Model-Based Testing in Practice: An Industrial Case Study using GraphWalker

Muhammad Nouman Zafar
muhammad.nouman.zafar@mdh.se
Mälardalen University
Sweden

Wasif Afzal
wasif.afzal@mdh.se
Mälardalen University
Sweden

Eduard Enoiu
eduard.enoiu@mdh.se
Mälardalen University
Sweden

Athanasios Stratis
athanasios.stratis@rail.bombardier.com
Bombardier Transportation AB
Sweden

Aitor Arrieta
aarrieta@mondragon.edu
Mondragon University
Spain

Goiuria Sagardui
gsagardui@mondragon.edu
Mondragon University
Spain

Abstract

Model-based testing (MBT) is a test design technique that supports the automation of software testing processes and generates test artefacts based on a system model representing behavioural aspects of the system under test (SUT). Previous research has shown some positive aspects of MBT such as low-cost test case generation and fault detection effectiveness. However, it is still a challenge for both practitioners and researchers to evaluate MBT tools and techniques in real, industrial settings. Consequently, the empirical evidence regarding the mainstream use, including the modelling and test case generation using MBT tools, is limited. In this paper, we report the results of a case study on applying GraphWalker, an open-source tool for MBT, on an industrial cyber-physical system (i.e., a Train Control Management System developed by Bombardier Transportation in Sweden), from modelling of real-world requirements and test specifications to test case generation. We evaluate the models of the SUT for completeness and representativeness, compare MBT with manual test cases written by practitioners using multiple attributes as well as share our experiences of selecting and using GraphWalker for industrial application. The results show that a model of the SUT created using both requirements and test specifications provides better understanding of the SUT from testers' perspective, making it more complete and representative than the model created based only on the requirements specification alone. The generated model-based test cases are longer in terms of the number of test steps, achieve better edge coverage and can cover requirements more frequently in different orders while achieving the same level of requirements coverage as manually created test cases.

CCS Concepts: **• Cyber Physical System of Systems → Embedded systems**; **• Modeling**; **• Requirement Specification**; **• Test Specification**;

Keywords: GraphWalker, Case Study, Model-Based Testing

ACM Reference Format:
Muhammad Nouman Zafar, Wasif Afzal, Eduard Enoiu, Athanasios Stratis, Aitor Arrieta, and Goiuria Sagardui. 2021. Model-Based Testing in Practice: An Industrial Case Study using GraphWalker. In *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference) (ISEC 2021), February 25–27, 2021, Bhubaneswar, Odisha, India.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3452383.3452388

1 Introduction

The technological transformation towards complex systems such as Cyber Physical Systems (CPSs), which are characterized by networks of embedded systems and strong human-computer interactions, amplify the need of rigorous verification and validation activities. Since software is a growing and an essential part of such complex systems [25], software testing increases the confidence in the correctness of the overall system properties. In the safety-critical domain, verification and validation through software testing assumes even more importance as a failure can result in substantial damage and even loss of life. Thus, several techniques have been introduced to increase the efficiency and effectiveness of software testing and to automate its processes. Model-based testing (MBT) [27] is one such technique that automates the software testing processes, including the generation of test artefacts. MBT has shown to provide increased effectiveness in terms of fault detection rate in software systems [22]. Also, as most of the processes involved in MBT are either semi or fully automated, it promises to provide a drastic fall in the cost for generating and maintaining test artefacts.

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 [26]. 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. Inline with this need, we present the results of an industrial case study conducted in collaboration with Bombardier Transportation (BT) in Sweden. We have modelled the fire indication system of the train control management system (TCMS) in GraphWalker (GW)[1] using two sources (requirement specifications and test specifications). GW is an open source MBT tool for generating test artefacts meeting certain coverage criteria. We have evaluated the generated models for completeness and representativeness. Furthermore, the test cases generated by GW have been compared with manual test cases from practitioners. Lastly, we documented our experience of selecting and on-going use of GW for both industrial application and research purposes. The main objectives of this study are as follows:

- Model the industrial system under test (SUT) based on two different input sources (requirements and test specifications) and analyse the differences in terms of model completeness and representativeness.
- Compare the tool-generated test cases with manual test cases written by practitioners.
- Document the experiences of selecting and using an MBT tool in the industrial safety-critical domain.

Our results indicate that the input sources/artefacts influence the completeness and representativeness of the model used for MBT. In our case, the requirements specification, together with the test specification, resulted in an improved representation of the structure and behavior of the system under test (SUT). The user-specified edge coverage criterion was optimized better by GW-generated test cases when compared with practitioners' test cases; also covering each requirement multiple times in different orders. Our experiences in using GW indicate that it supports an easy to adopt modeling notation and has the ability of both offline and online test generation. In addition, GW is an open-source tool actively developed and maintained by its community that is applicable to test generation in practice.

The rest of the paper is organized as follows. Section 2 presents a description of a representative set of related studies. Section 3 describes the case study design, including the research questions, case selection and data collection procedures. Section 4 describes the results of the case study, Section 5 presents the threats to the validity of the study, Section 6 presents a discussion on the results while the conclusions and future work appear in Section 7.

[1]https://github.com/GraphWalker/graphwalker-project/wiki

2 Related Work

Numerous researchers have provided a comparison between manual testing, semi-automated testing and MBT (e.g. [11], [15], [18], [19], [20], [22]) to evaluate the efficiency and effectiveness of MBT with respect to cost and time. We have also found two studies ([9, 24]) that evaluated the effectiveness of GW in terms of fault detection and cost. In this section, we briefly summarize these related studies, while refer the reader to several review papers on MBT for a more complete reading on the subject [10, 16, 17, 23, 26].

A notable comparison to test a web-based application through MBT has been presented in [22]. Results showed that MBT is a systematic approach and can detect more functional faults than manual testing. Nevertheless, the time required by MBT activities was higher than the time required by manual test design. Manual test design was also found to be more effective in detecting GUI-related issues. Similarly, a new MBT technique and a comparison between manual and automatic test generation has been presented in [15]. The authors proposed a test automation language framework for the creation of concrete test values by mapping it with abstract tests generated using MBT. They also presented an empirical comparison between automated and manual test generation in terms of mapping between abstract and concrete tests. The empirical evaluation showed that automatic test generation provides efficient tests with a better mapping.

Pretschner et al. [20] conducted an empirical study in the networking domain to compare the automatic and manual MBT techniques in terms of model coverage, detected faults, and code coverage. The focus of this study was to evaluate the quality of generated tests by both techniques and to study the correlation between model and implementation coverage. The evaluation highlighted a moderate positive correlation between the implementation Condition/Decision coverage and model coverage. Moreover, the study also concluded that MBT is more effective in detecting the logical and requirement faults [20]. Marques et al. [18] compared the manual and adhoc testing techniques with MBT. The study evaluated the performance of techniques in terms of efficiency, effectiveness, precision and relative recall for bug detection. The results showed that the techniques have almost equivalent time, effectiveness, precision and relative recall for bug detection. The results provided mathematical explanation of the comparison between both techniques and indicated that both techniques have almost equivalent impact on bug detection. However, manual tests provided better detection rate of logical faults whereas model-based tests provided better evaluation of system documentation and could cover all possible scenarios with more precision for complex use cases.

Mouchawrab et al. [19] analyzed state-based round-trip path coverage testing technique and structure-based edge coverage testing technique to assess the fault detection rate, factors that affect the effectiveness of these techniques and cost from tester's perspective. The empirical study showed that the state-based testing and structural testing could be used together to achieve better fault detection rate. Similarly, Enoiu et al. [11] conducted a detailed study to measure the effectiveness and efficiency of specification-based manually generated tests and implementation-based manually and automatically generated tests in terms of fault detection. The results suggest that specification-based tests are more effective for detecting faults than implementation-based automated tests, whereas implementation-based tests have higher efficiency rate for fault detection than the specification-based tests created manually.

Rashid et al. [9] developed a prototype tool (CANoe$^+$) using CANoe and GW to increase the functional coverage of generated test cases and evaluated its effectiveness in terms of fault detection by comparing it with CANoe. The results showed that CANoe$^+$ provided better function coverage by reporting all failed functionalities whereas CANoe did not report a single injected fault. Similarly, in [24], an exploratory study used GW to investigate the effectiveness of keyword and behaviour driven test automation framework in Agile Development (AD) in terms of cost. The results suggested that MBT improved the maintenance, functional coverage and flexibility for AD as well as provided effective continuous integration by reducing the cost.

Overall, there is a need to validate these MBT approaches against relevant industrial systems such that more knowledge is built on how to efficiently create models using different sources of information and how the resulting model-based test cases compare with manually created test cases in industry.

3 Case Study Design

To investigate the impact of input artifacts on modeling and to compare MBT with manual test design, as well as to reflect on our experiences of selecting and working with GW, we conducted an industrial case study, starting with the modeling of the fire indication sub-system of an on-going TCMS project at BT. Throughout our investigation in this case study, 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. With slight modifications, we report our case study results influenced by the guidelines of Runeson and Höst [21] and the paper by Schulze et al. [22].

3.1 Research Questions

Our aim in this case study is to answer the following research questions:

- RQ1: What are the differences in considering two types of artifacts, requirements specifications and test specifications, as an input to the modeling process of MBT?
- RQ2: How do the model-based test cases compare with test cases manually written by practitioners?
- RQ3: What are the initial experiences in selecting and using an open-source MBT tool for both research and industrial application?

3.2 Case Selection

In discussions with BT, we selected an on-going TCMS development project for the MOVIA[2] vehicle product family as a 'case' in this study. 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' in our case study translates to the SUT for the selected software development project, which is the TCMS. The following sub-section provides further details on the SUT.

3.2.1 SUT.

The case has been selected from an industrial system developed by BT, a world leader in the manufacturing of trains and railway systems. 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 also between the train and the ground. TCMS is designed to perform all tasks related to modern vehicle control.

As shown in Figure 1, 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 analog 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

[2]https://www.railway-technology.com/projects/bombardier-movia-metro-cars/

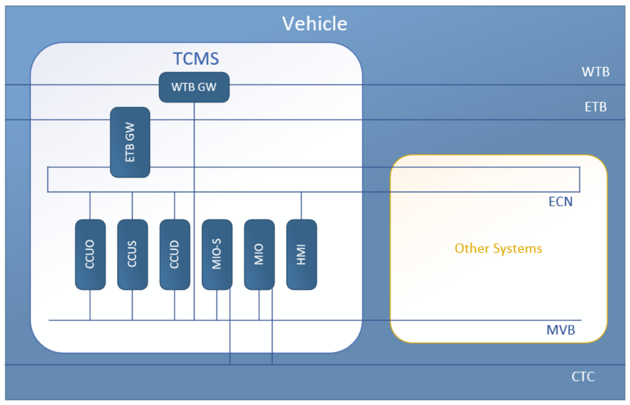


Figure 1. Illustration of TCMS architecture.

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.

3.2.2 Industrial scenario for modeling and test generation. In this case study, the selected requirements of the SUT were related to the multi-purpose TCMS devices to detect fire in the cab (the driver's compartment). The fire detection system in TCMS is used to detect two types of fire: external fire and internal fire. It uses two instances of a device known as the Fire Detection Control Unit (FDCU-1 and FDCU-2) to signal the TCMS about the status of both types of fire. Both devices can have two states: slave and master in order to detect fire. The TCMS indicates the external or internal fire based on the signals sent by the devices and the current state of these devices.

On the real train, the FDCU devices communicate with fire and smoke sensors. When a sensor detects smoke/fire, FDCUs 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 lit a LED as indication of fire/smoke on the driver's desk, via electrical wiring.

3.3 Data Collection Procedures

The data collection for the case study and inputs required for modelling was performed using two different data collection techniques [14]: 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.

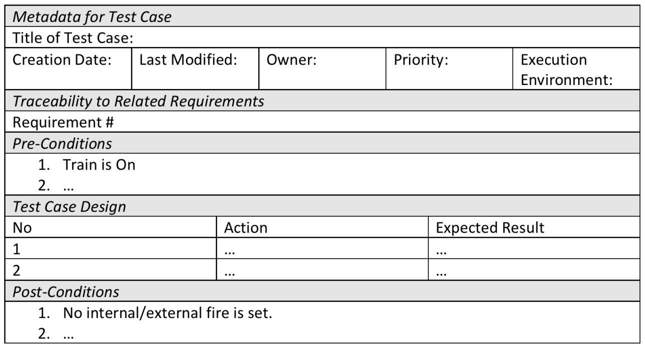
The engineering processes of software development at BT are performed according to safety standards and regulations (e.g., EN 50128 standard is used for designing test cases). Each test case should contribute to the demonstration that a specified requirement has indeed been covered and satisfied. Executing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

3.3.1 Requirements Specification. The requirements for the specific industrial scenario (Section 3.2.2) 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, similar to the 'Given-When-Then' template as common in Behavior Driven Development[3]. 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}

3.3.2 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 2. 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.

[3]https://dannorth.net/introducing-bdd/



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 2. An illustration of a manually written test specification.

4 Results

This section describes our findings in terms of modelling aspects, behavioural differences between test cases generated through GW and manual test cases, and our initial experiences for selecting and using GW for industrial use.

4.1 Model-based Testing Using GraphWalker

In this study, 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.

4.1.1 Modelling the SUT using Requirements Specification. The first step of modelling the SUT involved understanding 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 3).

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

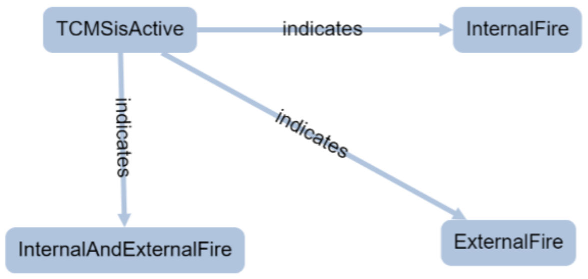


Figure 3. Initial FSM-based model created using the requirements specification

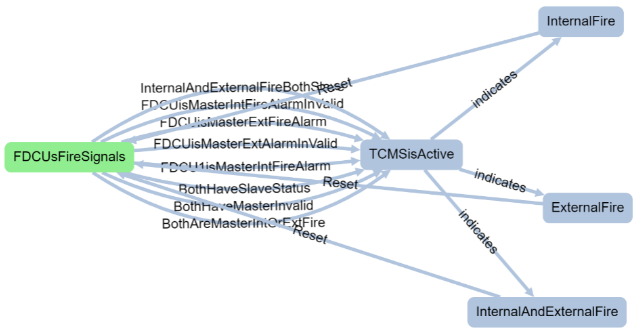


Figure 4. Final FSM-based model created using requirements specifications

sent by the FDCU are added as actions on edges covering the requirements of the SUT to generate test cases. Figure 4 represents the final model created using the requirements specification.

4.1.2 Modelling the SUT using Requirements 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 requirements 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 5 and Figure 6 depict the initial and final model created using both specifications, respectively. The final model (Figure 6) consists of three diagrams;

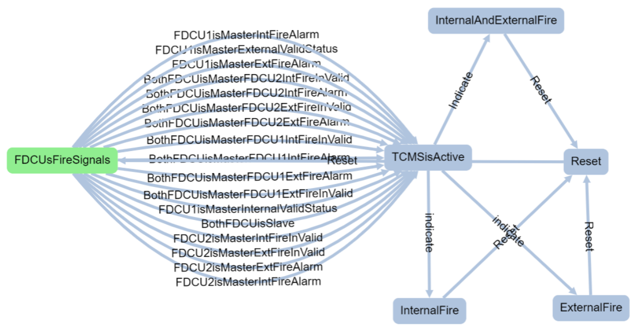

FDCU1isMasterIntFireAlarm
FDCU1isMasterExternalValidStatus
FDCU1isMasterExtFireAlarm
BothFDCUisMasterFDCU2IntFireInValid
BothFDCUisMasterFDCU2IntFireAlarm
BothFDCUisMasterFDCU2ExtFireInValid
BothFDCUisMasterFDCU2ExtFireAlarm
BothFDCUisMasterFDCU1IntFireInValid
BothFDCUisMasterFDCU1IntFireAlarm
Reset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

Figure 5. Initial FSM-based model created from requirements and test specifications.

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 behavior of the SUT more completely than the first version of the model which was created using the requirement specification alone.

4.1.3 Modelling Aspects.

MBT is known to provide a better understanding of the domain and the SUT to generate testware [13]. However, in order to create a *complete* and *representative* model of the SUT, all relevant details about the test object or SUT need to be made available.

The Degree of Test Model Completeness and Model Evolution. As shown in Figure 4, 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 as depicted in Figure 6. 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, in order to create a complete behavioral 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 behavior. This, in turn, is expected to help generate test cases that realistically cover more testing scenarios.

Model Representativeness. Moreover, in our case, the test specification also provided information about the SUT in a more clearer and 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 analyzing 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 more *complete* and *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.

We observed that input artifacts influence the usefulness of the resulting model for MBT. Input artifacts for the modelling of the SUT should include the documents from which the requirements can be inferred but also the test specification from manual test planning activities.

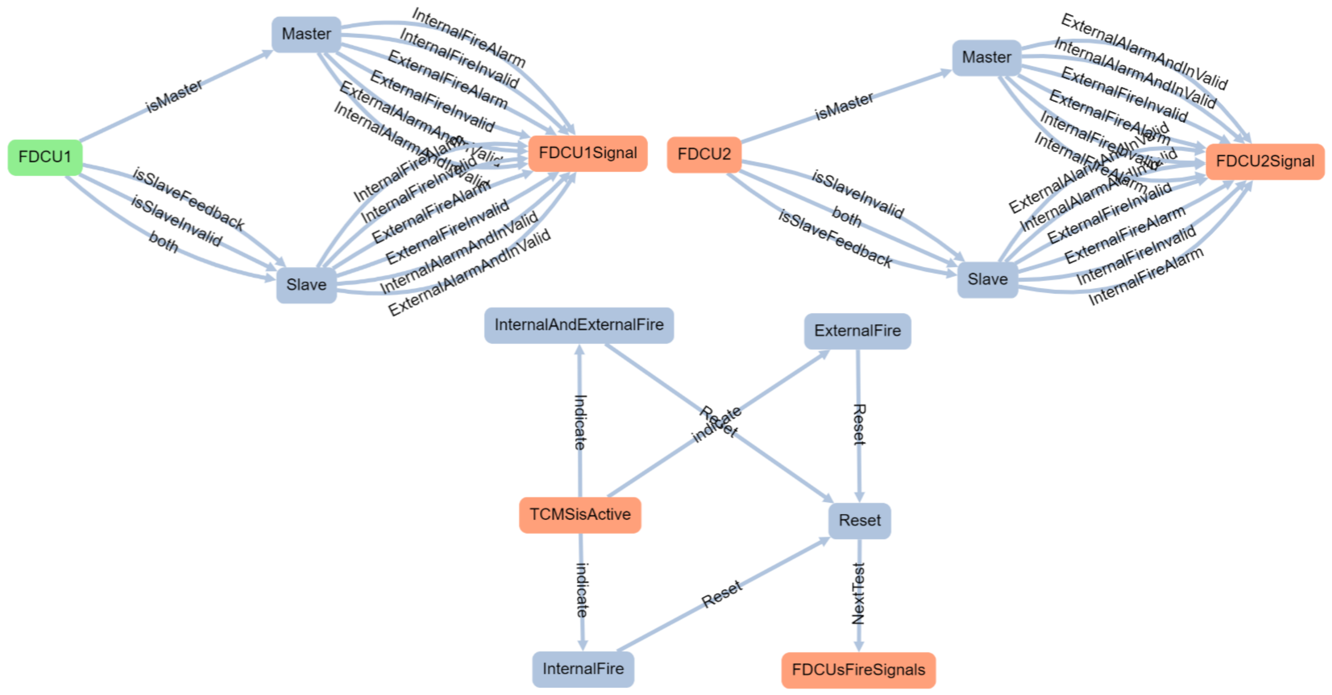


Figure 6. Final FSM-based shared model with multiple diagrams created from requirements and test specifications having shared nodes

4.2 Behavioural Differences Between MBT and Manually Created Test Cases

GW walks the directed graph of the SUT in a fashion determined 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 and we discuss this limitation in Section 6. 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; this limitation is discussed in Section 6.

Edge coverage is a stronger coverage criterion than node coverage alone as it includes traversing of both elements of a model (nodes and edges) [13]. 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 [8, 13], but we leave the measurement of those as a future work (Section 7). "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 1. 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 1; 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 fulfill requirements; an example of it is shown in Table 2. The manually written test cases, on the other hand, require one test step per requirement as shown in Table 3. 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 2, 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.

Model-based test cases are able to optimize user-specified edge coverage criterion better than manually-written test cases; they also tend to cover each requirement multiple times more in best case scenarios, in different order, while achieving the same level of requirements coverage as manually created test cases.

4.3 Initial Experience with GW for Industrial Case Modeling 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.

4.3.1 Selection of GW as an 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 [7] 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 of 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. In order 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 4. 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 modeling the SUT as a state machine that is understandable to both researchers and practitioners.

4.3.2 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.

The GW tool is an industrial applicable open-source and actively developed/maintained MBT tool, having the ability of offline and online test generation, along with the support of the well-known state machines graphical notation. The Eclipse plugin of GW is relatively more user-friendly for an industrial user, but supports a limited set of stopping criteria.

5 Validity Threats

The threats to the validity of this study are discussed in this section and are divided in the following three classes.

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

Table 1. 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 2. 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

continuous input on the correctness of both the SUT models from the testing team at BT.

External Validity & Reliability. We argue that if another 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 provide 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 take into account the possible variations in the number of test steps generated each time and reported the highest number of generated steps. We give the motivation of selecting the highest number of generated test steps in Section 4.2, however, this choice can be improved in the future based on a systematic experimental evaluation of the number of test generation trials.

Construct Validity. We looked into existing measures from the literature. For example, the modeling aspects used for comparison were inspired from the taxonomy of MBT provided by Kramer et al. [13]. We also used well-known behavioral 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.

6 Discussion

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 available 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

Table 3. An equivalent, manually created test step corresponding to the example in Table 2 (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

Table 4. 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

in our industrial scenario, thus *Weight_random* path generator 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 stopping condition. However, while experimenting, we found that the time provided as a parameter to *time_duration* stopping 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 affects of such parameter changes is left as a future work.

Broader Impact. Here we discuss any potential negative impact of our research, inline with the spirit of the blog post written by Hecht et al. [12]. 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, nuclear 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 provide 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.

7 Conclusion and Future Work

This 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 generator 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.

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.

Acknowledgments

The work in this study has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 871319 and from the Swedish Innovation Agency (Vinnova) through the XIVT project.

References

[1] [n.d.]. GraphWalker. https://graphwalker.github.io/. last checked: 08-Aug-2020.

[2] [n.d.]. MoMuT. https://momut.org/. last checked: 08-Aug-2020.

[3] [n.d.]. NModel. https://archive.codeplex.com/?p=nmodel. last checked: 08-Aug-2020.

[4] [n.d.]. SpecExplorer. https://www.microsoft.com/en-us/research/project/model-based-testing-with-specexplorer/. last checked: 08-Aug-2020.

[5] [n.d.]. UPPAAL. http://www.uppaal.org/. last checked: 08-Aug-2020.

[6] [n.d.]. Uppaal-Tron. https://people.cs.aau.dk/ marius/tron/. last checked: 08-Aug-2020.

[7] Shaukat Ali, Hong Lu, Shuai Wang, Tao Yue, and Man Zhang. 2017. Chapter Two - Uncertainty-Wise Testing of Cyber-Physical Systems. Advances in Computers, Vol. 107. Elsevier, 23 – 94.

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

[9] Rashid Darwish, Lynnie Nakyanzi Gwosuta, and Richard Torkar. 2017. A controlled experiment on coverage maximization of automated model-based software test cases in the automotive industry. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 546–547.

[10] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. 2007. A Survey on Model-Based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. Association for Computing Machinery, New York, NY, USA.

[11] Eduard P Enoiu, Adnan Causevic, Daniel Sundmark, and Paul Pettersson. 2016. A controlled experiment in testing of safety-critical embedded software. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–11.

[12] Brent Hecht, Lauren Wilcox, Jeffrey P. Bigham, Johannes Schöning, Ehsan Hoque, Jason Ernst, Yonatan Bisk, Luigi De Russis, Lana Yarosh, Bushra Anjum, Danish Contractor, and Cathy Wu. [n.d.]. It's Time to Do Something: Mitigating the Negative Impacts of Computing Through a Change to the Peer Review Process. ACM Future of Computing Blog, https://acm-fca.org/2018/03/29/negativeimpacts/. last checked: 11-Oct-2020.

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

[14] Timothy C Lethbridge, Susan Elliott Sim, and Janice Singer. 2005. Studying software engineers: Data collection techniques for software field studies. *Empirical software engineering* 10, 3 (2005), 311–341.

[15] Nan Li and Jeff Offutt. 2015. A test automation language framework for behavioral models. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–10.

[16] Wenbin Li, Franck Le Gall, and Naum Spaseski. 2018. A Survey on Model-Based Testing Tools for Test Case Generation. In *Tools and Methods of Program Analysis*, Vladimir Itsykson, Andre Scedrov, and Victor Zakharov (Eds.). Springer International Publishing, Cham, 77–89.

[17] Raluca Marinescu, Cristina Seceleanu, Hèléne Le Guen, and Paul Pettersson. 2015. Chapter Three - A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs. Advances in Computers, Vol. 98. Elsevier, 89 – 140.

[18] Arthur Marques, Franklin Ramalho, and Wilkerson L Andrade. 2014. Comparing model-based testing with traditional testing strategies: An empirical study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 264–273.

[19] Samar Mouchawrab, Lionel C Briand, Yvan Labiche, and Massimiliano Di Penta. 2010. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Transactions on Software Engineering* 37, 2 (2010), 161–187.

[20] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. 2005. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*. 392–401.

[21] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131.

[22] Christoph Schulze, Dharmalingam Ganesan, Mikael Lindvall, Rance Cleaveland, and Daniel Goldman. 2014. Assessing Model-Based Testing: An Empirical Study Conducted in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA.

[23] Muhammad Shafique and Yvan Labiche. 2015. A Systematic Review of State-Based Test Tools. *International Journal on Software Tools for Technology Transfer* 17, 1 (2015), 59–76.

[24] Sandeep Sivanandan et al. 2014. Agile development cycle: Approach to design an effective Model Based Testing with Behaviour driven automation framework. In *20th Annual International Conference on Advanced Computing and Communications (ADCOM)*. IEEE, 22–25.

[25] Martin Törngren and Ulf Sellgren. 2018. *Complexity Challenges in Development of Cyber-Physical Systems*. Springer International Publishing, Cham, 478–503.

[26] Mark Utting, Bruno Legeard, Fabrice Bouquet, Elizabeta Fourneret, Fabien Peureux, and Alexandre Vernotte. 2016. Chapter Two - Recent Advances in Model-Based Testing. Advances in Computers, Vol. 101. Elsevier, 53 – 120.

[27] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.