A Model-Based Test Script Generation Framework for Embedded Software

Muhammad Nouman Zafar*, Wasif Afzal*, Eduard Paul Enoiu*, Athanasios Stratis†, Ola Sellin†
*Mälardalen University, Sweden
{muhammad.nouman.zafar, wasif.afzal, eduard.enoiu}@mdh.se
†Bombardier Transportation AB, Sweden
{athanasios.stratis, ola.sellin}@rail.bombardier.com

***Abstract*—The abstract test cases generated through model-based testing (MBT) need to be concretized to make them executable on the software under test (SUT). Multiple researchers proposed different solutions, e.g., by utilizing adapters for concretization of abstract test cases and generation of test scripts. In this paper, we propose our Model-Based Test scrIpt GenEration fRamework (TIGER) based on GraphWalker, an open source MBT tool. The framework is capable of generating test scripts for embedded software controlling functions of a cyber physical system such as passenger trains developed at Bombardier Transportation AB. The framework follows some defined mapping rules for the concretization of abstract test cases. We have evaluated the generated test scripts using an industrial case study in terms of fault detection. We have induced faults in the model of the SUT based on three mutation operators to generate faulty test scripts. The aim of generating faulty test scripts is to produce failed test steps and to guarantee the absence of faults in the SUT. Moreover, we have also generated the test scripts using the correct version of the model and executed it to analyse the behaviour of the generated test scripts in comparison with manually-written test scripts. The results show that the test scripts generated by GW using the proposed framework are executable, provide 100% requirements coverage and can be used to uncover faults at software-in-the-loop simulation level of sub-system testing.**

***Index Terms*—Model-based Testing, Concrete Test Scripts**

I. Introduction

Safety-critical embedded system software is an essential part of modern cyber physical systems (CPSs) and errors in a CPS software can lead to catastrophic events. Needless to say, software testing for safety-critical embedded system software needs to be thorough and effective. Research into Model-Based Testing (MBT) [1] has shown that it can be a resource-efficient technique for generating test cases, test scripts, test verdicts as well as effective in terms of fault detection. MBT generates abstract test cases based on a model of the System Under Test (SUT) [2]. The abstract test cases can be transformed into concrete, executable test scripts that can eventually produce test verdicts. However, we have observed that this concretization step is not covered well enough in the literature or is discussed only in the context of web-based and mobile applications where somewhat mature script generation frameworks exist such as Selenium and Appium. Thus only few papers (i.e. [3] and [4]) report on the details of the concretization step and even fewer have addressed it for embedded software testing at integration and system levels.

In this paper, we have developed a Model-Based Test scrIpt GenEration fRamework (TIGER)[1] based on GraphWalker (GW)[2], a freely-available MBT tool. The generated test scripts validate the functionality of sub-systems controlled by the train control management system (TCMS) software developed by Bombardier Transportation (BT), Sweden. GW is utilized for generating abstract test cases that traverses the nodes and edges of the Extended Finite State Machine (EFSM) model representing the SUT. The developed framework uses the GW to generate abstract test cases, concretizes them by following the defined mapping rules and generates test scripts in C# language. We have evaluated the behaviour of the generated test scripts by injecting faults in the model of a fire detection system controlled by the TCMS. The injected faults are based on the work on mutation operators by Belli et al. [5]. The test scripts are then generated from the correct and fault-induced version of the models, and executed at software-in-the-loop (SiL) execution platform at BT. The aim of introducing faults in the model is to generate faulty test scripts. These should fail when executed on the SUT to guarantee the absence of induced faults in the SUT. We have also analysed the input combinations generated by GW to evaluate the requirements coverage of test scripts.

The main objective of this study is as following:

- Development and preliminary evaluation of a test script generation framework based on MBT principles for SiL testing of embedded software.

The preliminary results show that TIGER is able to generate test scripts that are readily executable on BT's SiL platform and have the potential to uncover interaction faults at sub-system level of TCMS testing. The rest of the paper is organized as follows. Section II presents a description of related studies. Section III provides a detailed description of the proposed framework. Section IV describes the validation methodology, case study, type of faults injected in the model, and results of the case study. Section V presents a discussion on the results and validity threats to the study whereas the conclusions and future work are presented in Section VI.

[1]TIGER is available at https://github.com/MuhammadNoumanZafar/TestScriptGeneration.git

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

II. Related Work

Since the past decade, researchers have been exploring multiple techniques to automate the software testing process and MBT is one of them. Automation of software testing involves the generation of test cases, test scripts and execution of test scripts to generate test verdicts. We have found numerous studies (e.g., [3], [4], [6], [7], [8], [9], [10], [11], [12], [13], [14]) that have provided multiple solutions for automatically generating test artefacts (i.e. test cases and test scripts) using MBT techniques. In the section, we briefly summarize these studies and for a more complete overview of MBT techniques in general, we refer the interested reader to sveral review papers on the topic [15], [16], [17], [18], [19].

A model-based solution has been presented in [6] to address the testing challenges and to automatically generate the test scripts for distributed Multi Mission User Services (MMUS) systems. They have developed the model representing the SUT, linked it with GUI and generated the test cases using a tool named 'TEMPPO Designer'. The generation of executable test cases/scripts has been done with Micro Focus TestPartner. The results showed the improvement in test quality, removal of redundancies, reduction in test effort and increase in test efficiency. Wang et al. [7] proposed a model-based framework for Cloud API testing. It uses the data modelling, individual API modelling and API scenario model using directed diagraph to generate the test scripts containing test cases as well as test data. Another MBT framework has been presented in [8] to generate test cases and test scripts for the embedded systems by extending EAST-ADL with timed automata semantics and also provided a chain of tools for automatically generating and executing the test script for code validation.

Xu et al. [3] proposed a novel automated model-based test script generation technique for functional and security testing. They have used the Petri nets to model the control and data flow for functional dependencies of the system. They also have provided a mapping between the model elements and implementation constructs for test script generation and found that the proposed technique is significantly effective in terms of fault detection. An automated framework to generate test cases and scripts for Android applications based on system sequence diagrams has been presented in [9]. The experimental evaluation showed a significant improvement in bug detection along with better effectiveness in terms of test coverage when compared with other methods. Similarly, a test script generation approach has been presented in [12]. They have proposed a tool that automatically generates the test scripts in C++ by translating the constructs of scenarios written in AValLa language based on some defined rules.

Gupta et al. [10] and Moura et al. [11] developed model-driven approaches for generating test cases and scripts based on Business Process Modelling Notations (BPMN) focused on functional testing for web-based applications. The empirical evaluation not only showed the completeness and correctness of the proposed approaches but also turned out to be efficient in terms of performance. Similarly, a model-based approach for generating test cases and test scripts (according to Robot Framework) from Requirements Specification Language (RSL) has been proposed in [4]. They have also defined a mapping for RSL constructs and GUI elements for the concretization of test data in test scripts. The results showed that the use of RSL improved the quality of requirements and reduced the manual effort and time for the generation of test artefacts.

Vanhecke et al. [13] developed a plugin to concretize abstract test cases by using a transformation and adaption approach. They have also provided the mapping of operations and verification with corresponding assertions and actions defined in abstract test cases to generate executable python scripts in QTaste format. The results show that the plugin increased the generality of QTaste by raising the abstraction level of the SUT interface and the test API. Similarly, a test reuse strategy to minimize the concretization effort has been proposed in [14]. After the concretization of FSM-based generated abstract test cases, they have used a selection algorithm to select the non-redundant concrete test cases, which can be used to test the unchanged behavior of a new product. The results showed that the reusablity of concretized test cases not only provided a reduction in effort but also minimized the cost of testing.

III. Proposed Framework

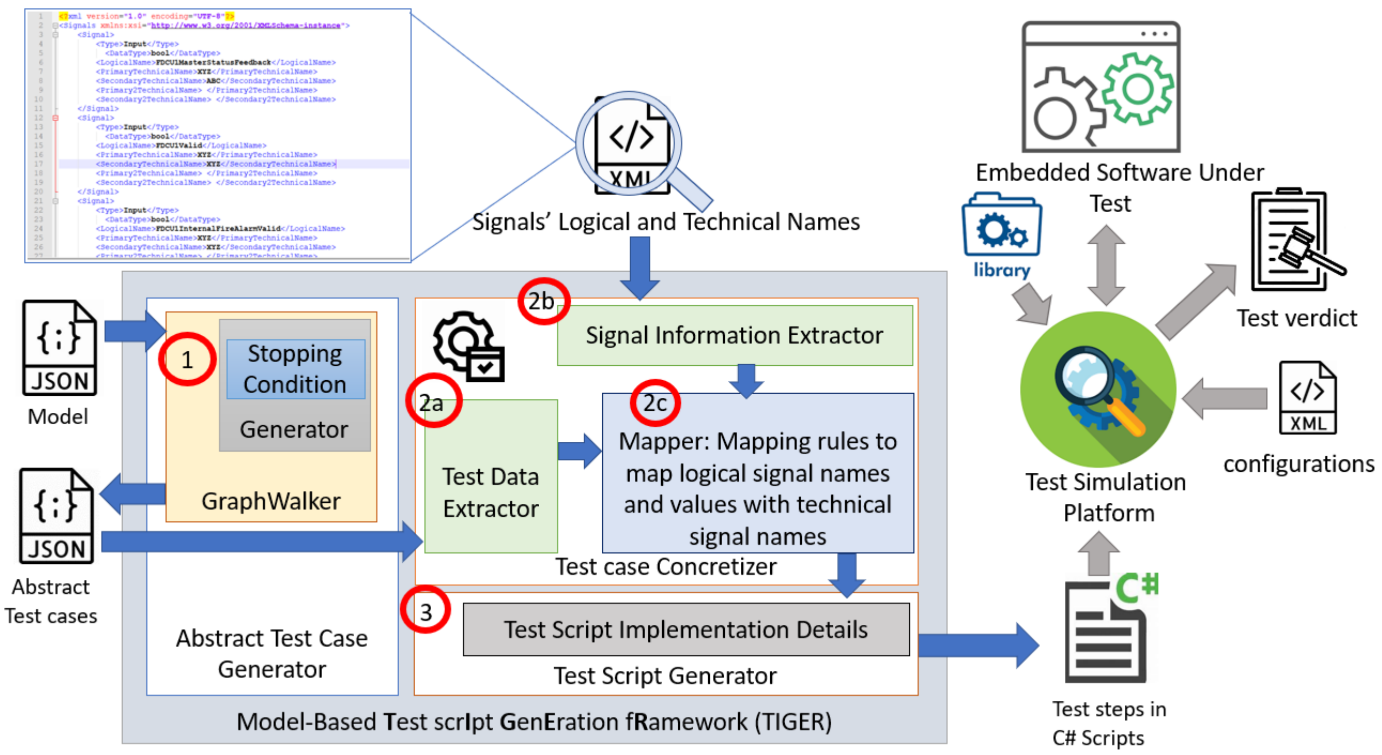
The proposed Model-Based Test scrIpt GenEration fRamework (TIGER) is intended to provide a cost and time-efficient solution to automatically test sub-systems controlled by BT's TCMS. The framework consists of three main parts (Figure 1):

1) Abstract test case generator
2) Test case concretizer
3) Test script generator

In the below text, we describe each of these main parts of TIGER in detail with respective constituent steps.

A. Abstract Test Case Generator

The framework is based on GraphWalker (GW), an open source MBT tool, that is able to generate abstract test cases. These test cases do not explicitly contain the precise information about the input/output signals involved in the operation of the SUT and thus can not be executed directly on the SUT to verify the correctness of results [2]. GW uses a model file in JSON or GraphML format and generates the abstract test cases in the form of multiple test steps by traversing through the model elements (i.e. nodes and edges) using a generator and a stopping condition. It provides multiple options that could be used to generate different information about the model in the test cases. For example, one could generate the test cases without including the test data (variables and their respective values) in it. In this case, the GW-generated test cases only contain the name of the nodes and edges. Similarly, options to include test data, filtering blocked elements could be used to include other pieces of information related to the model in test cases. However, in our case, the `verbose` feature for generating test cases is useful as it generates all the relevant

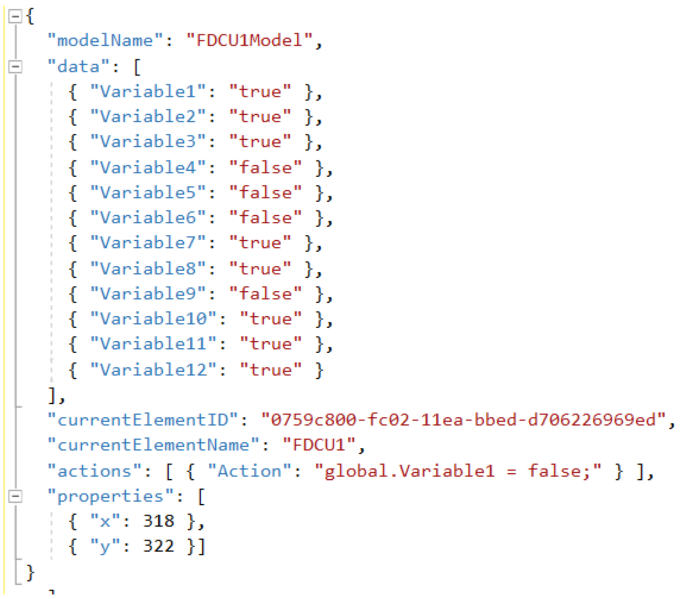

Signals' Logical and Technical Names
XML
Embedded Software Under Test
library
Test verdict
Test Simulation Platform
XML
configurations
Test steps in C# Scripts
C#
JSON
Model
JSON
Abstract Test cases
1
Stopping Condition
Generator
GraphWalker
Abstract Test Case Generator
2b
Signal Information Extractor
2a
Test Data Extractor
2c
Mapper: Mapping rules to map logical signal names and values with technical signal names
Test case Concretizer
3
Test Script Implementation Details
Test Script Generator
Model-Based Test scrIpt GenEration fRamework (TIGER)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Signals xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Signal>
        <Type>Input</Type>
          <DataType>bool</DataType>
        <LogicalName>FDCU1MasterStatusFeedback</LogicalName>
        <PrimaryTechnicalName>XYZ</PrimaryTechnicalName>
        <SecondaryTechnicalName>ABC</SecondaryTechnicalName>
        <Primary2TechnicalName> </Primary2TechnicalName>
        <Secondary2TechnicalName> </Secondary2TechnicalName>
    </Signal>
    <Signal>
        <Type>Input</Type>
          <DataType>bool</DataType>
        <LogicalName>FDCU1Valid</LogicalName>
        <PrimaryTechnicalName>XYZ</PrimaryTechnicalName>
        <SecondaryTechnicalName>XYZ</SecondaryTechnicalName>
        <Primary2TechnicalName> </Primary2TechnicalName>
        <Secondary2TechnicalName> </Secondary2TechnicalName>
    </Signal>
    <Signal>
        <Type>Input</Type>
          <DataType>bool</DataType>
        <LogicalName>FDCU1InternalFireAlarmValid</LogicalName>
        <PrimaryTechnicalName>XYZ</PrimaryTechnicalName>
        <SecondaryTechnicalName>XYZ</SecondaryTechnicalName>
        <Primary2TechnicalName> </Primary2TechnicalName>
```

Fig. 1. The Architecture of the Model-Based Test scrIpt GenEration fRamework (TIGER)

details related to the model in test cases, such as initialized variables, actions, currentElementID, and properties of nodes and edges.

1) Format of Abstract Test Cases: GW generates the abstract test cases in a JSON format. Figure 2 shows a test step in the JSON format generated using the `verbose` option. The model name in JSON object ("modelName") contains the name of the model that GW traversed while generating this test step, "data" holds the names of variables and their initial values (these variables are used to predict the behavior of the model based on guard conditions and actions), "currentElementID" represents the unique identifier for the model element, "currentElementName" is the user-specified name for the model element, "actions" indicate the change in the values of variables when a transition is taken through the element, and "properties" specifies the extra information if included in the model, i.e. a description regarding the node or edge.



```json
{
  "modelName": "FDCU1Model",
  "data": [
    { "Variable1": "true" },
    { "Variable2": "true" },
    { "Variable3": "true" },
    { "Variable4": "false" },
    { "Variable5": "false" },
    { "Variable6": "false" },
    { "Variable7": "true" },
    { "Variable8": "true" },
    { "Variable9": "false" },
    { "Variable10": "true" },
    { "Variable11": "true" },
    { "Variable12": "true" }
  ],
  "currentElementID": "0759c800-fc02-11ea-bbed-d706226969ed",
  "currentElementName": "FDCU1",
  "actions": [ { "Action": "global.Variable1 = false;" } ],
  "properties": [
    { "x": 318 },
    { "y": 322 }]
}
```

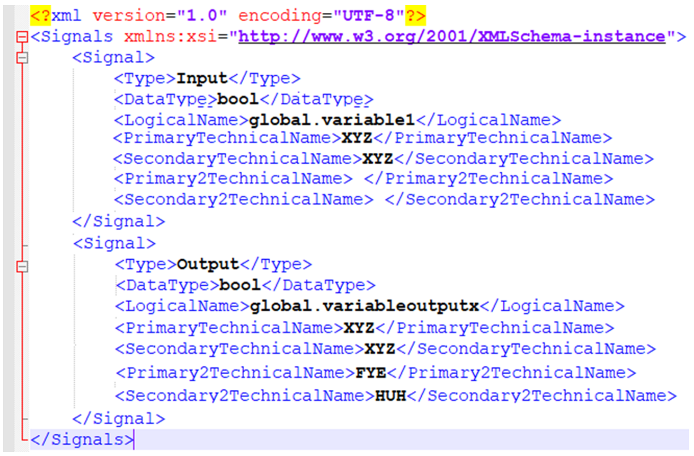
Fig. 2. Example of an JSON formatted test step generated by GW

B. Test Case Concretizer

To execute the test cases on BT's test execution platform, we need to have concrete test cases, which contain the precise actions, data values and signals' names not present in the abstract test cases [2]. Our framework uses transformation and adaption approach [20] for the concretization and execution of generated test scripts. The transformation of abstract test cases to concrete test cases is done while generating the test scripts whereas the adaption layer is added at execution level. The concretization of test cases include the extraction of the test data from the actions (generated in a test step) from the JSON file, and providing the system's signals in a XML format i.e. logical and technical signal names used by the TCMS as shown in Figure 3. Engineers at BT use these logical signal names in requirements and test specification at initial levels of development and testing phases. Logical names are the initial names for signals representation, which are available to the developers and testers. However, these logical signal names must map to one or more technical signal names, which are the actual signal names used by the SUT to complete operations. So, the last step in the concretization process includes rules to map these logical and technical signal names with test data generated in test cases.

1) Test Data Extractor: The test data extractor in the framework is responsible for reading the JSON file and extracting the useful information from it such as model name, currentElement, currentElementID and actions containing the test data for test cases.

2) Signals' Information Extractor: The signals' information extractor in the framework is responsible for reading the XML file and extracting the information about each signal such as the signal type (i.e. Input or Output), technical name(s) (i.e. primary and secondary technical names) and information related to data type of its respective values.



```
<?xml version="1.0" encoding="UTF-8"?>
<Signals xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Signal>
        <Type>Input</Type>
        <DataType>bool</DataType>
        <LogicalName>global.variable1</LogicalName>
        <PrimaryTechnicalName>XYZ</PrimaryTechnicalName>
        <SecondaryTechnicalName>XYZ</SecondaryTechnicalName>
        <Primary2TechnicalName> </Primary2TechnicalName>
        <Secondary2TechnicalName> </Secondary2TechnicalName>
    </Signal>
    <Signal>
        <Type>Output</Type>
        <DataType>bool</DataType>
        <LogicalName>global.variableoutputx</LogicalName>
        <PrimaryTechnicalName>XYZ</PrimaryTechnicalName>
        <SecondaryTechnicalName>XYZ</SecondaryTechnicalName>
        <Primary2TechnicalName>FYE</Primary2TechnicalName>
        <Secondary2TechnicalName>HUH</Secondary2TechnicalName>
    </Signal>
</Signals>
```

Fig. 3. Example of an XML file containing information about logical and technical names

3) Mapping Rules: The concretization of test cases is based on certain rules, which not only preserve the continuity of test cases and traversing order but also prevent information loss (Table I). These rules worked as a guideline for the framework to concretize the test cases and to map the logical signal names to the technical ones and with respective values.

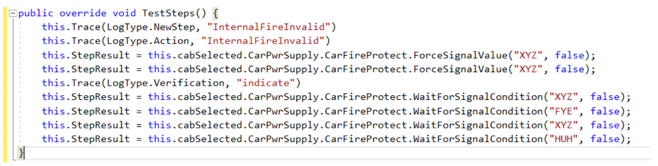
The rule 1 defines a mapping between the each action construct of a model element, from the JSON file, having a 'string' containing abstract test data with a 'list' of variable names and their respective values. A mapping of each variable names from the 'list' with 'logical names' of the signals (provided in the XML file) has been provided in rule 2. Similarly, these 'logical names' are mapped to their corresponding 'type and technical names' of the signals in rule 3 whereas rule 4 provides a mapping between the 'values' (extracted from the JSON file) and 'logical name' of the signals. Rule 5 maps the 'values of signal's logical names' with the 'values of respective technical names' of the signals. However, if a string contains a variable representing a timing constraint with a keyword 'Time', it has been mapped to the value of 'ResponseTime' in rule 6, which will be used as a parameter in the verification step of the generated test script.

TABLE I
MAPPING RULES FOR TEST CASE CONCRETIZATION

ID.	Constructs	Mapped Constructs
1	Model $\rightarrow$ actions $\rightarrow$ Action $\rightarrow$ Strings	List<VariableName, Value>
2	List<VariableName, Value> $\rightarrow$ VariableName	Signals $\rightarrow$ Signal $\rightarrow$ LogicalName
3	Signals $\rightarrow$ Signal $\rightarrow$ LogicalName	Signals $\rightarrow$ Signal $\rightarrow$ SignalType Signals $\rightarrow$ Signal $\rightarrow$ PrimaryTechnicalName Signals $\rightarrow$ Signal $\rightarrow$ Primary2TechnicalName Signals $\rightarrow$ Signal $\rightarrow$ SecondaryTechnicalName Signals $\rightarrow$ Signal $\rightarrow$ Secondary2TechnicalName
4	List<VariableName, Value> $\rightarrow$ Value	Signals $\rightarrow$ Signal $\rightarrow$ LogicalName $\rightarrow$ Value
5	Signals $\rightarrow$ Signal $\rightarrow$ LogicalName $\rightarrow$ Value	Signals $\rightarrow$ Signal $\rightarrow$ PrimaryTechnicalName $\rightarrow$ Value Signals $\rightarrow$ Signal $\rightarrow$ Primary2TechnicalName $\rightarrow$ Value Signals $\rightarrow$ Signal $\rightarrow$ SecondaryTechnicalName $\rightarrow$ Value Signals $\rightarrow$ Signal $\rightarrow$ Secondary2TechnicalName $\rightarrow$ Value
6	List<VariableName, Value> $\rightarrow$ Variable = 'Time' $\rightarrow$ Value	ResponseTime $\rightarrow$ Value

C. Test Script Generator

The test script generator uses the mapping rules, the data extracted from the JSON formatted test cases, the XML containing logical and technical signal names, and the implementation details of the test scripts (i.e. format, classes and methods to be executed on the target test execution platform) to generate executable test scripts. The generated test scripts contain two main steps for each test case: forcing the input signals and verifying the output signals. Based on the signal type, the test script generator generates the scripts to force and verify the signals accordingly. An example of a generated test script is shown in Figure 4 where line 1 of the code represents the method name, lines 2 and 3 are used to write the logs for actions of test script, lines 4 and 5 are used to force the signals, line 6 writes the log for verification steps, and lines 7, 8, 9 and 10 are used to validate the output signals.



```
public override void TestSteps() {
    this.Trace(LogType.NewStep, "InternalFireInvalid")
    this.Trace(LogType.Action, "InternalFireInvalid")
    this.StepResult = this.cabSelected.CarPwrSupply.CarFireProtect.ForceSignalValue("XYZ", false);
    this.StepResult = this.cabSelected.CarPwrSupply.CarFireProtect.ForceSignalValue("XYZ", false);
    this.Trace(LogType.Verification, "indicate")
    this.StepResult = this.cabSelected.CarPwrSupply.CarFireProtect.WaitForSignalCondition("XYZ", false);
    this.StepResult = this.cabSelected.CarPwrSupply.CarFireProtect.WaitForSignalCondition("FYE", false);
    this.StepResult = this.cabSelected.CarPwrSupply.CarFireProtect.WaitForSignalCondition("XYZ", false);
    this.StepResult = this.cabSelected.CarPwrSupply.CarFireProtect.WaitForSignalCondition("HUH", false);
}
```

Fig. 4. An Example of Generated Test Steps in C# script

The generated test scripts can be executed on the test simulation platform for TCMS. But for execution, the simulation platform also requires some libraries (specifically designed for the TCMS developed at BT) and a configuration file containing information related to SUT, paths for storing test logs etc. The test simulation platform, after executing the test scripts, generates test verdicts specifying the failed and passed test steps.

IV. Experimental Evaluation

This section presents the evaluation of generated test scripts using an industrial case study of a fire detection sub-system controlled by the TCMS. Initially, we have developed an EFSM model representing the SUT using requirements and test specifications. We have generated test cases and scripts with TIGER and executed the test scripts on the SUT. We have also injected some faults in the model, generated and executed test scripts to evaluate the framework in terms of fault detection from conformance perspective.

A. SUT

The case selected for the evaluation belongs to an ongoing TCMS development project for MOVIA vehicle product of BT's family of metro cars. TCMS is the centre of a distributed system, built upon an open standard IP-technology to communicate and control multiple sub-systems (i.e. doors, air conditioning, ventilation etc.) of the modern train. In discussion with BT, we have selected requirements of the SUT related to the fire detection sub-system of TCMS [21]. TCMS uses fire detection system for the indication of two type of fires in driver's cab: internal fire and external fire. The fire detection system uses two instances of a Fire Detection Control Unit (FDCU) device. Both FDCUs can have two states: slave and master. The fire detection system sends signal to the TCMS about the status of fires depending on the state of each FDCU. Based on these signals, TCMS indicates internal or external fire.

B. Modelling of SUT

We have modelled the expected behaviour of aforementioned sub-system in GW studio by exploring and understanding the requirements and test specifications as well as by getting continuous input from the testing team at BT. GW studio is one of the versions of GW to create and validate the EFSM model of the system. The model created in GW studio consists of nodes, edges and guard conditions. Nodes represent the states, edges represent the transition taken by the system from one state to another and guard conditions are Boolean expressions representing the expected behaviour of the model. To model the fire detection system, we have identified all possible states, transitions and guard conditions of the system based on requirements and test specification. Figure 5 depicts the EFSM model in three diagrams representing the SUT.

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

C. Fault Injection in EFSM Model

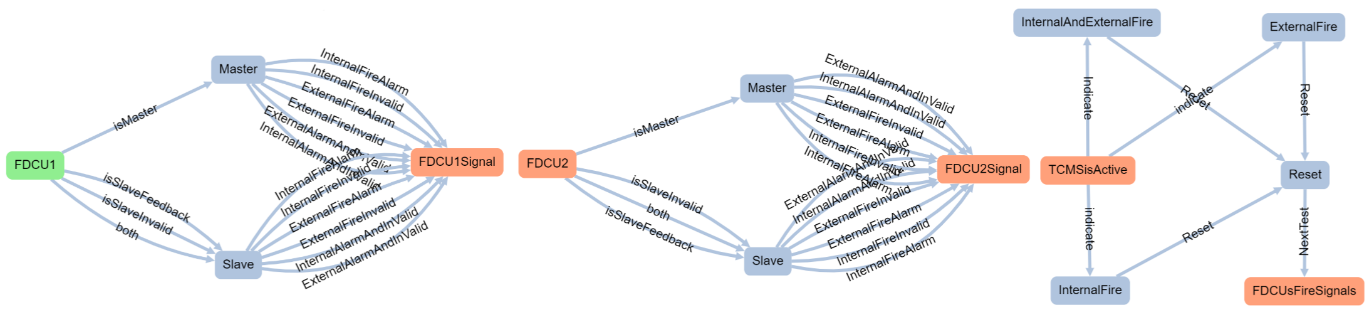
Due to unavailability of source code, we have injected some faults based on insertion and omission of mutant operators [5] in the model to produce faulty test scripts. We have used three mutant operators (output exchanged, change in guards/programming mistake and state missing) and created three versions of the faulty model. In first version of the faulty model, we exchanged the output values of internal and external fire such that if a system is supposed to indicate the internal fire, it will indicate the external fire and vice versa. Similarly, in second version, we made some changes in the guard conditions while we removed the `Master` state of one of the FDCUs in the third version to make these models contradictory to the original specification.

D. Results

This section describes our findings based on fault injection analysis to evaluate the test scripts generated by TIGER.

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

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



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

Fig. 5. EFSM Model representing the Fire Detection System controlled by BT's TCMS

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

TABLE II
COMPARISON BETWEEN TIGER-GENERATED TEST SCRIPTS USING CORRECT MODEL AND MANUALLY-WRITTEN TEST SCRIPTS.

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

V. Discussion & Validity Threats

We have used three different mutation operators to evaluate TIGER, however, we also tried with other mutation operators discussed in [5]. These other mutation operators did not result in any different behavior in our case. For example, we induced some faults based on 'arc missing', 'output missing', 'event missing', 'destination exchanged' and 'event exchange' operators. But event and destination exchange had similar affect on the model as output exchange. Similarly, output missing, event missing and arc missing showed no affect on the model, and induced faults based on these operators only resulted in less input combinations with no failed test steps, so we have neglected these operators in our study.

For a thorough evaluation of TIGER, a proper cost-benefit analysis is required. Although, a large number of generated test steps from TIGER yielded greater execution time than manually written test steps, an added benefit may be is better combinatorial coverage than manually-written test scripts. However, a proper quantitative comparison with combinatorial testing is left as an interesting future work, along with the optimization to reduce the generation time and configuration for use at a different level of simulation testing (application level) for testing BT's CPS.

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

VI. Conclusion

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

Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement Nos. 871319 & 957212.

TABLE III

Comparison between TIGER-generated test scripts using correct and faulty models. Red shade shows failed test steps and green shows no failures detected.

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

References

[1] R. V. Binder, B. Legeard, and A. Kramer, "Model-based testing: where does it stand?" *Comm. of the ACM*, vol. 58, no. 2, pp. 52–56, 2015.

[2] A. Kramer and B. Legeard, *Model-Based Testing Essentials*. Wiley Online Library, 2016.

[3] D. Xu, W. Xu, M. Kent, L. Thomas, and L. Wang, "An automated test generation technique for software quality assurance," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 247–268, 2014.

[4] A. C. Paiva, D. Maciel, and A. R. da Silva, "From requirements to automated acceptance tests with the rsl language," in *Intl. Conf. on Evaluation of Novel Approaches to SE*. Springer, 2019.

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

[6] S. Mohacsi, M. Felderer, and A. Beer, "A case study on the efficiency of model-based testing at the european space agency," in *8th Intl. Conf. on Software Testing, Verification and Validation*. IEEE, 2015.

[7] J. Wang, X. Bai, L. Li, Z. Ji, and H. Ma, "A model-based framework for cloud api testing," in *41st Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2017.

[8] R. Marinescu, M. Saadatmand, A. Bucaioni, C. Seceleanu, and P. Pettersson, "A model-based testing framework for automotive embedded systems," in *40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014.

[9] R. Anbunathan and A. Basu, "Automation framework for test script generation for android mobile," in *2nd IEEE Intl. Conf. on Recent Trends in Electronics, Information & Communication Tech.* IEEE, 2017.

[10] P. Gupta and P. Surve, "Model based approach to assist test case creation, execution, and maintenance for test automation," in *1st International Workshop on End-to-End Test Script Engineering*, 2011.

[11] J. L. de Moura, A. S. Charão, J. C. D. Lima, and B. de Oliveira Stein, "Test case generation from BPMN models for automated testing of web-based bpm applications," in *17th International Conference on Computational Science and Its Applications*. IEEE, 2017.

[12] S. Bonfanti, A. Gargantini, and A. Mashkoor, "Generation of behavior-driven development c++ tests from abstract state machine scenarios," in *International Conference on Model and Data Engineering*. Springer, 2018, pp. 146–152.

[13] J. Vanhecke, X. Devroey, and G. Perrouin, "Abscon: a test concretizer for model-based testing," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019, pp. 15–22.

[14] V. H. Fragal, A. Simao, A. T. Endo, and M. R. Mousavi, "Reducing the concretization effort in fsm-based testing of software product lines," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2017, pp. 329–336.

[15] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "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*. New York, NY, USA: Association for Computing Machinery, 2007.

[16] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 1, pp. 59–76, 2015.

[17] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux, and A. Vernotte, "Chapter two - recent advances in model-based testing," ser. Advances in Computers, A. Memon, Ed. Elsevier, 2016, vol. 101, pp. 53 – 120.

[18] R. Marinescu, C. Seceleanu, H. Le Guen, and P. Pettersson, "Chapter three - a research overview of tool-supported model-based testing of requirements-based designs," ser. Advances in Computers, A. R. Hurson, Ed. Elsevier, 2015, vol. 98, pp. 89 – 140.

[19] W. Li, F. Le Gall, and N. Spaseski, "A survey on model-based testing tools for test case generation," in *Tools and Methods of Program Analysis*, V. Itsykson, A. Scedrov, and V. Zakharov, Eds. Cham: Springer International Publishing, 2018, pp. 77–89.

[20] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[21] M. N. Zafar, W. Afzal, E. P. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui, "Model-based testing in practice: An industrial case study using GraphWalker," in *14th Innovations in Software Engineering Conference (formerly knownas India Software Engineering Conference)*. ACM, 2021.