ADEPTNESS – Design-Operation Continuum Methods for Testing and Deployment under Unforeseen Conditions for Cyber-Physical Systems of Systems

EUROPEAN COMMISSION

Horizon 2020

H2020-ICT-01-2019

GA No. 871319

Deliverable No.	ADEPTNESS D7.2	
Deliverable Title	Report on the Results of the Cost-Benefit Assessment	
Deliverable Date	M.36	
Deliverable Type	Report	
Dissemination level	Public	
Written by	Alstom, MDH	
Checked by	Alstom	
Approved by	Alstom	
Status	Public	

H2020-ICT-01-2019 – 871319 – ADEPTNESS: Design-Operation Continuum Methods for Testing and Deployment under Unforeseen Conditions for Cyber-Physical Systems of Systems

Acknowledgement

The author(s) would like to thank the partners involved with the project for their valuable comments on previous drafts and for performing the review.

Project partners

1 – MGEP – Mondragon Goi Eskola Politeknikoa – ES

2 – ORO – Orona S. Coop – ES

3 – UES – Ulma Embedded Solutions S. Coop – ES

4 – SRL – Simula Research Laboratory S. Coop – NO

5 – BT – Bombardier Transportation Sweden – SE

6 – IKL – Ikerlan S. Coop – ES

7 – EGM – Easy Global Market SAS – FR

8 – MDH – Maelardalens Hoegskola – SE

9 – TUW – Technische Universitaet Wien – AT

Disclaimer:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319.

Document Information

Additional author(s) and contributing partners

Name	Organisation
Wasif Afzal, Muhammad Nouman Zafar, Amirali Piadehbasmenj	MDH
Henrik Gustavsson, Nedim Zaimovic, Inderjeet Singh	Bombardier Transportation/Alstom

Document Change Log

Name	Date	Comments
V0.1	15-March-2023	Initial draft
V0.2	31-March-2023	Final Version

Exploitable results

Exploitable results	Organisation(s) that can exploit the result
Model-Based Test scrIpt GenEration fRamework (TIGER)	MDH, Bombardier Transportation/Alstom
Passive Testing Toolchain using T-EARS	MDH, Bombardier Transportation/Alstom

Contents

1 PURPOSE OF THE DOCUMENT

The purpose of this document is to assess the impact of Adeptness methods in the case study in the railway domain. The overall objective is to compare the quality and the cost of a new software release.

1.1 Document structure

Section 2 presents an introduction to the deliverable. Section 3 consists of an overview of the use case and the current process at Bombardier Transportation. The detailed description of the use case is in Section 4, which also contains the methodology of relevant Adeptness solutions. The results of the investigations are in Section 5, which also contains a discussion of results as well as the limitations. Conclusions are presented in Section 6, while future work is in Section 7.

1.2 Deviations from the original Description in the Grant Agreement Annex 1 Part A

1.2.1 Description of work related to deliverable in GA Annex 1 – Part A

There are no deviations with respect to work of this deliverable.

1.2.2 Time deviations from original planning in GA **Annex** *1 – Part A*

There are no deviations with respect to work of this deliverable.

1.2.3 Context deviations from the original plan in GA Annex 1 – Part A

There are no deviations from the Annex 1.

2 INTRODUCTION

Software controls safety-critical functions of systems in different domains, e.g., in avionics and vehicular. However, the failure of software in such systems directly affects the physical world. Therefore, failures in safety-critical software can lead to substantial risk to the safety of human lives, serious environmental damage, and severe economic problems. Engineering safety-critical systems typically require a certain degree of certification according to safety standards and defined processes for thoroughly analysing the system requirements, together with software testing to ensure its reliability.

The analysis of the preliminary system requirements is performed by one of the qualitative or quantitative safety analysis techniques such as expert analysis, failure mode and effect analysis, reliability block diagrams, and fault tree analysis [1]. After analysing system requirements, safety-critical functions are usually implemented in specific industrial control software systems. For example, Programmable Logic Controllers (PLCs) support multiple programming languages for the development of industrial applications and have been widely adopted in several domains. The testing of such software is then carried out to ensure adequate functional and non-functional operations of the system according to specific system requirements. However, testing such systems is costly. As a solution to reducing the cost of testing and assuring the reliability of such systems, several techniques for automated test generation such as model-based testing (MBT), and combinatorial testing (CT) exist.

Despite the promise of MBT, the industrial adoption of it is slow and there is a need for more industrial case studies that evaluate the strengths and weaknesses of MBT. Moreover, since manual test generation is still widely used in industrial practice, more systematic studies on how manual test design compares with MBT and how it can be adopted in the industrial safety-critical domain are needed. Given that for safety-critical systems, rigours testing needs to be performed according to certain standards, it is important to bring more evidence on how MBT tools compare with, what is perceived as rigorous manual test design performed by industrial practitioners.

MBT generates abstract test cases based on a model of the System Under Test (SUT). The abstract test cases can be transformed into concrete, executable test scripts that can eventually produce test verdicts. However, we have observed that this concretization step is not covered well enough in the literature or is discussed only in the context of web- based and mobile applications where somewhat mature script generation frameworks exist such as Selenium and Appium. Thus, only few papers report on the details of the concretization step and even fewer have addressed it for embedded software testing at integration and system levels.

Besides, manual testing is still considered a prevailing technique for the testing of real-world industrial applications [2], with some evidence suggesting that both technical and non-technical skills are required for effective fault detection [3]. There is a scarcity of research and consequently empirical evidence into the fault detection effectiveness of automated test generation techniques and industrial manual testing, especially if the aim is to generate test cases at the system level of a safety-critical system. Thus, the use of automated test generation alongside industrial manual testing requires a thorough evaluation of the fault detection capabilities in an industrial setting.

This report summarizes the impact of methods developed in Adeptness in the railway use case. In summary, this report focusses on the following contributions:

- Modelling of the fire indication system of the train control management system (TCMS) in GraphWalker (GW) using two sources (requirement specifications and test specifications).
- Evaluation of the generated models for completeness and representativeness.
- Comparison with manual test cases from practitioners.
- Documentation of our experiences of selecting and on-going use of GW for both industrial application and research purposes.
- Development of a Model-Based Test script Generation framework (TIGER) based on GW.
- Evaluation of the behaviour of TIGER-generated test scripts by injecting faults in the model of a fire detection system controlled by the TCMS.
- Investigation of MC/DC adequacy of two automated testing techniques (CT and MBT) and industrial manual testing at system-level, along with measuring requirement coverage and performance efficiency in terms of time.
- Investigation of the fault detection effectiveness of manual and two well-known automated test generation techniques (i.e., CT and MBT) through mutation analysis, at the system level of the safety-critical train control management system (TCMS).
- Experiences from introducing passive testing at the system level showing how this technique can be adopted and deployed in the safety-critical domain.

3 OVERVIEW OF THE USECASE AND CURRENT PROCESS

3.1 Company and Context

In early 2021, Bombardier Transportation was acquired by Alstom. However, for the purpose of consistency in Adeptness deliverables, we will keep referring to its use case as Bombardier Transportation's (BT's) use case.

Bombardier Transportation (BT) has an extensive portfolio of railway transportation solutions for diverse customers all over the world. This portfolio includes high-speed trains, metros, monorails, trams, to turnkey systems, services, infrastructure, signalling and digital mobility (https://www.alstom.com/company). BT's ambition is to contribute more and more to green and smart mobility solutions, leading societies to a low carbon future, and providing sustainable foundations for the future of transportation (https://www.alstom.com/company). In 2021/2022, the total number of BT employees exceeds 74000, with 250 sites across 70 countries, with 150000 vehicles in commercial service worldwide.

3.2 Product

BT's Train Control and Management System (TCMS) runs the central processor units that provide connections to all the electronic equipment, traditional train wiring and activators. TCMS implements several of the required train functions, including train command and control, train safety, maintenance, passenger information systems, passenger comfort system, video surveillance, and train to wayside data transfer. The TCMS team that is part of the Adeptness project, programs the applicative software of the central processor units to make sure that the requested behaviour of the related train function is achieved. This software, after validation, is installed into the train central processor units. Thus, TCMS provides functional support mainly with the help of software.

TCMS is an integral subsystem embedded in a complex railway vehicle architecture together with other subsystems and conventional train control. TCMS is connected to any other subsystem which requires support by software functions, such as power converters, braking system, train radio and others, as shown in Figure 1 below.

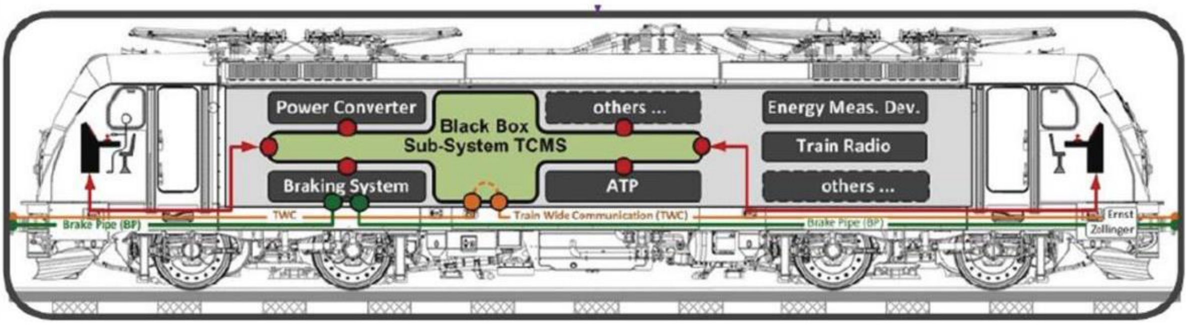


Figure 1. TCMS is connected to any other subsystem that requires support by software functions.

The validation and verification engineers at TCMS ensure that the Applicative Software corresponds to the required behaviour. This must be confirmed in nominal and degraded scenarios. The validation activities include requirement management/test management, configuration and change management of the software releases, software specification verification, tests specification and scripting, software review/code review, supporting software integration on trains, processes and tools optimizations, and release note restriction.

The application SW of the TCMS System is largely developed using a design centric integrated development environment which is based on the function block language according to norm IEC 1131. Using this language minimises the risk for program faults because it does not allow dangerous language constructs (e.g., null pointer assignment).

TCMS is an integral part of the vehicle architecture. As mentioned before, the TCMS system together with the conventional train control and other systems (e.g., brake, traction chain, car body, boogies etc.) contributes to the realisation of the overall functions on vehicle or train levels. The vehicle breakdown to system level is shown in Figure 2 where the range of scope of the work in Adeptness is also highlighted.

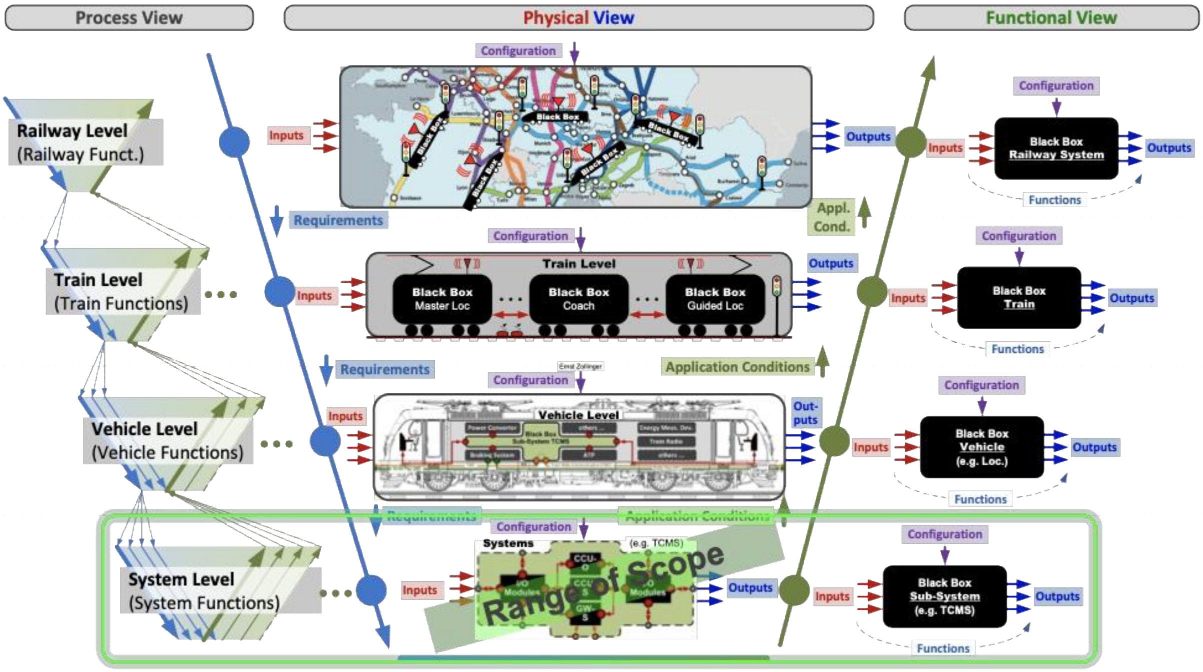
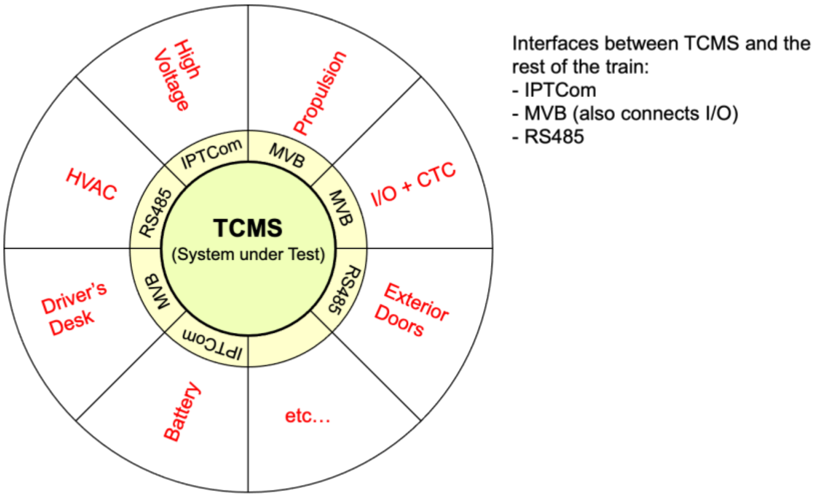


Figure 2. Breakdown of the complete system covering various levels.

3.3 Process

There is a defined process of testing and validation of TCMS. There is a test level hierarchy as well as scope of testing defined, TCMS acts as a system under test, where the different intelligent units are connected to each other and the other intelligent units on the train via different communication links such as MVB- and IP networks. This is shown in Figure 3.



Interfaces between TCMS and the rest of the train:
- IPTCom
- MVB (also connects I/O)
- RS485

Figure 3. The system under test (TCMS) has defined interfaces between TCMS and other units of the train.

The testing and validation of the TCMS follow certain guiding principles: (1) Stable TCMS software application before delivery to commissioning (there should not be any serious stopping faults on the train) (2) Minimal commissioning time on the train (reduction from months to weeks) and (3) Efficient update and retest procedure (reduction from weeks to hours). Moreover, the testing strategy should fulfil the demand of the safety-integrity in compliance with the norm (EN50128/EN50657).

The testing strategy at the company is defined by the following:

§ Level for formal testing including the demand on test coverage

§ Test Coverage & Traceability

§ Selection criteria for specific test environments for the TCMS System Test

§ Specific test techniques

§ Comparison between different test level

§ Passed / failed criteria, suspension criteria and resumption requirement

§ Incremental test for changes

§ Usage of a laboratory for the test

In addition, it relies on the following complementary measures:

§ Continuous integration & test

§ Developer tests

The test strategy utilises the incremental development model for the continuous integration & test. The following subsections will briefly discuss each of these elements of the testing strategy.

3.3.1 Test level hierarchy at the company

As was shown in Figure 2, there are different levels of test and validation, moving from the sub-system to vehicle to train to eventually to the complete railway system. The first two levels, along with their corresponding testing activities are shown in more detail in Figure 4 below. At the basic level, there is software level testing where component testing and TCMS function testing is done. At the system level, TCMS sw/hw integration testing, TCMS system and validation testing takes place. At the vehicle level, there is vehicular functional testing that is performed. For Adeptness, the scope of testing remains at the software and the system level with TCMS as the system under test.

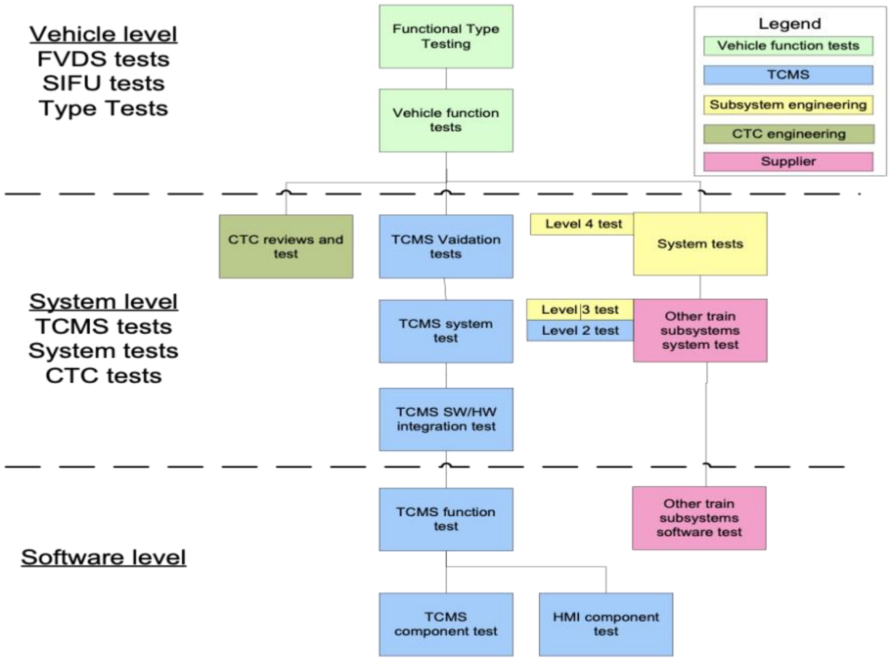


Figure 4. Test level hierarchy at the use case company.

The test level, their content and their correspondence to the test level demanded by the norms EN 50657 / EN 50128 are shown in Table 1.

Table 1. TCMS test levels and types of testing at the software and the system test levels.

Test level				Corresponding test level of EN 50657/50128			
Name	BI	SIL 0	SIL 1/2	Name	BI	SIL 0	SIL 1/2
M – Mandatory C – Conditional Mandatory				R – Recommended HR – Highly Recommended			
TCMS SW Test (at SW application level)	C 1)	C 1)	M	SW Component Test	R	R	HR
				SW Integration Test	R	HR	HR
TCMS System Integration Test	M	M	M	SW/HW Integration Test	R	HR	HR

<table>
<tr><td>TCMS System Test</td><td>M</td><td>M</td><td>M</td><td>Overall Software Test</td><td>HR</td><td>HR</td><td>HR</td></tr>
<tr><td colspan="8">[1] The TCMS SW Test (at SW application level) shall apply for those parts of BI (Basic Integrity) and SIL0 applications which are known to be error-prone, complex or were an error may have an impact on the availability of the vehicle or where the integration of SW components shows critical parts. SIL stands for Safety Integrity Level.</td></tr>
</table>

The SW Test at the SW application level is based on the TCMS SW Requirement Specification and the TCMS SW Architecture & Design Specification and is documented in the SW Test Specification.

The main target of the TCMS SW Test at SW application level is to ensure the correctness of

§ integration of the SW components into the SW application

§ SW application regarding the SW Requirement Specification

The TCMS System Integration Test is based on the TCMS System Architecture Specification and is documented in the TCMS Integration Test Specification. It corresponds to the HW/SW Integration Test as demanded by the norm EN 50128 and EN 50657. It also covers a regression test to check the correct behaviours of the TCMS system after changes.

The TCMS System Test is based on the TCMS System Requirement Specification including the Interface Signal List which defines the logical interface signals and the TCMS Interface Control Documents which define the physical interfaces. The design of the TCMS System Test is documented in the TCMS System Test Specification whereas the implementation is done via test scripts. The main target of TCMS system testing is to ensure the compliance of the TCMS System with the TCMS System Requirement Specification and its intention.

3.3.2 TCMS Test Coverage and Traceability

The establishment and maintenance of test-related traceability is an essential part of test analysis, test design and test execution process that is performed continuously according to the project progress. It demonstrates the coverage of all implemented requirements in terms of the validating test cases as well as the test execution results and contains justifications if test-related coverage is intentionally not given. Clearly, it does not show whether the content of the validating test cases fully "proves" the requirement, but this is done by checking the contents of the test cases during test case review. The test-related requirement coverage is formally documented by the Test Traceability Matrix. The requested test coverage is shown in Table 2 under consideration of the selection of the test environments for the system test.

Table 2: TCMS Requirement Test Coverage.

Test Level	Integrity Level	Test coverage
TSMS System Test	all	100% of all testable[1] requirements in all selected test environments

TCMS Integration Test	all	100% of all testable[1] requirements
TCMS SW Test (at SW application level)	BI, SIL0	selected testable[1] requirements (see below table)
	SIL1/2	100% of all testable[21] requirements
[1] Testable requirement: requirement which can be validated by test		

3.3.3 TCMS Test Techniques & Methods

During the test of the TCMS, the certain specific techniques and methods shall apply for the different test levels. At the software application level, functional black-bpx testing with boundary value analysis, equivalence classes and input partition testing is used. At the integration test and system test levels, functional black-box testing is complemented with performance testing of responses and memory constraints.

3.3.4 TCMS Test Passed/Failed Criteria

The passed/failed decision shall be done at test step level by comparing the expected test result with the actual test result. It shall be recorded in the corresponding test record. Each test case execution shall provide a summarization of the results of the test steps where the summarised result is passed only if all test steps in scope have the result passed. Pass/fail criteria are defined in each test case description in the column expected results corresponding to each test activity. The overall result of a test at a given level is determined by the evaluation of the results of the individual test and may be passed, conditional passed or failed. Conditional passed means the found malfunctions are evaluated as sufficiently safe (SIL1/2) and not operation critical for the intended use of the test object. In this case, the SW application or the TCMS System may be released but the malfunctions must be stated as known errors and application conditions must be promoted to the next level to show the impact of the malfunction. For that purpose, the change management process shall be utilized. If the overall test result is failed, the test must be interrupted, and the error corrected.

3.3.5 Test Suspension & Resumption

There are reasons to suspend a test.

§ The test environment shows a malfunction, or a misconfiguration of the test environment has been detected

§ The SW application or the TCMS System shows a critical malfunction

§ Further test activities of the TCMS SW application or TCMS System has been administratively stopped for the release under test

In any case, the test records shall be archived to be able to resume the test activities. To resume the test activities in case of a critical malfunction of the TCMS System, an impact analysis must be performed which shall, depending on the test level, follow the intention of the incremental verification, test, and validation activities.

3.3.6 Comparison between different test levels

To improve the error detection probability, the test result of a failed test execution shall be evaluated regarding the ability to uncover them in a lower test level. If feasible and efficient, the test cases and/or the test environment shall be adapted accordingly.

3.3.7 Incremental test for changes

According to the validation strategy, it is possible to rely on test results which were obtained from predecessor releases if the Change Impact Analysis allows this. In this case, testing shall be done at least to the following extent:

TCMS System Test

§ execute all system test cases which validates requirements identified by the Change Impact Analysis as affected, implemented, or tracked by work items demanding a new or changed functions or solving defects

TCMS Integration Test

§ execute all integration test cases which validates requirements identified by the Change Impact Analysis as affected, implemented, or tracked by work items demanding a new or changed functions or solving defects

§ execute the Integration Regression Test part of the TCMS Integration Test

TCMS SW Test

§ execute all SW test case of all safety relevant SW applications which are identified by the Change Impact Analysis as affected

§ execute all test cases of all SW applications which are identified by the Change Impact Analysis, and which are identified to be validated according to the test coverage.

3.3.8 Usage of laboratory

The test of the TCMS System shall rely entirely on the test in the laboratory where the test rack hosts the original TCMS devices for the formal test of the TCMS System including bus systems.

The test rack also provides the environment for the test execution (stimulation, observation of signals) and the Environment Model which simulates the environment of the TCMS System at the vehicle to the extent necessary by the test.

The test of the SW Application is preferably performed at the Virtual CCU which provides an emulated environment which runs at a PC, and which is compliant to the HW and firmware of the corresponding CCU type. Building of a Virtual CCU is done by utilising the original SW of the SW application without any modification.

If it is not possible to use a virtual CCU due to a strong dependency with the real firmware or HW, the test shall be done using the real TCMS device.

Table 3 shows the test setups used for the different test levels.

Table 3: Test Setup used for formal tests.

Test Level	Test Specifics	Test Equipment
TCMS System Test		Test Rack
TCMS Integration Test		Test Rack
TCMS SW Test	SW application at CCU	Virtual CCU
	HW related functions	Real TCMS device
	SW application at HMI	Real HMI

The compliance of the behaviour of the test rack with the behaviour of the train is demonstrated by the laboratory qualification. The qualification relies mainly on the correct implementation of the environment model according to the specifications of the other systems with an optionally comparison of selected test results obtained at the test rack and obtained at the vehicle.

Table 4 shows the test environments used for developer test.

Table 4: Test Environment used for developer test.

Test Level	Test Specifics	Test Equipment
TCMS System Test	Usage of HMI	Test Rack
		Virtual Train
TCMS Integration Test		Test Rack
	Integration Regression Test (functional test)	Virtual Train
TCMS SW Test		Virtual CCU

	HW related functions	Real TCMS device
	SW application at HMI	Real HMI

In the context of Adeptness, one can map the whole process of testing at the software application level to Software or Model in the loop (MIL/SIL) level while testing on the test rack is synonymous with testing at Hardware in the loop (HIL) level. This is shown, together with Adeptness specific scientific and technical objectives (STOs) in Figures 5 & 6 below.

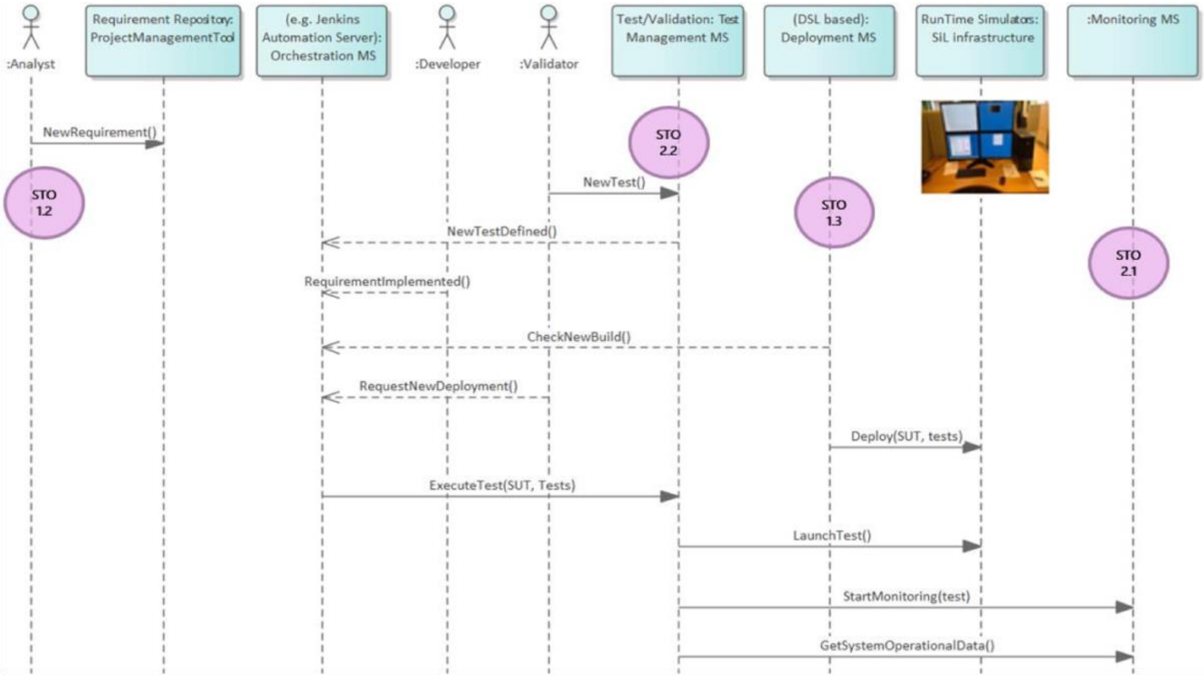


Figure 5. TCMS development at MIL/SIL level and mapping with Adeptness STOs.

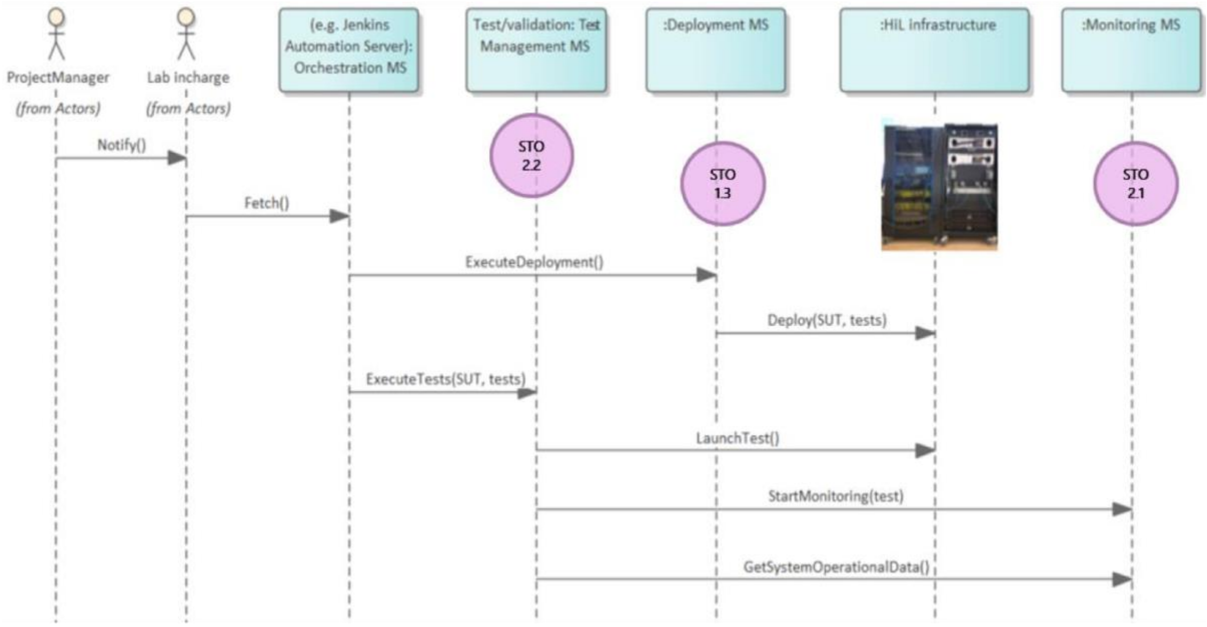


Figure 6. TCMS development at HIL level and mapping with Adeptness STOs.

3.3.9 Continuous Build & Test

The continuous build & test is a complementary informal measure to demonstrate continuously the correctness of the TCMS System to a certain extent already during the development phase.

The continuous build & test is based on an automated build system that takes advantage of the incremental development of the TCMS system. Continuous build & test is done at the level of the TCMS System and utilises the Virtual Train as the test environment. It assumes that a Virtual Train can be built successfully from the current snapshot of the source code and the configuration data of the TCMS system together with the Environment Model.

The continuous build & test starts as soon as a minimum part of the TCMS System has been implemented which allows the start-up of the TCMS devices. The test executed during the continuous build & test is at the minimum the TCMS Integration Regression Test. Due to the incremental development, the test provides first a reduced set of test steps which is extended according to the progress of the development of the TCMS System.

The continuous build & test allows fast feedback for the SW developers about the quality of their contribution to the overall TCMS SW or its configuration. Therefore, it is done at least every night.

In addition, the build can be performed by any developer to individually check the impact of their contribution to the overall system behaviour.

The continuous build & test also allows an early development of test scripts because it provides an environment where the test scripts can run without additional measures (test of the test).

3.3.10 Developer Test

The developer test is a complementary informal measure used by every developer to ensure the correctness of a contribution to the overall TCMS SW and its configuration before delivery. It utilises the automated build system and the test environment already in place for the TCMS SW Test and the TCMS System Test at the Virtual CCU or the virtual train. That is, a developer may trigger an individual build of the Virtual CCU or

Virtual Train to check the impact of their contribution to the behaviour of a SW application or the overall system behaviour.

3.4 Environment/Infrastructure/Tools

3.4.1 Test Setup

The test of the TCMS System uses test setups utilising the following test equipment:

- Virtual CCU
- Virtual Train
- Test rack

Figure 7 shows the test setup utilising the Virtual CCU and the test executors for automated or manual test execution.

The Virtual CCU is a tool which runs at a test PC, bases on the Soft-TCMS technology and exposes a virtualized environment for the SW applications at an CCU. It provides the same functional environment as the real CCU (process scheduling, OS calls, communication means) including the capability for signal stimulation, observation and debugging. The Virtual CCU can be built directly from the TCMS project files.

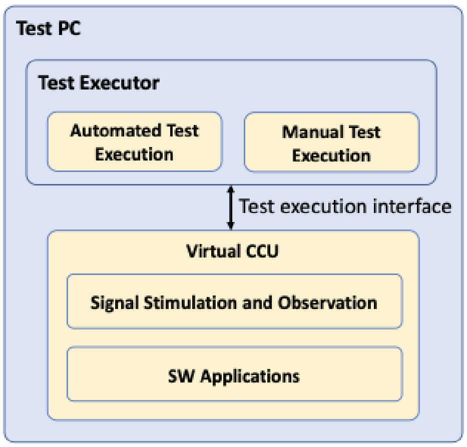


Figure 7. Test setup with virtual CCU.

Figure 8 shows the test setup utilising the test rack, the Virtual Train, and the test executors for automated or manual test execution in a single-unit operation environment.

The test rack is a setup which hosts the TCMS System of one vehicle comprising the real TCMS devices. In addition, the test rack hosts the test execution environment and supporting tools (e.g., remote controllable switch to power TCMS devices, coupling controller). Like the Virtual CCU, the Virtual Train is a tool which runs at a test PC, bases on the Soft-TCMS technology and exposes a virtualized environment for the SW applications at an CCU as well as the configuration of the additional TCMS devices (switches, gateways) including the TCMS bus systems and the test execution environment. The Virtual Train can be built directly from the TCMS project files and the configuration of the test execution environment as used for the test rack.

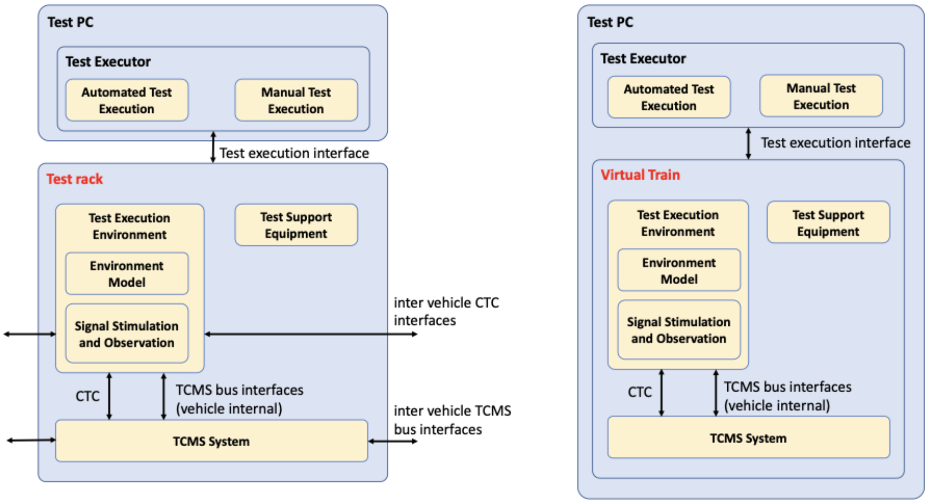


Figure 8. Test setup utilising test rack and Virtual Train.

Figure 9 shows the setup used for the test at the system level if the interworking between different TCMS Systems at different vehicles needs to be checked. The setup is chosen because a complete simulation of a vehicle at the inter TCMS System interface is not in place. In case of a test rack, the test is done by utilising multiple test racks where each test rack represents one vehicle. In case of the Virtual Train, the Virtual Train itself provides the virtualization of multiple vehicles.

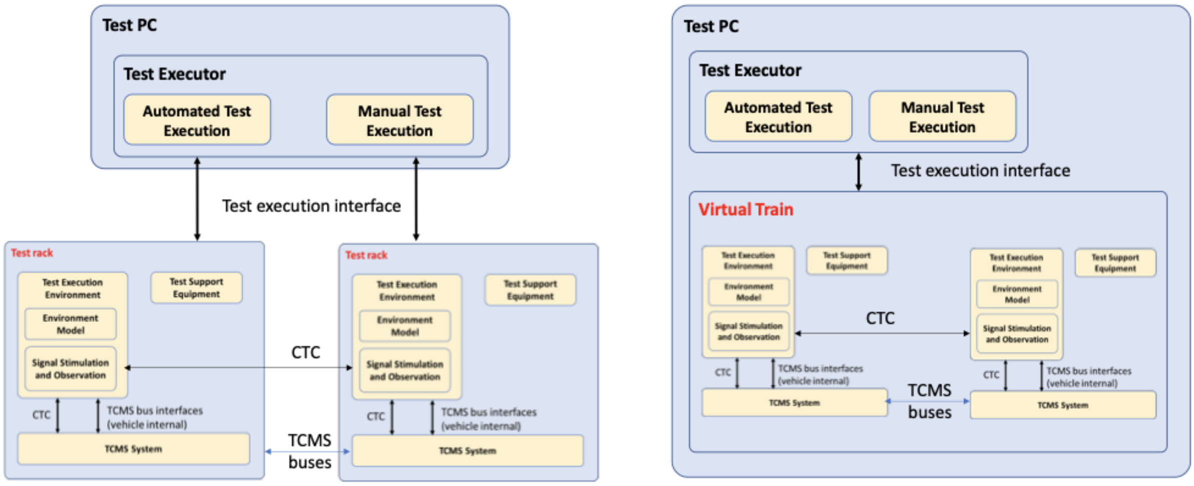


Figure 9. Test setup utilising test rack and Virtual Train for multiple-unit operation.

3.4.2 Test Executor

The test executor is a tool which triggers the execution of test steps, gathers the results, and provides the test records (test logs) which includes gathering of metadata also.

3.4.3 CSharp Bench

The CSharp-Bench allows to implement test scripts and execute them automatically or semi-automatically. The latter option allows manual intervention during automated test execution. The CSharp Bench can trigger the RTSIM to observe or stimulate signals at TCMS according to the need of the test script which includes the observation and manipulation of visual elements (capture screenshots and press soft buttons). The CSharp-Bench can be used for all levels of testing. The observation and stimulation of visual elements is a new method. The productive usage requires the qualification of the tool chain including proof of robustness. Introduction into existing projects requires upgrade of existing test script frameworks.

3.4.4 Manual Test Framework

The manual test framework supports the manual test by providing means to design a graphical user interface which allows to stimulate and observe signals similar as it is possible with the CSharp-Bench. It may be used to provide a graphical simulation of the cab.

3.4.5 Test Execution Framework – RTSIM

The test execution environment is a tool called RTSIM which interfaces between the test executor and the TCMS System for stimulation and observation of the TCMS System. In addition, it provides a runtime environment for the Environment Model which simulates the vehicle environment of the TCMS System. The RTSIM is a component which simulates in conjunction with the Environment Model the environment of the TCMS system and which can stimulate the input and monitor the output of the TCMS system on request of the CSharp Bench. For that purpose, the RTSIM interfaces with the TCMS System via the TCMS bus systems and the test PC hosting the CSharp Bench via IP.

The RTSIM comes in two flavours, as HW component which allows a real time operation at a test rack or as SW component which is part of the Virtual Train.

The current RTSIM device is shown in Figure 10.

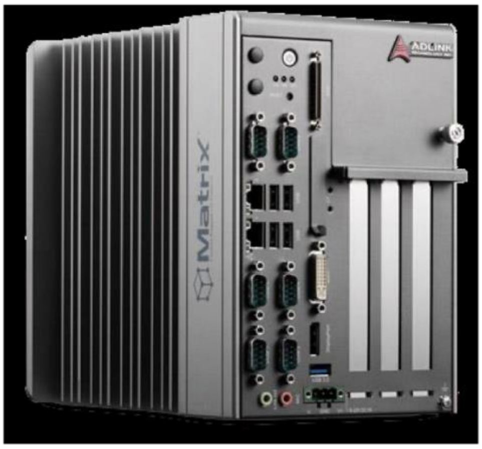


Figure 10. RTSIM hardware device.

3.4.6 Environment Model

The Environment Model runs on the RTSIM and is used to simulate the real environment of the TCMS system to the extent necessary by the TCMS system test but does not provide the simulation of all aspects of the subsystem. For that purpose, the Environment Model can automatically set a large portion of the TCMS input signal vector without any support of the test scripts. The test script needs to set only the signals relevant to the test case. The Environment Model can also be stimulated by the test script itself and can execute complex signal flows, which individual subsystems or a group of sub-systems would generate upon external stimulation. The same also works in the opposite direction. Complex reactions of the TCMS system can be verified by the Environment Model.

The conventional train circuits part of the environmental model is completely provided by direct generation from the technical documentation. This may provide more functions than required for the test, but with a higher reliability than would be possible manually. Thus, uncovering malfunctions in the conventional train circuits is possible even if it is not directly part of the TCMS test.

The Environment Model is defined by Simulation Specification which relies on the Interface Control Documents and Functional Vehicle Design Specifications of the subsystems and the schematics of the conventional train circuits.

The Soft TCMS Environment is a tool chain for software-in-the-loop tests. This environment makes it possible to execute and test CCU and CCUS code on a normal Windows PC. By that it is possible to equip every software developer or tester with his own Virtual Train and consequently making the testing much more efficient.

3.4.7 Virtual Train

The Virtual Train Configuration Wizard is a tool with a graphical user interface designed to help users set up a valid runtime environment for Virtual Trains. It interfaces with the various executables and configuration files and options involved.

3.4.8 TCMS Test Rack

Not all requirements can be easily tested in simulation environments. Complex interfaces and performance requirements mean that they can only be completely tested in the real environment, see Figure 11. It must also be proven that the simulation environment corresponds to the real behaviour of the TCMS devices and the TCMS system.

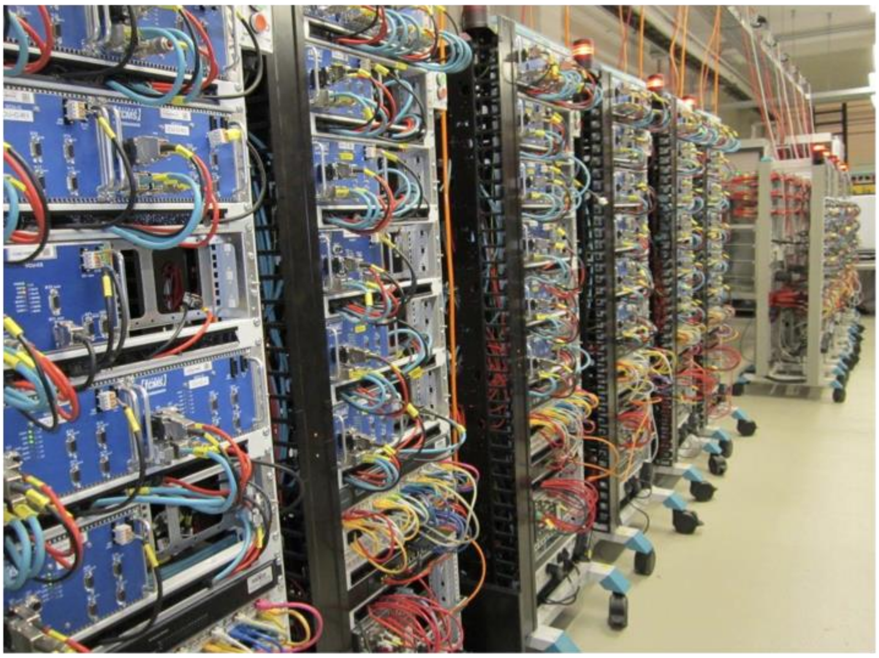


Figure 11. TCMS Test Racks.

4 DESCRIPTION OF THE USE-CASE AND METHODOLOGY OF SOLUTIONS

As was mentioned before, Bombardier's system under test is the TCMS. TCMS is the centre of the distributed system that controls the flow of information both on the train between the different subsystems (like heating and ventilation) and between the train and the ground. A general architecture of the TCMS is shown in Figure 12 below.

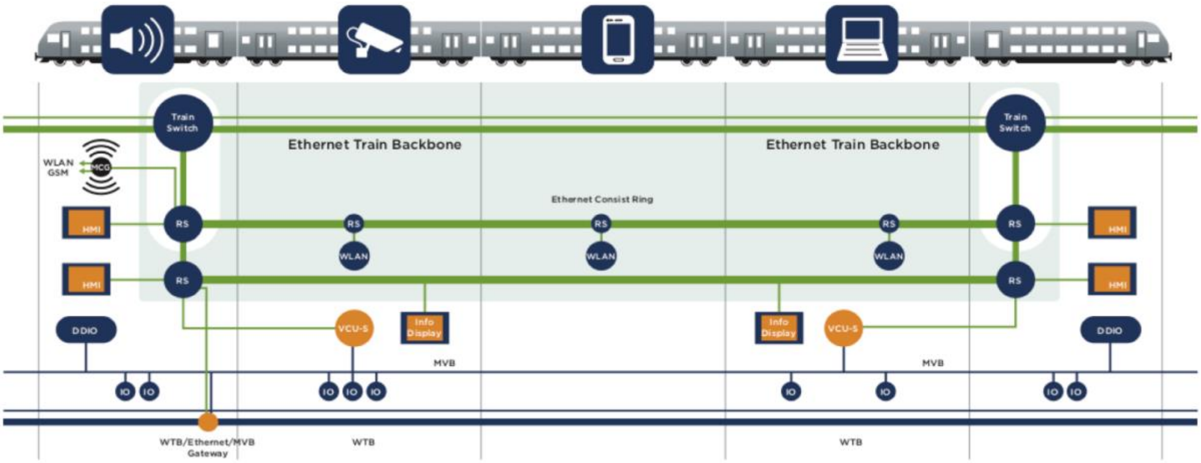


Figure 12. A general architecture of the system under test, TCMS, from Bombardier's use case.

For developing Adeptness solutions with Bombardier's use case, an ongoing delivery project was selected. It is called the R151 Movia Metro Train project. Movia is a family of metro trains. Movia metros are operational in several countries, including Canada, China, India, and Sweden. An example Movia metro in Sweden is called C30 and is shown in Figure 13.

Figure 13. C30 Movia train arriving at a station in Sweden.

R151 is the upcoming 7th generation electric-driven rolling stock that will be introduced in Singapore's mass rapid transit system. These metro trains are expected to enter service from 2023.

When it comes to the optimization of the lifecycle of the development, testing and operation of TCMS, it is convenient to think in terms of the model that is in place at the company. This is shown in Figure 14 below.

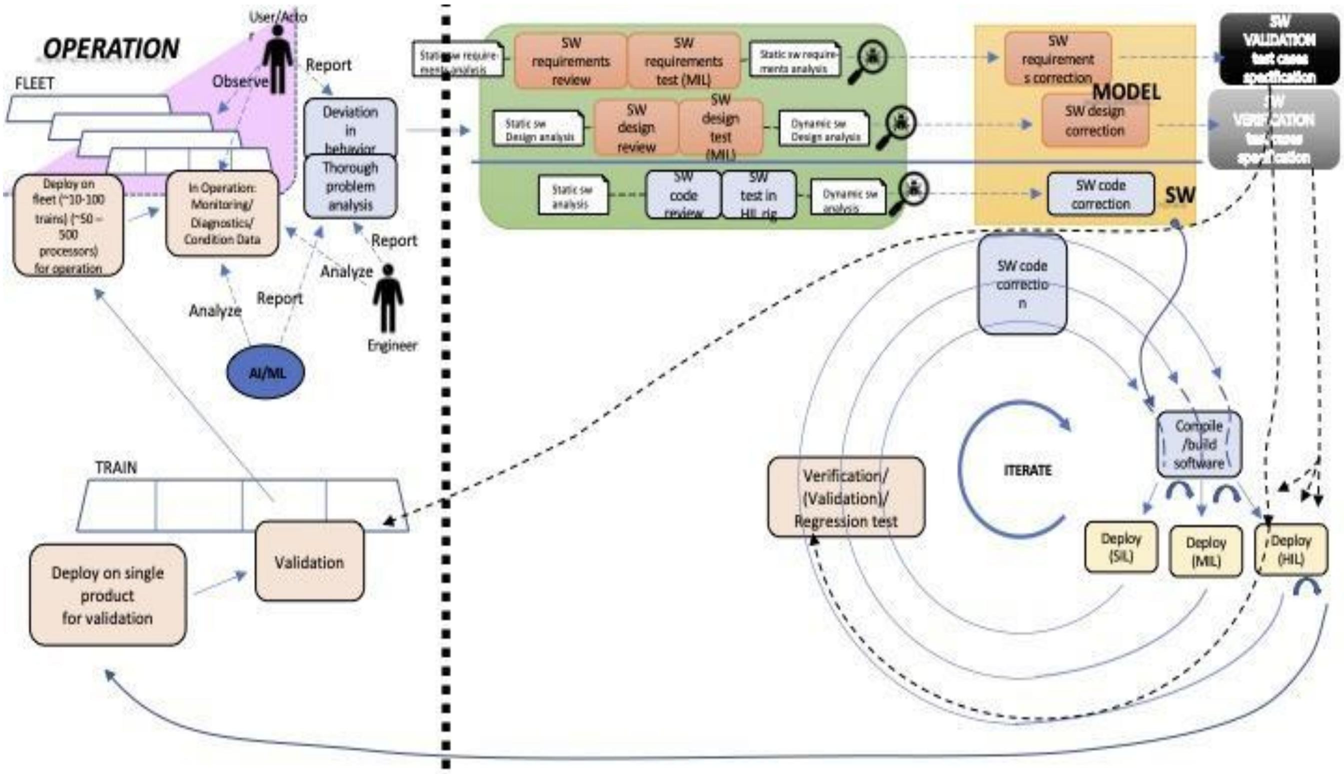

OPERATION
User/Acto
FLEET
Observe
Report
Deviation in behavior
Thorough problem analysis
Deploy on fleet (~10-100 trains) (~50 – 500 processors) for operation
In Operation: Monitoring/ Diagnostics/ Condition Data
Report
Analyze
Report
Analyze
Engineer
AI/ML
TRAIN
Deploy on single product for validation
Validation
Static sw requirements analysis
SW requirements review
SW requirements test (MIL)
Static sw requirements analysis
Static sw Design analysis
SW design review
SW design test (MIL)
Dynamic sw Design analysis
Static sw analysis
SW code review
SW test in HIL rig
Dynamic sw analysis
SW requirements correction
MODEL
SW design correction
SW code correction
SW
SW VALIDATION test cases specification
SW VERIFICATION test cases specification
SW code correction
Compile /build software
ITERATE
Verification/ (Validation)/ Regression test
Deploy (SIL)
Deploy (MIL)
Deploy (HIL)

Figure 14: The V-model of design and operation of the TCMS.

This figure separates the operation phase from the design phase. The design phase includes the software requirements specification and review, software design and review, implementation and review and the different cycles of the test and validation in different simulation platforms: MIL, SIL and HIL. On the left side of the figure is the operation phase, that includes validation on the train and fleet side.

This use case is set up to primarily focus on the improvement on the validation front but other Adeptness microservices, such as deployment, will also be evaluated.

4.1 Involved Adeptness Solutions for KPI measurement

The Adeptness technical solutions that have been evaluated for KPI measurement on the railway use case are as following:

- Model-based test generation and execution on Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) execution platforms.
- Continuous monitoring using T-EARS.

4.2 Use case Selection

To investigate the impact of adeptness solutions, we conducted a series of industrial case studies. We modelled the fire indication sub-system of an on-going TCMS project at BT. Throughout our investigations, we also consulted with the test team at BT to understand their test generation process and analyzed their test artefacts to compare them with artefacts generated using MBT.

4.2.1 Case Selection

In discussions with BT, we selected an on-going TCMS development project for the MOVIA vehicle product family as a 'case'. The MOVIA is BT's family of metro train cars, and they are currently operational in various metro rail networks across the globe. The 'unit of analysis' translates to the SUT for the selected software development project, which is the TCMS. The following sub-section provides further details on the SUT.

4.2.2 SUT

The SUT is a TCMS that is currently being developed with a testing process highly influenced by safety standards and regulations. TCMS is a high capacity, infrastructure backbone built upon an open standard IP-technology that allows easy integration of all control and communication functions on-board the train. It is considered as the centre of the distributed system that controls the flow of information both on the train between the different subsystems like converters, doors, heating, ventilation, and air-conditioning and between the train and the ground. TCMS is designed to perform all tasks related to modern vehicle control.

As shown in Figure 15, the TCMS consists of multiple TCMS devices of specific types, which are connected internally via the system Multi-function Vehicle Bus (MVB) and Ethernet Consist Network (ECN). Both bus systems are also used to interconnect other systems at the vehicle. The Modular Input/Output (MIO) devices are used to interconnect with the conventional train lines of the vehicle whereas Modular Input/Output Unit – Safe (MIO-S) deals with safety critical Input/Output analogue signals. The connection between the TCMS instances of different vehicles is established via the bus systems Wired Train Bus (WTB) and Ethernet Train Bus (ETB). The Centralized Traffic Control (CTC) is used to consolidate the train routing decisions. A TCMS device of a certain type may appear multiple times depending on the required scalability or redundancy demands for a certain product. TCMS uses Central Control Units (CCU) to control multiple train functions; CCUO controls the basic functions, CCUS executes the safety critical functions and CCUD is a logical component that manages the diagnostic history database. The HMIs (Human Machine Interfaces) are also foreseen for each driver's desk.

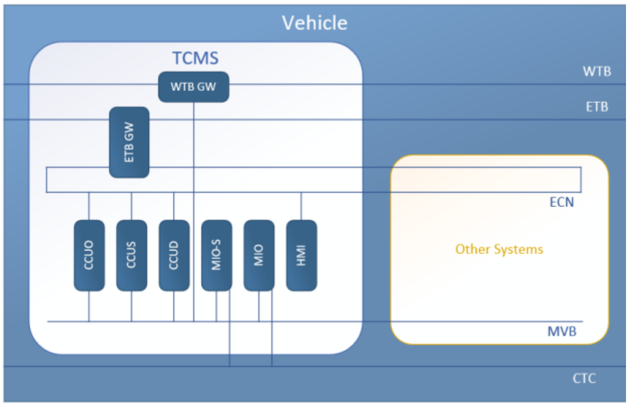


Figure 15. Illustration of TCMS architecture.

4.2.3 Industrial Scenario for Modelling and Test Generation

In our case studies, the selected requirements of the SUT were related to the multi-purpose TCMS devices to detect fire in the cab (the driver's compartment). Figure 16 represents an architecture of the fire detection system used by TCMS. The fire detection system of TCMS uses two instances of Fire Detection Control Units (FDCUs) to detect fire based on their current states i.e., Master or Slave. A FDCU is considered as 'Master' if two of its signals hold the value as true. Both FDCUs communicate with smoke and fire sensors to receive signals indicating two types of fire i.e., internal, and external. The TCMS MIOS device receives these signals from both FDCUs and communicates with CCUS via the MVB network. The CCUS computes the logic based on system requirements and responds with an output signal. The MIOS receives the corresponding output signal from the CCUS and indicates the type of fire on the driver's desk.

On the real train, the FDCU devices communicate with fire and smoke sensors. When a sensor detects smoke/fire, FD- CUs send signals informing TCMS MIO-S device. The MIO-S device communicates using MVB network with the CCUS device. CCUS computes some logic (based on the system requirements) and provides an output signal. This signal is transmitted to the MIO-S device to light a LED as indication of fire/smoke on the driver's desk, via electrical wiring.

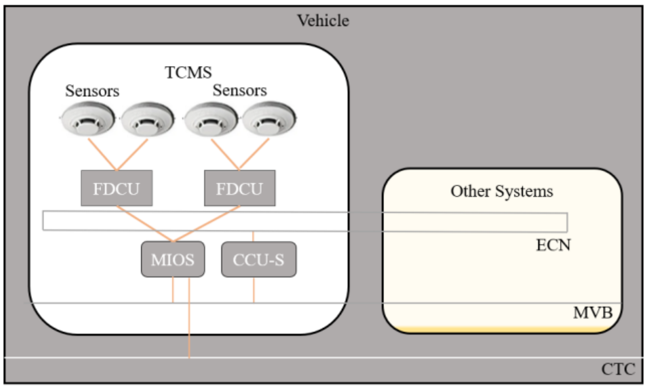


Figure 16. Architecture illustration of the fire detection subsystem of TCMS.

4.2.4 Data collection procedures

The data collection for the case study and inputs required for modelling was performed using two different data collection techniques: direct contact with the testing team at BT and independent analysis of the artefacts produced by practitioners.

Through consultations with the testing team at BT, the SUT to be modelled was selected from an on-going development project, for which the practitioners developed test cases manually and provided access to relevant artefacts (such as requirements specification and test specification) related to the SUT. One member from academia spent numerous hours to understand BT's test process, SUT and the test cases written by industrial professionals. He also underwent trainings by an experienced test lead at BT to get acquainted with the test procedures and the software-in-the-loop testing process at BT. The academic team coordinated efforts to bring clarity in understanding the SUT and the testing process at BT. This involved numerous email exchanges and meetings between the industrial and academic parties as well as thorough scrutiny of the test related artefacts by the academic team. The manual test data was collected by using a post-mortem analysis of the available artifacts.

The engineering processes of software development at BT are performed according to safety standards and regulations (e.g., EN 50128 standard is used for designing test cases). Each test case should contribute to the demonstration that a specified requirement has indeed been covered and satisfied. Executing test cases on TCMS is supported by a test framework that includes the comparison between the expected results with the actual outcome.

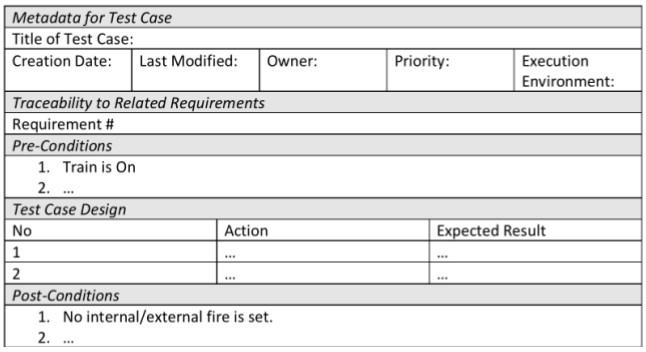
4.2.5 Requirements specifications

The requirements for the specific industrial scenario contained all the details about under what circumstances the TCMS should indicate external and internal fires in the cabs. These requirements have been specified in natural language but follows a pattern of 'Given-Then-Within' scenario description, like the 'Given-When-Then' template as common in Behaviour Driven Development. The 'Given' clause specified the actions, 'Then' clause specified the observable outcome and 'Within' clause specified the timing constraints of each requirement. The 'Given' and 'Then' clauses for a requirement occasionally included multiple Boolean operators (AND/OR) to join conditions together. The requirements thus followed the following template:

GIVEN {Statement 1} AND/OR {Statement 2} **THEN** TCMS shall {Statement 3}
WITHIN {**t** Seconds}

4.2.6 Test specification

The main component in the test specification document included manually written test cases in natural language, corresponding to the specified requirements. An example of a test specification is shown in Figure 17. Each test case was designed in a series of test steps, where for each test step, the action and the expected result was specified. In addition, the pre-conditions and the post-conditions for each test case were also specified, along with essential metadata (such as priority, execution environment etc.) and traceability to related requirements.

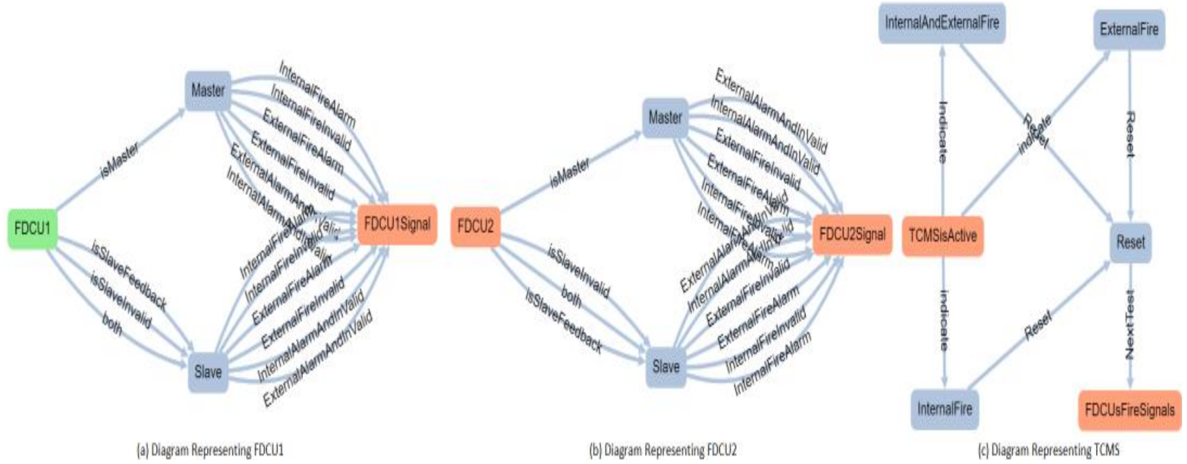


Metadata for Test Case				
Title of Test Case:				
Creation Date:	Last Modified:	Owner:	Priority:	Execution Environment:
Traceability to Related Requirements				
Requirement #				
Pre-Conditions				
1. Train is On 2. …				
Test Case Design				
No	Action	Expected Result		
1	…	…		
2	…	…		
Post-Conditions				
1. No internal/external fire is set. 2. …				

Figure 17. An illustration of a manually written test specification.

4.2.7 Modelling of SUT

We have modelled the expected behaviour of fire detection sub-system in GraphWalker (GW) studio by exploring and understanding the requirements and test specifications as well as by getting continuous input from the testing team at BT. GW studio is one of the versions of GW to create and validate the EFSM model of the system. The model created in GW studio consists of nodes, edges, and guard conditions. Nodes represent the states, edges represent the transition taken by the system from one state to another and guard conditions are Boolean expressions representing the expected behaviour of the model. To model the fire detection system, we have identified all possible states, transitions and guard conditions of the system based on requirements and test specification. Figure 18 depicts the EFSM model in three diagrams representing the SUT.



(a) Diagram Representing FDCU1
(b) Diagram Representing FDCU2
(c) Diagram Representing TCMS

Figure 18. EFSM Model representing the Fire Detection System controlled by BT's TCMS

Two diagrams (a) and (b) represent the FDCUs and one diagram (c) represents the TCMS as black box. FDCU1 is an initial as well as shared node of the model. FDCU1Signal, FDCU2, FDCU2Signal, TCMSisActive and FDCUsFireSignals are rest of the shared nodes of the models. These shared nodes are used by the GW to traverse between different models while validating the model and generating test cases. Master, Slave, InternalFire, ExternalFire, InternalAndExternalFire and Reset nodes represent the other states of the SUT based on requirements and test specifications. Similarly, 39 edges were added based on the expected behaviour of the SUT.

4.2.8 Fault Injection in EFSM Model

Due to unavailability of source code, we have injected some faults based on insertion and omission of mutant operators [4] in the model to produce faulty test scripts. We have used three mutant operators (output exchanged, change in guards/programming mistake and state missing) and created three versions of the faulty model. In first version of the faulty model, we exchanged the output values of internal and external fire such that if a system is supposed to indicate the internal fire, it will indicate the external fire and vice versa. Similarly, in second version, we made some changes in the guard conditions while we removed the Master state of one of the FDCUs in the third version to make these models contradictory to the original specification.

4.3 Methodology

4.3.1 Methodology for Experimentation on Requirement and MC/DC Coverage, Differences and Overlaps Between Test Cases and Performance Efficiency

Figure 19 represents an overview of the methodology that we designed for one of the investigations.

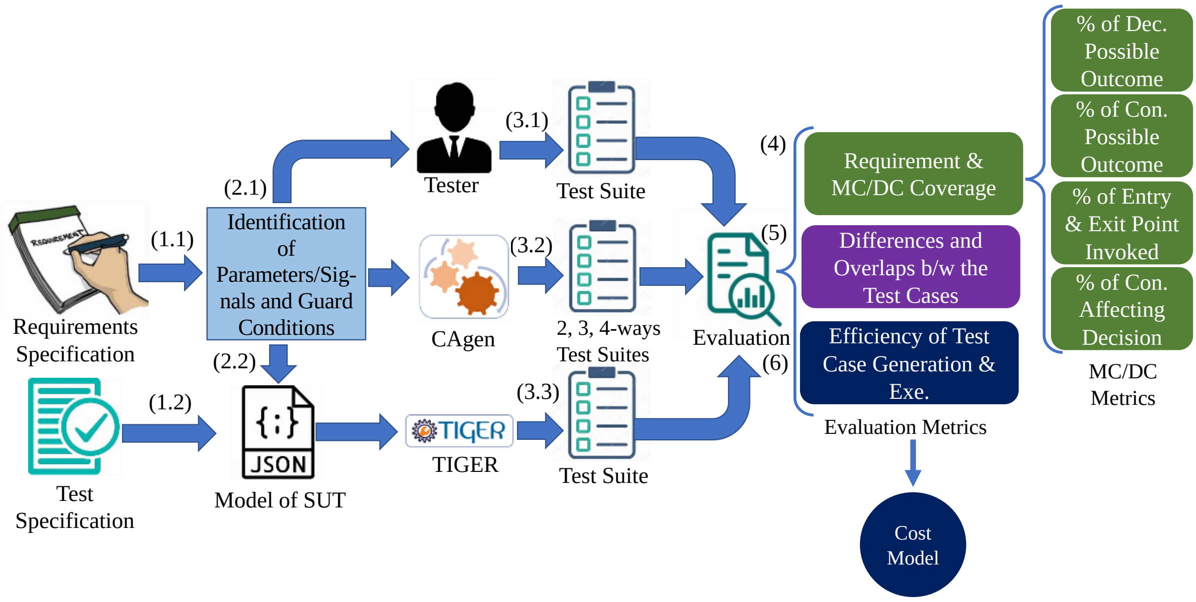

Requirements Specification
(1.1)
Identification of Parameters/Signals and Guard Conditions
(2.1)
Tester
(3.1)
Test Suite
(4)
CAgen
(3.2)
2, 3, 4-ways Test Suites
(5)
Evaluation
(2.2)
Test Specification
(1.2)
Model of SUT
JSON
TIGER
(3.3)
Test Suite
(6)
Requirement & MC/DC Coverage
Differences and Overlaps b/w the Test Cases
Efficiency of Test Case Generation & Exe.
Evaluation Metrics
% of Dec. Possible Outcome
% of Con. Possible Outcome
% of Entry & Exit Point Invoked
% of Con. Affecting Decision
MC/DC Metrics
Cost Model

Figure 19. An overview of the experimental methodology for one set of experiments.

The sequence of steps in our methodology are represented with numbers in Figure 19 and described below:

- (1.1) & (1.2) the analysis of requirements specification and test specification using an industrial case study of TCMS.
- (2.1) & (2.2) identifying the parameters involved in defining the behaviour of the SUT and creating a model of the system in JSON format using GW.
- (3.1), (3.2) & (3.3) generation of manual test suite by testers at BT and automated test suites by CAgen and TIGER.
- (4) evaluating the test suites based on requirement and MC/DC coverage metrics.
- (5) measuring the efficiency of test suites based on a cost model.
- (6) assessing the differences and overlapping between the test suites.

4.3.1.1 Test Suite Creation

We took manual test suite created for the selected requirements by the testers at BT according to EN 50128 and EN 50657 safety standards and regulations. The test suite consists of a set of test steps specifying test inputs, expected outputs and timing constraints in natural language based on specification-based testing. The design process of the test suite ensures that each requirement of the system has been covered and executed. After the designing of test cases, test scripts are created manually, executed on Software- in-the-Loop (SiL) level by a testing framework and a test verdict is generated.

For the generation of test suites for CT and MBT, we have utilized CAgen and TIGER tools respectively. Particularly, we have considered all the signals used by the SUT for communication be- tween FDCUs, sensors, and TCMS as parameters and generated 2-ways, 3-ways, and 4-ways test suites, in the form of decision tables, through the CAgen tool. Furthermore, TIGER generated the test suite based on FSM model using 100% edge coverage and through random algorithm. It is important to mention that test scripts are developed manually in case of CT but are generated automatically in TIGER for MBT. The reason behind limiting CT at 2, 3 and 4-ways interaction strength is the combinatorial explosion of test cases generated for complex systems with higher strength [5]. Similarly, the random generation algorithm and edge coverage criterion of GW provides adequate model and requirement coverage, which is an important metric for our industrial partners.

4.3.1.2 Coverage Assessment

A coverage metric is used to analyse the quality of a test suite and measures the degree to which the code has been exercised by a test suite. The testing of safety-critical subsystems of TCMS at BT need to follow EN 50128 and EN 50657 standards. These standards require that the code related to each requirement should be executed by the test suite. Hence, in this study, we have defined two metrics, i.e., requirement coverage and MC/DC to analyse the design and structural coverage provided by a test suite.

The requirement coverage requires that requirements of a system are exercised by test cases at least once [6]. Hence, we identified the total number of requirements specified in requirement specification and analysed the requirements covered by each test suite to measure the requirement coverage using the following formula:

$RequirementCoverage(RC)\% = (NRC/TNR)x100$

Where NRC represents number of requirements covered by test cases and TNR depicts total number of requirements.

As the requirement coverage does not ensure the coverage of logical conditions and decisions of a system from the implementation perspective, we have also used MC/DC to examine the structural coverage of the test suites. We have defined the MC/DC metric according to the coverage points defined for a MC/DC analysis tool in [7]. Moreover, we have also analysed each condition of MC/DC metric to investigate the least dominant condition affected by each technique to generate MC/DC adequate test suites. We used the following formulas to determine the MC/DC metric of test suites in percentage:

- Condition's possible outcomes (C) % = (No. of conditions having all possible outcomes/Total no. of conditions) x 100
- Decision's possible outcomes (D) % = (No. of decision having all possible outcomes/Total no. of decisions) x 100
- Conditions affecting decisions independently (AC) % = (No. of conditions affected the decisions / Total no. of independently effecting conditions) x 100
- Entry and exit point invoked (E) % = (Sum of no. of entry and exit points invoked/Sum of total no. of entry and exit points) x 100

Hence, we have calculated the overall MC/DC of a test suite using the formula below.

$$MC/DC\% = (C + D + AC + E)/4$$

4.3.1.3 Efficiency Evaluation Criteria

The efficiency of testing techniques can be measured by analysing the cost/time benefit of each technique based on direct and indirect costs [8]. The direct costs are the expenditures that are directly associated with testing techniques such as time for test creation and execution of test cases. The indirect costs are the expenditures that are indirectly associated with the testing techniques such as maintenance time, development of testing tools etc. Hence, to analyse the efficiency of a testing technique, we have developed an economic cost model based on direct costs and have neglected the indirect costs as the effect of indirect costs diminishes over time, for example, development of a tool is only a one-time effort. There- fore, in this model, we have considered the following parameters based on variable cost factors and time required to complete the activities defined in [9] and [10] as well as the actual activities adopted by testers at BT.

The cost of testing activities that we have considered in our efficiency evaluation metrics for automated and manual testing include analysis of requirements specification (C_R), test suite development (C_{TS}) and test execution (C_E). The requirements analysis involves understanding the behaviour of a system by thoroughly investigating functional (C_{Rf}) and non-functional (C_{Rnf}) requirements as well as reviewing (C_{Rr}) these requirements to determine un- feasible requirements before the start of the development process. After requirements analysis, test cases are designed followed by the development of test scripts and identifying concrete values (e.g., technical signal names) (C$_{S}$). The test scripts containing test steps (C$_{Ets}$) are then executed along with pre and post conditions (C$_{Epre}$, C$_{Epost}$) to generate test verdicts. However, in case of MBT in our context, a model is created (C$_{M}$) by identifying the states, transitions (C$_{Mcst}$), guard conditions (C$_{Mcg}$) and test scenarios (C$_{Mct}$) based on requirements analysis. Moreover, the test suite (C$_{Gts}$) is generated automatically by creating an XML file containing information about the signals (C$_{Gf}$) (e.g., technical signal names, primary signal names and data type etc.) and ensuring the correctness and conformance of a model (C$_{Mv}$) with system requirements. On the other hand, in case of manual testing, a tester designs test cases (C$_{Tc}$) and scripts (C$_{Tsd}$) manually by specifying and developing system libraries (C$_{sl}$).

4.3.1.3.1 Parameters for cost calculation of MBT

C$_{R}$ = Cost for Requirements Specification Analysis (C$_{Rf}$ + C$_{Rnf}$ +C$_{Rr}$)
C$_{Rf}$ = Cost to analyse functional Requirements
C$_{Rnf}$ = Cost to analyse non-function Requirements
C$_{Rr}$ = Cost to review requirements
C$_{M}$ = Cost for modelling the SUT (C$_{Mc}$ + C$_{Mv}$)
C$_{Mc}$ = Cost to create and modify model (C$_{Mcst}$ + C$_{Mcg}$ + C$_{Mct}$)
∗ C$_{Mcst}$ = Cost to identify states and transitions ∗ C$_{Mcg}$ = Cost to identify guard conditions
∗ C$_{Mct}$ = Cost to identify test scenarios
C$_{Mv}$ = Cost to validate the correctness of model
C$_{T\ S}$ = Cost to generate MBT test cases and test scripts (C$_{Gf}$ + C$_{Gts}$)
C$_{Gf}$ =Cost to prepare XML file containing logical and technical signal names
C$_{Gts}$ = Cost to generate MBT abstract test cases and concrete test scripts
C$_{E}$ = Cost to execute MBT test scripts
C$_{Epre}$ = Cost to execute pre-conditions
C$_{Ets}$ = Cost to execute test steps
C$_{Epost}$ = Cost to execute post-conditions
Based on the above defined parameters, the cost for MBT (C$_{MBT}$) can be summed below:

$$C_{MT} = C_R + C_M + C_{TS} + C_E$$

4.3.1.3.2 Parameters for cost calculation of manual testing

C$_{R}$ = Cost for Requirement Specification Analysis (C$_{Rf}$ + C$_{Rnf}$ +C$_{Rr}$)
C$_{Rf}$ = Cost to analyse functional Requirements
C$_{Rnf}$ = Cost to analyse non-function Requirements
C$_{Rr}$ = Cost to review requirements

C$_{TS}$ = Cost for the development of test suite (C$_{S}$ + C$_{Tc}$ + C$_{Ts}$)

C$_{S}$ = Cost to Identify technical signals to use in test design

C$_{Tc}$ = Cost for writing/designing test specification – test cases

C$_{T\ s}$ = Cost for writing concrete test scripts (C$_{T\ sl}$ + C$_{Tsd}$)

∗ C$_{Tsl}$ = Cost for specifying and developing library functions

∗ C$_{Tsd}$ = Cost to develop test scripts

C$_{E}$ = Cost to execute test scripts

C$_{Epre}$ = Cost to execute pre-conditions

C$_{Ets}$ = Cost to execute test steps

C$_{Epost}$ = Cost to execute post-conditions

Based on the above defined parameters, the total cost for manual testing (C$_{MT}$) can be summed below:

$$C_{MT} = C_R + C_{TS} + C_E$$

4.3.1.4 Parameters for cost calculation of CT

The testing activities required for CT include all the activities like manual testing, just that test case development is automated in CT in our case. Hence, we have used the parameters of manual testing to determine the total cost for CT (C$_{CT}$) too.

$$C_{CT} = C_R + C_{TS} + C_E$$

4.3.1.5 Differences and Overlaps Assessment

To measure the differences and overlaps between the test suites produced by each testing technique, we investigated the similar test sets and created a Venn diagram to show the distribution and intersection of different test sets between test suites. A Venn diagram shows the measure of union for sets using the measure of intersections between them. Hence, the illustration of relationships between the elements of test suites in Venn diagram is done using Meta-Chart which is an online data visualization tool. We provided the total number of elements and intersections between different combinations of test suites to create the Venn diagram.

4.3.2 Methodology for Experimentation on Detailed Fault Detection Effectiveness, Sensitivity of Test Suites, and the Relationship Between MC/DC Coverage and Mutation Score

Figure 20 represents an overview of the methodology that we designed for another investigation.

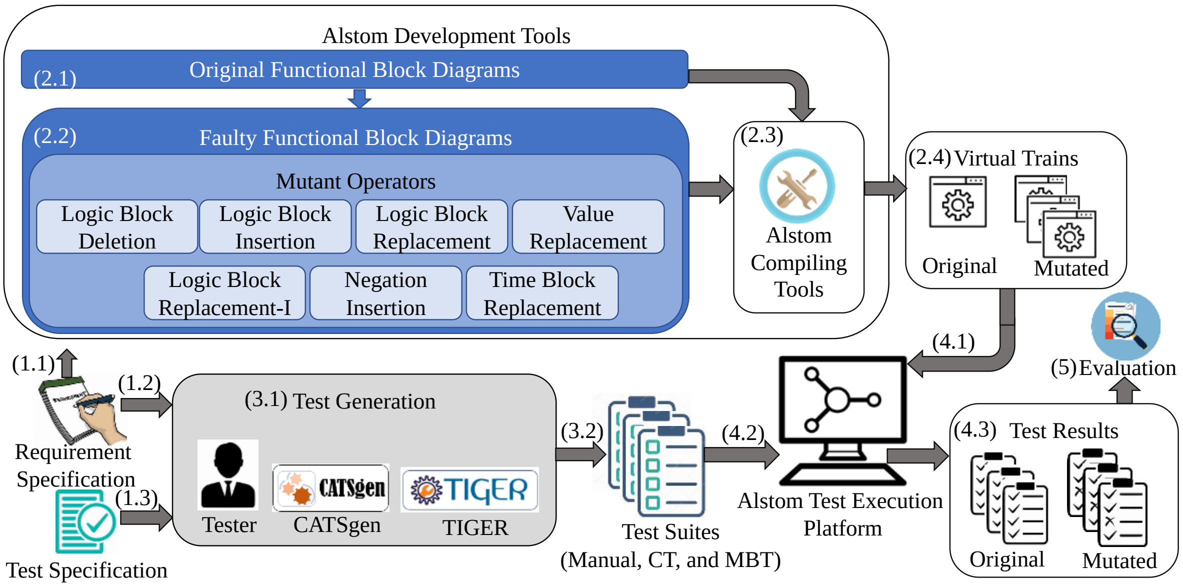


Figure 20. An overview of the experimental methodology for another set of experiments.

The methodology consists of five main steps:

(1) analysis of system requirements and test specification (i.e., Steps (1.1), (1.2), and (1.3)),

(2) development of mutated versions of the original program (i.e., (2.1), (2.2), and (2.3)),

(3) creation of test suites using each testing technique (i.e., (3.1) and (3.2)),

(4) deployment and execution of programs and test scripts to generate test results (i.e., (4.1), (4.2), and (4.3)),

(5) evaluation of test scripts based on test results.

4.3.2.1 Development of SUT and Mutation Injection

The general principle of mutation analysis is to examine the detection of injected faults in an original program. These faults are injected based on some mutant operators that can be used to mimic a programmer's common mistakes. So, we have used the original FBD program of the industrial case study developed by a developer at BT and created a set of mutants based on selected mutant operators manually. The selection of mutant operators is carried out by thoroughly reviewing the previous studies from the literature [11] [12] [13] that applies mutation analysis specifically for the evaluation of testing technique, test coverage, or to generate test suites using FBD programs. Moreover, the development of safety- critical FBD programs at BT requires only specific FBD operators. So, by considering the safety-critical industrial case study and FBD- specific faults, we have used seven mutant operators as follows:

- Logic Block Deletion Operator (LDO): to delete a logical block from the FBD program (e.g., deleting AND block).
- Logic Block Insertion Operator (LIO): to insert another logical block between the logical blocks or the input signals of the FBD program (e.g., inserting AND block between the output of two OR blocks).
- Logic Block Replacement Operator (LRO): to replace a logical block of the FBD program with another logical block of the same category (e.g., replacing the AND block with the OR block).
- Logic Block Replacement Operator-Improved (LRO-I): to replace the logical block of the FBD program with a logical block of the same category and another logical block with Boolean input (e.g., replacing OR with RS).
- Negation Insertion Operator (NIO): to insert the negation block at the inputs or outputs of other logical blocks.
- Time Block Replacement Operator (TRO): to replace the timer block with another timer block (e.g., replacing TOF with TON).
- Value Replacement Operator (VRO): to replace the constant value of a variable provided to a block with another value (e.g., replacing the timer variable from 3s to 6s).

After injecting the faults based on the selected mutants, we used the BT -specific compiling tools to generate the builds of a train containing the mutated program. We used these builds to generate the simulations of a train, also known as virtual trains, to execute the test scripts at the software- in-the-loop level.

4.3.2.2 Test Suite Creation

To evaluate the test suite developed using manual testing, we used the manually created test suite by a tester at BT for the selected subsystem. Whereas, for MBT and CT, we have utilized automated test script generation tools i.e., TIGER (Model-Based Test scrIpt GenEration fRamework) [14] and CATSgen (Covering Array Test Script generator) to generate the test suites, respectively. Both automated tools are based on different abstract test case generation tools but use a similar procedure, and format of the XML file containing the information about the signals (i.e., data type, logical and technical signal names) to generate the executable test scripts. Moreover, the test scripts developed by the selected techniques are implemented in the C# language. A brief description of the activities and tools for test suite generation of each testing technique is given in the subsequent subsections.

4.3.2.2.1 Manual Test Suite Creation

The testers at BT follow EN 50128 and EN 50657 safety standards and regulations to create the test suites based on Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) testing techniques. However, in some cases, MC/DC coverage criterion is also used for the creation the test suites for testing complex systems. The test cases are written in natural language and consist of a set of test steps specifying test inputs for the system, expected output, and response time according to each requirement specified in the requirement specification. Requirement coverage is considered a de facto criterion at BT for test suite development to ensure that each requirement has been covered and executed by the test cases. After the creation of test cases, BT -specific libraries are used to write the test scripts manually.

4.3.2.2.2 Model-Based Test Suite Creation

For model-based test suite generation, we have utilized GraphWalker studio version to create the FSM model of the SUT and provided the model to TIGER along with an XML file to generate C# implemented test scripts. TIGER uses the CLI version of Graph- Walker to generate the abstract test cases in JSON for- mat by traversing through the model elements based on the selected generator algorithm (e.g., random, quick random, etc.) and coverage criteria (edge, vertex, requirement, etc.). It contains the implementation of some defined mapping rules for logical and technical signal names as well as information specific to BT's testing framework (i.e., configurations, classes, and methods). After the generation of abstract test cases, it processes the data in a JSON file and utilizes the mapping rules along with testing framework-specific information to generate concrete test scripts. Hence, we generated the test suite by selecting the 'random' generator algorithm and 100% edge coverage criteria using TIGER.

4.3.2.2.3 Combinatorial Test Suite Creation

There exist multiple combinatorial test generation tools to generate test cases using different algorithms and combinatorial interaction strengths [15]. However, these tools can only be used to generate abstract test cases in the form of a covering array. To execute the test cases on the BT -specific testing framework, these test cases need to be concretized and implemented in the C# language. Hence, to generate the combinatorial-based test suites, we have developed our own CT test script generator called CATS-gen based on a state-of-the-art combinatorial test generation tool known as CAgen. CAgen is an open- source tool and available in two versions (i.e., online web GUI and offline command-line) with comparatively high performance than other combinatorial test generation tools [16]. It provides three state-of-the-art metaheuristic search algorithms (i.e., FIPOG, FIPOG-F, FIPOG-F2) to generate the test cases based on the t-ways testing strategy. We have considered the logical names of all the signals specified in the requirement specification as input parameters and utilized the web GUI online version of CAgen and selected the FIPOG heuristic algorithm to generate the test cases. We have also provided the test redundancy value '1' with the 'randomization of don't care values' for the generation of non-redundant test cases and limited the interaction strength to 2, 3, and 4-ways to avoid combinatorial explosion [17].

After the creation of the test cases, we exported the generated test cases in an excel file and added the expected outputs and timing constraints against each test case manually by thoroughly analysing the requirements of the system. Then we provided the exported file along with an XML file as input to CATS- gen. CATSgen contains the implementation of mapping rules, like TIGER, to map the logical signal names and their respective values to the technical signal names. Moreover, it also contains the implementation details specific to the BT's testing framework and libraries (e.g., configurations, classes, methods, etc.) to generate the executable test scripts. Hence, CATSgen extracted the data from the excel and XML files, used the mapping rules along with implementation details, and generated the test scripts in the C# language.

4.3.2.3 Deployment and Execution of the SUT and Test Scripts

After the generation of virtual trains and test scripts, we deployed the virtual trains on a laptop containing the software compatible with the BT -specific testing framework and test simulation platform for TCMS. Moreover, we used the BT-specific libraries and con- figuration files to set up the testing environment in a project using Visual Studio 2019 and executed the test scripts to generate test results in the form of test verdicts. The generated test verdicts contained passed and failed test steps that can be used to identify the detection of a fault produced by a mutant operator in a program.

4.3.3 Methodology for Experimentation on the Practical Implications of Adopting Passive Testing using T-EARS and the Accompanying Toolchain

The objective is to study the practical implications of adopting passive testing using T-EARS and the accompanying toolchain to a safety grade industrial context. We analyse the challenges encountered and our solutions to those challenges when writing, executing, and analysing passive tests written in T-EARS and its accompanying toolchain. The study's context is the system level testing in an embedded vehicular software system.

The studied case organization is responsible for developing TCMS (Train Control and Management System), an embedded safety-critical system controlling and monitoring software and hardware systems in a train. The different functions of a TCMS system constitute safety-related parts (SAFE), con- trolling safety-related functions, and non-safety-related parts (REGULAR) for non-safe control and monitoring functions written using the IEC 61131-3 programming language [18]. The focus of this study is on the safety-critical part (SAFE) of TCMS. The engineering processes of TCMS software development are performed according to safety standards and regulations (e.g., EN 50128 [19]). Testing can be performed on a real train or different configurations of a simulated virtual train environment, illustrated in Figure 21. The "Train" includes everything required to perform end-to-end testing of the system under test (SUT). Typically, either simulated systems and train environment or a hardware (HIL-Rig) are used. In either case, the automated and manual test cases are not changed. Such tests can either be performed by a tester (in manual testing) or implemented as automated test scripts (performing the same sequence of actions and reaction checks as the manual tester would have done). Both manual and automated test cases can be used to log signals using the same logger, which is vital for applying a passive testing approach.

The unit of analysis is the set of safety-related requirements, implementation, and system-level tests for a Safety Integrity Level 2 (SIL2) compliant TCMS application.

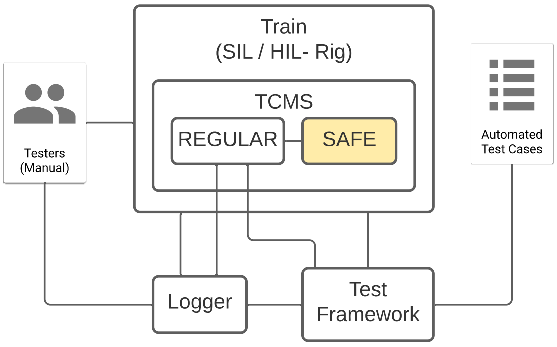
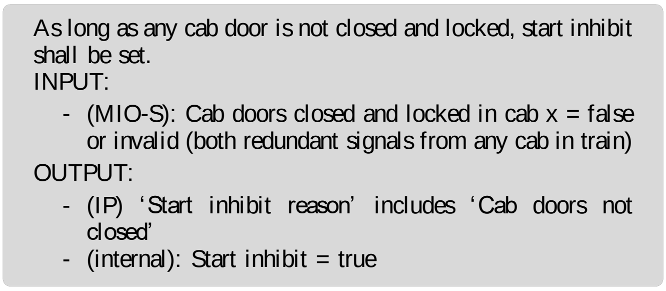


Figure 21. Conceptual overview, TCMS Testing Environment.

The system-level test cases are primarily written in natural language for manual test execution. These must be carefully written and reviewed to ensure that they cover all requirements and combinations and are feasible for testing in the intended test environment (i.e., the actual train). These tests take hours of manual labour to complete. However, while these manual tests provide valuable information about irregularities and contradicting results that would be hard to observe by strictly automated test cases, intermittent failures in unexpected situations are still difficult to catch with such a traditional testing approach.

The studied requirements are described in a semi-formal format, as shown in Figure 22. Since there are two cab compartments for the train driver, A1 and A2, these are typically referred to as Cab x, meaning that the requirement applies to both Cab A1 and Cab A2. The example in Figure 2 starts with a natural language description of the requirements. The SAFE requirements have an additional semi-formal description, starting with an INPUT section, describing the condition under which the requirement shall be fulfilled. The condition includes a list of logical input signals, their respective values, and logical relations. In this example, the means of communication (e.g., MIO-S or IP) is also given to provide a bridge to the concrete system. This structure corresponds quite well to a guard in the G/A concept. The OUTPUT section lists logical signals and expected values (response) to the INPUT section's conditions. This part corresponds well to an assertion in the G/A concept.



As long as any cab door is not closed and locked, start inhibit shall be set.
INPUT:

- (MIO-S): Cab doors closed and locked in cab x = false or invalid (both redundant signals from any cab in train)

OUTPUT:

- (IP) 'Start inhibit reason' includes 'Cab doors not closed'
- (internal): Start inhibit = true

Figure 22. Example Requirement (slightly adjusted for readability).

Safety-critical requirements typically summarize these logical INPUT and OUTPUT signals, so the same names are used throughout an entire function or system.

The study is carried out in three phases. In the first phase, a gold standard is set, and a set of requirements are chosen. This gold standard is used for evaluating a successful translation of the requirements to passive tests (G/As) concerning false positives. The absence of such false positives is the stopping criterion for the second phase, the adoption of the translation process [20] using T-EARS and its accompanying toolchain. Although the core of this paper refers to the implications of adopting and deploying passive testing in practice, the third phase is an evaluation that the results are applicable, the produced G/As do not elicit false positives and can find faults that are hard to find using the existing tests.

4.3.3.1 Phase 1, Gold Standard and Requirements Selection

The first phase, setting the gold standard, requires selecting a set of requirements to translate and a set of existing test cases testing those requirements. The following steps are followed:

- First, the TCMS system requirements are analysed to find automated test cases and, ultimately, a set of log files, serving as the gold standard. The chosen test cases shall stem from a system that is well tested, sufficiently observable, and offer reliable automated test scripts. Automation is key here since it should be possible to re- execute the test cases with any extra logging required by the G/As.

- Second, a set of requirements is selected where the selected automated test cases test each requirement, and dependent signals are observable from the selected set of test cases.

This phase's outcome is a set of automated test cases and a set of requirements to be translated into passive test cases (G/As).

4.3.3.2 Phase 2, Process Adoption

This phase's starting point is the T-EARS language and its accompanying toolchain, together with the proposed process of translating natural language requirements to passive test cases as G/As [20]. The purpose of this step is to explore how to apply this in practice in an industrial safety-critical context.

Inputs to this phase are a set of automated test cases for a set of well-tested system functions and a set of requirements tested by those test cases. During the adoption procedure iterations, requirements are translated using the translation process [20] and compared to the gold standard. Iterations could cover the entire process from the requirement to tuned G/A or a particular step in the process, meeting a particular challenge. This iterative adoption procedure is based on the work of Staron [21]. Each iteration contains the following activities:

- Diagnosing: Observations and analysis of challenges met during the different steps of the translation process.
- Action-Planning: Discussions with other industry experts on possible solutions.
- Action-Taking: Implementation or stubbing of suggested features.
- Evaluation: Occurs at each iteration that ends with a set of tuned G/As. The resulting G/As are evaluated against the set of correct logs.
- Learning: Information and reflections on the work per- formed are collected. The notes are then analysed and structured into a process, and tests are executed.

This phase's result is a set of improvements to the original process, a set of challenges that could not be solved or requirements on new tools, and a set of tuned G/As, translated from the requirements using the improved process.

4.3.3.3 Phase 3, Final Evaluation

The last phase in the case study is the final evaluation.

- The complete set of 116 SAFE requirements is analysed given the improvements from phase two. The suggestions are implemented to see the extent to which the results are useful.

- Further, the resulting G/As from the second phase are evaluated over two signal logs from a manual test session with an expert.

The results of this phase are a measurement of false positives from a well-tested system, as well as means to assess the degree to which the produced G/As can find injected faults that are hard to find using existing test cases.

5 ANALYSIS OF THE RESULTS

This section describes our findings in terms of:

- Modelling aspects.
- Behavioural differences between test cases generated through GW and manual test cases.
- Initial experiences for selecting and using GW for industrial use.
- Fault injection analysis to evaluate the test scripts generated by Model-Based Test scrIpt GenEration fRamework (TIGER).
- MC/DC adequacy of two automated testing techniques (CT and MBT) and industrial manual testing at system-level.
- Measuring requirement coverage and performance efficiency in terms of time.
- An assessment of the differences and over- laps between the test suites generated by automated testing and manual industrial testing to better demonstrate the potential gains and trade-offs among them.
- A detailed fault detection effectiveness of test suites.
- Sensitivity of test suites to specific mutant operators.
- Relationship between MC/DC coverage and mutation score.
- The practical implications of adopting passive testing using T-EARS and the accompanying toolchain.

5.1 Results on Modelling Aspects, Behavioural Differences Between Test Cases and Initial Experience

5.1.1 Model-based Testing Using Graphwalker

In our first investigation, we developed two versions of the model for test case generation: in the first version, we modelled the expected behaviour of the SUT by exploring the requirements specification document alone and discussing the result with the testing team, whereas in the second version, the model was created by developing an understanding of the SUT using both requirements specification as well as the test specification and then discussing the result with the testing team. In this section, we discuss the modelling process and the difference between the two versions of the model.

5.1.1.1 Modelling the SUT using Requirements Specification

The first step of modelling the SUT involved under- standing the SUT by talking to the testing team and exploring the requirements specification document. The researchers examined the requirements specification and identified the possible states of the SUT and then added the guard conditions according to the expected behaviour (as shown in Figure 23).

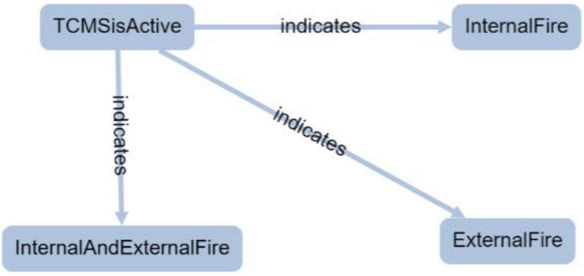


Figure 23. Initial FSM-based model created using the requirements specification.

Guard conditions are Boolean expressions that affect the behaviour of the FSM model by enabling or disabling the actions or transitions upon evaluation. The FSM-based model in GW consists of nodes (round-edged rectangular boxes) and directed edges (arrows). The nodes represent the state of the SUT, whereas edges represent the requests/decisions when a certain event occurs. TCMSisActive node shows the active state of TCMS while InternalFire, ExternalFire and ExternalAndInternalFire nodes represent types of fire indicated by the TCMS. The indication of fire depends on the signals sent by FDCUs, so the node and edges were added showing the active state of the FDCUs. The signals sent by the FDCU are added as actions on edges covering the requirements of the SUT to generate test cases. Figure 24 represents the final model created using the requirements specification.

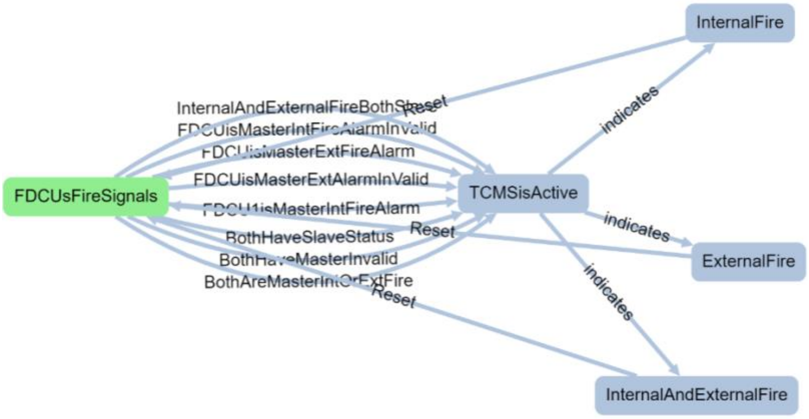


Figure 24. Final FSM-based model created using requirements specifications.

5.1.1.2 Modelling the SUT using Requirements Specification and Test Specification

In the second version, we refined the model using the previous knowledge as well as exploring the test specification document. The test specification helped us to understand the test objectives, test scenarios and behaviour of the SUT from a tester's perspective. One new nodes Reset was added to the model representing states where the SUT can be reset to its initial state with corresponding input values. This node was missing in the first version as it was not specified in the requirements specification. On the other hand, a tester considered it as an obvious requirement while designing the tests in the test specification document. Similarly, 10 edges were added in the initial model of the second version which helped us identify the possible signals of FDCUs in the final model to cover the scenarios needed to generate data for complete test suite. Figure 25 and Figure 26 depict the initial and final model created using both specifications, respectively. The final model (Figure 26) consists of three diagrams: two diagrams representing FDCUs and one diagram representing the TCMS as black box. These diagrams contain shared nodes, which are used by GW while traversing from one diagram to another. FDCU1 is an initial and shared node whereas FDCU1Signal, FDCU2, FDCU2Signal, TCMSisActive and FDCUsFireSignals are other shared nodes. Moreover, MasterState and SlaveState nodes represent the 'Master' and 'Slave' states of the FDCU. The purpose of developing two separate diagrams explicitly representing the FDCUs in the final version of the model was to cover more test scenarios. GW can traverse through the elements of FDCU diagrams providing the TCMS different combinations of signals using random walks. The single- diagram model, on the other hand, can only provide the specific values of signals provided by the user as actions on its edges. Moreover, by generating random walks for the final model, we were able to identify an incorrect guard condition, which was not identified in the single-diagram model.

In discussions with the test team at BT, it was also concluded that the final model represented the behaviour of the SUT more completely than the first version of the model which was created using the requirement specification alone.

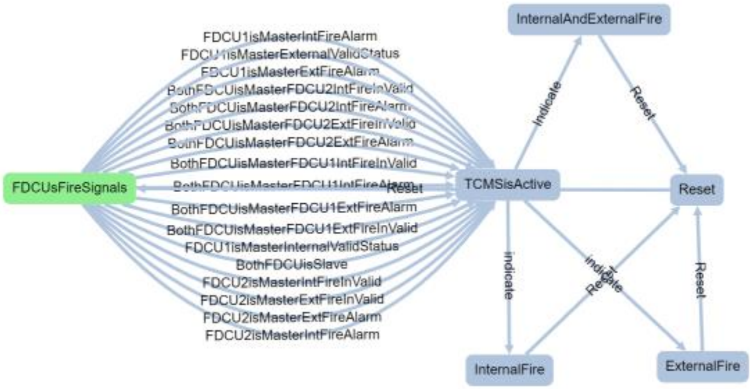

InternalAndExternalFire
FDCU1isMasterIntFireAlarm
FDCU1isMasterExternalValidStatus
FDCU1isMasterExtFireAlarm
BothFDCUisMasterFDCU2IntFireInValid
BothFDCUisMasterFDCU2IntFireAlarm
BothFDCUisMasterFDCU2ExtFireInValid
BothFDCUisMasterFDCU2ExtFireAlarm
BothFDCUisMasterFDCU1IntFireInValid
FDCUsFireSignals
BothFDCUisMasterFDCU1IntFireAlarm
Reset
TCMSisActive
Reset
BothFDCUisMasterFDCU1ExtFireAlarm
BothFDCUisMasterFDCU1ExtFireInValid
FDCU1isMasterInternalValidStatus
BothFDCUisSlave
FDCU2isMasterIntFireInValid
FDCU2isMasterExtFireInValid
FDCU2isMasterExtFireAlarm
FDCU2isMasterIntFireAlarm
indicate
Reset
indicate
indicate
Reset
Reset
InternalFire
ExternalFire

Figure 25. Initial FSM-based model created from requirements and test specifications.

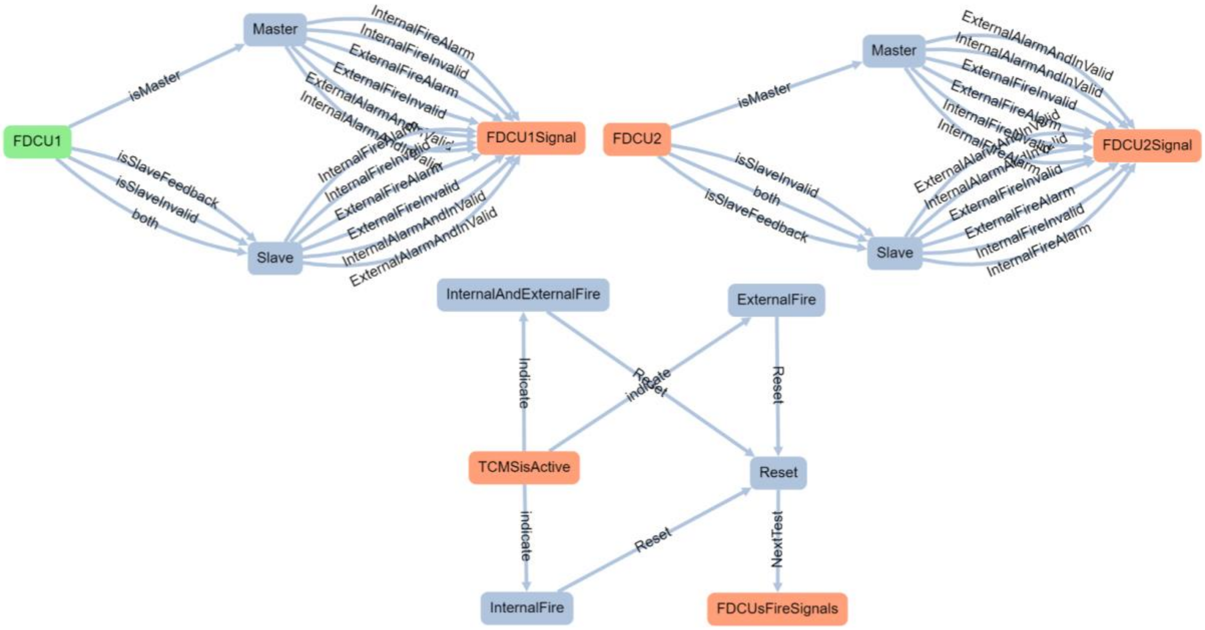


Figure 26. Final FSM-based shared model with multiple diagrams created from requirements and test specifications having shared nodes.

5.1.1.3 Modelling Aspects

MBT is known to provide a better understanding of the domain and the SUT to generate test ware [23]. However, to create a complete and representative model of the SUT, all relevant details about the test object or SUT need to be made available.

5.1.1.3.1 The degree of test model completeness and model evolution

The first version of the model was created using requirements specifications and it contained behavioural aspects of the SUT. However, the second version of the model, created using both requirements and test specifications, contained more complete information about the expected behaviour of the SUT as well as incorporated tester’s perspective to cover all the scenarios. The second version of the model includes additional information about the states of the SUT, which were not mentioned in requirements specification. For example, there is a condition where the TCMS can be reset to its initial state before testing the next scenarios for indicating internal or external fire in both cabs. This specific additional information about the condition, which was only available in the test specification, resulted in the addition of one new node in the model representing TCMS as well as creation of the models representing FDCUs in the second version as compared to the first version of the model of the same SUT.

Thus, to create a complete behavioural model of the SUT, relying only on requirements specification resulted in less-than-optimal model (having fewer edges and states) in our case. While the model created using both types of artifacts (requirements and test specifications) led to a more complete representation of the scenarios and expected behaviour. This, in turn, is expected to help generate test cases that realistically cover more testing scenarios.

5.1.1.3.2 Model representativeness

Moreover, in our case, the test specification also provided information about the SUT in a clearer and more concrete way, which included information about signals sent by the FDCUs to the TCMS. Requirements specification did contain information about the FDCUs (i.e., FDCUs could be in a master or slave state) but did not clarify the total number of FDCUs used to signal the TCMS. Similarly, it was also not clear from the requirements specification alone if FDCUs should send the signals about its states along with fire indication to the TCMS or not. Thus, in our case, understanding and analysing both requirements and test specifications resulted in a more representative model of the SUT. Therefore, according to our analysis, to generate quality tests using MBT, one should consider different characteristics of the test object and test objectives while creating the MBT model. And the model of the SUT created using both requirements and test specifications is completer and more representative than the model created using the requirements specification alone.

Additionally, as we have created the final model in an incremental approach by getting continuous input from the testing team at BT; we also observed that the lack of domain knowledge or access to team members working on SUT could cause conformance issues between the model and the SUT. Moreover, it can directly affect the effort for developing the model and quality of generated test cases.

5.1.2 Behavioural Differences Between MBT and Manually Created Test Cases

GW walks the directed graph of the SUT in a fashion deter- mined by the “generator” algorithm and generates tests on every run until the “stopping condition” is met. There are a variety of generator algorithms available in GW to traverse the model, however, not all of them were suitable for comparison with manual test generation. Thus, we used the random generator for the comparison with manual test generation, which navigates the model in a random fashion or takes a random walk from each node. In this comparison, we have evaluated four different stopping conditions for GW: edge_coverage(100) in which the test generation is stopped when 100% of unique edges are traversed, vertex_coverage(100) when test generation is stopped when 100% of unique nodes/states are traversed, requirement_coverage(100) in which the test generation is stopped when 100% of unique requirements are traversed and length(100) when 100 edge-node pairs are traversed. There are other supported stopping conditions in GW that were found not suitable for comparison with manual test cases.

Edge coverage is a stronger coverage criterion than node coverage alone as it includes traversing of both elements of a model (nodes and edges) [23]. We have used "edge coverage %" as a measure to quantify how thoroughly a model has been validated for each stopping condition. However, for manually written test cases, we have calculated the "edge coverage %" by comparing "action" and "expected result" defined for each step with the edges representing similar actions or transitions in the model. There are several other model coverage criteria available [24, 23], but we leave the measurement of those as a future work. "Requirements coverage frequency" depicts the number of times a certain requirement gets covered using multiple test steps; it is particularly beneficial to cover a requirement multiple times in a safety critical system (as is provided by TCMS) as well as to uncover interaction faults at system level. As GW generates the test cases using random walks through the model, the "number of test steps generated" can vary for each attempt. So, we generated test cases 3 times and selected the test cases with maximum number of test steps and have reported the figures in "Number of Test Steps Generated" and "Edge Coverage %" columns in Table 5. The reason to select the highest number of test steps in the three test generation attempts is to optimize the edge coverage as well as to improve the frequency of requirements coverage. Similarly, requirements coverage frequency of each requirement can get affected through random walks. Subsequently, we calculated the minimum and maximum requirements coverage frequency provided by the generated test steps for the three execution attempts, as reported in the column "Requirements Coverage Frequency (Min–Max)" of Table 5; the value of this metric for manual test cases remains constant due to deterministic, sequential order of coverage.

The results show that manually written test cases have 26 test steps and covered the requirements in a sequential order whereas MBT generated higher numbers of test steps using different stopping conditions in a random order. Model-based test cases resulted in an increased requirements coverage frequency, when looking at the maximum figures, than the manually written test cases. All the test cases generated by GW using different stopping conditions provided 100% requirements coverage. The same is also true for manual test cases. However, test cases generated using edge_coverage(100) provided full edge coverage whereas partial edge coverage was achieved by other stopping conditions and manually written test cases.

After analysing manually written and GW-generated test cases, we found that as GW generates the test steps by traversing the edges and nodes, thirteen test steps are required in our case to cover one specific scenario. However, these steps can vary depending on the number of diagrams representing the SUT as well as the edges and nodes traversed by GW to fulfil requirements; an example of it is shown in Table 6. The manually written test cases, on the other hand, require one test step per requirement as shown in Table 7. Furthermore, test steps generated by GW also include complete and clear details (i.e., signals from each FDCU and its respective values) in test steps while manual test steps do not contain such details for each signal. In Table -- , the "Test Steps" column for GW-generated test cases represent nodes and edges, "Action/Expected Result" represents the specified actions at edges and resulting nodes.

Table 5. Comparison between manually written and MBT generated test cases using different stopping conditions provided by GW.

Test Design Technique	Stopping Conditions	Order of Coverage	Number of Test Steps Generated	Edge Coverage%	Requirements Coverage Frequency (Min–Max)
Manual	Until the end of manually designed test scenarios	Sequential	26	70	4
MBT (1)	edge_coverage(100)	random	2703	100	7–84
MBT (2)	vertex_coverage(100)	random	900	57.5	1–26
MBT (3)	requirement_coverage(100)	random	1202	55	1–41
MBT (4)	length(100)	random	2021	62.5	6–53

Table 6. An example of test steps generated by GW.

Sr.#	Test Steps	Action/Expected Result
1	FDCU1	FDCU is Active
2	isMaster	“Signal1 = true; Signal2= true;"
3	MasterState	FDCUisInMasterState
4	ExternalFire	“Signal3=false;"
5	FDCU1Signal	FDCU1FiresSignals
6	FDCU2	FDCU is Active
7	isSlaveInValid	“SignalX = true;"
8	SlaveState	FDCUisInSlaveState
9	InternalAlarmAndInValid	“SignalZ = false; SignalY = false;"
10	FDCU2Signal	FDCU2FiresSignals
11	TCMSisActive	TCMS get the Signals
12	indicate	“SignalToIndicateNofire=false; Time= 200;"
13	ExternalFire	External Fire Indicated

Table 7. An equivalent manually created test step corresponding to the example in Table 6 (the pre- and post-conditions are omitted for clarity).

Step No.	Action	Expected Result
1	Set external fire from master FDCU-1 by setting signalA=false	External fire is indicated in both cabs

5.1.3 Initial Experience with GW for Industrial Case Modelling and Test Generation

In this section, we summarize our experiences of selecting and making use of GW as the MBT tool of choice for application on the TCMS industrial case.

5.1.3.1 Selection of GW as the MBT Tool

Both industrial and academic/research considerations played its role in the selection of GW as the model-based tool of choice in this work. Previous research [25] shows that applicability, usability, and expressiveness are important attributes for MBT tool selection from an industrial uptake point of view. Two other important attributes of tool selection for our industrial partner were open-source availability and continuous development/maintenance of the tool. From an academic/research point of view, in addition to all the above-mentioned attributes, we were particularly interested in the less-researched aspect of online test generation capability as compared to the more prevalent offline test generation capability of the tool. To compare a representative sample of the available tools, we did a non-exhaustive search for MBT tools and compared them against important attributes. The results are summarized in Table 8. GW was selected due to its active development/maintenance in terms of latest year of modification, its availability as an open-source tool, its features of both offline and online test generation as well as usability in terms of modelling the SUT as a state machine that is understandable to both researchers and practitioners.

Table 8. A non-exhaustive comparison of a brief selection of available MBT tools.

Tools	**Modelling Language**	**Type**	**License**	**Mode of Testing**	**Last Modified**
Uppaal [5]	Timed automata	Academic & Industrial	Free for non-profit use only	Offline	2019
Uppal Tron [6]	Timed automata	Academic & Industrial	Free for non-profit use only	Online & Offline	2009
SpecExplorer [4]	Model programs in C#	Academic & Industrial	Commercial	Offline	2013
MoMuT [2]	UML, Timed Automata & Textual	Academic & Industrial	Free for non-profit use only/Partly open source	Offline	2020
NModel [3]	Model programs in C#	Academic & Industrial	Open Source	Online & Offline	2010
GraphWalker [1]	State Machines	Academic & Industrial	Open Source	Online & Offline	2020

5.1.3.2 Using GW for Test Generation

GW is available in three versions: GraphWalker studio, GraphWalker CLI, and GW4E as Eclipse plugin. All the three versions provide some detailed and well-formatted online documentations and tutorials for support and learning purposes. We have found GW4E more user-friendly than GraphWalker studio and GraphWalker CLI in terms of debugging, execution of tests, and generating information for testers and developers but it supports limited stopping criteria (i.e., time duration in our case). In our experience, GraphWalker studio and GraphWalker CLI provide more functionality and options, but are less usable and user-friendly. GraphWalker studio can be used for modelling and validating the model by traversing through the model elements but does not explicitly generate test artifacts. Whereas the lack of a user interface in the GraphWalker CLI results in cumbersome activities due to the repetitive and manual use of CLI commands when using the test generation functionality. Furthermore, GraphWalker CLI requires the use of an additional tool for modelling of the SUT in JSON/graphML modelling language and it generates the test cases in JSON format.

5.2 Results of Fault Injection Analysis to Evaluate TIGER-Generated Test Scripts

5.2.1 Generation and Execution of Test Scripts

After creating different versions of model, we have generated the test cases and scripts multiple times using each model. We provided logical and technical names of the signals to TIGER and executed the test scripts on the SUT. Here we report results from five test generations based on the variation in number of generated test steps and the failed test steps. We have also executed the manually written test scripts to compare the test verdicts with TIGER-generated test scripts.

5.2.2 Fault Injection Analysis

As shown in Table 9, no fault was identified in the SUT by executing the manually written test scripts. The correct version of the model conforms with the requirements specification of the SUT; hence no failed test steps were reported on each execution of test scripts generated using it. On the other hand, different number of failed test steps were identified by the test scripts generated from faulty models (as shown in Table 10). However, number of generated test steps were different in each test script due to random walks of GW for the generation of test cases. We have also analysed the combinations of inputs to evaluate the requirements coverage of the generated test scripts. We observed that all test scripts generated using correct version of the model contained at least one combination for each requirement and provided 100% requirements coverage. It was also observed that one of test scripts generated using the 'change guard' mutant model missed the combination required to produce failed test steps. Similarly, no failed test step(s) was identified in some of the test scripts with 'state missing' mutant. It was attributed to the missing Master state and GW made the FDCU 'slave' in initial steps while generating the test cases, hence provided no combination that could produce failed test step.

Table 9. Comparison Between TIGER-Generated Test Scripts Using Correct Model and Manually Written Test Scripts.

Test Generation Source	No. of Test Steps (Min-Max)	No. of Failed Test Steps
Correct Model	264-514	0
Manually Written	24	0

Table 10. Comparison Between TIGER-Generated Test Scripts Using Correct and Faulty Models. Red Shade Shows Failed Test Steps and Green Shows No Failures Detected.

Test Generation #		Test Generation 1		Test Generation 2		Test Generation 3		Test Generation 4		Test Generation 5	
Test Generation Source	**Mutant Operators**	**No. of Test Steps**	**No of Failed Test Steps**	**No. of Test Steps**	**No of Failed Test Steps**	**No. of Test Steps**	**No of Failed Test Steps**	**No. of Test Steps**	**No of Failed Test Steps**	**No. of Test Steps**	**No of Failed Test Steps**
Faulty Model (1)	Output Exchange	329	28	364	24	349	28	544	37	179	19
Faulty Model (2)	Change Guard	624	6	239	0	133	1	514	2	254	4
Faulty Model (3)	State Missing	270	1	399	1	187	3	248	0	365	2
Correct Model	NA	264	0	454	0	514	0	294	0	299	0

In summary, the results show that when the faulty models did not conform to requirements, TIGER-generated test scripts had failed test steps on execution, except for two instances where GW did not generate the required failure- triggering combinations. In comparison, the test scripts generated using the correct version of the model discovered no fault in the SUT due to its conformance with requirements specification as well as the implemented SUT.

5.3 Results of MC/DC Adequacy, Requirements Coverage, Overlap of Test Cases and Performance Efficiency

5.3.1 Requirement and MC/DC Coverage

The requirement and MC/DC coverage of test suites generated by the three different techniques (MBT, manual and CT) is shown in Figure 27 whereas Figure 28 represents the breakdown of MC/DC coverage according to the metric parameters.

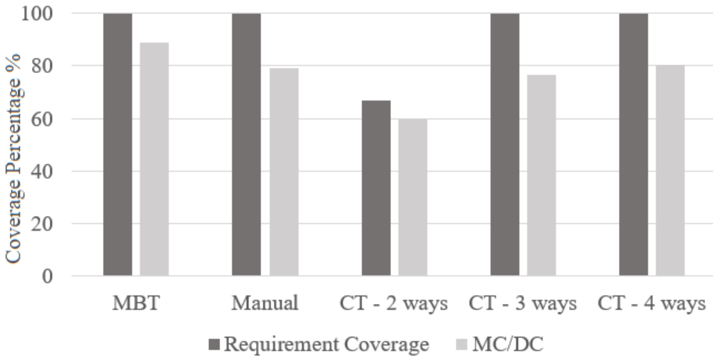


Figure 27. Requirement and MC/DC coverage of test suites.

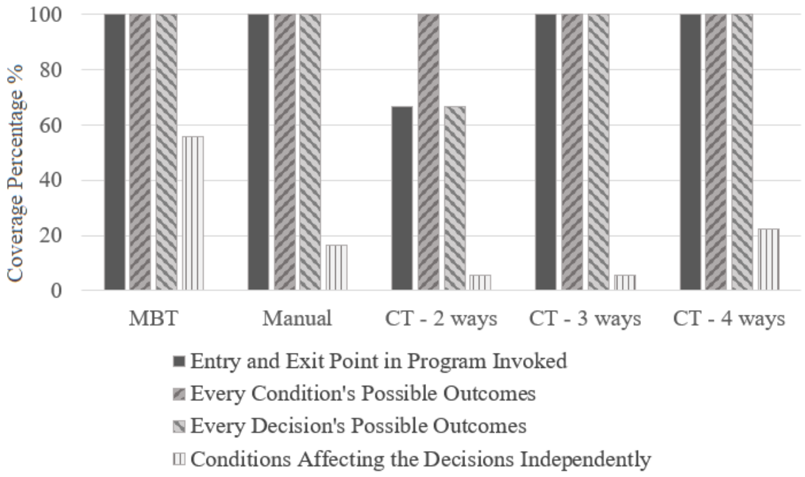


Figure 28. MC/DC coverage of test suites according to selected parameters.

The results showed that all the test suites contained test cases covering each requirement of the system at least once except the test suite generated by 2-ways combinatorial test strategy that provided 66% requirement coverage. However, the MBT-generated test suite is the most MC/DC adequate by providing 88% MC/DC coverage. Test suites generated using 2-ways, 3-ways, 4-ways combinatorial test strategy and by manual testing provided 59%, 76%, 80% and 79% MC/DC coverage, respectively.

The analysis of test suites in terms of MC/DC also showed the effect of each technique on the conditions of MC/DC. The results indicated that each technique affected the generation of such test cases which can be used to verify the 'effect of a condition independently on the outcome of a decision'. Consequently, the MBT-generated test suite covered 55% of conditions to verify the independent effects on a decision, whereas test suites generated using 2-ways, 3-ways, 4-ways combinatorial test strategy, and created manually covered 5%, 5%, 22% and 16% of such conditions, respectively. However, in case of CT test strategies, two other conditions were affected to a larger extent with an increase in the strength of CT (i.e., for 3- and 4-ways). These two conditions are 'invocation of each entry and exit point in a program' and 'every decision possible outcome'.

Moreover, we have also analysed the MBT and CT test generation process to assess the impact of achieving the higher MC/DC on the number of test cases. As MBT can generate different numbers of test cases due to random walks, so we have generated multiple test suites using TIGER and selected three based on the number of test cases (min, max, median) and performed a comparative analysis with 2-, 3-, and 4-ways testing strategy of CT. The results showed that random walks of MBT had an intermittent effect on the number of test cases and a minimal effect on the MC/DC coverage In test suites as shown in Table 11, MBT generated 150, 245 and 145 test cases, including the redundant test cases, and provided 88%, 90% and 87% of MC/DC coverage respectively. In cases of CT, the number of test cases increase exponentially while achieving the higher MC/DC by increasing the strength of CT.

Table 11. MC/DC coverage of MBT and CT.

Test technique	No. of test cases	MC/DC %
MBT (Median)	150	88.88
MBT (Max)	245	90.27
MBT (Min)	145	87.5
CT 2-ways	10	59.71
CT 3-ways	22	76.38
CT 4-ways	50	80.55

The analysis of test suites in terms of MC/DC and requirement coverage showed that MBT and manually developed test suites were more MC/DC and requirement coverage adequate and contained complete test cases (we designate a test case as complete having input(s), expected output(s) and timing constraint(s)). The completeness of test cases is dependent on the availability of the required information about the system; thus, one can argue that the test engineer and the model of the SUT utilized all the relevant information such as the behaviour of the system (functional and non-functional requirements), test scenarios known to the tester based on domain knowledge and experience, and information regarding system environment to create test suites. Our results also showed that adequacy of CT-generated test suite increased with an increase of interaction strength. Moreover, it contained no in- formation about the expected behaviour of the system and test environment. Hence, tester's assistance was required to specify the expected outputs to complete the test suites (this cost is captured as part of the parameter, C$_{Tsd}$ = Cost to develop test scripts).

5.3.2 Efficiency of Test Suites

Efficiency of test suites is generally calculated based on three factors i.e., time required for requirements analysis, time required for creation of a test suite and time required for execution of it. However, for simplicity, we have not reported the time for the analysis of requirements specification in the results as there was no significant difference between the time spent on the analysis phase by each technique. Moreover, we have divided the test suite development time according to the activities required by each testing technique as shown in Table 12. We have also reported the accumulated time of different activities required for modelling and execution phases. It is also important to mention here that the time required by a tester for adding expected outputs to complete the CT-generated test cases is also included in test script development time of CT.

Table 12. Efficiency measurements (in seconds) of test suites created by each technique. N/A is short for not applicable.

Techniques	No. of test cases	Development time of test cases and scripts (C″), (C) ( )					Execution time (C...)	Total time
		Modelling of SUT	Verification of model	Signal identification	Test case development	Test script development		
Manual	17	N/A	N/A	2400	15600	9600	300	27900
MBT	150	16800	1800	2700	5	2	600	20107
CT 2-ways	10	N/A	N/A	2400	2	7200	240	9842
CT 3-ways	22	N/A	N/A	2400	2	10200	360	12962
CT 4-ways	50	N/A	N/A	2400	2	13800	420	16622

The results showed that the development of a test suite using MBT was less efficient than CT due to additional activities i.e., modelling of the SUT and verification of the model's conformance with system requirements. The identification of signals in case of MBT also had a significant effect on the development time as it also involved the creation of XML file to generate executable test scripts. Whereas regardless of manual test script development in CT, CT required a significantly less test suite development time as compared to MBT and Manual testing.

The analysis also showed that the number of generated test cases in a test suite had a significant effect on the execution time. Consequently, the test suite generated using 2-ways test strategy required minimum execution time. The execution times required by the test suites generated using 3-ways, 4-ways testing strategies and MBT were greater than the manual test suite due to the additional number of test cases (more specifically, the test steps). However, based on the total time of all activities required by each testing technique, CT resulted in being the most efficient, MBT stood second and manual testing came out to be the least efficient technique.

5.3.3 Differences and Overlaps Between the Test Generation Techniques

To measure the differences and overlaps between the test suites, we have identified similar test cases in each test suite generated from different techniques and created a subsequent Venn diagram as shown in Figure 29. However, we have removed the redundant test cases within each test suite to provide one-to-one mapping between the similar test cases generated by each technique.

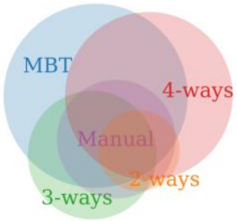


Figure 29. Venn Diagram representing the differences and overlaps between the test suites.

Table 13 presents an illustration of the total number of non-redundant test cases within the test suite, unique test cases across the test suites, and percentage of similar test cases across the test suites on average, whereas Tables 14 and 15 show the differences and number of test cases overlapping between the pair of each test suite, respectively.

Table 13. Number of non-redundant test cases within each test suite and unique test cases across the test suites.

Technique	No. of non-redundant test cases within each test suite	No. of unique test cases across the test suites	Percentage of similar test cases across the test suites on average %
MBT	39	9	71
Manual	17	2	54
CT 2-ways	8	1	25
CT 3-ways	20	2	44
CT 4-ways	34	3	53

Table 14. Number of different test cases in each pair of test suite

Technique	MBT	Manual	CT 2-ways	CT 3-ways	CT 4-ways
MBT	N/A	26	32	28	16
Manual	4	N/A	10	7	4
CT 2-ways	1	1	N/A	3	2
CT 3-ways	9	10	15	N/A	12
CT 4-ways	11	21	28	26	N/A

Table 15. Number of overlapped test cases in each pair of test suite

Technique	Manual	CT 2-ways	CT 3-ways	CT 4-ways
MBT	13	7	11	23
Manual	N/A	7	10	13
CT 2-ways	-	N/A	5	6
CT 3-ways	-	-	N/A	8

The analysis of Figure 29 showed that the MBT-generated test suite contained the highest numbers of test cases generated by other techniques (approx. 71% of test cases on average). The test suites developed by CT using 4-ways, 3-ways, 2-ways, and manual testing contained approx. 53%, 44%, 25% and 54% of similar test cases on average, respectively. MBT and 2-ways had the greatest overlap (MBT suite contained almost 90% (7 out of 8) of test cases generated using 2-ways). MBT-generated test cases had the second greatest overlap with manual test cases whereby MBT suite contained 76% of test cases from manual (13 out of 17). The analysis of test suites also showed that the test suite generated by MBT was more similar, and hence complete, to specification-based manual testing as it contained constraints (i.e., time) and test inputs along with expected outputs, which can be used to determine requirement coverage and traceability. However, the generation of MBT-based test suite is dependent on the conformance of the model representing the SUT. CT-generated test suites did not contain expected outputs but provided more test scenarios that can be used to validate the model as well as to complement the manual testing of the SUT.

It can be analysed from Figure 29 that MBT generated most of the unique test cases. Particularly, it is interesting to see the impact of unique test cases in each test suite on MC/DC and requirement coverage. Hence, for experimentation purposes, we discarded all the unique test cases from each test suite and measured their impact. Subsequently, results indicated that MC/DC coverage reduced from 88% to 84% in case of MBT. However, no change was observed in requirement coverage of the MBT-generated test suite. In case of other testing techniques, no major effect on MC/DC and requirement coverage was reported. We conclude that MBT-generated unique test cases are also highly relevant for covering MC/DC in a safety critical system as compared to the unique test cases generated by other approaches. Moreover, addition of these unique test cases will improve the effectiveness of manually created test suite in terms of MC/DC coverage.

5.4 Results of Detailed Fault Detection Effectivess, Sensitivity of Test Suites and the Relationship Between MC/DC Coverage and Mutation Score

In this section, we provide the experimental results in terms of fault detection effectiveness for each test suite using mutation score, the sensitivity of each test suite towards mutation operators, and an analysis of the relationship between MC/DC and mutation score in detecting faults.

5.4.1 Fault Detection Effectiveness of Test Suites

To measure the fault detection effectiveness of each test suite, we created 50 mutants of the original FBD program based on mutation operators and calculated the mutation score for each test suite. The mutation scores are shown in Column 5 of Table 16. It is important to mention here that we have considered only non-equivalent mutants in our results and excluded the equivalent mutants after carefully examining the test results and alive mutants manually. For example, we added an XOR between two operators in one of the LIO mutants, and none of the test suites detected this fault. On examining this mutant, we found out that it had an apparent effect at the internal level, but that effect did not propagate towards the overall output of the program. Hence, we declared this mutant an equivalent and excluded it from the results. Table 16 shows the result of each test suite in terms of mutation score, and the total number of killed and alive mutants.

Table 16. Mutation score of each testing technique.

Techniques	No. of Test Cases	Mutants Killed (M_K)%	Mutants Alive (M_A)%	Mutation Score%
Manual	17	43	7	86
CT 2-ways	10	41	9	82
CT 3-ways	22	45	5	90
CT 4-ways	50	45	5	90
MBT	150 (39 after excluding identical test cases)	44	6	88

Table 17 depicts the number of common mutants killed and alive between the pairs of test suites. Whereas Figure 30 illustrates the overlaps and differences between alive mutants of each test suite.

Table 17. No. of common killed and alive mutants between the pairs of test suites.

Techniques	CT 2-ways	CT 3-ways	CT 4-ways	MBT
	(M_K / M_A)	(M_K / M_A)	(M_K / M_A)	(M_K / M_A)
Manual	41/7	42/4	43/5	41/4
CT 2-ways	N/A	41/5	41/5	39/4
CT 3-ways	-	N/A	41/5	43/4
CT 4-ways	-	-	N/A	43/4

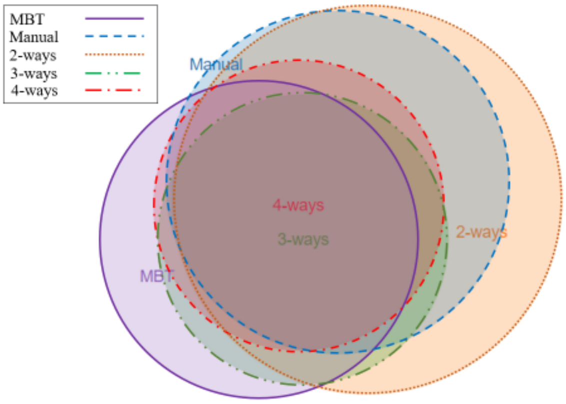


Figure 30. Venn diagram representing the overlaps and differences between alive mutants.

The results show that CT-generated test suites provide the highest mutation scores by detecting 90% of the mutants when using 3-ways and 4-ways interaction strength while requiring 1140 and 1680 seconds as an average execution time per mutant, respectively. In contrast, CT 2-ways, manual, and MBT achieved 82%, 86%, and 88% mutation scores respectively, and 780, 600, and 2760 seconds of average execution time per mutant, respectively. Moreover, the number of un- detected mutants in manual, CT 2-ways, CT 3-ways, CT 4-ways, and MBT were 7, 9, 5, 5, and 6, respectively. We observed that MBT generated the highest number of test cases in a test suite and required the highest execution time, on average per mutant, but still provided a slightly low mutation score than 3- ways and 4-ways testing strategy. In addition, we also examined the test verdicts of the MBT-generated test suite and found that, due to duplicate test cases in the MBT-generated test suite, if a test case in the MBT- generated test suite detects a fault, its identical test case(s) also show the presence of the fault in the mutated program. In our industrial context, each failed test step in a test case requires a maximum waiting time for a signal response specified in the requirements. Similarly, in the case of other test suites, the number of test cases detecting a mutant and the waiting time for a signal response have a significant effect on the execution time. However, the number of MBT- generated test cases in a test suite can be minimized by removing these identical test cases, which consequently will reduce the execution time too.

The analysis of the data shown in Table 17 suggests a similar number of killed mutants whereas only 4 common alive mutants were found among all the test suites. We have also examined these alive mutants and observed that 2 out of these 4 mutants were affecting a part of the code that could not be invoked by the generated test cases' inputs and requires inputs from another subsystem upon integration. However, test suites developed using Manual, 2- ways, 3-ways, 4-ways, and MBT left 3, 5, 1, 1, and 2 mutants alive, respectively. Hence, all the test suites achieved a reasonably high level of mutation scores, within the range of 82% to 90%. However, the test suites generated by CT 3-ways and 4-ways provided higher fault detection rates than the test suites generated by other testing techniques. Moreover, each technique missed the generation of some signal combinations that could be used to achieve a mutation score of 100%. However, we still found manual testing as a better-performing technique in terms of average execution time per mutant while achieving a 86% of mutation score. A summary of our observations regarding alive mutants across all test suites is as follows:

- The total number of mutants not detected by manual testing, 2-ways, 3-ways, 4-ways, and MBT were 7, 9, 5, 5, and 6, respectively. 4 of these alive mutants were common among all the test suites.

- The manual test suite did not detect 3 alive mutants that were also included in the subset of alive mutants in CT 2-ways, whereas 1 mutant amongst the 3 was also not detected by the 2-ways and 4-ways generated test suites.

- The use of CT 2-ways generated test suite did not detect 1 unique alive mutant which was killed by each of the other test suites.

- CT 3-ways and 2-ways generated test suites had 1 alive mutant which was not killed by either of the test suites.

- Lastly, 2 unique mutants were not detected by the MBT-generated test suite.

5.4.2 Sensitivity of Test Suites to Specific Mutation Operators

To examine the type of faults prone to be detected by each test suite, we calculated the mutation scores as per each mutation operator. We also developed a bar graph based on the results to analyse the breakdown of mutation scores of each test suite as shown in Figure 31.

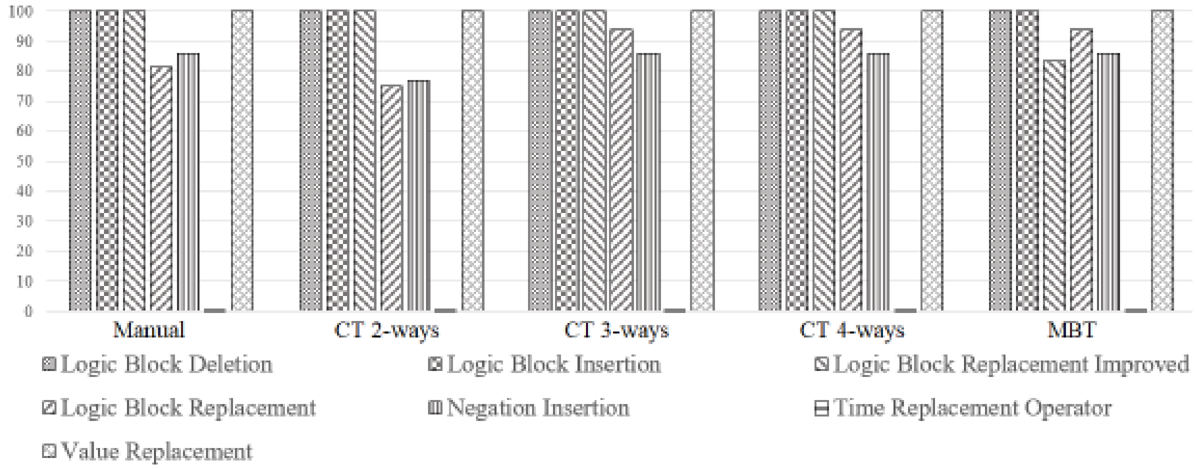


Figure 31. Percentage of mutants killed by each test suite per operator.

Figure 31 shows some meaningful implications based on the mutants killed by different test suites. All test suites detected each of the mutants injected by 3 out of 7 mutation operators i.e., LDO, LIO, and VRO, and provided 100% mutation scores for these mutation operators. On the other hand, none of the test suites detected any mutant induced in the original program based on the TRO mutation operator, consequently providing 0% mutation scores. In the case of LRO-I, all the test suites achieved 100% mutation scores except MBT, which achieved only 83%. For the NIO-based mutants, all the test suites achieved 86% mutation scores except 2-ways, which achieved 77%. Similarly, 3-ways, 4-ways, and MBT attained a similar mutation score of 93% by detecting mutants injected based on LRO, whereas 2-ways and manual achieved 75% and 81%, respectively.

Based on these results, we observe that for each mutation operator except TRO, each testing technique did not generate such combinations (i.e., combinations of inputs invoking the faulty area of the code affected by the alive mutant) that could be used to reach 100% mutation score. To detect TRO- based mutants, each test suite requires special test cases targeting the basic integrity (i.e., ensuring the starting states of the system) while entering a state within specified time, which can be used to vali- date the timing related requirements of the system. Hence, the results show that the CT 3-ways and 4- ways generated test suites are the most effective in detecting all types of faults except the faults related to TRO. MBT's mutation score closely follows, where it achieved an equal mutation score in detecting all types of faults when compared to CT 3-ways and 4-ways, except faults related to LRO-I. CT 2-ways- generated test suites are least effective in detecting NIO and LRO-based mutants due to the generation of a smaller number of input combinations having lower interaction strength.

5.4.3 Relationship Between MC/DC Coverage and Mutation Score

Figure 32 presents the mutation score and MC/DC coverage of each test suite. It shows that the MBT-generated test suite provided the highest MC/DC coverage, i.e., 88%. The test suites developed using CT 2-ways, 3-ways, 4-ways, and manual testing techniques provided 59%, 76%, 80%, and 79% of MC/DC coverage respectively. The test suite generated by MBT provided an equal mutation score as its MC/DC coverage, whereas manual, 2-ways, 3-ways, and 4- ways generated test suites provided higher mutation scores as compared to the MC/DC coverage achieved by the respective test suites. We also observed that the differences between the mutation scores and MC/DC coverage in 2-ways and 3-ways generated test suites were greater than for the test suites generated by 4- ways and manual testing. Moreover, regardless of its high MC/DC, MBT shows a slightly lower mutation rate when compared with CT 3-ways and 4-ways techniques.

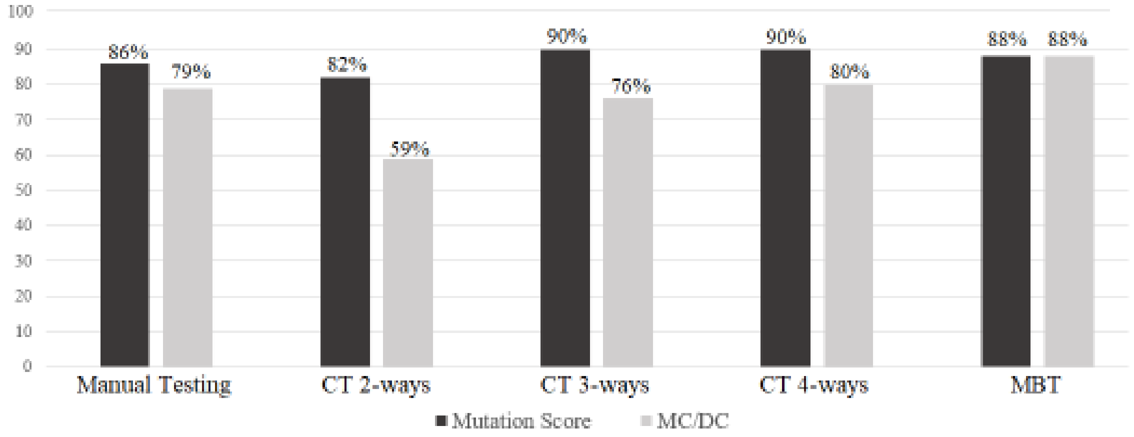


Figure 32. Mutation score and MC/DC of each test suite in percentage.

To thoroughly analyse the relationship between MC/DC coverage and mutation scores, we examined the breakdown of MC/DC coverage achieved by each test suite according to the selected parameters as shown in Figure 33, the SUT, as well as the killed and alive mutants. The analysis showed that if a system shares two or more similar requirements, then one FBD program can be used to generate two instances of the code. Therefore, each mutant induced at the FBD level can affect different areas of the code (i.e., induce more than one fault) and it may be possible that test suites with lower MC/DC cannot detect each fault induced per mutant. Similarly in our case, a fault induced at the FBD level affected the behaviour of the selected program at the system level where the combi- nations of input signals have a significant effect on the output of the system. Hence, the test suites developed using CT with higher interaction strength i.e., 3-ways and 4-ways and manual testing techniques contained such combinations, which were required to achieve an adequate mutation score at the system level. Whereas, MBT has provided higher MC/DC coverage by generating such test cases, especially targeting the MC/DC parameter 'coverage of all the conditions having an independent effect on a decision at least once', where other techniques gave relatively lower coverage per- centages. Based on the graphs presented in Figure 32 and Figure 33, we also observed a generally positive relationship between MC/DC coverage and mutation score and argue that test suites achieving an adequate level of MC/DC coverage also provide higher mutation scores.

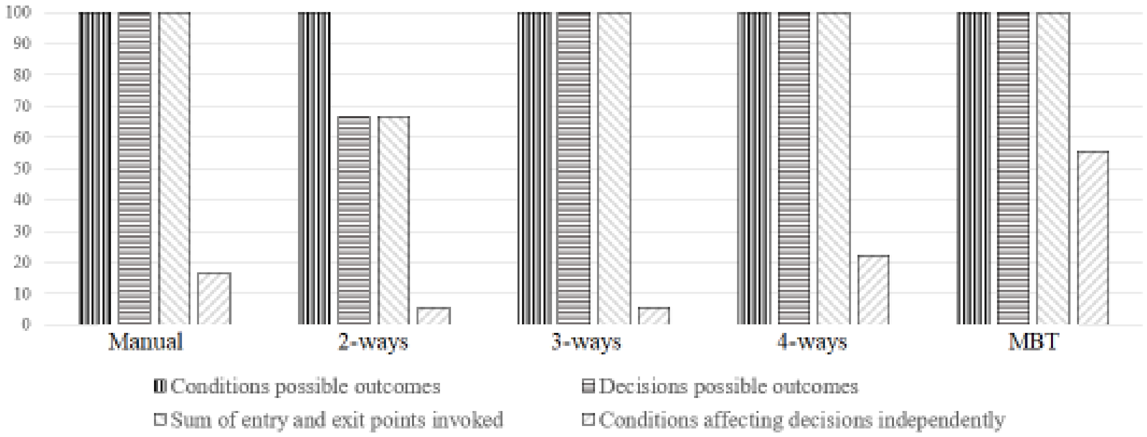


Figure 33. Breakdown of MC/DC according to selected parameters.

5.5 Results Regarding the Practical Implications of Adopting Passive Testing using T-EARS and the Accompanying Toolchain

The results of this investigation are presented in three phases: Phase I - Gold Standard and Requirements Selection, Phase II - Process Adoption, and finally, Phase III Final Evaluation.

5.5.1 Phase I – Gold Standard and Requirements Selection

The first phase aims to identify a set of automated test cases to use as a reference gold standard when translating a subset of the requirements to G/As and a set of requirements to translate. We identified 14 automated regression test cases from the Drive-and-Brake Functions, fulfilling our criteria on automation, observability, and priority by the case organization. The tested requirements were selected for translation, while the final evaluation is done over 116 SAFE requirements from the overall TCMS system.

The translation process [20] we earlier outlined is used as a starting point and framework for structuring the results. Further, since one of the potential drawbacks of the method is the risk of false positives [22], we use a subset of the regression test log files from a well-tested system as a gold standard. Sixteen of the tested requirements were translated and tuned until all false positives were removed, and no remaining problems were left unresolved. We then analysed each challenge and possible solution and formed a generalized workflow.

5.5.2 Phase II – Requirement Analysis Results

Given the choice of the Drive-and-Brake Functions, the expected result of this activity is the set of requirements concerning the Drive-and-Brake Functions, dependencies, and a list of logical signals. Typically, these logical signals result from the harmonization activity [20]. However, using logical signals already in the requirements is common in safety-related requirements. Such standardization radically reduced effort in the later steps of the translation process. We argue that writing requirements this way is worthwhile on non- safe requirements as well. Besides speeding up the translation work, using logical signals disconnects the passive test cases from a particular release of the system, which was imperative since signals tend to be frequently reallocated between data buses or modules, especially in early releases. Finally, since the list of used signals is known at an early stage, the implementation analysis can start in parallel, so logging of the required signals can be done early, which ultimately reduces the time spent in the concretization step. The ability to transparently observe this mapping while translating the requirement was beneficial since the test engineer is more confident in the technical signals' meaning.

For requirements that contain more complex expressions such as sequence, many signals, or negated logical expressions, we can manually create examples signals to facilitate the next step, namely the Abstract G/A Construction. Due to observability limitations in the testing framework available to us, we selected 16 of the identified Drive-and-Brake requirements for translation. In parallel with this step, parts of the Implementation Analysis and Concretization were performed. In short, the required technical signals were identified and added to the regression test cases to create the gold standard log files. Further, the mapping from logical to technical signals was prepared.

Throughout the upcoming sections, we use these logs to show the progression from a drafted G/A to a tuned G/A with a minimum of false positives.

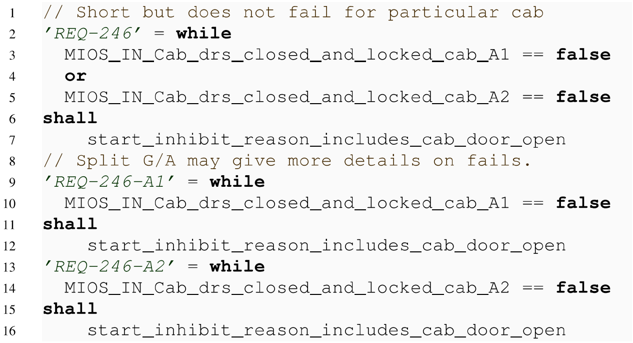
5.5.3 Phase II – Abstract G/A Construction Results

The expected outcome from this step is a G/A that is complete with respect to its logical guard and assertion expressions but using logical signals. The main steps in this activity are [20] a) Language Harmonization, b) Extraction Of G/A information, c) Pattern Selection, and d) Abstract G/A Formalization. Since (a) and (b) steps are already given by the semi-formal notation of the requirement, we focus on (c) and (d) steps.

One observed challenge is the case when there are alternative states for the guard (from any cab in the train), which in this case is cab A1 or cab A2. We observed that, when possible, splitting the passive test into one G/A for each OR expression allows a more fine-grained test, e.g., allowing each cab to be tested separately.

Further, when the validity of a signal S (i.e., S and valid_S) is important to test, (S or not(valid_S) could be used to create two tests as previously mentioned.

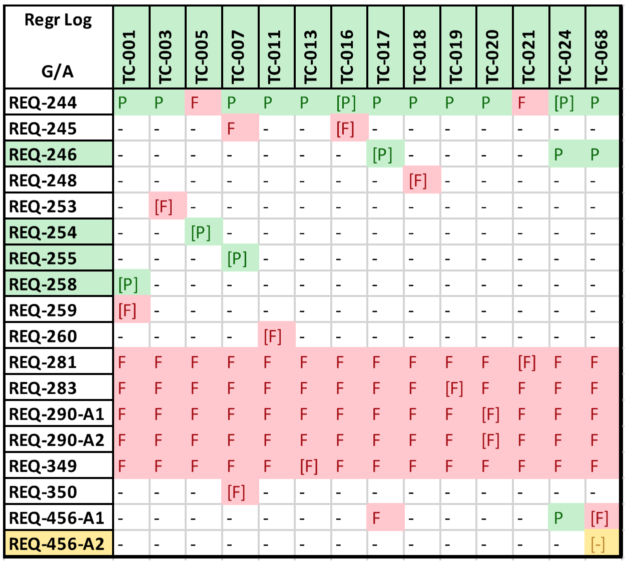
Listing 1 shows how the G/A at Line 2-7 can be split into one G/A for each cab. The example requirement did not contain any timing information. We note here that it is not always possible to have precise timing information for each requirement at this level. Often such timing behaviour is described in separate non-functional requirements covering one or more functional requirements. Even if a requirement contains sufficient timing information, valuable information can be obtained by observing the results of a G/A without timing specifications added.



```
1  // Short but does not fail for particular cab
2  'REQ-246' = while
3    MIOS_IN_Cab_drs_closed_and_locked_cab_A1 == false
4    or
5    MIOS_IN_Cab_drs_closed_and_locked_cab_A2 == false
6  shall
7      start_inhibit_reason_includes_cab_door_open
8  // Split G/A may give more details on fails.
9  'REQ-246-A1' = while
10   MIOS_IN_Cab_drs_closed_and_locked_cab_A1 == false
11 shall
12     start_inhibit_reason_includes_cab_door_open
13 'REQ-246-A2' = while
14   MIOS_IN_Cab_drs_closed_and_locked_cab_A2 == false
15 shall
16     start_inhibit_reason_includes_cab_door_open
```

Listing 1. Abstract G/A

Each requirement was translated to one or more G/As using the requirements' INPUT/OUTPUT sections and the list of logical signals. No timing information was specified yet. Where necessary, we will add timing to the G/As during the upcoming tuning session. The G/As were evaluated on the set of gold standard log files, and the result is presented in Figure 34. The rows show the G/As are named according to the requirement (e.g., REQ-245) to maintain traceability. Some requirements result in more than one G/A. We use a suffix for those G/As (e.g., -A1, -A2 to denote testing Cab A1 and Cab A2, respectively). For each TC-G/A combination, a P, F, or - denotes passed, failed or not-activated, respectively. For each log, the tested requirements (G/As) are marked with a bracket. For example, TC-001 tests requirement number 259. The corresponding G/A REQ-259 is thus expected to be activated (and passed). Since all logs stem from passed regression test cases of a well-tested system, we expect all such cells with a bracket to carry a [P]. However, without any adaptation, it turned out that a) conditions for all expected G/A were not present in the test data set ([-]) and b) several G/A failed, although the test data was expected to show passed ([F]), and c) some G/A failed in all test runs (F). In our case, any fails or ([-]) are false positives since we use reliable logs from a well-tested system.



Regr Log G/A	TC-001	TC-003	TC-005	TC-007	TC-011	TC-013	TC-016	TC-017	TC-018	TC-019	TC-020	TC-021	TC-024	TC-068
REQ-244	P	P	F	P	P	P	[P]	P	P	P	P	F	[P]	P
REQ-245	-	-	-	F	-	-	[F]	-	-	-	-	-	-	-
REQ-246	-	-	-	-	-	-	-	[P]	-	-	-	-	P	P
REQ-248	-	-	-	-	-	-	-	-	[F]	-	-	-	-	-
REQ-253	-	[F]	-	-	-	-	-	-	-	-	-	-	-	-
REQ-254	-	-	[P]	-	-	-	-	-	-	-	-	-	-	-
REQ-255	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-258	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-259	[F]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-260	-	-	-	-	[F]	-	-	-	-	-	-	-	-	-
REQ-281	F	F	F	F	F	F	F	F	F	F	F	[F]	F	F
REQ-283	F	F	F	F	F	F	F	F	F	[F]	F	F	F	F
REQ-290-A1	F	F	F	F	F	F	F	F	F	F	[F]	F	F	F
REQ-290-A2	F	F	F	F	F	F	F	F	F	F	[F]	F	F	F
REQ-349	F	F	F	F	F	[F]	F	F	F	F	F	F	F	F
REQ-350	-	-	-	[F]	-	-	-	-	-	-	-	-	-	-
REQ-456-A1	-	-	-	-	-	-	-	F	-	-	-	-	P	[F]
REQ-456-A2	-	-	-	-	-	-	-	-	-	-	-	-	-	[-]

Figure 34. Initial Translation evaluated over a set of passed logs.

5.5.4 Phase II – Implementation Analysis Results

This step's expected outcome is a mapping between the abstract (logical) signals and the concrete (technical) signals that can be used for directly evaluating the abstract G/As. We performed this step concurrently with the requirement analysis step to facilitate the translation.

This step's most significant challenges concern consistency, technical signal identification, observability, maintenance, and signal scoping. In an initial attempt, as a preparation for this case study using other requirements without standardized logical signals, the mapping consistency quickly eroded into multiple/duplicate definitions and resulted in constant updates of the automated test cases to obtain the required signals logged. Identifying the correct signal among the tens of thousands available was another challenge aside from the experienced observability issues. Some requirements include non-observable internal signals. Often, other signals could be used as a proxy. The challenge is to understand how the non-observable signal affects the observable signal. The next challenge is maintaining the mappings when the implementation changes, such as an updated source, emitting the signal. Lastly, there are always more signals available in the general case than it is possible to log. For a practical application of passive testing, this has a significant impact on the method's usefulness. If a single signal is missing, a whole passive test case is rendered useless and will not even evaluate, resulting in a potential false positive. Further, for each set of signals to log, all the test cases need to be executed. Keeping track of the log-sets by hand was tedious and error-prone.

In the upcoming paragraphs, we present identified solutions to these challenges. Firstly, having a standardized set of logical signals used for all (at least a relevant subset) requirements is critical for consistency. Listing 2 shows examples of the outcome of this step using this strategy.

```
1  // Separating from implementation / versions
2  alias S_DrEnRgtTrLn_P =
3        MWT.xxxxx.C2M22m1In3_S_DrEnRgtTrLn_P
4  alias V_DrEnRgtTrLn_P =
5        MWT.xxxxx.C2M22m1In3_V_DrEnRgtTrLn_P
6  // Abstracting away validity handling
7  def intervals DrEnRgtTrLn_P =
8      S_DrEnRgtTrLn_P == true
9      and valid_DrEnRgtTrLn_P
10 // ...and so on
11 // Redundant validated signals using above aliases
12 def intervals
       MIO_S_Safe_door_enable_right_TRUE_and_VALID =
13   (S_DrEnRgtTrLn_P == true and
14    V_DrEnRgtTrLn_P == true) or
15   (S_DrEnRgtTrLn_R == true and
16    V_DrEnRgtTrLn_R == true)
```

Listing 2. Partially Obfuscated Signal Definitions

Further, Lines 2-3 and 4-5 show a low-level separation of logical signals and their binding to a particular implementation. Such low-level separation allows for automating the logical to technical signal mapping, contributing to solving the maintenance challenge. In the example, the logical signals gradually increase the abstraction level (Line 7 and finally, Lines 12-16). The "final" signal on Line 12-16 is the "standardized" logical signal used. Theoretically, it would be possible to use the logical names in the INPUT/OUTPUT section as-is, but the presence of white-space and the slight variations (e.g., CCU IP OUT / OUT CCU IP) are deemed to produce hard to find errors. Hence, the T-EARS name was created by replacing white-space with an underscore, reformatting the IN/OUT and BUS info to the same order everywhere. While translating, the tester copy/pastes the logical signal's natural language name into a search view and gets the closest matching T-EARS names. The above method had a substantial impact on the translation effort.

In the case study, we identified two approaches for increasing the logging-efficiency and possibly reducing the scoping challenge. The first approach concerns telegrams where each bit corresponds to a digital signal. Depending on the logging framework, logging the telegram rather than the individual signals may drastically increase the number of logged signals. If the logged telegram is a 16-bit integer, we can log the entire telegram and let T-EARS mask out the individual signals using the bitmask function. Such a mapping can typically be automated using a template, as shown in Listing 3.

```
// Logging several signals in same telegram
// Bitmask out individual signals
def interval my_bin_signal =
   bitmask(TelegramXYZ, my_bin_sig_mask) ==
        my_bin_sig_mask
// Defining an optional VALID signal
def intervals valid_DrEnRgtTrLn_P =
      select(exists(Valid_DrEnRgtTrLn_P)
                    Valid_DrEnRgtTrLn_P,
                    true)
def interval DrEnRgtTrLn_P =
        S_DrEnRgtTrLn_P and
        valid_DrEnRgtTrLn_P
```

Listing 3. Tricks For Increasing Log Information

Using the T-EARS select and exists, the validity signal of fail-safe signals can be made optional, as shown in Line 6-9 in Listing 3. The defined signal is equal to the validity signal if it is logged or always true if it is not logged. Using optional validity signals only works on fail-safe signals with a guaranteed fail state whenever the validity signal is false.

5.5.5 Phase II – Concretization Results

The concretization step's expected outcome is a G/A evaluated using concrete (technical) signals. Given our suggestions for the implementation step, this is accomplished by specifying the signal mapping. We did not observe any challenges in this step.

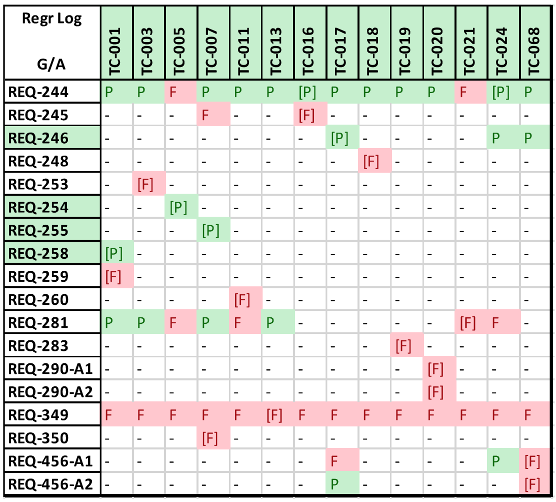
5.5.6 Phase II – Tuning and Validation Results

This step's expected result is an executable and complete G/A with a minimum of false positives. A G/A is considered complete when there are no unknown dependencies that would cause a wrong verdict. A fictitious example is a G/A only checking that the brake light is turned off when lifting the brake pedal, but another subsystem issues a brake order that causes the test to fail. False positives include such conditions and G/As are not being activated when supposed to or failing where there is no underlying fault. In this step, the tester logs required signals while operating the system to activate the G/A. Eliminating false positives turned out to be the most challenging part of the entire process. To find out how to do this systematically, we tuned our G/As against our gold standard until no more false positives were encountered. The result is presented as a systematic process containing the steps outlined in the following sub-sections.

5.5.6.1 Validating Guard Activation

The expected outcome of this step is a guard that is activated when expected and not activated when not expected. In the general case, the tester operates the system (while logging the appropriate signals) until she knows that the system state is correct. For our gold standard, we already know which G/A should be active for each log. Thus, this step's expected outcome is that each G/A is activated at least once for the logs testing the G/As requirement. Such expected activations are marked with brackets in Figure 34. G/A activations for other logs are not necessary but welcome if they do not produce false positives. Notably the REQ-456-A2 in Figure 34 has not been activated when expected.

We identified possible sources of missing activation(s), including input stimuli sequence, log, signal mapping, and requirement. The first source, input stimuli sequence, is the easiest to investigate. Examining the test case actions should reveal if the test does not put the system in a testable state (as it should have). In our case, we know that the logs are correct, so the cause of the missing activation of TC-068 and REQ- 456-A2 in Figure 34 must be related to some other reason. If the evaluation completely fails for one log but looks fine for another, the log probably lacks one or more signals. Another reason for not activating (or giving a faulty evaluation) may be that a signal is captured with the right name but with the wrong values. This may happen if the test framework injects faults or alter signals for testing purposes. Another common mistake is that a logical signal is defined with the wrong technical signal (e.g., a logical signal from cab A1 and cab A2 are mapped to the same technical signal by mistake). In this study, we experienced all the above, mostly due to a lack of proper automation of these tasks. The last category, requirement, includes reasons such as the requirement lack of information to capture the testable state, or the requirement text is misunderstood. It may also be missing signals or unknown dependencies as in the example of the brake pedal. In our example, it turned out that REQ-456- A2 suffered from misunderstandings of the requirement text. Updating the conditions of the guards activated the G/A for TC-017 and TC-068, as seen in Figure 35. This also shows that the G/A still fails for TC-068 and more tuning is required for the G/A.



Regr Log G/A	TC-001	TC-003	TC-005	TC-007	TC-011	TC-013	TC-016	TC-017	TC-018	TC-019	TC-020	TC-021	TC-024	TC-068
REQ-244	P	P	F	P	P	P	[P]	P	P	P	P	F	[P]	P
REQ-245	-	-	-	F	-	-	[F]	-	-	-	-	-	-	-
REQ-246	-	-	-	-	-	-	-	[P]	-	-	-	-	P	P
REQ-248	-	-	-	-	-	-	-	-	[F]	-	-	-	-	-
REQ-253	-	[F]	-	-	-	-	-	-	-	-	-	-	-	-
REQ-254	-	-	[P]	-	-	-	-	-	-	-	-	-	-	-
REQ-255	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-258	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-259	[F]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-260	-	-	-	-	[F]	-	-	-	-	-	-	-	-	-
REQ-281	P	P	F	P	F	P	-	-	-	-	-	[F]	F	-
REQ-283	-	-	-	-	-	-	-	-	-	[F]	-	-	-	-
REQ-290-A1	-	-	-	-	-	-	-	-	-	-	[F]	-	-	-
REQ-290-A2	-	-	-	-	-	-	-	-	-	-	[F]	-	-	-
REQ-349	F	F	F	F	F	[F]	F	F	F	F	F	F	F	F
REQ-350	-	-	-	[F]	-	-	-	-	-	-	-	-	-	-
REQ-456-A1	-	-	-	-	-	-	-	F	-	-	-	-	P	[F]
REQ-456-A2	-	-	-	-	-	-	-	P	-	-	-	-	-	[F]

Figure 35. Activation and Start-up (ignore) Tuned

5.5.6.2 Systematic Issues

This step is applicable if there exists a set of logs and a suspicion that the false positives are due to some systematic disturbance. One such observed systematic disturbance was identified at the beginning of most log files. Such turbulence may occur in some signals when starting up and tearing down the environment. This is especially true for simulated environments and may generate enormous amounts of false fail indications. However, cutting the log in either end is not without risks, especially when a passive test case relies on a sequence to occur or complete. Even trivial sequences like a button toggle may wreck the entire evaluation of a passive test case, e.g., if the first press in the log file is ignored, so this activity must be done in the context of each passive test case.

To investigate whether false positives stem from such start-up problems, we need to bring up a detailed evaluation view over several logs, preferably from different testing sessions.

Figure 36 illustrates such a view for spotting systematic fails. The figure shows the evaluation of the G/A REQ-281 over six logs. Each sub-figure shows the evaluation over the corresponding log file. The plots in each sub-figure show the guard passes and fails as binary signals. The grey signal at the bottom (guard interval) goes high whenever the system is in a state where the requirement should be validated. The (red) signal in the middle graphs goes high whenever the requirement is not met during the guard interval (fail). Similarly, the topmost (green) signal goes high, where the requirement is met (pass).

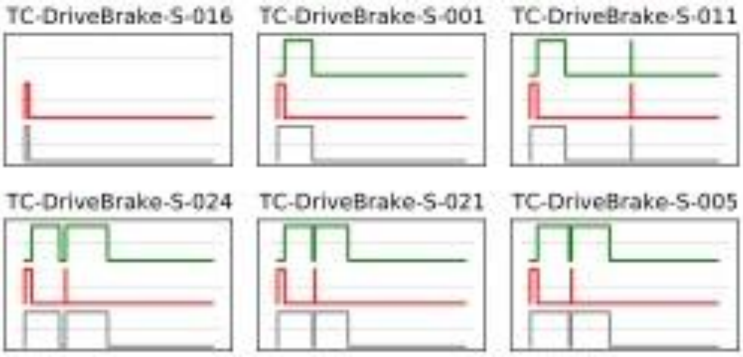


Figure 36. Systematic Fail Analysis, REQ-281

Focusing on the very first part of each log evaluation, we observe a fail at the very beginning of each evaluation for REQ-281. Further, the fail interval is too long to be explained by natural latencies or sampling effects in the system. According to our domain expert, this requirement was sensitive to some start-up adjustments in the simulator. In this study, only 4 out of 17 passive test cases were susceptive to such start-up disturbances, emphasizing the recommendation against a default ignore. For those four test cases, we ignored the first 32 seconds using the keyword ignore, as shown at line 2 in Listing 4.

```
1   // Tuning for Simulated Rig
2   ignore < 0s
3   allow 500ms fail
4   const TIMEOUT = 500ms
5   // ...
6   'REQ-456-A2' =
7   while Cab_doors_closed_and_locked_in_cab_A2 == false
          and
8         Standstill == true
9         and
10        Bypass_active_in_ready_to_run == true
11  shall
12        Traction_safe_command == false
13        and
14        bitmask(CabDrOp,
15                MWT_traction_block_reason) == CabDrOp
16        and
17        Allow_traction == false
18  within TIMEOUT
19  // ...
```

Listing 4. Example G/A With Timing Specifications

This removed all confirmed false positives during start-up. However, still, many passive test cases failed while they should pass.

5.5.6.3 Latencies and Sampling Effects on Assertions

Aside from the start-up turbulence, one distinct type of false fail, observed in the study, is shown in Figure 37 at approximately 300s. In the example, the guard starts with a very short fail period but then passes until the end of the guard period.

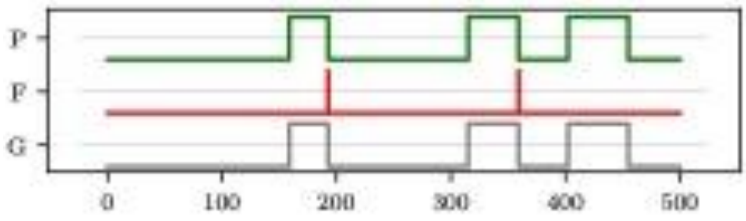
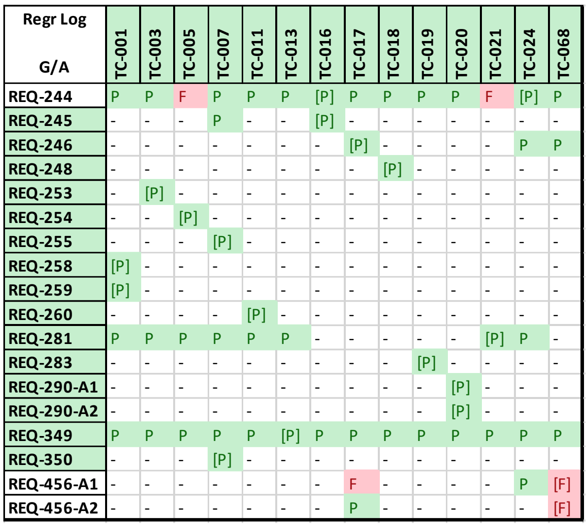


Figure 37. Example of Natural Latency Fails of a (Should-Be) Passing G/A.

A typical example in this study is REQ-245; when the doors are open, a start-inhibit signal should be set. According to our observations, when the doors are open, it takes a (short) while until the start-inhibit signal is set. Even though the requirement contains timing information, using G/As to explore assertion latencies, together with proper domain knowledge, allows establishing better, worst, and acceptable latency margins. Further, some safety-related signals are only indirectly observable at the system level, which adds to the specified timing in the requirement. The tester needs to judge whether the latencies are reasonable or not, a process that would be facilitated by such a tool. Although we call it assertion latency, other probable sources of such delays include sampling issues or different time-domain effects (e.g., A machine is used to simulate parts of the system and a real-time simulator other parts).

A tool that discovers harmless assertion latencies needs to make sure that the fail starts simultaneously as the guard and that it is followed by a substantially larger pass period to rule out other, potentially severe fails. Confirmed harmless assertion latencies can be ignored by adding the within, as demonstrated on line 18 in Listing 4, the passive test case ignores latencies fails up to the specified time limit (500ms). Although it is tempting to use a sizeable global time limit, this may conceal severe problems in the system. On the other hand, setting the within too narrow will give false fails due to variations in the active test case logs. The added within statements (between 200-500ms) solved all assertion latencies as defined above and shown in Figure 38. REQ-254 and REQ- 456 are still failing.



Regr Log G/A	TC-001	TC-003	TC-005	TC-007	TC-011	TC-013	TC-016	TC-017	TC-018	TC-019	TC-020	TC-021	TC-024	TC-068
REQ-244	P	P	F	P	P	P	[P]	P	P	P	P	F	[P]	P
REQ-245	-	-	-	P	-	-	[P]	-	-	-	-	-	-	-
REQ-246	-	-	-	-	-	-	-	[P]	-	-	-	-	P	P
REQ-248	-	-	-	-	-	-	-	-	[P]	-	-	-	-	-
REQ-253	-	[P]	-	-	-	-	-	-	-	-	-	-	-	-
REQ-254	-	-	[P]	-	-	-	-	-	-	-	-	-	-	-
REQ-255	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-258	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-259	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-260	-	-	-	-	[P]	-	-	-	-	-	-	-	-	-
REQ-281	P	P	P	P	P	P	-	-	-	-	-	[P]	P	-
REQ-283	-	-	-	-	-	-	-	-	-	[P]	-	-	-	-
REQ-290-A1	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-290-A2	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-349	P	P	P	P	P	[P]	P	P	P	P	P	P	P	P
REQ-350	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-456-A1	-	-	-	-	-	-	-	F	-	-	-	-	P	[F]
REQ-456-A2	-	-	-	-	-	-	-	P	-	-	-	-	-	[F]

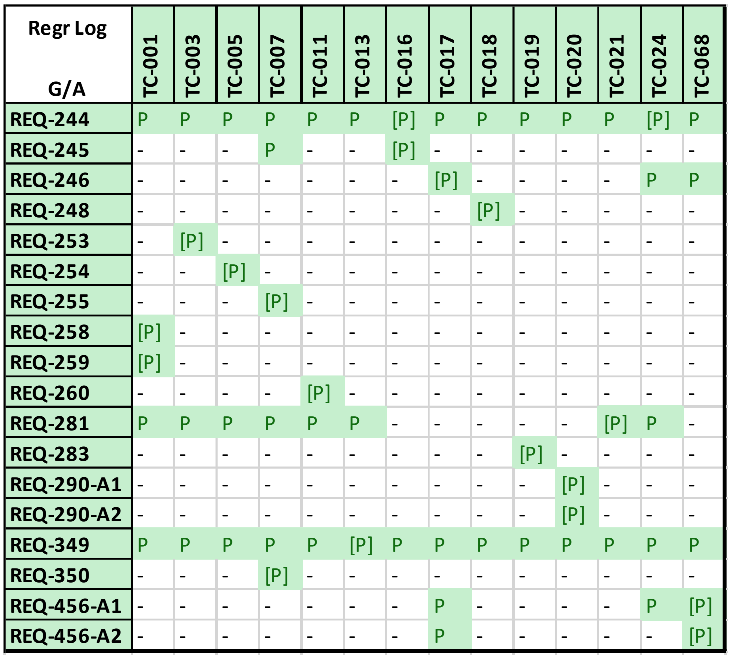
Figure 38. Assertion Latency (within) Tuned

5.5.6.4 Latencies and Sampling Effects on Guards

Another distinct source of false positives observed in the study is exemplified in Figure 37 at approximately 180s. The guard mostly passes but fails for a tiny part at the end of the guard interval. It appears as if the guard condition ended too late. This behaviour was observed in, e.g., REQ-245, where traction seems to be allowed just before the doors were closed and locked. Again, exploring fail intervals, together with proper domain knowledge, allows establishing an acceptable range for guard latencies. Although we call it guard-latency for simplicity, there may be other sources of such delays as for the assertion latencies. A tool that automatically finds such fails would match all fail-intervals within a guard-interval, immediately preceded by a pass-interval and ending where the guard ends.

Confirmed harmless guard-latencies can be ignored by adding the allow, as demonstrated at line 3 in Listing 4. The keyword specifies to the G/A to ignore guard latencies up to a specified time limit like within, although it is tempting to use a sizeable global time limit, this may conceal severe problems in the system. On the other hand, setting the allow too narrow will give false fails due to variations in the active-test-case logs.

In our case, the engineers concluded that the delays up to twice the sample time were acceptable. Given an analysis of the failing G/As, allow was added up to double the sampling time. The remaining test cases were left unchanged. The added slack addressed all remaining latency/sampling effect problems for all our G/A over the available log files, as presented in Figure 39.



Regr Log G/A	TC-001	TC-003	TC-005	TC-007	TC-011	TC-013	TC-016	TC-017	TC-018	TC-019	TC-020	TC-021	TC-024	TC-068
REQ-244	P	P	P	P	P	P	[P]	P	P	P	P	P	[P]	P
REQ-245	-	-	-	P	-	-	[P]	-	-	-	-	-	-	-
REQ-246	-	-	-	-	-	-	-	[P]	-	-	-	-	P	P
REQ-248	-	-	-	-	-	-	-	-	[P]	-	-	-	-	-
REQ-253	-	[P]	-	-	-	-	-	-	-	-	-	-	-	-
REQ-254	-	-	[P]	-	-	-	-	-	-	-	-	-	-	-
REQ-255	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-258	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-259	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-260	-	-	-	-	[P]	-	-	-	-	-	-	-	-	-
REQ-281	P	P	P	P	P	P	-	-	-	-	-	[P]	P	-
REQ-283	-	-	-	-	-	-	-	-	-	[P]	-	-	-	-
REQ-290-A1	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-290-A2	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-349	P	P	P	P	P	[P]	P	P	P	P	P	P	P	P
REQ-350	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-456-A1	-	-	-	-	-	-	-	P	-	-	-	-	P	[P]
REQ-456-A2	-	-	-	-	-	-	-	P	-	-	-	-	-	[P]

Figure 39. Guard Latency (allow) Tuned

5.5.6.5 Remaining Issues and Root Cause Analysis

When all of the above standard procedures fail to explain a fail interval in a passive test case, the chances are that we are facing a real bug, unknown feature interaction, or insufficient/faulty requirement description. In either case, the remaining fails need to be examined closely. Figure 40 shows such a fail from an early stage in the translation process. In this case, the failure turned out to be a misinterpretation of the signal redundancy, which led to the use of an AND operator instead of an OR operator.

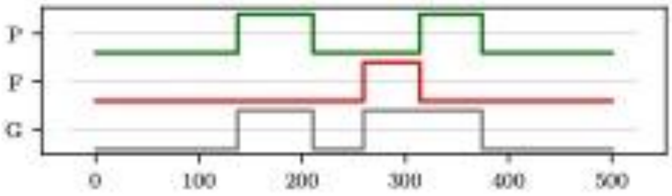
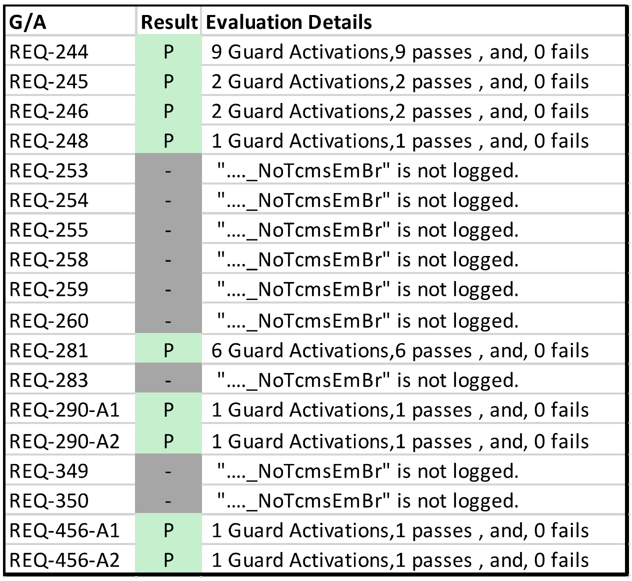


Figure 40. Fail in test case that requires root cause analysis.

5.5.7 Phase III – Final Evaluation

The 116 SAFE requirements were examined and are estimated to be applicable for 74 of the 116 reviewed SAFE requirements, which is in line with a previous case study using an early prototype [22].

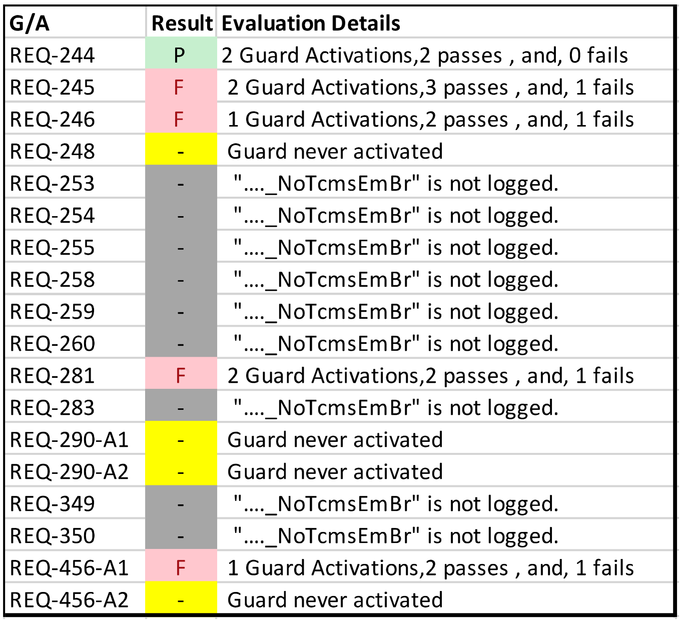
In the remaining section, we show the results from two logged manual sessions performed by an expert. The first session is performed on a well-tested and released system in a HIL-rig, as opposed to our gold standard produced in a completely simulated environment. The expert tester's goal is to cover as much of the functionality during the session. The purpose of the first session is to validate that the false positives have been tuned away and demonstrate how passive test cases can be used to understand requirement coverage. A fail for the REQ-246 revealed that the timing was slightly different for the HIL-rig signals compared to the simulator. After adding an allowance of 250ms, the G/A passed. The results are presented in Figure 41. Each row shows the evaluation of a G/A, as translated in previous sections. A green "P" shows that the G/A could be evaluated, did not fail, and passed at least once. The "Evaluation Details" column show how many times the G/A was activated (guard activations) and the total number of fails and passes during the logged session. The grey "- " denotes a G/A that could not be evaluated. In this case, due to a missing signal. Having missing signals is a common situation since there is often a restriction on how many and which signals can be logged. Since the expert did not log that signal, the corresponding G/As could not be evaluated. However, the remaining G/As show that the achieved coverage was at least 9 out of 18 G/As. Since this system was extremely well tested, the absence of fails shows that the tuning has been successful. A closer look at REQ-244 in Figure 41 reveals that the requirement has been tested nine times during the session, and REQ-281 six times.



G/A	Result	Evaluation Details
REQ-244	P	9 Guard Activations,9 passes , and, 0 fails
REQ-245	P	2 Guard Activations,2 passes , and, 0 fails
REQ-246	P	2 Guard Activations,2 passes , and, 0 fails
REQ-248	P	1 Guard Activations,1 passes , and, 0 fails
REQ-253	-	"…._NoTcmsEmBr" is not logged.
REQ-254	-	"…._NoTcmsEmBr" is not logged.
REQ-255	-	"…._NoTcmsEmBr" is not logged.
REQ-258	-	"…._NoTcmsEmBr" is not logged.
REQ-259	-	"…._NoTcmsEmBr" is not logged.
REQ-260	-	"…._NoTcmsEmBr" is not logged.
REQ-281	P	6 Guard Activations,6 passes , and, 0 fails
REQ-283	-	"…._NoTcmsEmBr" is not logged.
REQ-290-A1	P	1 Guard Activations,1 passes , and, 0 fails
REQ-290-A2	P	1 Guard Activations,1 passes , and, 0 fails
REQ-349	-	"…._NoTcmsEmBr" is not logged.
REQ-350	-	"…._NoTcmsEmBr" is not logged.
REQ-456-A1	P	1 Guard Activations,1 passes , and, 0 fails
REQ-456-A2	P	1 Guard Activations,1 passes , and, 0 fails

Figure 41. Resulting G/As, Expert Session I (Well-Tested)

In the second session, the test engineer injected two intermittent faults in the system that would be difficult to detect using traditional scripted testing. The result is presented in Figure 42. Again, a signal was not logged, so the grey "-" does not provide any information. There are, however, some yellow "-" (Guard never activated); a never activated guard means that the tester has not covered the corresponding requirement in the session, which was confirmed by the expert tester in this session. The red "F" shows where the G/A detected violations of the requirements. Analysis of the failed requirements (G/As) against the injected faults concluded that the two faults affected the system in a way that make it violate exactly these four requirements (G/As) during the short time the faults were injected.



G/A	Result	Evaluation Details
REQ-244	P	2 Guard Activations,2 passes , and, 0 fails
REQ-245	F	2 Guard Activations,3 passes , and, 1 fails
REQ-246	F	1 Guard Activations,2 passes , and, 1 fails
REQ-248	-	Guard never activated
REQ-253	-	"...._NoTcmsEmBr" is not logged.
REQ-254	-	"...._NoTcmsEmBr" is not logged.
REQ-255	-	"...._NoTcmsEmBr" is not logged.
REQ-258	-	"...._NoTcmsEmBr" is not logged.
REQ-259	-	"...._NoTcmsEmBr" is not logged.
REQ-260	-	"...._NoTcmsEmBr" is not logged.
REQ-281	F	2 Guard Activations,2 passes , and, 1 fails
REQ-283	-	"...._NoTcmsEmBr" is not logged.
REQ-290-A1	-	Guard never activated
REQ-290-A2	-	Guard never activated
REQ-349	-	"...._NoTcmsEmBr" is not logged.
REQ-350	-	"...._NoTcmsEmBr" is not logged.
REQ-456-A1	F	1 Guard Activations,2 passes , and, 1 fails
REQ-456-A2	-	Guard never activated

Figure 42. Resulting G/As, Expert Session II (Fault-Injected)

5.6 Analysis of the Results for O1 and O2

The first objective O1 in Adeptness was set as: "To demonstrate an increment in software quality for CPSoS".

The second objective O2 in Adeptness was set as: "To demonstrate a reduction in the re-commissioning cost of software releases for CPSoS while guaranteeing its reliability".

The relevant solutions and figures that are developed and validated in the railway use case relevant for these objectives are mentioned below:

- Automatic generation of test cases providing 100% edge coverage.
- Automatic generation of test cases providing higher frequencies of requirements coverage than manually written test cases, the maximum requirements coverage frequencies vary in the range of 26 to 84. In comparison, manually written test cases cover each requirement only once.
- Automatic generation of test scripts that provide 100% requirements coverage.

- Validation of the correctness of the test scripts as no faults identified on the correct model, confirming the 100% correct generation and execution, ensuring conformance to the requirements specifications and the implemented SUT.

- Validation of the correctness of the test scripts as faults identified with the faulty model.

- Automatic generation of model-based test cases achieves highest median MC/DC coverage of 90.27% in comparison with manual testing (79%) and 4-ways combinatorial testing (80%).

- Automatic generation and execution of test scripts using model-based testing is more cost efficient in terms of time than manual testing, 27900 seconds (manual) vs. 20107 seconds (automatic).

- Model-based testing generated the greatest number of unique test cases (9) in comparison with manual (2) and 4-ways combinatorial testing (3).

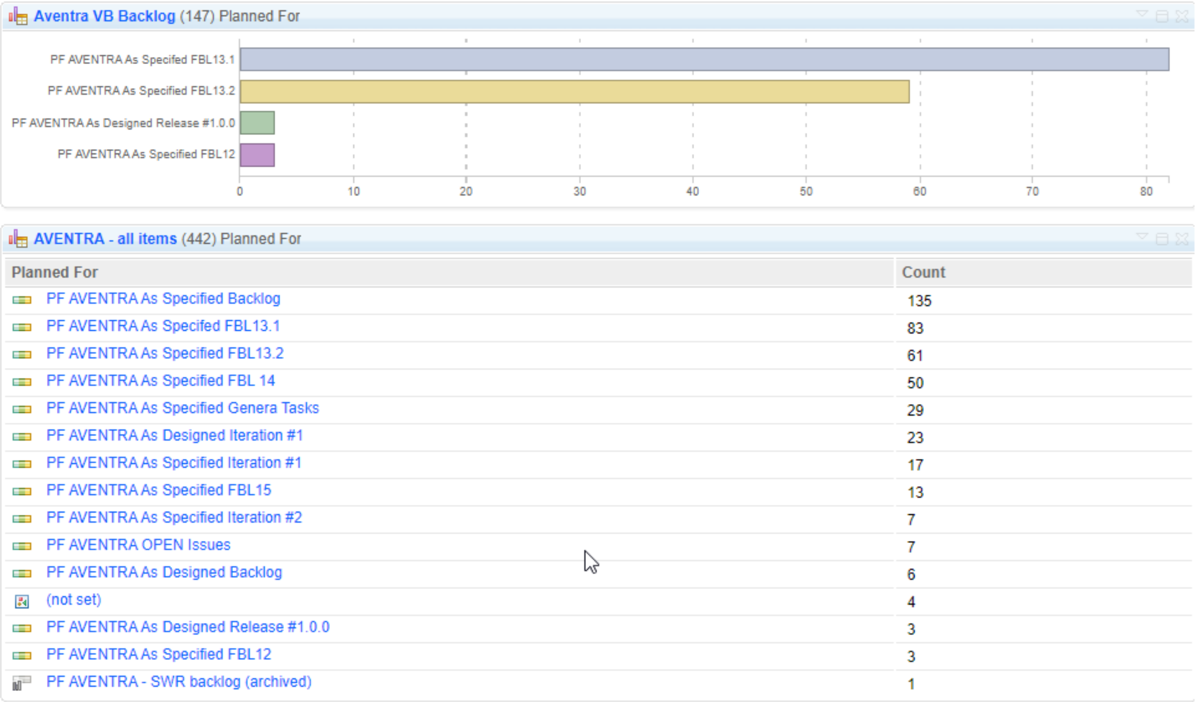
- Automatically generated model-based test cases contained approximately 71% of similar test cases, on average, generated by manual and combinatorial testing.

- Automatically generated model-based test cases achieve better mutation score (88%) as compared to manually written test cases (86%).

- Automatically generated model-based test cases have a smaller number of alive mutants (6) than manually written test cases (7).

- 100% mutation score achieved through continuous monitoring of the system under test using T-EARS.

The above results are in in line with internal Bombardier investigations on modelling and coverage enhancement. For modelling, an internal Bombardier investigation reveals that modelling leads to an increased identification of issues. This is shown in Figure 43 below, where the count of issues shows an increasing trend because of modelling of requirements.

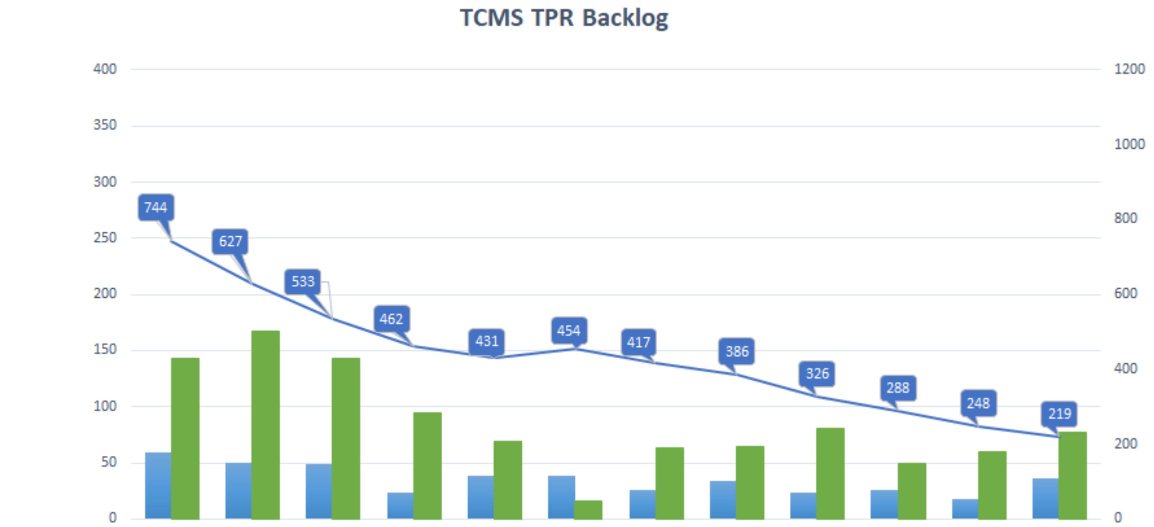

Aventra VB Backlog (147) Planned For
PF AVENTRA As Specifed FBL13.1
PF AVENTRA As Specified FBL13.2
PF AVENTRA As Designed Release #1.0.0
PF AVENTRA As Specified FBL12
0
10
20
30
40
50
60
70
80

AVENTRA - all items (442) Planned For

Planned For	Count
PF AVENTRA As Specified Backlog	135
PF AVENTRA As Specifed FBL13.1	83
PF AVENTRA As Specified FBL13.2	61
PF AVENTRA As Specified FBL 14	50
PF AVENTRA As Specified Genera Tasks	29
PF AVENTRA As Designed Iteration #1	23
PF AVENTRA As Specified Iteration #1	17
PF AVENTRA As Specified FBL15	13
PF AVENTRA As Specified Iteration #2	7
PF AVENTRA OPEN Issues	7
PF AVENTRA As Designed Backlog	6
(not set)	4
PF AVENTRA As Designed Release #1.0.0	3
PF AVENTRA As Specified FBL12	3
PF AVENTRA - SWR backlog (archived)	1

Figure 43. Modelling of requirements leads to an increased count of issues.

Similarly, internal Bombardier investigations also reveal that an increased test coverage is good surrogate measure for keeping the issue backlog smaller since a lot of issues are identified and solved quickly. So, the set of open and closed issues get smaller, which is a positive quality indicator. This is shown in Figure 44.



	Jun-21	Jul-21	Aug-21	Sep-21	Oct-21	Nov-21	Dec-21	Jan-22	Feb-22	Mar-22	Apr-22	May-22
TPR Created	59	50	49	24	38	39	26	34	23	26	18	36
TPR Closed	143	167	143	95	69	16	63	65	81	49	60	77
TPR Backlog	744	627	533	462	431	454	417	386	326	288	248	219

Figure 44. Bombardier’s analysis of issue backlog shows a declining trend in open and closed issues due to increased test coverage.

The above two figures indicate that the results achieved in Adeptness are in line with Bombardier’s internal investigations.

5.7 Analysis of the Results for O3

The third objective O3 in Adeptness was set as: “To increase synergies and collaborations between sector-leading companies and academic partners”.

The relevant measures relevant for this objective are mentioned below:

- Development of several tool prototypes to expedite collaboration with companies and academic partners. These tool prototypes include a test script execution framework, a cloud-based GUI for integrating deployment and testing processes and a toolchain based on passive testing.
- Development of a workflow where several different technologies get integrated, belonging to deployment, testing, and monitoring.
- Continuous dissemination of results in leading global conferences.

5.8 Discussion of the Results

5.8.1 Discussion on the Results on Modelling Aspects, Behavioural Differences Between Test Cases and Initial Experience

Requirements and test specifications help in understanding the behavioural aspects of the SUT in a better way. These documents not only provide information about the SUT but also cover the testers' perspective, thus covering all scenarios meeting the test objectives.

At our industrial partner, manually written test specification follow a sequential order to cover requirements. The test steps to cover the first requirement are written and executed first and engineers follow this order in a systematic way. Conversely, through GW when the random generator is selected, test cases are randomly generated. Hence, MBT-generated test cases can exploit the SUT through exploring different paths, increasing the chances of uncovering unknown faults and interesting interaction scenarios not possible through a sequential execution order of test steps.

During GW test generation, we also tried with other avail- able path generators and stopping conditions but identified certain limitations. For this study, we only considered the path generator and stopping conditions that provide 100% requirements coverage as this is an important metric for BT at this level of testing. While using the quick_random path generator with different stopping conditions, we were unable to generate test cases because GW started traversing a specific path in a continuous loop, unless a threshold value was reached for the test steps, resulting in too many test cases with similar test steps. Weight_random path generator requires weight at each edge, which represents the probability of an event to happen; this information was not available in our industrial scenario, thus Weight_random path genera- tor was not used. Similarly, A_star path generator generates the test steps for a specified node or edge, hence is unable to provide 100% requirements coverage. Moreover, GW4E does not support the time_duration stopping condition, so we were unable to generate the test cases using this stop- ping condition. However, while experimenting, we found that the time provided as a parameter to time_duration stop- ping condition in GraphWalker studio can affect the number of generated test steps, model, and requirements coverage as well as frequency of requirements coverage. Similar effects were also observed using the length stopping condition as number of generated test steps depends on the value of length provided. A systematic experimentation to quantify the effects of such parameter changes is left as a future work.

5.8.1.1 Broader Impact

Here we discuss any potential negative impact of our research. The evidence regarding the prevailing use of MBT is rather limited. This is especially problematic if we consider relying on MBT for thoroughly testing industrial safety-critical systems (e.g., trains, cars, nu- clear power plants) where failures can lead to loss of human lives. Our aim is to provide aid to the testers in test case and test script generation, so that they can invest their time in more productive activities of investigating root cause analysis of bugs and to design better testing scenarios, helping them to optimize relevant coverage criteria. The purpose is not to replace them. Moreover, MBT requires human effort, such as in creation of correct models and to correctly generate concrete test cases. Our results investigate the use of MBT and identify the empirical evidence for, or how to improve, the use of it in practice when testing industrial safety-critical systems. In addition, our results aim to pro- vide more evidence on how to improve the adoption and deployment of MBT in an industrial setting as well as how the resulting test cases can perform comparably with manual test design performed by industrial engineers.

5.8.2 Discussion on the Results on Fault Injection Analysis to Evaluate TIGER-Generated Test Scripts

We have used three different mutation operators to evaluate TIGER, however, we also tried with other mutation operators. These other mutation operators did not result in any different behaviour in our case. For example, we induced some faults based on 'arc missing', 'output missing', 'event missing', 'destination exchanged' and 'event exchange' operators. But event and destination exchange had similar effect on the model as output exchange. Similarly, output missing, event missing and arc missing showed no effect on the model, and induced faults based on these operators only resulted in less input combinations with no failed test steps, so we have neglected these operators in our study.

5.8.3 Discussion on the Results of MC/DC Adequacy, Requirements Coverage, Overlap of Test Cases and Performance Efficiency

Our results regarding the requirement coverage show that all the techniques, except 2-ways, achieved 100% requirement coverage. The techniques differed more with respect to the fulfilment of different conditions for MC/DC. Overall, the MC/DC condition that evaluates the effectiveness of the test suites in terms of 'independent effect on the outcome of a decision' was the least dominant condition fulfilled by all the test suites. The rest of the MC/DC conditions were met 100% by MBT, manual testing and higher strength CT (3- and 4-ways).

Our results have also shown that MBT gave far greater number of unique test cases when compared with CT and manual testing. In terms of efficiency, MBT is found to be better than manual testing but worse than CT. For CT, we observed that increasing the strength also increased the number of unique test cases along with increasing the overall number of test cases. CT was the most efficient of all the techniques but the absence of knowledge regarding expected outcomes makes CT dependent on tester's skills.

In an industrial setting where manual testing is performed, our results indicate the MBT and CT can both add important test cases, with MBT providing the greatest number of additions. Whether such additions are fault revealing or not is left as a future investigation but at least in terms of MC/DC coverage, MBT test suite also has given evidence in support of its advantage. Our results also showed that 2-way CT strategy does not perform better than higher strength CT strategies and 90% of its test cases are covered by MBT as well as higher strength CT gives more unique test cases, thus if CT is to be adopted, our results support utilising greater than 2-way strength CT.

What is also clear from our study is that there is no silver bullet when it comes to testing of safety-critical systems in terms of effectiveness and efficiency. A mix of techniques with an understanding of advantages and disadvantages in terms of efficiency and effectiveness seems like the best advice for industrial practitioners.

5.8.4 Discussion on the results of Fault Detection Effectiveness, Sensitivity of Test Suites and the Relationship Between MC/DC Coverage and Mutation Score

We see from our results that a high fault detection rate can be achieved using the higher interaction strength of CT. However, CT has an increasing cost of completing the test suite with expected outputs and timing constraints according to the system requirements. On the other hand, MBT generated a complete set of test suites containing inputs, expected outputs, and timing constraints from the model conforming to system requirements while achieving an 88% fault detection rate. Moreover, we also found the MBT-generated test suite is complete (i.e., contained expected output, and timing constraints) and ready to be executed as well as like specification-based manual testing used in the industry.

The analysis of different mutant operators illustrates that all the test suites do not detect faults related to time constraints and thus special test cases should be included to target testing of timing proper- ties. Our results further suggest that test suites generated by CT 3-ways, CT 4-ways, and MBT achieved similar effectiveness in detecting all functional level faults according to the achieved mutation scores. A deeper analysis, however, shows that MBT did not achieve a high mutation score for the operator LRO-I when compared with CT 3-ways and CT 4-ways and missed generating specific fault-revealing input combinations due to the random coverage criteria.

Our results also indicate that, in general, the higher MC/DC coverage corresponds to a higher mutation score. However, there are subtle differences among different techniques. For example, the difference between the achieved MC/DC coverage and mutation scores in the case of CT 2-ways and CT 3-ways is greater between other techniques.

Lastly, we argue that a mutation analysis at the FBD level alone is not sufficient to measure the fault detection effectiveness of test suites, particularly when safety critical systems are concerned, and further analysis should be conducted at the code level in industrial settings. The reason is that for similar requirements, if one FBD program is used to generate different instances of code, then a limited number of mutants can be induced as well as a fault at the FBD level can also produce multiple faults in different areas of the generated code. This has an impact on the mutation score that can be achieved per test technique. Similarly, it is also possible that a test suite may not achieve significant effectiveness in terms of fault detection rate at the code level or if different FBD pro- grams are used to generate the code for similar requirements.

5.8.5 Discussion on Results Regarding the Practical Implications of Adopting Passive Testing using T-EARS and the Accompanying Toolchain

Faulty software in safety-critical systems may cause economic damage and, in some cases, loss of human lives. At the same time, the embedded software in today’s safety-critical systems grows more and more complex, which emphasizes the importance of efficient and realistic testing. One of the methods suggested allowing that is passive testing. The idea is to only observe the system under test, which allows, e.g., parallel execution of test cases. In this approach, test cases only monitor the system under test (SUT) and do not alter the state of the system at all. Instead, requirements are verified whenever appropriate, independently of the input stimuli sequence. Most of these approaches target non-critical software testing, such as protocol testing in, e.g., web applications and telecom applications. Such applications often involve sending and receiving complex data. While the current passive testing methods have a long history for these applications, the safety-critical software we address does not require handling comprehensive data transfers and complex state machines. Further, due to the complexity of the data and the state machines in the original domain, the specification languages used are very complex and often based on mathematical expressions. Although there have been attempts to introduce passive testing to the system-level testing of safety-critical systems, these studies are all somewhat limited when it comes to industrial validation and adoption. Also, these do not consider practical issues and tooling requirements.

5.9 Limitations (Threats of Validity)

5.9.1 Limitations of the Results on Modelling Aspects, Behavioural Differences Between Test Cases and Initial Experience

5.9.1.1 Internal Validity

Internal Validity. One internal threat relates to learning since the second version of the model could be modelled in a better way because we gained more experience with GW and understood the industrial domain better with time. However, we mitigated this factor by consulting and getting continuous input on the correctness of both the SUT models from the testing team at BT.

5.9.1.2 External Validity & Reliability

We argue that if an- other person with similar experience of the modelling environment and testing domain knowledge will replicate this study, the results should be similar, disregarding GW's randomness in test generation. However, different modelling notations and different test generation algorithms may pro- vide different results. Another issue is the number of repeated trials of test case generation to have valid data for evaluation. We repeated the test generation process for each stopping condition three times to consider the possible variations in the number of test steps generated each time and reported the highest number of generated steps.

5.9.1.3 Construct Validity

We investigated existing measures from the literature. For example, the modelling aspects used for comparison were inspired from an existing taxonomy of MBT. We also used well-known behavioural measures of test generation, while few were influenced by the industrial applicability point of view, such as frequency of requirements coverage as it is an important aspect of testing from an industrial perspective as it can uncover interaction faults on a system level.

5.9.2 Limitations of the Results on Fault Injection Analysis to Evaluate TIGER-Generated Test Scripts

One internal validity threat is regarding the correctness of the model of the SUT. It took us several rounds of modelling to completely understand the requirements and the test specifications to arrive at a correct model that was eventually confirmed as correct by a BT test engineer. Other threats relate to external validity and reliability such as human experience, modelling notations and generator algorithms. We expect that if a person with similar modelling and testing experience will replicate this study using random walk and edge coverage criterion of GW, similar results should be achieved. However, different modelling notations and generator algorithms may produce different results. Another issue is that the framework is specifically designed for the CPS testing at BT, so it has particularities that may not be applicable to other CPSs but still be applicable to multiple projects inside BT. Nevertheless, the description of the framework and the mapping procedure can give clues to companies operating in similar domains to apply MBT in practice. We may also want the mutation testing at the model level to be supplemented with lower, code-level mutations and then validate our framework. We did not have access to code for this study but if it becomes a possibility, this research direction is worth investigating.

5.9.3 Limitations of the Results of MC/DC Adequacy, Requirements Coverage, Overlap of Test Cases and Performance Efficiency

5.9.3.1 Internal Validity

One of the main threats to internal validity is the validation and conformance of the model with system requirements. However, we have alleviated this factor by creating the model in multiple iterations and continuously consulting with the testing team at BT, which eventually confirmed it as correct. Another issue relates to the indirect cost of testing techniques, such as maintenance cost of test suites and development time of tools like TIGER. In this study, we have not considered the effect of indirect costs and therefore, considering both direct and indirect costs may affect efficiency results. We have spent a fair amount of time to analyse all the test suites for coverage and efficiency manually; a manual analysis of such large amount of data can result in small errors but these should not be large to affect our results in any meaningful way.

5.9.3.2 Reliability and External Validity

The threats related to reliability and external validity include generalization of the MBT model, size of state space, complexity of the system, impact of human experience, generator algorithms and modelling notations. We are working with modelling and generation of test suites at system level for the subsystems of TCMS developed at BT, so it contains particularities for the generation of test suites that may not be applicable to other domains. However, for the replication of this study in a similar domain, we have tried to provide enough information about the experimentation setup. We cautiously argue that if another researcher with similar experience of the testing domain and modelling will replicate this study, similar results should be produced, however different modelling languages, testing tools and generator algorithms may affect the results. Moreover, for this study, we have used a part of the TCMS system with their actual number of parameters and constraints. Nevertheless, more case studies are required to generalize the results of this study to larger systems.

5.9.3.3 Construct Validity

The operational measures used in this study for cost estimation of testing techniques were inspired by an existing taxonomy of MBT and industrial practices at BT, Sweden. Moreover, the measure for requirement and MC/DC coverage for comparative analysis of test suites were determined by a thorough investigation of literature and industrial applicability of testing strategies.

5.9.4 Limitations of the results of Fault Detection Effectiveness, Sensitivity of Test Suites and the Relationship Between MC/DC Coverage and Mutation Score

The threats related to the internal validity include the conformance of the model to system requirements and the existence of equivalent mutants. The modelling of system requirements in MBT is a manual pro- cess, and it requires a complete understanding of the system requirements and environment. One can misinterpret these requirements, impacting the conformance of the model with actual system requirements and thus impacting the test suite generated. Hence, to mitigate this factor, we have developed the model in an iterative manner and by getting continuous feed- back from the testers at BT.

Similarly, in mutation testing, the existence of equivalent mutants is also a possible risk to the evaluation of test suites. We tried to eliminate this threat by spending a fair amount of time examining the alive mutants and test results manually.

The factors that can affect the reliability and external validity of this study include the particularities of the MBT model and test suites specific to the BT's environment, test generation tools, modelling notations, size of the subsystem, human experience, and generation algorithms. We created the model and test suites by using the requirements related to a subsystem of the TCMS developed at BT and it contains some particularities related to BT's specific testing environment, libraries, and development tools that may not be relevant to other domains. However, we have provided sufficient information on the experimentation methodology and argue that if a researcher with similar testing and modelling experience will replicate this study, similar results can be obtained. Furthermore, different test generation tools, modelling notations, and generator algorithms may also affect the results. The research into the effectiveness of test suites should therefore require more industrial case studies for creating generalizable knowledge.

The measures used for MC/DC and mutation score in this study were inspired by the literature and evaluation metrics used in industrial safety standards, i.e., ISO 26262, IEC 61508, EN 50128, and EN 50657. Moreover, the selection of mutants is done by thoroughly investigating the operators used for the development of FBD programs from the literature, their industrial applicability, and dependencies on the tools used by the BT for developing safety-critical programs.

5.9.5 Limitations of the Results Regarding the Practical Implications of Adopting Passive Testing using T-EARS and the Accompanying Toolchain

We observed a risk that false positives quickly affect the results without proper tuning of the passive test cases. Further, our results suggest that mapping of the logical (abstract) signals to technical (concrete) signals is a major challenge. Thus, we further improve the translation process's test case tuning steps and suggest some lessons learned when applying passive testing in a real industrial context.

The translation and tuning of requirements are done manually, which can be a time-consuming activity, so future improvements include automating it, e.g., using automated latency analysis. In addition, more empirical work on the cost-effectiveness of passive testing in the embedded software industry is needed as well as further support for industrial uptake and adoption.

6 CONCLUSION

This deliverable contains the work done in Adeptness with regards to BT use case. The focus is on model-based testing, continuous monitoring through T-EARS and the corresponding empirical results.

The first study has a focus on the modelling aspect of MBT as well as explores the behavioural differences between manually written and MBT generated test cases using an open- source MBT tool (i.e., GW). Based on the case study done in close collaboration with BT developing the safety critical TCMS, our results show that a testing team can create a complete and representative model of the SUT using both requirements and test specifications. This study also shows that GW-generated test cases provide higher frequency of requirements coverage than manually written test cases. GW can generate a complete test suite with random path genera- tor fulfilling the edge coverage criterion. Lastly, the attractive features of GW and our experience of adapting it for a real- world, industrial scenario can help further research on the capabilities of MBT and GW in other domains.

Later, we have proposed a MBT framework, TIGER, focused on the concretization of abstract test cases and generation of test scripts for CPSs where embedded software plays an important part. There are three main parts of TIGER: abstract test case generator, test case concretizer and finally, test case generator. We have evaluated TIGER in terms of fault detection by inducing faults in the model representing the SUT and then generating and executing the test scripts. The results show that test scripts generated by TIGER are executable, contains concrete test data and can be used to uncover interaction faults at SiL simulation level. The test scripts generated through the correct model did not result in any failed execution step, confirming the correct generation and execution, ensuring conformance to the requirements specifications and the implemented SUT.

We further performed comparative analysis between MBT, CT and manual testing techniques in terms of MC/DC and requirement coverage. The efficiency of testing techniques was also evaluated, along with determination of the differences and overlaps of test suites generated by each testing technique. The experimentation results based on an industrial case study showed that regardless of test objectives used to develop test cases for each technique (i.e., boundary value analysis (BVA) and equivalence partitioning (EP) for manual testing, edge coverage in MBT and t-ways interactions of parameters in CT), each test suite achieved a substantial level of MC/DC coverage. However, the test suite generated using MBT provided a higher MC/DC coverage. MBT-generated test suite contained ap- proximately 71% of similar test cases, on average, generated by other testing techniques as well as highest number of unique test cases, which also had an observable effect on MC/DC adequacy. Furthermore, the analysis also showed that MBT-generated test suite is highly relevant to manual specification-based testing in terms of complete test case generation due to model's conformance with requirements. Hence, we argue that MBT-generated test suite is most likely to uncover system-level faults and it could be used to improve manual testing. On the other hand, CT was the most efficient technique when compared to MBT and manual testing but exponential growth of test cases while achieving higher MC/DC could affect its efficiency. Also, MBT-generated test suite contains redundant test cases, and their exclusion can reduce the execution time, consequently efficiency could be improved.

As our last investigation within MBT, an experimental evaluation of industrial manual testing and two popular system-level automated test generation techniques, MBT and CT, in terms of fault detection effectiveness is carried out. In addition, we measure the sensitivity of each test suite towards each type of fault induced by different mutant operators. Moreover, we examine the relationship between the MC/DC coverage and mutation scores at the system level. The experimental results show that the test suites achieved mutation scores within the range of 82% (CT 2-way) and 90% (CT 3-ways and CT 4- ways), whereas other techniques' mutation scores lying within this range. Thus, CT with higher interaction strength (3-ways and 4-ways) was found to be the most effective testing technique in terms of achieved mutation score, closely followed by MBT with a mutation score of 88%. This means that we found higher-interaction strength CT and MBT as most effective in detecting induced faults based on the selected mutant operators, the exception being the TRO, where none of the techniques were able to find faults based on this operator. MBT was found to be the least effective in detecting faults induced by the LRO- I operator, while manual testing achieved a low mutation score in detecting LRO mutants. CT 2-ways was found to be the least effective testing technique in our case. On the other hand, manual testing was also found to be efficient in terms of execution time. The results also showed that the MBT-generated test suite achieved the highest MC/DC coverage when compared with other techniques. Lastly, the analysis of mutation and MC/DC coverage scores showed a general positive relationship between both measures for all test suites. Hence, we put forward the hypothesis that the test suite achieving adequate MC/DC coverage tends to also provide a higher mutation score.

With respect to continuous monitoring using passive testing, we conclude that passive testing is adequate for testing the studied system-level requirements in a real industrial setting. In addition, we show how passive testing can be used to understand requirement coverage and finding faults.

7 FUTURE WORK

As future work, we see the need to perform a more thorough and rigorous evaluation of different coverage criteria provided by GW, generate test scripts using GW, evaluate it with manually created test scripts in terms of efficiency and fault-detection effectiveness, explore the online testing capabilities of GW and running it as a Restful service. In future, we also intend to investigate approaches to reduce test suite generated by MBT. Moreover, a thorough evaluation and statistical analysis are also warranted to analyse the mutation score and MC/DC coverage at the structural level. Lastly, passive testing needs to be launched on a greater scale and needs continuous validation to solidify its strengths.

8 REFERENCES

[1] L. Rouvroye and E. van den Bliek. Comparing safety analysis techniques. Reliability Engineering & System Safety, 75(3):289–294, 2002.

[2] Kasurinen Taipale, K. Karhu, and K. Smolander. Trade-off between automated and manual software testing. International Journal of System Assurance Engineering and Management, 2(2):114–125, 2011.

[3] M. Sanchez-Gordon, L. Rijal and R. Colomo-Palacios. Beyond technical skills in software testing: Automated versus manual testing. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pages 161–164, 2020.

[4] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, “Model-based mutation testing—approach and case studies,” Science of Computer Programming, vol. 120, pp. 25–48, 2016.

[5] Rudolf Ramler, Theodorich Kopetzky, and Wolfgang Platz. Combinatorial Test Design in the TOSCA Testsuite: Lessons Learned and Practical Implications. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.569–572, 2012.

[6] Erik Van Veenendaal, Dorothy Graham, and Rex Black. “Foundations of Software Testing: ISTQB Certification. Cengage Learning EMEA (2008), 30, 2008.

[7] Kelly J Hayhurst. A practical tutorial on modified condition/decision coverage. DIANE Publishing, 2001.

[8] Hareton KN Leung and Lee J White. A cost model to compare regression test strategies. In ICSM, Vol. 91. 201–208, 1991.

[9] Anne Kramer and Bruno Legeard. Model-based testing essentials-guide to the ISTQB certified model-based tester: foundation level. John Wiley & Sons, 2016.

[10] Stefan Mohacsi, Michael Felderer, and Armin Beer. 2015. Estimating the Cost and Benefit of Model-Based Testing: A Decision Support Procedure for the Application of Model-Based Testing in Industry. In 2015 41st Euromicro Conference on Software EngineeringandAdvancedApplications.382–389, 2015.

[11] Y. Oh, J. Yoo, S. Cha and H. S. Son. Soft- ware safety analysis of function block diagrams using fault trees. Reliability Engineering & System Safety, 88(3):215–228, 2005.

[12] E. Enoiu, D. Sundmark, A. Causevic, R. Feldt and P. Pettersson. Mutation-based test generation for plc embedded software using model checking. In IFIP International Conference on Testing Software and Systems, pages 155–171. Springer, 2016.

[13] D. Shin, E. Jee, and D, -H. Bae. Empirical evaluation on fbd model-based test coverage criteria us- ing mutation analysis. In International Conference on Model Driven Engineering Languages and Systems, pages 465–479. Springer, 2012.

[14] M. N. Zafar, W. Afzal, E. P. Enoiu, A. Stratis, and O. Sellin. A model-based test script generation framework for embedded software. In The 17th Work- shop on Advances in Model Based Testing, 2021.

[15] S. K. Khalsa, and Y. Labiche. An orchestrated survey of available algorithms and tools for combinatorial testing. In 2014 IEEE 25th International Symposium on Software Reliability Engineering, pages 323– 334. IEEE, 2014.

[16] M. Wagner, K. Kleine, D. E. Simos, R. Kuhn, and R. Kacker. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In 2020 IEEE International Conference on Software Testing, Verification and Valida- tion Workshops (ICSTW), pages 191–200. IEEE, 2020.

[17] R. Ramler, T. Kopetzky, and W. Platz. Combinatorial test design in the tosca test suite: lessons learned and practical implications. In 2012 IEEE Fifth Inter- national Conference on Software Testing, Verification and Validation, pages 569–572. IEEE, 2012.

[18] M. Tiegelkamp and K.-H. John, IEC 61131-3: Programming industrial automation systems. Springer Publishing Company, Incorporated, 2010.

[19] J.-L. Boulanger, CENELEC 50128 and IEC 62279 standards. John Wiley & Sons, 2015.

[20] F. Daniel, E. Eduard, A. Wasif, S. Daniel, G. Thomas, and K. Avenir, "From natural language requirements to passive test cases using guarded assertions," in The Intl. Conf. on SW Qual., Rel., and Sec. IEEE, 2018.

[21] M. Staron, Action Research as Research Methodology in Software Engineering. Cham: Springer Intl. Publishing, pp. 15–36, 2020.

[22] D. Flemström, T. Gustafsson, and A. Kobetski, "A case study of interactive development of passive tests," in the 5th International Workshop on Requirements Engineering and Testing. ACM, 2018.

[23] Anne Kramer and Bruno Legeard. Model-based testing essentials- guide to the ISTQB certified model-based tester: foundation level. John Wiley & Sons, 2016.

[24] Paul. Ammann and Jeff. Offutt. 2016. Introduction to Software Testing. Cambridge University Press.

[25] S. Ali, H. Lu, S. Wang, T. Yue, and M. Zhang. Chapter Two - Uncertainty-Wise Testing of Cyber-Physical Systems. Advances in Computers, Vol. 107. Elsevier, 23 – 94, 2017.

9 ACKNOWLEDGEMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 871319.

Disclaimer

This document reflects the views of the author(s) and does not necessarily reflect the views or policy of the European Commission. Whilst efforts have been made to ensure the accuracy and completeness of this document, the ADEPTNESS consortium shall not be liable for any errors or omissions, however caused.