Towards a Workflow for Model-Based Testing of Embedded Systems

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

ABSTRACT

Model-based testing (MBT) has been previously used to validate embedded systems. However, (i) creation of a model conforming to the behavioural aspects of an embedded system, (ii) generation of executable test scripts and (iii) assessment of test verdict, requires a systematic process. In this paper, we have presented a three-phase tool-supported MBT workflow for the testing of an embedded system, that spans from requirements specification to test verdict assessment. The workflow starts with a simplistic, yet practical, application of a Domain-Specific Language (DSL) based on Gherkin-like style, which allows the requirements engineer to specify requirements and to extract information about model elements (i.e. states and transitions). This is done to assist the graphical modelling of the complete system under test (SUT). Later stages of the workflow generates an executable test script that runs on a domain-specific simulation platform. We have evaluated this tool-supported workflow by specifying the requirements, extracting information from the DSL and developing a model of a subsystem of the train control management system developed at Alstom Transport AB in Sweden. The C# test script generated from the SUT model is successfully executed at the Software-in-the-Loop (SIL) execution platform and test verdicts are visualized as a sequence of passed and failed test steps.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Software and its engineering** → **Software verification and validation**; • **Computing methodologies** → **Modeling methodologies**.

KEYWORDS

Domain-Specific Language, Model-Based Testing, Software-in-the-Loop

ACM Reference Format:
Muhammad Nouman Zafar, Wasif Afzal, and Eduard Enoiu. 2021. Towards a Workflow for Model-Based Testing of Embedded Systems. In *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '21), August 23–24, 2021, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3472672.3473956

A-TEST '21, August 23–24, 2021, Athens, Greece

ACM ISBN 978-1-4503-8623-4/21/08.
https://doi.org/10.1145/3472672.3473956

1 INTRODUCTION

Model-Based Testing (MBT) [16] uses an explicit model that captures the intended behavior of the system under test (SUT). Generally, some modelling notations are used such as various UML diagrams, finite state machine (FSM), etc. that depend on requirements mostly written in a natural language. In some cases, requirements are specified using a semi-structured notation, e.g., the Gherkin (*Given-When-Then*) style. To facilitate the modelling efforts, a Domain-Specific Language (DSL) could be used to define the requirements in a consistent format and to extract certain information (such as states, transitions, variables and their values) that can be used for generating an explicit model. A DSL contains an abstract syntax based on a meta-model to provide a custom and platform independent support for a specific domain [18]. A DSL is implemented by exploring the domain concepts and analysing the structure of specification documents [13].

In this paper, we propose a tool-supported, end-to-end MBT workflow, starting from requirements specification and ending at test verdict assessment. This workflow includes an initial implementation of a DSL based on the Gherkin-like style using XText. XText is one of the well-known open source languages for the development of DSLs supported by the Eclipse Modelling Framework (EMF). We have defined a grammar for the DSL in XText representing the structure of the Gherkin-like format and used it to generate the artefact containing the states, transitions, variables, and their corresponding values that should be included in a FSM model representing the SUT. We have provided a mapping between the information extracted from the requirements specified in the DSL and the model representing the system under test (SUT). We have also evaluated the abstract test cases generated from the model by transforming them into a concrete and executable test script. The proposed workflow is then successfully applied to test a subsystem of the train control management system at Alstom Transport AB, Sweden.

The overall objective of this paper is thus to propose, implement and apply a tool-supported MBT workflow describing domain specific requirements, utilizes these for modelling and generates concrete and executable test scripts to validate the embedded system under test. This paper is an extension of our recent work on this topic [19, 20] that augments an MBT workflow through the addition of a simple but practical DSL for assisting domain expert with model generation.

2 BACKGROUND

Testing of an embedded system is usually done at several levels of simulation. Development and testing the models representing the system happens at model tests (MT) and Model-in-the-Loop (MIL) test levels [3]. Further, in Software-in-the-Loop (SIL) level, the

software is developed and tested using experimental hardware or simulations. In Hardware-in-the-Loop (HIL) level, the real hardware is tested in a simulated environment.

MBT [15] is an automated test generation technique, which involves fully or semi-automated processes to produce test artefacts such as test cases, test oracles and test scripts to validate a system based on behavioral or functional models representing the SUT. It supports multiple algorithms and coverage criteria to traverse through the model elements (i.e. states and transitions) and generates test sequences known as abstract test cases. The abstract test cases are then transformed into concrete and precise actions to validate a system.

Several model-based tools are available [10] that are based on a variety of modelling notations and test generation methods. GraphWalker (GW)[1], which is used in this paper, is one such MBT tool based on finite state machine (FSM) models. It provides multiple generator algorithms (i.e. random, quick_random, A_Star etc.) and coverage criteria (requirement_coverage, edge_coverage, vertex_coverage etc.) to traverse through the model elements.

A Domain-Specific Language (DSL) [17] is a powerful tool for expressing domain concepts in a consistent and well-structured format, particularly in terms of specifying domain-specific system requirements. Implementing a DSL entails the development of a software capable to process and interpret the program or requirements, e.g. Xtext [7] is a framework to help implement a DSL and it is available as a Java plugin for the Eclipse IDE. For creating a DSL, one needs to define a grammar and then use Xtend [2] to generate the language support infrastructure (i.e., editor ecore meta-model, API).

Behaviour-Driven Development (BDD) is an agile methodology that supports automated testing. It provides a common view of domain between business partners, domain experts, testers, and developers to ratify a concrete understanding of the system and its behaviour [6]. One way of documenting BDD requirements and test cases is the Gherkin DSL (*Given-When-Then*) format. The preconditions of the system are specified with a keyword starting with *Given*, actions are expressed using *When* whereas the expected outcomes are stated with a keyword containing *Then* [11]. There is some evidence to suggest that BDD is a cost and time effective methodology to test an embedded system [12].

3 RELATED WORK

Olajubu [13] proposed a domain specific modelling notation to specify requirements. The paper also describes a tool that generates test cases to satisfy the Modified Condition/Decision Coverage (MC/DC) by applying model transformation approach to textual test cases generated from the DSL. The results showed that the requirements specified using DSL are atomic, complete, and unambiguous and experimental evaluation of generated test cases guaranteed the accuracy of the proposed methodology. Similarly, a DSL-based tool to automatically generate test cases and oracles from functional requirements has been presented in [1]. Results showed a satisfactory expressiveness of DSL to write requirements and efficient generation of test steps that can be used to validate a system.

[1] https://graphwalker.github.io/

Galinier [8] proposed a DSL-based seamless approach to validate the requirements for embedded systems using Eiffle solver and to reduce the inconsistencies by providing a bridge between different stakeholders involved in a project. The results showed that the proposed approach is efficient for identifying errors in requirements. In [9], Hoisl et al. specified scenarios in Given-When-Then format, transformed them into executable test scenarios providing traceability of domain requirements and executed the generated test scenarios on an execution platform.

Broenink et al. [4] developed a tool and defined a workflow to automate the testing of cyber-physical systems. The tool consists of a testing language, a simulator and a post processor. After executing the tests generated from the DSL on the simulator, the test data was extracted and processed by the post processor to obtain final output. The empirical study showed that the tests generated by the tool are robust and executable. Similarly, Bucaioni et al. [5] have proposed a model-based approach to automatically generate test scripts for product variants in the railway domain. They have also defined a DSL to specify abstract test cases and used it to generate test scripts using a model to text transformation approach. The results showed the applicability of the proposed approach in the industrial setting.

Our work builds on these previous works to propose an end-to-end MBT workflow for embedded system testing. It is supported by associated tooling and is capable of expressing requirements in a DSL that generates an artefact utilized for modeling. The workflow then continues to generate a concrete test script that is executable on a domain-specific simulation platform and finally the test verdicts are available for assessment.

4 PROPOSED WORKFLOW

Our proposed MBT workflow has been divided into three phases as shown in Fig. 1.

- **Phase 1:** Requirements description and automatic generation of supporting artefact for modelling.
- **Phase 2:** Generation of executable test scripts using Model-Based Test scrIpt GenEration fRamework (TIGER) [20].
- **Phase 3:** Execution of generated test scripts on simulation levels and test verdict assessment.

4.1 Phase 1

The first phase deals with the modelling aspect of the SUT. In our case, we have the requirements specification as well as the test specification (to include the tester perspective) as an input to the modelling. The requirements specification in our case is written in a specific DSL that resembles the Gherkin format. The result of Phase 1 is a FSM model in `JSON` or `GraphML` format, that is in turn used as an input to Phase 2 to generate the executable test scripts.

4.1.1 Requirements Specification in Gherkin-like DSL and the Xtext Grammar. In order to model the SUT, the first input is a descriptive form of requirement scenarios written in a Gherkin-like DSL. These scenarios have pre-conditions, post-conditions and actions to specify the behaviour of the SUT.

Fig. 2 represents the meta-model for describing the concepts of requirements specification in a Gherkin-like format. In the standard

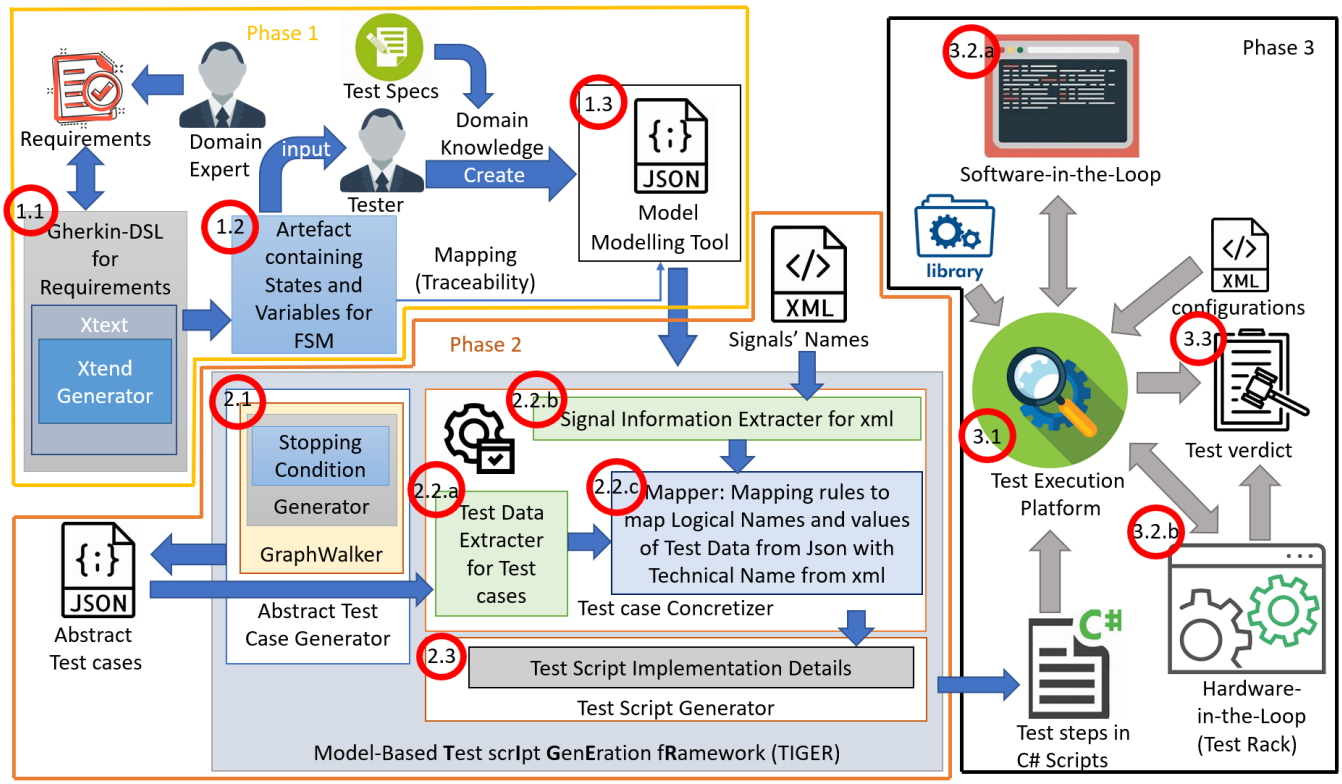

Figure 1: A MBT workflow for an embedded system.

Gherkin format, the pre-condition is expressed using the *Given* keyword, action is specified using the *When* keyword and outcome is described using the *Then* keyword. In our case, after analyzing the requirements and domain concepts, we have developed the DSL for requirements in Gherkin-like format which contains a detailed description about each state of an embedded system and every event is determined by one or more signals (logical names of input/output signals and their values). Moreover, pre-conditions, actions and system responses could be represented as a sequence of multiple events.

In the meta-model (Fig. 2), the top-level element is the *RequirementSpecification*. The *RequirementSpecification* contains *Requirements*, each with an unique identifier to validate the atomicity of the requirement. Each requirement contains the definition of *Precondition* that specifies the initial state of the system, *Trigger* defining the actions required for an event and state achieved after an event, *SystemResponse* specifying the state that will be achieved after the response of a system and *Time* to define the timing constraint for a system to respond. Moreover, *Trigger* also contains a reference for the initial state of the system. Each *State* in pre-condition, trigger and system response contains the definition of a *State Variable*, which represents the name of the input/output signal with their corresponding value. The implementation of the Gherkin-like DSL in concrete syntax using the Xtext grammar is shown in Fig. 3.

4.1.2 Automatic Generation of Supporting Artefact for Modelling. We have also implemented the Xtend generator as shown in Fig. 4 to extract the information from the Gherkin-like DSL produced in the Eclipse-based Xtext editor. The Eclipse editor provides multiple built-in features such as highlighting the syntax based on DSL (i.e. preferences for font and color, style for comments and keywords), predefined templates, an outline view, and assistance for code completion, and error handling [14]. The Xtend generator contains the mapping between each meta-model element of the defined DSL and model elements of the FSM (such as states and transitions). It extracts the information such as model name, state name, transitions, transition variables and their corresponding values from each requirement specified by the requirements engineer in the editor. The requirements specified using DSL are structurally complete, however, the validation checks to validate each requirement's atomicity and unambiguity can be included as an advanced feature in Xtend, which is currently done manually in our case.

The result of running the Xtend generator is an automatic generation of the artefact containing the information about modelling elements such as states and transitions. Fig. 6 shows a sample of the extracted information from the Xtend-generated artefact, that is discussed in detail in Section 5.2.

4.1.3 Modelling of the SUT. Our experience with modelling the SUT [19] shows that the information generated from requirements specification alone is not sufficient to embed all behavioural aspects in the model. The modelling activity has to be iterative, also embedding tester's expertise and perspective regarding the desired behavioral aspects in the model. While the initial model of the SUT

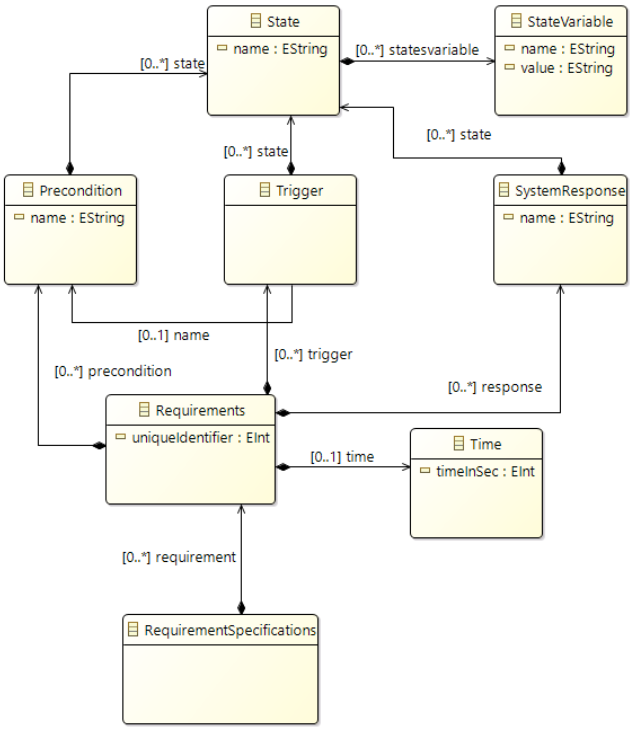

Figure 2: Meta-model for Gherkin-like DSL

is created based on the requirements specification, our experience suggests that it evolves by including the tester's domain knowledge. The inclusion of tester's perspective, viewpoint and expertise is an important step towards making the model complete with respect to scenarios that are missed in the requirements specification but still considered obvious by the tester [19].

Thus, while the automatic generation of the artefact containing the states and variables (based on the Gherkin-like DSL for the requirements specification) helps create an initial version of the SUT model, it still needs tester's domain expertise to reflect complete behavioral aspects of the SUT.

The final model can be made in different editors, however we have used GW as the modelling environment where the model is exported as a `JSON` file, details regarding which appear in Section 5.3.

4.2 Phase 2

Following a transformation approach for MBT [15], we have defined the mapping rules to transform the abstract test cases into concrete ones using the TIGER framework. TIGER consists of three parts: abstract test case generator, test case concretizer and test script generator. While the detailed description of these steps is explained in [20], here we briefly summarize these parts.

4.2.1 Abstract Test Case Generator. GW takes as an input the model file in `JSON/GraphML` format and generates the abstract test cases by traversing through the model elements (i.e. states and transitions) based on a generator algorithm (such as random, quick_random, Astar etc.) and a stopping condition (such as edge_coverage, vertex_coverage etc.).

4.2.2 Test Case Concretizer. The test case concretizer converts the abstract test cases into concrete by mapping the logical signal names with their technical counterparts and corresponding values. Testers and developers at our case organization (Alstom Transport AB) use these logical names as initial names of the signals in the early phases of development. Later in the development, technical signal names become available that represent the actual signal names used by the SUT for its normal operations. Hence, test case concretizer extracts the test data i.e. (variable names and its respective values) from the generated abstract test cases (available in a `JSON` file), extracts the required information about technical signal names from a XML file, and maps the logical signal names with technical signal names and their corresponding values based on defined mapping rules [20].

4.2.3 Test Script Generator. Once the abstract test cases are converted into concrete test cases, the test script generator generates the test script in C# language using the implementation details of the SUT (i.e. script format, libraries and methods to be executed on the target test execution platform, SIL & HIL). The generated test script contains two types of steps for each test case, forcing the input signals and verifying the expected output signals, to validate the expected behaviour of the SUT. An example test script is shown later in Section 5.4.

4.3 Phase 3

The test scripts generated by TIGER can be executed on Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) levels after including specifically designed libraries and integrating a configuration file in our case company's `C#` generic test framework. After the execution of generated test scripts on either SIL or HIL level, the test framework generates a test verdict containing a detailed list of passed and failed test steps, described further in Section 5.4.

5 CASE STUDY

The implementation and evaluation details of the workflow are presented as an industrial case study in the following subsections.

5.1 SUT

The case selected for evaluation is the part of an ongoing Train Control Management System (TCMS) development project at Alstom Transport AB, Sweden, concerning one of the metro train projects. TCMS is a distributed control system built upon highly complex infrastructure and uses an open standard IP-technology for communication to control the subsystems such as ventilation, doors, heating, and air conditioning. We have selected requirements of TCMS related to the fire detection subsystem to evaluate our proposed workflow. The fire detection subsystem in TCMS is responsible for the indication of two types of fire, internal and external, in the driver's cab. It uses two instances of the Fire Detection Control Unit (FDCU) device to achieve this function. Each FDCU device can have two states, Master and Slave. Both FDCUs communicate with fire and smoke sensors in order to detect fire. When a sensor detects a fire, the fire detection subsystem sends signals to the TCMS representing the current state of each device along

```
grammar org.xtext.adaptness.btdsl1.BtDsl with org.eclipse.xtext.common.Terminals

generate btDsl "http://www.xtext.org/adaptness/btdsl1/BtDsl"

RequirementSpecifications:
    requirement+=Requirements*
;
Requirements:
    'Requirement#' uniqueIdentifier=INT ':'
    'Given' precondition+=Precondition
    'When' trigger+=Trigger
    'Then' response+=SystemResponse
    'Within' time=Time 'milisec';
Precondition:
    ('a'|'both'|'no') name=ID ('are'|'is') state+=State ;
Trigger:
    ('a'|'both'|'no') name=[Precondition] ('reports'|'report') state+=State;
SystemResponse:
    name=ID 'shall' 'indicate' state+=State ;
Time:
    timeInSec=INT;
State:
    name=ID '(' (statesvariable+=StateVariable ('AND'|'OR')?)* ')';
StateVariable:
    name=ID '=' value=('0'|'1')?;
```

Figure 3: Concrete syntax of the Gherkin-like DSL using the Xtext grammar

```
class BtDslGenerator extends AbstractGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
    IGeneratorContext context) {

        var Specs = resource.contents.head as RequirementSpecifications

            fsa.generateFile('ExtractedInfo.txt', Specs.genSpec)
    }
    def genSpec(RequirementSpecifications Specs)'''
        «FOR Requirements R: Specs.requirement»
            Requirement# «R.uniqueIdentifier»

                «FOR Precondition t: R.precondition»
                    Model Name: «t.name»
                    State Name:«t.name»
                    «FOR State s: t.state»
                        State Name: «s.name»
                        Transition/Action Variables:
                        «FOR StateVariable statevar: s.statesvariable»
                            Variable Name: «statevar.name» = «statevar.value»
                        «ENDFOR»
                    «ENDFOR»
««««                        «t.genStateInf»
                «ENDFOR»
                «FOR Trigger con: R.trigger»
                    «FOR State s: con.state»
                        State Name: «s.name»
                        Transition/Action Variable:
                        «FOR StateVariable statevar: s.statesvariable»
                            Variable Name: «statevar.name» = «statevar.value»
                        «ENDFOR»
                    «ENDFOR»
««««                        «con.genCstateInf»
                «ENDFOR»
                «FOR SystemResponse SR: R.response»
                    Model Name: «SR.name»
                    State Name: «SR.name»
                    «FOR State s: SR.state»
                        State Name: «s.name»
                        Transition/Action Variable:
                        «FOR StateVariable statevar: s.statesvariable»
                            Variable Name: «statevar.name» = «statevar.value»
                        «ENDFOR»
                    «ENDFOR»
««««                        «SR.genSRStateInf»
                «ENDFOR»
            Time: «R.time.timeInSec»

        «ENDFOR»
        '''
```

Figure 4: Implementation of Xtend generator

with the signals representing the fire detected by the sensor of each device. Based on these signals, TCMS takes the desired action of turning on the LED light in the driver's desk via electrical wiring, indicating detection of fire in the train.

5.2 Generated Artefact

In contrast to a typical 'Given-When-Then' Gherkin format, Alstom Transport's domain expert used 'Given-Then-Within' format to specify requirements where the 'Within' clause specifies the timing constraint of each requirement. The general format of requirements specification is as follows:

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

After specifying the requirements in Xtext-based Eclipse editor as shown in Fig. 5, the Xtend generator generated a text file containing the information about model elements as shown in Fig. 6. For a complex system, having multiple diagrams/models representing the sub-parts of the SUT, provides a high level of abstraction and can be helpful in determining errors and their causes due to improved traceability. Moreover, these models need to be connected through shared nodes. So, the Xtend generator generated the information by dividing the system in different models. The generated artefact contains `Requirement#`, `Model Name`, `State Name`, `Transition/Action`, and `Variable Name` along with their corresponding values for each requirement specified in the editor.

As mentioned in Section 4.1.3, incorporating the tester's perspective in the model makes behavioral aspects more complete

```
Requirement# 01: Given a FDCU is Master (variable1 = 1 AND variable2 = 1)
 When a FDCU reports InternalFire (variable3 = 0 OR variable4 = 0)
 Then TCMS shall indicate InternalFire (variable5 = 0) Within 2500 milisec
```

Figure 5: An example of a requirement specified in Xtext

```
Requirement# 1

    Model Name: FDCU
    State Name:FDCU
    State Name:  Master
    Transition/Action Variables:
    Variable Name: variable1 = 1
    Variable Name: variable2 = 1
    State Name:   InternalFire
    Transition/Action Variable:
    Variable Name: variable3 = 0
    Variable Name: variable4 = 0
    Model Name: TCMS
    State Name: TCMS
    State Name: InternalFire
    Transition/Action Variable:
    Variable Name: variable5 = 0
Time: 2500
```

Figure 6: A sample of extracted information from the Xtend generated artefact

and representative of the SUT, therefore a test specification document based on the specified requirements was made available. It included manually-written test cases in natural language, where each test case was composed of a series of steps (actions) and their corresponding expected results.

5.3 Modelling of the SUT

Based on the information in the Xtend-generated artefact alone, we have manually created the initial models representing FDCU and TCMS in GW, as shown in Fig. 7. On both sides of the models, we have also provided an outline view of elements generated by Xtend generator and mapping of some generated elements with initial model elements using arrows. However, initial models contain all the generated elements in it. After refining and including tester's domain knowledge in the modelling process, the final model of the system is shown in Fig. 8, containing two diagrams representing each FDCU and one diagram representing TCMS. In this final model, we have merged some initial states of the FDCUs: `InternalFireIndication`, `ExternalFireIndication` and `InternalAndExternalFireIndication` (`FDCU1Signal` in diagram representing FDCU1 and `FDCU2Signal` in diagram representing FDCU2, in Fig. 8), as these states represent the actions of a sensor to detect the presence of fire. Whereas, TCMS indicates the fire as the final output based on signals received from FDCUs. `FDCU1`, `FDCU1Signal`, `FDCU2`, `FDCU2Signal`, `TCMSisActive` and `FDCUFireSignals` represent the shared states used by the GW for traversing between the model elements to generate abstract test cases. Similarly, `Master`, `Slave`, `InternalFire`, `ExternalFire`, `InternalAndExternalFire` and `Reset` states represent the output states of corresponding transitions. Moreover, the addition of one new state `Reset` and 21 new transitions are the result of scenarios added based on tester's domain expertise and perspective.

5.4 Executable Test Script Generation

After creating the model, we have generated the abstract test cases and executable test scripts. We have provided the model in `JSON` format to TIGER and generated abstract test cases based on random generator algorithm and 100% edge coverage criterion. After the generation of abstract test cases, TIGER used the `XML` file containing the logical and technical signal names, and type of the signals (i.e. input/output) to generate the executable test scripts as shown in Fig 9.

5.5 Execution of Generated Test Scripts at Software-in-the-Loop (SIL) Level

The generated test script is successfully executed at the SIL level by creating a generic C# project in Visual Studio and by importing TCMS libraries and a configuration file (as mentioned in Section 4.3). After the execution, the test execution platform generates test verdict in an `html` format as shown in Fig. 10. The test verdict `html` contains the configuration steps (in the beginning), test steps (forced signal values) along with their pass or fail verdicts and a summary of the whole execution.

6 FUTURE WORK

The developed DSL helped us to describe the requirements in a well-structured format. The requirements specified using DSL are structurally complete and exclusive of errors, but DSL can be extended by adding validation checks to ensure other quality attributes of requirements (i.e. atomicity and unambiguity). The Xtend generator successfully generated artefact containing information about the states and transitions but to model a system, one should have the domain knowledge and experience of modelling to understand the generated information. Moreover, a semi or fully automatic transformation of the textual description of requirements to an FSM model in `JSON` or `GraphML` format is left as an interesting future work.

While the generated test script was successfully executed at the SIL level, a thorough cost-benefit analysis of the workflow is still required. Also, optimization of the generated test steps and comparison with combinatorial testing can provide better insights into the effectiveness of the proposed workflow.

7 VALIDITY THREATS

The threat related to internal validity of this study concerns the correctness of the SUT model. The DSL helped us to identify the model elements but it took multiple iterations to develop the correct version of the model, that was ultimately achieved by including the tester's domain knowledge. The final model was also declared correct by a domain expert. Another threat is with respect to the general applicability of the defined workflow. Both, the DSL and the TIGER, are particularly designed for the testing of the embedded system developed at Alstom Transport AB, thus they are applicable to other projects inside the company. However, due to dependencies on the proprietary platforms and frameworks, the workflow may not be applicable as it is to other embedded systems, where the workflow must change to cater for domain-specific dependencies.

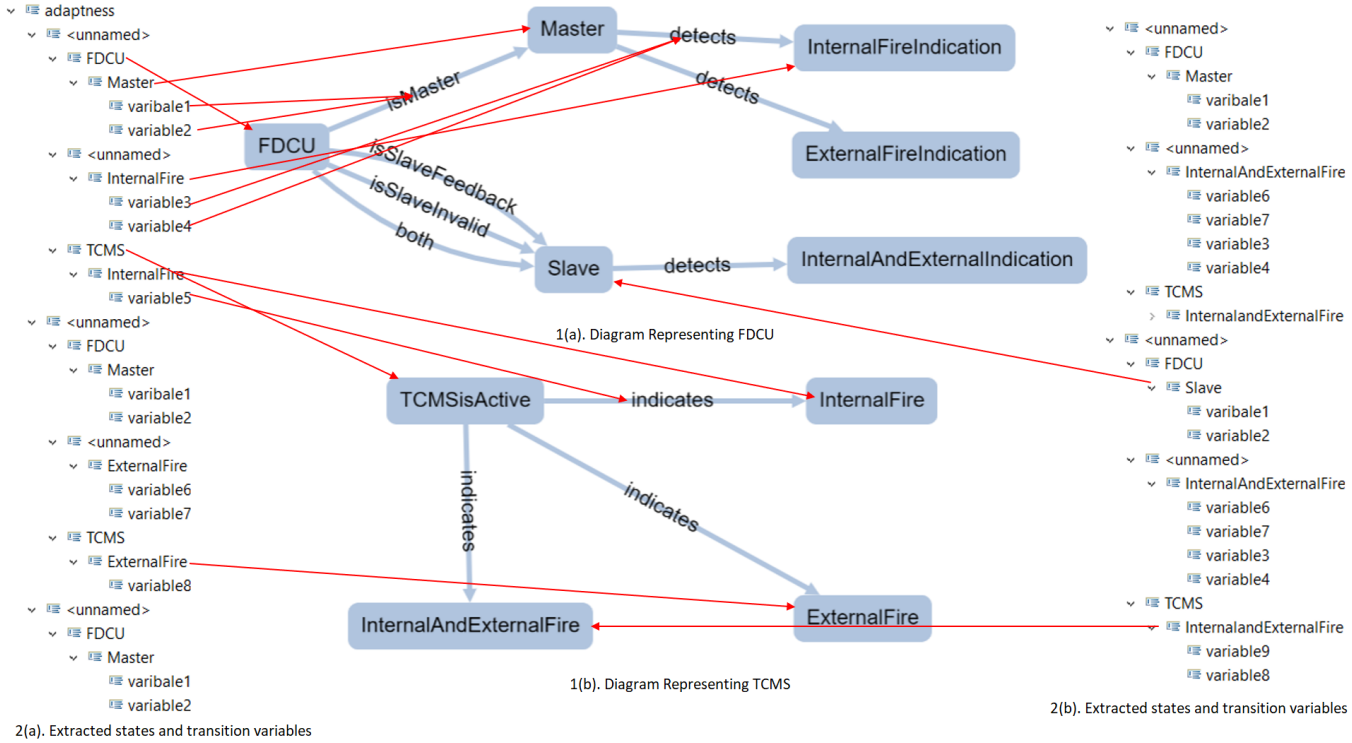

Figure 7: Initial model representing the SUT and the mapping of Xtend-generated elements with model elements

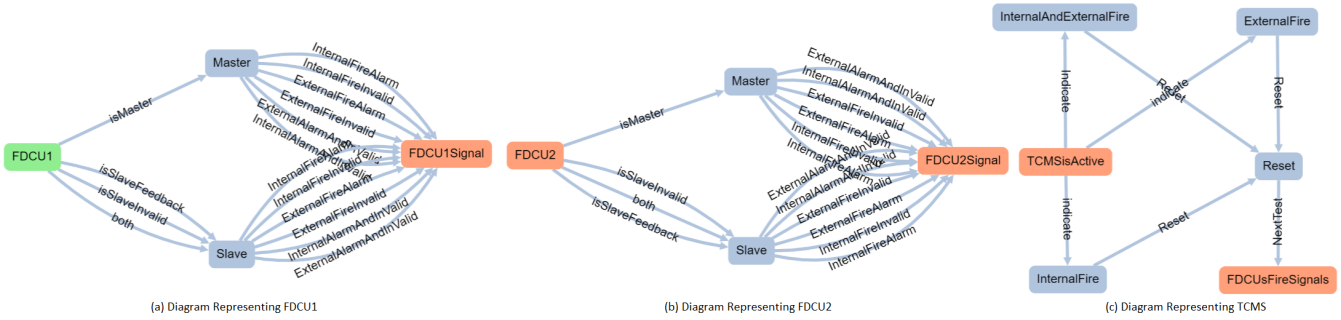

Figure 8: Final model representing FDCUs and TCMS as a black box

```
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);
}
```

Figure 9: An example of the generated test script

Script Names	Test_Gen
Script Version	01
Tester	Nouman
Configuration	XYZ
Result	Successful .

No.	Command	Request\ Parameter\ Expected value	Response\ Details\ Current value	Result	Timestamp	Comment
0	The test case is running on framework version ...				2021-01-04T23:07:06.445+01:00	
Pre-Execution Steps					2021-01-04T23:07:06.445+01:00	
Test Step 1: isSlaveFeedback!						
546	Force	Signal1.Name	System.Object->True	OK	2021-01-04T23:26:01.252+01:00	
547	TEST STEP 1 RESULT				2021-01-04T23:26:01.252+01:00	
548	TraceText			Passed		
PostExecution Steps:						
Final test result and summary:						
3388	TraceText			Passed	2021-01-04T23:26:01.252+01:00	Finished
PreExecution Steps: 1, Passed: 1, Failed: 0 Preparation Steps: 1, Passed: 1, Failed: 0 Test Steps: 384, Passed: 384, Failed: 0 Termination Steps: 1, Passed: 1, Failed: 0						

Figure 10: An example of the generated test verdict `html`

8 CONCLUSION

We have proposed a systematic MBT workflow and tool support to facilitate the simulation-based testing process of an embedded system. The workflow is divided into three main phases: 1) requirements description and automatic generation of supporting artefact for modelling, 2) generation of executable test scripts using Model-Based Test scrIpt GenEration fRamework (TIGER) [20], and 3) execution of generated test scripts on simulation levels and test verdict assessment. We have evaluated the workflow as well as the supported tooling using an industrial case study at Alstom Transport AB. The results show that the tool-supported workflow is practically feasible and the generated test script is executable at the SIL level to produce the test verdict required for an embedded system validation.

ACKNOWLEDGMENTS

The work in this study has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement Nos. 871319, 957212; from the Swedish Innovation Agency (Vinnova) through the XIVT project and from the ECSEL Joint Undertaking (JU) under grant agreement No 101007350.

REFERENCES

[1] A. Arrieta, J. A Agirre, and G. Sagardui. 2020. A Tool for the Automatic Generation of Test Cases and Oracles for Simulation Models Based on Functional Requirements. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops*. IEEE.

[2] L. Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[3] B. Broekman and E. Notenboom. 2003. *Testing embedded software*. Pearson Education.

[4] T. Broenink, B. Jansen, and J. Broenink. 2020. Tooling for automated testing of cyber-physical system models. In *2020 IEEE Conference on Industrial Cyberphysical Systems*. IEEE.

[5] A. Bucaioni, F. D. Silvestro, I. Singh, M. Saadatmand, H. Muccini, and T. Jochumsson. 2021. Model-based Automation of Test Script Generation Across Product Variants: a Railway Perspective. In *2nd ACM/IEEE International Conference on Automation of Software Test*. IEEE Computer Society.

[6] R. Bussenot, H. Leblanc, and C. Percebois. 2018. Orchestration of Domain Specific Test Languages with a Behavior Driven Development approach. In *2018 13th Annual Conf. on SoS Eng*. IEEE.

[7] M. Eysholdt and H. Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM.

[8] F. Galinier. 2018. A DSL for Requirements in the Context of a Seamless Approach. In *33rd Intl. Conf. on Automated SE*. IEEE.

[9] B. Hoisl, S. Sobernig, and M. Strembeck. 2014. Natural-Language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*. IEEE.

[10] W. Li, F. Le Gall, and N. Spaseski. 2017. A survey on model-based testing tools for test case generation. In *International Conference on Tools and Methods for Program Analysis*. Springer.

[11] M. Micallef and C. Colombo. 2015. Lessons learnt from using DSLs for automated software testing. In *8th International Conference on Software Testing, Verification and Validation Workshops*. IEEE.

[12] A. S. Nezhad, J. J. Lukkien, and R. H. Mak. 2018. Behavior-driven Development for Real-time Embedded Systems. In *2018 IEEE 23rd Intl. Conf. on Emerging Technologies and Factory Automation*. IEEE.

[13] O. Olajubu. 2015. A textual domain specific language for requirement modelling. In *10th Joint Meeting on Foundations of SW Engineering*. ACM.

[14] A. Rahman and D. Amyot. 2014. A DSL for importing models in a requirements management system. In *2014 IEEE 4th International Model-Driven Requirements Engineering Workshop*. IEEE.

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

[16] M. Utting, A. Pretschner, and B. Legeard. 2012. A taxonomy of model-based testing approaches. *SW Test, V and R* 22, 5 (2012), 297–312.

[17] M. Vidal, T. Massoni, and F. Ramalho. 2020. A domain-specific language for verifying software requirement constraints. *Science of Computer Programming* 197 (2020), 102509.

[18] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, K. Czarnecki, and B. von Stockfleth. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

[19] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui. 2021. Model-Based Testing in Practice: An Industrial Case Study using GraphWalker. In *14th Innovations in Software Engineering Conference*. ACM.

[20] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, and O. Sellin. 2021. A Model-Based Test Script Generation Framework for Embedded Software. In *The 17th Workshop on Advances in Model Based Testing*. IEEE.