Automation of the Creation and Execution of System Level Hardware-in-Loop Tests through Model-Based Testing

Viktor Aronsson Karlsson
vkn17002@student.mdu.se
Mälardalen University
Västerås, Sweden

Ahmed Almasri
aai17011@student.mdu.se
Mälardalen University
Västerås, Sweden

Eduard Paul Enoiu
eduard.paul.enoiu@mdh.se
Mälardalen University
Västerås, Sweden

Wasif Afzal
wasif.afzal@mdu.se
Mälardalen University
Västerås, Sweden

Peter Charbachi
peter.charbachi@volvo.com
Volvo Construction Equipment AB
Eskilstuna, Sweden

ABSTRACT

In this paper, we apply model-based testing (MBT) to automate the creation of hardware-in-loop (HIL) test cases. In order to select MBT tools, different tools' properties were compared to each other through a literature study, with the result of selecting GraphWalker and MoMuT tools to be used in an industrial case study. The results show that the generated test cases perform similarly to their manual counterparts regarding how the test cases achieved full requirements coverage. When comparing the effort needed for applying the methods, a comparable effort is required for creating the first iteration, while with every subsequent update, MBT will require less effort compared to the manual process. Both methods achieve 100% requirements coverage, and since manual tests are created and executed by humans, some requirements are favoured over others due to company demands, while MBT tests are generated randomly. In addition, a comparison between the used tools showcased the differences in the models' design and their test case generation. The comparison showed that GraphWalker has a more straightforward design method and is better suited for smaller systems, while MoMuT can handle more complex systems but has a more involved design method.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**.

KEYWORDS

MoMuT, GraphWalker, Hardware-in-Loop, Model-Based Testing

A-TEST '22, November 17–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9452-9/22/11.
https://doi.org/10.1145/3548659.3561313

ACM Reference Format:
Viktor Aronsson Karlsson, Ahmed Almasri, Eduard Paul Enoiu, Wasif Afzal, and Peter Charbachi. 2022. Automation of the Creation and Execution of System Level Hardware-in-Loop Tests through Model-Based Testing. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '22), November 17–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3548659.3561313

1 INTRODUCTION

Vehicular systems have developed rapidly, and their software and hardware parts have increased to cover most of the vehicle, either by controlling the vehicle autonomously or interacting with the driver's commands to ensure safe driving in different environments [24]. Future software development for vehicular systems requires more complex requirements to be satisfied, which consecutively requires an increase in test cases. The industry has found that the validation process for functional requirements has been costly and time-consuming throughout the years, which the industry wants to alleviate [23][20].

The traditional way to test systems is by doing it manually. However, since the demand for testing and the cost of manual testing has increased, the desire for a better test creation technique to satisfy the demand has emerged. Software and requirements updates are needed to maintain the high efficiency of the systems, which increases the pressure of developing new test cases [1][27]. Manually written test cases are often time-consuming and costly to perform for the company. By automating the creation, the quantity of the tests can increase, which will allow more test cases to be executed with less time invested.

From a tester's perspective, maintenance of test cases is a hassle and a time-consuming activity [2]. When a requirement gets updated, so do all test cases related to that requirement. With model-based testing (MBT), only the model has to be modified while the test cases will automatically correct themselves; the impact of this change will give the testers more time to focus on more complex tasks [23].

This research, performed at Volvo Construction Equipment AB, is based on reducing human interaction and increasing the quality of the validation process for their products. The proposed solution for achieving this goal is the automation of the creation and execution of test cases with the help of MBT. Verification of the created test cases is performed on the actual hardware constructed in a controlled environment that can simulate realistic conditions; this

concept is formally known as Hardware-in-Loop (HIL) [4] and is commonly used to supplement field testing to reduce its cost [15]. The challenge of creating HIL test cases is that all functions before the function that needs to be tested have to be configured during the test. This is in contrast to the common application of MBT for software applications, where the software's features can be easily tested in isolation from each other. In contrast, HIL testing contains most of the system's functions, resulting in the testing process being applied and adapted for the whole system instead of individual features.

In this paper, we propose and evaluate a method for designing models to automate the creation and execution of HIL test cases to verify functional requirements.

2 BACKGROUND

This section outlines information concerning MBT, MBT simulation and testing, and finite state machines (FSMs) for creating models.

2.1 Model-Based Testing

MBT automates the generation of validation tests for the system functions. MBT functions by creating models depicting the system's behaviour and its requirements through FSM or textual representations [22]. The main advantage of MBT is the ability to dynamically modify the models and quickly generate new test cases for the updated system [23][12]. The main disadvantages of MBT are its high learning curve and the knowledge needed to maintain an older model [3]. The MBT process is divided into the following five main steps [29]:

(1) **Model the system under test (SUT) and/or its environment.** An abstract model of the desired system to be tested is created, and it is called abstract because it is simpler than the SUT itself and forgoes some of its details.
(2) **Generate abstract tests from the model.** When generating test cases in MBT, several coverage criteria are possible, e.g., choosing a path generator's condition to decide how to traverse a model, such as covering all transitions. The output of is a sequence of operations from the model.
(3) **Concretize the abstract tests to make them executable.** This step is where the tester/designer will transform operations into an executable test suite. The process can be done by using a transformation tool to create appropriate test cases to fit the test execution tool, alternatively by writing some adaptor code to filter out the unnecessary data.
(4) **Execute the tests on the SUT.** There are two ways to execute test cases on the SUT, online and offline. With online testing, the tests are executed on the SUT as they are generated and the execution results will be recorded. In offline testing, the test cases are generated and stored in a file, e.g., as a sequence of commands, stored data or lines of code. These files can be executable on different machines or in entirely different environments.
(5) **Analyse the test results.** The last step is to analyse the result of the test execution, and if a test reports a failure, the team must investigate the reason for the fault.

2.2 Hardware-in-Loop Testing

HIL testing allows the developers to validate their creations for the actual hardware in a lab setting where some parts of the actual hardware are simulated while other parts are same as in the final system. The main benefit of HIL testing is the cost reduction on the development since the development and verification of the system can be made in parallel, and the testing of functionalities can be performed individually instead of having to wait for a specific time slot, e.g. with field testing a location has to be booked and can often only be performed at specific time slots [4]. The HIL system contains the hardware components of the final system to ensure the correctness in the test environment and accuracy in test results [11]. The HIL system will simulate the vehicle's environment in order to make the Electronic Control Unit (ECU) believe it is connected to the actual vehicle. The HIL system simulates the analogue, digital and bus signals to the ECU, aiming to emulate real-world operation. The perfect HIL system would be where the ECU could not differentiate between the actual environment and the HIL system [5].

2.3 Finite State Machines

FSM allows the developers to visualise the flow of control in any given system by utilising states and transitions. FSM collects all relevant, unique states for a given system and binds them together with conditional transitions [8]. FSM design consists of two primary forms, Moore and Mealy [30]. The difference between the two is how the state changes and the output generation (output here refers to variable assignments). Moore changes state only depending on the state variable, while Mealy can change by either state or input values. The outputs for Moore generates at the state change, and it occurs for Mealy during the transition between states. When applying FSM on a complex system with many states and transitions, the Harel state chart should be used instead of Moore and Mealy. Harel utilises composite states and sub-diagrams to reduce clutter and simplify the diagram.

3 RELATED WORK

In the context of system testing, MBT is nothing new. Using model-based tests compared to manually written tests has numerous advantages and can describe any system behaviour, shorten the development cycle and deliver a high-quality product with fewer errors [9].

Enoiu et al. [16] analysed the difference between automatically generated test cases with manually written tests. The tests were created for a safety-critical system written in the programming language IEC61131-3. The comparison of the two techniques was through the cost and quality measurements aspects. The study results show that the automatically generated and manual test cases achieved similar results in terms of code coverage. Manual creation of tests had tremendous success in detecting logical, timer and negation faults than automated.

The study [21] by Keränen and Räty presents an analysis of novel validation methodologies that apply MBT in a HIL platform to validate the test system itself. The novel methodologies presented in this paper are component, integration, acceptance and performance testing. Mutation testing was applied during the analysis process to

measure the effectiveness of the test cases and the coverage of non-functional requirements. A UML-based MBT tool 'Qtronic', was used to create the models and generate the test cases to perform this study. The generation and execution of test cases were performed online and offline, where the tool "TTworkbench" was utilised to execute the offline format.

Hussain's master thesis [18] aimed to investigate how MBT could improve industrial practices when it comes to validating the functionality of a system. To answer "How the MBT approach could improve the current testing processes?" In a case study at Volvo CE, Hussain used the tool 'Conformiq Creator' [6] to analyse the accelerator pedal position and the brake pedal position. Conformiq Creator is a functional black-box testing tool that utilises activity diagrams and structure diagrams to create the models; structure diagrams specify the structure of all interfaces in the system, and activity diagrams describe the flow of control between the states in the diagram. The created models were based on functional and non-functional requirements, which the company provided. Five criteria categories to guide the generation of test cases were used: Default, Requirements, Exhaustive, All paths and Boundary Values. The result of the thesis showcased that there is no significant difference in coverage between manual and automated testing.

In [19], the researchers investigated the possibility of generating models from natural language requirements using GraphWalker, an open-source tool, to create models based on FSM. Zafar et al. [31] point out the benefits of using an open-source MBT tool like GraphWalker. The researchers describe how it differs from manual test cases. The researchers investigated two methods of generating test cases using MBT, either by modelling SUT using only requirements specification or modelling SUT using requirements and test specification. A comparison between the generated and manually written test cases was then performed. The study highlights a promising result when utilising a MBT tool for creating test cases using both requirements and test specifications.

4 RESEARCH METHOD

Our aim here is to investigate the possibility of using MBT to automate the creation of HIL test cases. The MBT models are based on the behaviour and requirements of an accelerator pedal of a Wheel Loader System. The model shall then generate executable test cases applied on the HIL platform to validate the system's functional requirements. The research started with a literature study on the subject, followed by a case study based on the outcomes of the literature study. The case study begins after the literature study and the appropriate MBT tools have been selected. The case study investigates the MBT approach for generating test cases for a HIL system. The MBT models will generate offline test cases. The generated test cases are then transferred into SE-Tools, an internal tool used at VCE, where the tests are sent to the HIL system for execution.

4.1 Case Study Design

The case study follows the design described in the guidelines [26].

4.1.1 The Case and Context. The case under investigation is the Accelerator Pedal Detection System, that is part of a Wheel Loader and the associated HIL system, which simulates the Wheel Loader.

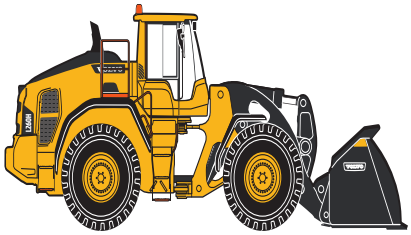

Figure 1: The profile view of a typical Wheel Loader

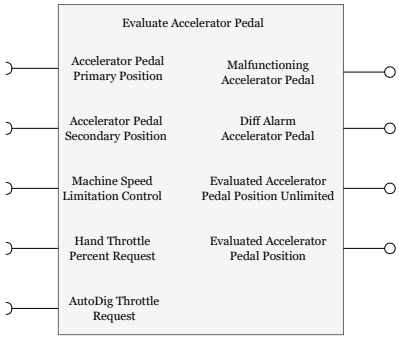

Figure 2: Description of the SUT.

These are made available from Volvo CE (VCE) (see Figure 1 for the profile view of a Wheel Loader). The Accelerator Pedal Detection System is chosen for its simplicity and early execution in the system process.

System Function - Accelerator Pedal. Accelerator Pedal Detection testing is the act of validating the functionality of the accelerator pedal and its interconnected alarms. The accelerator pedal consists of a primary and secondary sensor giving the system redundancy, meaning that it can still be operated with one damaged sensor. However, this would mean that the system can no longer be safely operated. The sensor signals are utilised in the internal logic of the vehicle system, where it decides the finalised position of the accelerator pedal and the status of the interconnected alarms. The system is also affected by three internal signals, "Machine Speed Limitation Control", "Hand Throttle Percent Request", and "AutoDig Throttle Request", which control the finalised pedal position that is sent out of the system (Figure 2). The outputs from the system are the finalised pedal position and the status of the alarms.

The scenario Table 1) can be defined from the requirements. These are the eight different scenarios when validating the accelerator pedal position detection system. The primary and secondary sensors can fundamentally change between two states' regular operation and erroneous operation. The two sensors have a value range between 0 and 100, and their maximum allowed value difference is 10% of the total value range, e.g., if sensor A has a value of 50 and sensor B have a value of 55, there will be no alarm, but if sensor B changes the value to 65, the alarm will trigger. Only when the first scenario is being tested will the malfunction alarm not be active; every other scenario will trigger the alarm. The difference alarm will trigger whenever there is a sensor difference of more

than 10%, but only if both sensors are in the normal state or both are in an erroneous state.

SE-Tools. The tool used for interfacing within and between teams at VCE is SE-Tools, a modified version of the tool SystemWeaver [7]. SE-Tools contain most of the system documentation, from high-level requirements down to design requirements, where all functions are interconnected to understand the system from start to finish entirely. It also contains the verification results and has version handling for extensive documentation. The validation test cases are written into SE-Tools, and through SE-Tools, other users can access the tests and execute them on their test system.

Manual System Level Testing at VCE.. VCE adopts the traditional manual testing process, which includes system requirements review, designing test cases based on requirements, reviewing the test cases and executing them in HIL, which simulates the behaviour of the desired vehicle. Requirements and tests are loaded in the SE-tool platform, where Volvo's experienced verification engineers have read-write access to the data. The manual testing process starts by reading the test script and applying it step by step in the HIL. The system will then respond to the commands and show them in a LabView interface. The tester can follow and see changes in the system according to the sent commands and how the vehicle should react.

Automated System Level Testing at VCE.. The test case's Automated Test Generation (ATG) script is generated in an XML file and can be executed directly from SE-Tools to the HIL. The execution process is automated and occurs quicker than the previous one. The tester might not catch all responses from the machine, but each step of the test script will be printed as either PASS or FAILED. However, if a step fails a test, the system will indicate the fault and give feedback on what went wrong.

4.1.2 *Research Questions.*

RQ1: *Which MBT tool can be of use to capture the behaviour of the functional requirements for a vehicular system model?*

RQ2: *How do manually created test cases compare with model-based test cases when applied on HIL testing, in terms of coverage of functional requirements?*

RQ3: *In which manner do the MBT tools differ from each other in terms of functional requirements coverage?*

5 COMPARISON OF MBT TOOLS

The criteria for tool selection were as follows:

(1) **Formatting of the output**: The desired format for VCE is XML. Thus, the output needs to be in the XML format or a format which can easily be converted into the XML format.
(2) **Tool generation ability**: The study focused on offline testing, and the chosen tool shall have the capability to operate in offline testing mode.
(3) **Usability/Documentation**: The application domains supported by the tool and and how good are the instructions.
(4) **Community support**: When using a tool in industrial application, it is essential to have prompt support. An important aspect is how active the user base is and how often the tool gets updated.
(5) **Company preference**: The main tool aspects important for VCE are its capability for requirements coverage, handling time aspects of the system and the ease of use.

It was decided to aim at open-source and academic tools while ignoring commercial tools to avoid unnecessary licenses. The investigated tools are GraphWalker, Tcases, CAgen, Pymodel, fMBT, JSXM, Modbat, QuickCheck, OSMO and MoMu::UML.

Conclusion of Literature Study. Despite the positive aspects of each tool, the comparison observed some limitations that influence the tool selection. The first aspect examined of each tool was their life-support. Most of the tools passed the first criterion since either the developer or community kept the tool updated. The secondary area was the support in the form of documentation, forum or wiki, and this is where most tools failed to achieve the criteria since if instructions were found, they were often lacking or only covered the bare minimum of the needed information. The third area examined was the output and the modelling method. The critical aspect of the output is its existence, and that it is generated in a usable format, this means that the tools need to be operated in offline test generation mode, and the output information is in a usable format. There are plenty of variances when modelling the SUT behaviour, but not every modelling technique will work to fulfil the criterion. The modelling technique should be able to generate random inputs and deterministic outputs, and the technique should also be able to handle time aspects, such as waiting for the RPM of a motor to reach the desired value before continuing the test.

The tools that fulfilled the most criteria and were selected to be used in the case study were GraphWalker (GW) and MoMuT:UML. The main appeals of the two tools can be seen in Table 2. The standout aspect of GW was its popularity in the MBT community [10] [28] [25] and its user-friendly appearance. MoMuT, on the other hand, utilises UML models created in Eclipse Papyrus [13], which is a popular program with a lot of online resources available [14]. MoMuT was also the only tool that advertised its ability to handle time aspects, which is an aspect VCE desire.

6 RESULTS

The two main sections in the results correspond to the two chosen MBT tools, GW and MoMuT. The case study ended with comparing the results with the manual testing process.

6.1 GraphWalker - Creation of Model

The model generation in GW went through a number of iterations, resulting in a total of four model versions, the last being the final. In the first version, the designed model aimed to cover one requirement at a time and once, and the issue with this design is that it does not cover all possible instances of input signals, not all possible generating paths either. This version was designed to go through the operation's vertices once in order according to a preset condition. The second version improved the first one when generating random paths between signals/operations, and a new method was used, based on global variables, which helped a lot to change variables values between models. The new variables worked as fine as expected, but all global values did not get printed out when generating test cases using CLI. This issue was discussed in the

Table 1: The eight combinations of the sensor signal states, the alarm states and the value difference of the sensor signals.

	Primary Sensor State	Secondary Sensor State	Sensors Differ More the 10%	Malfunction Alarm State	Difference Alarm State
Scenario 1	Normal	Normal	No	Normal	Normal
Scenario 2	Normal	Normal	Yes	Alarm	Alarm
Scenario 3	Normal	Erroneous	No	Alarm	Normal
Scenario 4	Normal	Erroneous	Yes	Alarm	Normal
Scenario 5	Erroneous	Normal	No	Alarm	Normal
Scenario 6	Erroneous	Normal	Yes	Alarm	Normal
Scenario 7	Erroneous	Erroneous	No	Alarm	Alarm
Scenario 8	Erroneous	Erroneous	Yes	Alarm	Alarm

Table 2: The essential tool aspects.

GraphWalker	MoMuT:UML
Graphical	
Easy to design and maintain	
Active Community	Support by mail
-	Time Aspect
Offline Testing	
Web-Based Studio	Used with Eclipse
Open-Source	
Related work found	
Recommended by the community	Favoured by Volvo Supervisor
Different path generating algorithms	

GW Forum [17], but we did not find any suggested solution. An upgraded model was designed that included all actions and expected signals at the same model and a vertex (Expected) to recognise where each signal set starts or ends when filtering the generated test cases. This model skips the usage of global variables, and they are not needed anymore since all signals are included in just one model. However, this version of the model ran into mismatch of signal definitions. HIL is signal case sensitive, which means that all the declared signals must exactly match as they are declared in the system's signals definition. The design of this model version followed the accelerator pedal functional requirements written in the SE-Tools, and to use signals in ATG test cases, they must be defined, which is not valid for all signals. Finally, after some removed signals in the new version, a complete model version was ready as shown in Figure 3.

Generation of Test Suites. The result of forgoing global variables is a more apparent generation of data where both the actions and expectations are expressed in the data section of the GW test case. For the finalised test case, the data only needs to be extracted at the end of the iteration since all values are relocated to local variables and thus collected in one place.

6.2 MoMuT - Creation of Model

As with the model creation in GW, the model in MoMuT was also finalized after two iterations. The final MoMuT model (Figure 4) consists of two main control regions, two sub-control regions and two observation regions. The control regions are duplicated and contain the actions for the two sensors, assigning new values and changing the erroneous state. The main state controls the sub-regions which are used for the internal logic. The observation regions contain the logic for the alarms and represent the state

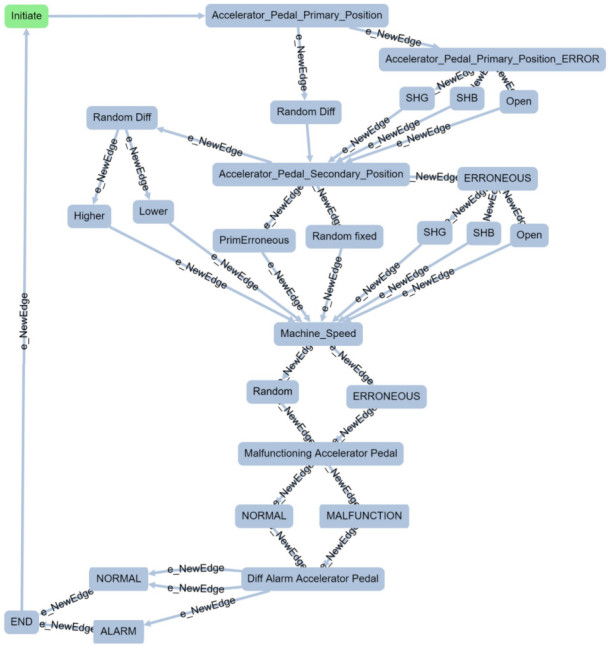

Figure 3: The final version of the GraphWalker model.

of the alarms. Signal activates the transitions, and both contain effects, one to generate a new value and the other for changing the state. The sensor state region changes state depending on the signal trigger from the previous region; this state change is used in the internal logic in the regions to the right. The two rightmost regions are both trigger-less state systems, meaning no signal triggers and only guards are preventing the state transition; each state has an entry action which triggers an observation.

Generation of Test Suites. MoMuT takes the developed model and transforms it into multiple-choice trees, where the branches represent the model's paths. MoMuT traverses the tree by selecting values from a given range that will best traverse the tree. When a branch has been traversed and does not lead into unexplored paths, MoMuT will execute the command "inconc" which will undo its latest action. Due to multiple regions being used for the design of the models, multiple trees were also generated, resulting in test cases that repeatedly test the guard commands and, in turn, validate the requirements more than once.

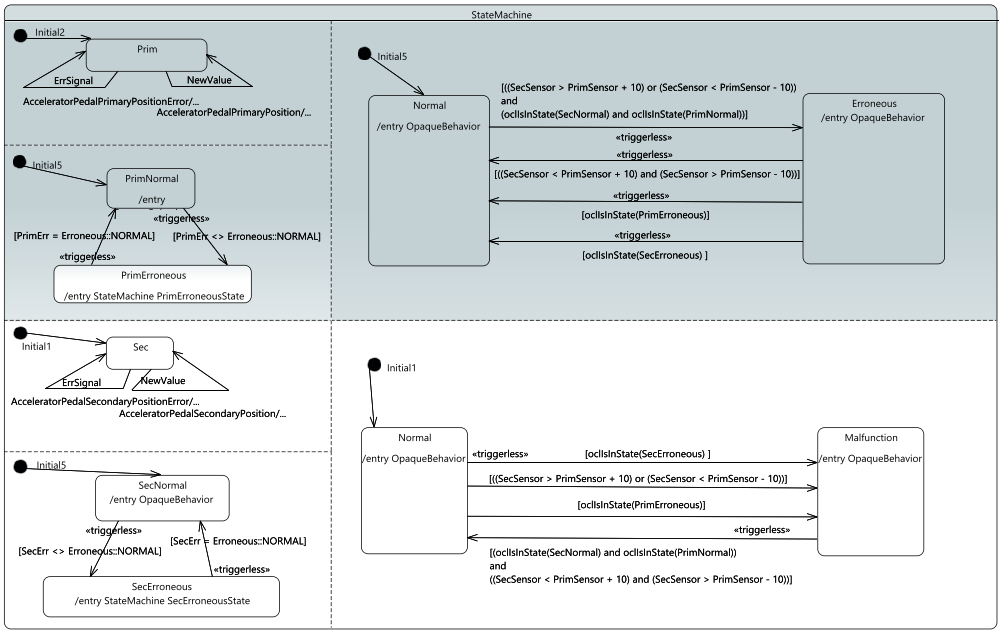

Figure 4: MoMuT State Machine, Containing the Abstraction of the SUT

6.3 Serialisation Script

To execute the generated test cases on the HIL system, the test cases first need to be serialised into the correct format. The GW script starts with reading each line of the raw JSON file, searching for the desired line where all the data for the new test cases is stored. Once the line is found, the data segment is extracted and divided into actions and expected. When the data have been extracted, the test will be formatted and written into the XML file. The MoMuT script function in a similar manner, where it reads each line of the raw test case. However, instead of searching for the correct line, it will identify the line as either action, expected or inconc. For action and expected, it will extract the important information and format it into the XML format, whiles with inconc it will remove the previous line and continue with the next line.

6.4 Comparison Between GraphWalker, MoMuT and Manual Testing

The manually written test cases are gathered from SE-Tools, automatic testing (ATG), and written by a VCE verification engineer. The manual test cases are a collection of five test cases, one for every requirement, and the test cases are then analysed for how many times the requirements are covered and how often each scenario is covered. The MBT test cases from GW and MoMuT are analysed similarly to the manual test cases. In Table 3, the proportion of requirements being tested in the three test suits can be seen, where two values are given: the first value is the number of times the requirement is tested, and the second value is the percentage of the test suites covering that specific requirement. In Figure 5, it can be seen that all three methods test all requirements, but the manual favour validating the first requirement (Normal Operation) while

Table 3: The requirements tested by techniques.

	Requirement 1	Requirement 2	Requirement 3	Requirement 4	Requirement 5
Manual (75)	49 (65.3%)	8 (10.6%)	8 (10.6%)	5 (6.6%)	5 (6.6%)
GraphWalker (54)	7 (12.9%)	14 (25.9)	5 (9.3%)	3 (5.5%)	25 (46.3%)
MoMuT (57)	8 (14.03%)	19 (33.3%)	5 (8.7%)	10 (17.5%)	15 (26.3%)

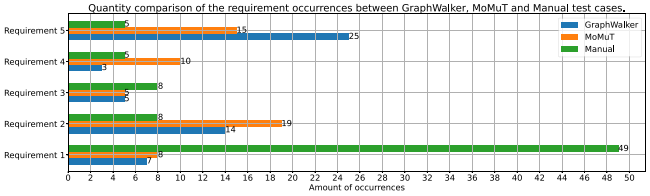

Figure 5: Comparison of requirements tested between manual testing, GW and MoMuT.

the MBT tests favour the fifth requirement (Double Erroneous Detectors). However, when the test cases are analysed for the scenarios in Figure 6, it can be seen that the MBT tools cover more scenarios than the manual. Due to the randomisation in MBT, more scenarios will be tested even though the tested requirement will not change. The only scenario where this phenomenon happened in the manual testing was when the requirement "Normal Operation" was tested. When creating tests for a system function with an extensive range of values to choose from, it is crucial to validate the requirements multiple times with different values from the range. In Figures 7 and 8, the diversity of values tested for each input sensor can be viewed. The MBT test cases utilise comparable values to the manual test case, showing that the generated test values are similar to the manual.

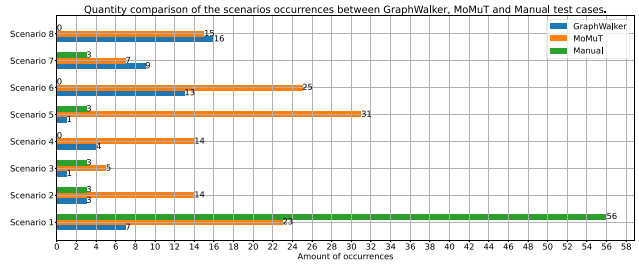

Figure 6: Comparison of scenarios being tested between manual testing, GW and MoMuT.

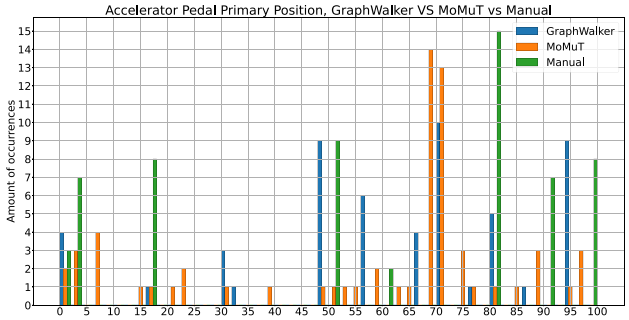

Figure 7: Comparison of the Primary sensor value distribution between manual testing, GW and MoMuT.

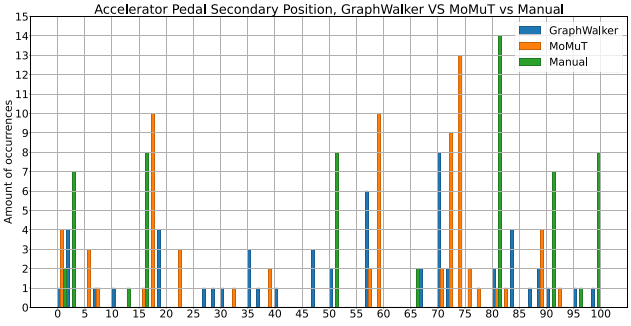

Figure 8: Comparison of the Secondary sensor value distribution between manual testing, GW and MoMuT.

The comparison of Manual, GW and MoMuT results shows that all methods achieve 100% requirement coverage, but the main difference is that the MBT tools generate a higher quantity of test suites. Moreover, manually created tests have a bias for validating "Normal Operation" because VCE demands higher priority to test it, while the MBT follows the probability of the path traversal. Furthermore, in comparison to manual testing, MBT does not validate threshold values because the logic in the path generation is missing this criteria.

Table 4: Time distribution where: N/A: Not performed, †: Manual time estimation, ††: Time for understanding VCE tool, *: When creating a test case with (Edge_Coverage(100%) as a stop condition for the path generator) and ** : The test generation mode = TCG and no. of steps = 300.

	GW				MoMuT		Manual
Versions	**V1**	**V2**	**V3**	**V3.1**	**V1**	**V2**	-
System Understanding	8 h						
Tool Understanding	16 h				40 h		2 h ††
Modelling	10 h	24 h	8 h	2 h	6 h	2 h	N/A
Test Case Creation	N/A	5.962 sec*	6.63 sec*	5.778 sec*	29.43 sec**	47.16 sec**	10 h ±2 †
Test Execution	N/A	N/A	N/A	134 sec	160 sec	784 sec	1 h†
Issue Analysis	2 h	32 h	2 h	-	2 h	-	N/A
Total Time	36 h	80 h	34 h	26 h	56 h	50 h	21±2 h

6.5 Effort

The efforts are calculated based on the following parameters (Table 4):

(1) System Understanding: Analysis of the system, and functional requirements to understand the system and signals.
(2) Tool Understanding: Time consumed for the needed experience to start using it.
(3) Modelling: Creation of each version of the test models.
(4) Test Case Creation: Time needed to generate a single test case in GW and a suite of test cases in MoMuT.
(5) HIL Test Execution: The time taken for the execution of a test case in HIL.
(6) Issue Handling Process: The estimated time spent analysing the results and finding a solution.

Modelling a system using MBT for the first time requires some effort, especially with no previous experience in modelling. In addition to that, understanding the system to model and the used MBT tool is a time-consuming task. From the first day of using them until analysing their results, the efforts needed are represented in Table 4. By following the table, it can be seen that the time needed to re-design the model decreases with each subsequent version of the model; the phenomenon can be observed for both tools. The exception is version one to version 2 for GW, where the model design changed entirely, and none of the previous designs were used in the new design. The observation of the total time shows that the needed time for both MBT methods is decreasing and starting to align with the manual testing. The test case creation time for the manual is the equivalent of the modelling time for the MBT tools. The factor that hinders the MBT time is the initial learning curve and the modelling process, but the maintenance and re-designing time is lower than manual, leading to a gain in time over a more extended period.

7 SUMMARY OF ANSWERS TO RQS

RQ1: The selection process of MBT tools aimed to focus on open-source and academic tools while ignoring commercial tools to avoid unnecessary licenses. Ten tools were chosen initially, and despite the positive aspects of each tool, the comparison observed some limitations that influence the tool selection. The tools that fulfilled

the most criteria and were selected to be used in the case study were *GraphWalker* and *MoMuT*.

RQ2: The iterations done on both tools gave several test suites that differ from each other in quantity since both tools use different generation criteria. The automation of generating test suites helped cover the desired system and assess the applicability of using MBT as the modelling and generation method. Moreover, manual tests put weight on some requirements since human is the one who performs the testing procedure, and it is natural to test one requirement more than other, e.g. testing manually "Normal Operation"/Scenario 1 in the modelled system had been performed many times compared with the rest of requirements (See Figures 5 and 6). On the contrary, MBT tools follow random probabilities when choosing requirements to test, which results in testing more requirements randomly (See Section 6.4).

RQ3: Both tools, GraphWalker and MoMuT, have the ability to design a model for the SUT. In addition, they were able to generate test suites in a different type of output file format, but a Python script (See Section 6.3) was able to serialise the data into an *XML* file that can be run in the HIL. The requirement coverage criteria differ between GW and MoMuT, which depends on each tool's technique when creating test suites.

8 CONCLUSIONS & FUTURE WORK

The main goal of this paper was to investigate an approach for automating the creation and execution of HIL test cases. Furthermore, we also evaluated the feasibility of implementing an MBT solution to accomplish this goal. In order to fully automate the creation of test cases in an industrial workflow, it requires a significant amount of effort to understand both the desired system and the MBT tool. However, the results show it is worth the effort and time. Automating the generation of test cases does not make the system behave differently during the testing process. Instead, the generated test cases perform similarly to their manual counterparts. The advantage of MBT compared to the manual is the reduction in creation-effort of new test cases, resulting in an increase in the quantity of requirement coverage and an earlier faults detection. The MBT tools applied in this work are GraphWalker and MoMuT. Both tools were available for academic work and free to use, but for using MoMuT in an industrial area, a licence must be obtained to get access to the tool. As a proof-of-concept, a test suite created by either GraphWalker or MoMuT covered the system requirements, was executed in HIL, and helped discover errors and bugs within the requirements and HIL rig.

An interesting future work is to investigate the maintainability and usability of the tools and models. Examining how traceability can be implemented within a model of various sizes would be interesting, and how different modelling designs could affect traceability.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 program under grant agreement Nos. 871319, 957212, 101007350 and the ITEA3 SmartDelta project.

REFERENCES

[1] 2017. Why Software Updates Are So Important | McAfee Blog. *McAfee* (9 2017). https://www.mcafee.com/blogs/internet-security/software-updates-important/
[2] 2018. How Model Based Testing Benefits the End-user. https://www.uk.sogeti.com/content-hub/blog/how-model-based-testing-benefits-the-end-user/.
[3] 2018. Model Based Testing: Testing Type You Must Know! - Software Testing Class. https://www.softwaretestingclass.com/model-based-testing/
[4] 2020. What Is Hardware-in-the-Loop? *National Instruments* (12 2020). https://www.ni.com/sv-se/innovations/white-papers/17/what-is-hardware-in-the-loop-.html#section--380192003
[5] 2021. *What is Hardware-in-the-Loop (HIL) Testing?* http://www.genuen.com/blog/what-is-hardware-in-the-loop-hil-testing
[6] 2022. CONFORMIQ CREATOR. https://www.conformiq.com/products/conformiq-creator/.
[7] 2022. SystemWeaver. https://www.systemweaver.se/
[8] 2022. *What is a state machine?* https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines
[9] Larry Apfelbaum and John Doyle. 1997. Model Based Testing.
[10] Automated-360. 2022. 9 Great Tools to work with Model-based Testing (MBT). https://automated-360.com/model-based-testing/model-based-testing/#graphwalker
[11] M. Bacic. 2005. On hardware-in-the-loop simulation. In *Proceedings of the 44th IEEE Conference on Decision and Control.*
[12] Mark R. Blackburn, Robert Busser, and Aaron Nauman. 2002. Interface-Driven, Model-Based Test Automation. In *Proceedings of the International Conference On Software Testing Analysis Review.*
[13] Eclipse. 2022. Eclipse Papyrus. https://www.eclipse.org/papyrus/
[14] Eclipse. 2022. Eclipse Papyrus™ Documentation. https://www.eclipse.org/papyrus/documentation.html
[15] George Ellis. 2012. Chapter 13 - Model Development and Verification. In *Control System Design Guide* (fourth edition ed.), George Ellis (Ed.). Butterworth-Heinemann, Boston, 261–282. https://doi.org/10.1016/B978-0-12-385920-4.00013-8
[16] Eduard Enoiu, Daniel Sundmark, Adnan Čaušević, and Paul Pettersson. 2017. A Comparative Study of Manual and Automated Testing for Industrial Control Software. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).*
[17] GraphWalker. 2022. GraphWalker forum. https://groups.google.com/g/graphwalker
[18] Aliya Hussain. 2018. *An Evaluation of Model-based Testing in Industrial Practice: From System Modelling to Test Generation.* Master's thesis.
[19] Korhonen Joakim. 2020. *Automated Model Generation Using GraphWalker Based on Given-When-Then Specifications.* Master's thesis.
[20] Massila Kamalrudin and Safiah Sidek. 2015. A Review on Software Requirements Validation and Consistency Management. *International Journal of Software Engineering and its Applications* x, x (2015), 20.
[21] Janne Keränen and Tomi Räty. 2013. Validation of Model-Based Testing in Hardware in the Loop Platform. In *2013 10th International Conference on Information Technology: New Generations.*
[22] Anne Kramer and Bruno Legeard. 2016. Model-based testing essentials: guide to the ISTQB certified model-based tester foundation level. John Wiley & Sons.
[23] Yasir Masood Malik. 2010. *Model Based Testing: An Evaluation.* Master's thesis.
[24] L. Pan, X. Zheng, H.X. Chen, T. Luan, H. Bootwala, and L. Batten. 2017. Cyber security attacks to modern vehicular systems. *Journal of Information Security and Applications* 36 (2017), 90–100.
[25] ProfessionalQA. 2022. Model Based Testing. https://www.professionalqa.com/model-based-testing-tools
[26] Per Runeson and Martin Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Softw. Engg.* 14, 2 (2009), 131–164.
[27] Steve Symanovich. 2021. 5 reasons why general software updates and patches are important. *NortonLifeLock* (1 2021). https://us.norton.com/internetsecurity-how-to-the-importance-of-general-software-updates-and-patches.html
[28] Ministry Of Testing. 2022. Model-based testing open source Tools? https://club.ministryoftesting.com/t/model-based-testing-open-source-tools/20159/8
[29] Mark. Utting and Legeard. Bruno. 2006. *Practical model-based testing a tools approach.* Morgan Kaufmann Publishers, San Francisco, CA.
[30] Peter Wilson and H. Alan Mantooth. 2013. Chapter 6 - Block Diagram Modeling and System Analysis. In *Model-Based Engineering for Complex Electronic Systems*, Peter Wilson and H. Alan Mantooth (Eds.). Newnes, Oxford, 169–196.
[31] 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.* ACM, New York, NY, USA.