05.2026.
- [15] J. Ousterhout. “What is Tcl.” [Online]. Available: <https://wiki.tcl-lang.org/page/What+is+Tcl>. Accessed: 23.03.2026.
- [16] Alstom. “Alstom University.” [Online]. Available: <https://www.alstom.com/careers/alstom-university>. Accessed: 18.03.2026.
- [17] J. D. Hunter, “Matplotlib: A 2D graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.
- [18] J. Sköld, J. Mattisson, U. Söderström, J. Unebrand, and M. Backman, “Godstrafik på järnväg,” *Trafikverket*, 2025. [Online]. Available: <https://trafikverket.diva-portal.org/smash/get/diva2:2011928/FULLTEXT01.pdf>, Accessed: 23.03.2026.
- [19] U. Nyström, “Långa godståg på prov,” *Järnvägar.nu*, 2025. [Online]. Available: <https://jarnvagar.nu/langa-godstag-pa-prov>, Accessed: 15.05.2026.
- [20] Jenkins contributors, *Jenkins*, comp. software, 2026. [Online]. Available: <https://jenkins.io/>, Accessed: 15.05.2026.
- [21] Google. “Gemini: A Family of Multimodal AI Models.” [Online]. Available: <https://deepmind.google/technologies/gemini/>. Accessed: 12.05.2026.

- [22] Microsoft. “Microsoft 365 Copilot for Enterprise.”[Online]. Available: <https://www.microsoft.com/en-us/microsoft-365/copilot>. Accessed: 12.05.2026.
- [23] UNIFE - The European Rail Supply Industry, “ERTMS/ETCS Levels: Different Levels to Match Customer’s Needs,” UNIFE, Factsheet Factsheet #2, 2024. [Online]. Available: <https://www.ertms.net/wp-content/uploads/2024/03/ERTMSETCS-Levels-Updated-2024-Edition.pdf>, Accessed: 28.04.2026.
- [24] I. Dahlström. “Repository appendix.”[Online]. Available: <https://gitlab.com/thesis-id/appendix.git>. Accessed: 21.05.2026.

The source code and anonymised input data are available at <https://gitlab.com/thesis-id/appendix> [24]. The repository contains the scripts used for the thesis methods, together with small example files that can be used to run the demo workflow. The setup commands are listed in Section A.4.

A.1 Template-based test generation

Generates test scripts from site data using a template.

`gitlab.com/thesis-id/appendix/.../rq2/generate_tests.py`

A.2 Boundary value coverage

Scripts used for research question 3.

A.2.1 Struct range parser

Parses site-data files to extract the documented minimum and maximum values for each parameter.

`gitlab.com/thesis-id/appendix/.../rq3/parse_struct.py`

A.2.2 Test script finder

Scans existing test scripts to verify if the documented minimum and maximum parameter values are used.

`gitlab.com/thesis-id/appendix/.../rq3/find_tests.py`

A.2.3 Boundary-value workflow

Executes the boundary-value analysis and exports the results to a CSV file.

gitlab.com/thesis-id/appendix/.../rq3/main.py

A.2.4 Struct parser tests

Verifies that documented minimum and maximum values are correctly parsed from site-data files.

gitlab.com/thesis-id/appendix/.../rq3/tests/test_parse_struct.py

A.2.5 Test script finder tests

Verifies that parameter values are correctly extracted and checked against documented boundaries.

gitlab.com/thesis-id/appendix/.../rq3/tests/test_find_tests.py

A.2.6 Boundary-value workflow tests

Verifies that the boundary-value analysis workflow generates the correct CSV output.

gitlab.com/thesis-id/appendix/.../rq3/tests/test_main.py

A.3 Data processing

Scripts to clean logs, extract data and create plots.

A.3.1 Log cleaning script

Condenses large logs into a clean log file containing only failed and unresolved tests.

gitlab.com/thesis-id/appendix/.../scripts/create_clean_log.py

A.3.2 CSV generation script

Fetches and structures cleaned log data into CSV format for analysis.

gitlab.com/thesis-id/appendix/.../scripts/generate_csv_from_logs.py

A.3.3 CSV plotting script

Generates plots and summary figures from the processed CSV data.

gitlab.com/thesis-id/appendix/.../scripts/analyze_csv.py

A.4 Execution instructions

To run the demo locally, clone the repository and install the dependencies.

A.4.1 Git

Clone the repository:

```
git clone https://gitlab.com/thesis-id/appendix.git  
cd appendix
```

Linux/macOS

Create the virtual environment once:

```
python3 -m venv .venv  
source .venv/bin/activate  
pip install -r requirements.txt
```

Run the demo workflow:

```
./scripts/demo/demo.sh
```

Windows

Create the virtual environment once:

```
py -3 -m venv .venv  
.\.venv\Scripts\activate.bat  
pip install -r requirements.txt
```

Run the demo workflow:

```
scripts\demo\demo.bat
```



LUND
UNIVERSITY

Series of Bachelor's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2026-1159
<http://www.eit.lth.se>