Ref. Ares(2021)4241624 - 29/06/2021

ADEPTNESS – DESIGN-OPERATION CONTINUUM METHODS FOR TESTING AND DEPLOYMENT UNDER UNFORESEEN CONDITIONS FOR CYBER-PHYSICAL SYSTEMS OF SYSTEMS

EUROPEAN COMMISSION

HORIZON 2020

H2020-ICT-01-2019

GA NO. 871319

Deliverable No.	ADEPTNESS D2.1	
Deliverable Title	Report on test benches for simulation-based multi-level testing as a service of CPSoS	
Deliverable Date	2021-06-31	
Deliverable Type	Report	
Dissemination level	Public	
Written by	**MGEP**, MDH, SRL	2021-05-26
Checked by	MGEP, TUW	2021-06-22
Approved by	Executive board	2021-06-29
Status	Final version	2021-06-29

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

Acknowledgement

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

Project partners

1 – MGEP – Mondragon Goi Eskola Politeknikoa – ES

2 – ORO – Orona S. Coop – ES

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

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

5 – BT – Bombardier Transportation Sweden – SE

6 – IKL – Ikerlan S. Coop – ES

7 – EGM – Easy Global Market SAS – FR

8 – MDH – Maelardalens Hoegskola – SE

9 – TUW – Technische Universitaet Wien – AT

Disclaimer:

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

Document Information

Additional author(s) and contributing partners

Name	Organisation
Goiuria Sagardui, Aitor Arrieta, Iñigo Aldalur	MGEP
Franck Le Gall	EGM
Wasif Afzal	MDH

Document Change Log

Name	Date	Comments
V0.1	2021-01-30	Initial draft with sections and subsection
V1.0	2021-05-26	Initial document with version to be reviewed by the executive board.
V1.1	2021-06-18	Document version considering comments provided by Executive Board

Exploitable results

Exploitable results	Organisation(s) that can exploit the result
Validation orchestrator microservice	The validation orchestrator has been exclusively developed by MGEP on top of the ADEPTNESS microservice template developed by Ikerlan. MGEP is the main exploiter of the validation orchestrator and Ikerlan can provide maintenance exploitation activities related to the ADEPTNESS microservice template.
Validation agent microservice	The validation agent has been exclusively developed by MGEP on top of the ADEPTNESS microservice template developed by Ikerlan. MGEP is the main exploiter of the

	validation orchestrator and Ikerlan can provide maintenance exploitation activities related to the ADEPTNESS microservice template.
External tool microservice	The external tool microservice has been exclusively developed by MGEP on top of the ADEPTNESS microservice template developed by Ikerlan. MGEP is the main exploiter of the validation orchestrator and Ikerlan can provide maintenance exploitation activities related to the ADEPTNESS microservice template.
NGSI-LD broker	The NGSI-LD broker ,has been exclusively developed by EGM and has been made available as open-source under the Apache 2.0 licence
TaaS Front-end	The TaaS front-end is exclusively developed by EGM and is intended to become a close for the management of the STELLIO broker.
GraphWalker-based Test Generation	The work related to GraphWalker-based test generation is exclusively done by MDH, in close collaboration with the use case of Bombardier Transportation.

CONTENTS

1. PURPOSE OF THE DOCUMENT 1

1.1. DOCUMENT STRUCTURE 1
1.2. DEVIATIONS FROM THE ORIGINAL DESCRIPTION IN THE GRANT AGREEMENT ANNEX 1 PART A 1
1.2.1. Description of work related to deliverable in GA Annex 1 – Part A 1
1.2.2. Time deviations from original planning in GA Annex 1 – Part A 1
1.2.3. Context deviations from the original plan in GA Annex 1 – Part A 1

2. INTRODUCTION 2

3. BACKGROUND 3

3.1. TESTING STANDARDS FOR CPSS AND CPSOS 3
3.1.1. UTP standard 3
3.1.2. ASAM-XiL 4
3.1.1. TTCN-3 4
3.2. TESTING OF EMBEDDED SYSTEMS AT DIFFERENT LEVELS 5
3.2.1. Model-based testing (MBT) and GraphWalker (GW) 5
3.2.2. Behaviour driven development and Gherkin 6
3.2.3. Automating MBT using Gherkin 6
3.3. TEST EXECUTION FRAMEWORKS AND TEST LEVELS 6
3.3.1. Test levels at TCMS 8
3.3.1.1. TCMS SW Test at SW application level 8
3.3.1.2. TCMS System Integration Test 9
3.3.1.3. TCMS System Test 9
3.3.2. Generalistic testing frameworks for CPSoS 10

4. VERIFICATION AND VALIDATION TEST BENCHES FOR CPSOS 12

4.1. REUSED CONCEPTS 12
4.2. DATA MODEL FOR CPSS 12
4.3. APPROACHES TO THE AUTOMATED TESTING OF CYBER-PHYSICAL SYSTEMS OF SYSTEMS 15

5. DEVELOPMENT OF THE AGILE TESTING FRAMEWORK FOR THE ADEPTNESS ECOSYSTEM 17

5.1. VALIDATION ORCHESTRATOR 17
5.1.1. Introduction 17
5.1.2 Architecture 18
5.1.2.1 REST and MQTT APIs 18
5.1.2.2 Interaction with other components 18
5.1.2.3 Internal architecture 19
5.1.2.4 Deployment 19
5.1.3 Test execution workflow 20
5.1.3.1 Parsing of the input validation plan 20
5.1.3.2 Obtaining additional configuration from external data sources 21
5.1.3.3 Agent configuration and plan execution 21
5.1.3.4 Verdict arrival, aggregation and publication 21
5.1.4 Execution example in the Elevation use-case 21
5.2. VALIDATION AGENT 23
5.2.1. Introduction 23
5.2.2. Architecture 24
5.2.2.1. REST and MQTT APIs 24
5.2.2.2. Internal architecture 25
5.2.2.3. Deployment 27

5.2.3. Execution flow 28
5.2.4. Reception of validation plan 28
5.2.5. Child component configuration 29
5.2.6. Validation plan execution 29
5.2.7. Verdict arrivals & aggregation 29
5.2.8. Submission of verdicts 30
5.2.9. Execution example in the Elevation use-case 30
5.2.9.1. Reception of validation plan by SiL validation agent 30

6. INSTANTIATION OF THE AGILE TEST EXECUTION FRAMEWORK ON THE ELEVATION USE CASE AT MIL, SIL AND HIL TEST BENCHES 34

6.1. INTRODUCTION 34
6.2. ARCHITECTURE 35
6.2.1. REST and MQTT API endpoints 35
6.2.2. Internal architecture 35
6.3. EXECUTION FLOW 35
6.3.1. Subtool execution 36
6.3.2. Status reporting 36
6.4. APPLICATION TO THE ELEVATION USE-CASE 37
6.4.1. Elevate external tool 37
6.4.2. Exposed capabilities for tool 37
6.4.3. Execution of the tool in validation example 38

7. INSTANTIATION OF THE AGILE TEST EXECUTION FRAMEWORK ON THE RAILWAY USE CASE AT MIL, SIL AND HIL TEST BENCHES 40

7.1. CASE STUDY 45
7.1.1. SUT 45
7.1.2. Generated Artefact 45
7.1.3. Modelling of the SUT 46
7.1.4. Executable test script generation 47
7.2. A CLOUD & MICROSERVICE-BASED ARCHITECTURE AND REALIZATION FOR MIL SIMULATION TEST BENCH IN THE RAILWAY DOMAIN 48
7.2.1. GraphWalker as a RESTful Web Service 48
7.2.2. GraphWalker's RESTful API Commands Flow 49
7.2.3. A General Purpose Architecture for Model-Based Testing as a Web Service 49
7.2.4. Variant 1 - The Simple Architecture 51
7.2.4.1. Setting up AWS and Realization of the Simple Architecture 52
7.2.5. Variant 2 - The Dynamic Architecture 56
7.2.6. Realization of Variant 2 - The Dynamic Architecture 57
7.2.7. Variant 3 - The Advanced Dynamic Architecture 58
7.2.8. Realization of Variant 3 - The Advanced Dynamic Architecture 59
7.2.9. Variant 4 - The HTML Server Enhanced Dynamic Architecture 61
7.2.9.1. Realization of Variant 4 - The HTML Server Enhanced Dynamic Architecture 62

8. CONCLUSION AND FUTURE WORK 66

ACKNOWLEDGMENTS 68

DISCLAIMER 68

BIBLIOGRAPHY 68

LIST OF FIGURES

FIGURE 1 TTCN-3 architecture 5
FIGURE 2 HiL test bench in orona 8
FIGURE 3 data flow within ADEPTNESS 14
FIGURE 4 monitoring agent model 15
FIGURE 5 An Example of Generated Test Steps in C# script 16
FIGURE 6 Validation orchestrator schema 17
FIGURE 7 Validation orchestrator validation plan execution workflow 20
FIGURE 8 Validation agent microservice interaction overview 23
FIGURE 9 Execution flow for the validation agent 28
FIGURE 10 Interaction of the External tool microservice with other validation components 35
FIGURE 11 Execution flow for the validation agent 36
FIGURE 12 Execution of the Elevate simulator with the external tool 38
FIGURE 13 A model-based testing workflow for an embedded system, from requirements specification to test verdict assessment, in SiL and HiL simulation platforms 40
FIGURE 14 Meta-model for Gherkin-like DSL shown as an UML class diagram 41
FIGURE 15 Concrete syntax of the Gherkin-like DSL using Xtext grammar 42
FIGURE 16 Implementation of Xtend generator 43
FIGURE 17 An example of a requirement specified in Xtext following the Gherkin-like DSL grammar 43
FIGURE 18 A sample of extracted information from the Xtend generated artefact 46
FIGURE 19 Initial model representing FDCU and TCMS as a blackbox 47
FIGURE 20 Final model representing FDCUs and TCMS as a blackbox 47
FIGURE 21 Graphical User Interface of TIGER 48
FIGURE 22 GW's REST API commands 49
FIGURE 23 A General-Purpose Architecture for MBT as a Web Service 51
FIGURE 24 Variant 1: The Simple Architecture for MBT as a Web Service 52
FIGURE 25 AWS Lambda code for authorizations 53
FIGURE 26 The Body parameter in Postman having the GW-exported JSON file of the fire indication system's model 54
FIGURE 27 The load request sent from Postman and the received payload shown in Postman's user interface 55
FIGURE 28 The *PUT* request sent from the Postman to update *num of inputs* to the value of 32455 56
FIGURE 29 Variant 2 The Dynamic Architecture for MBT as a Web Service 57
FIGURE 30 Loading the JSON file using EC2's public IP in variant two architectures 58
FIGURE 31 Variant 3 The Advanced Dynamic Architecture for MBT as a Web Service 59
FIGURE 32 Creating an Amazon DynamoDB table through AWS Lambda 60
FIGURE 33 Recording of the result of *getStatistics* in the Amazon DynamoDB table 61
FIGURE 34 Variant 4 The HTML Server Enhanced Dynamic Architecture for MBT as a Web Service 62
FIGURE 35 AWS Lambda code for including the HTML front-end 63
FIGURE 36 The HTML front-end of variant 4 64
FIGURE 37 Setting the value of *FDCU2Valid* to true through the HTML user interface 65

1. Purpose of the document

1.1. Document structure

This document shows the developed implementation for the Agile Testing Framework for CPSoS and compatible with the ADEPTNESS ecosystem. Section 2 provides a brief introduction. Section 3 explains the background. Section 4 introduces the re-used concepts from the analysed standards, the data model used for CPSoS testing and the approaches to the automated execution of tests. Section 5 describes the developed generic microservices for the execution of tests within the ADEPTNESS ecosystem. Section 6 describes the instantiation of the framework for the execution of tests within the elevation use-case. Section 7 describes the instantiation of the framework for the execution of tests within the rail-way use-case. Section 8 concludes the deliverable.

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

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

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

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

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

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

There are no deviations from the Annex 1.

2. Introduction

Testing activities are of extreme importance on CPSs testing. Usually, CPSs are tested by using simulation. Simulation allows to integrate the physical layer of the CPS with the software in charge of controlling it through high fidelity CPSs models. In occasions, since CPSs involve multiple engineering domains (e.g., software engineering, electrical engineering, mechanical engineering), co-simulation tools are required. However, testing activities are usually time-consuming, which is, to a large extent, caused due to the lack of test automation. The lack of test automation in CPSoS engineering is at several phases, such as, test generation, One such phase that makes it difficult to provide a high degree of test automation when testing CPSs and CPSoS is the lack of reusability and interoperability between test artifacts (e.g., test cases, test oracles) among the different test levels (i.e., MiL, SiL, HiL and Operation). While there are standards that have tried to enhance the interoperability of tests (e.g., ASAM-XiL), these have a strong focus and dependability on certain domain (i.e., for the case of ASAM-XiL, automotive domain). Besides, CPSoS phase new problems, such as testing at Operation-time.

In the first months of the ADEPTNESS project, an architecture based on microservices was designed to allow for the design-operation continuum engineering of the software development process. Such architecture was envisioned to allow for the reusability and interoperability of the different testing artifacts and to allow for the Testing as a Service (TaaS) paradigm in this context. Two core components were designed to allow this: (1) the validation orchestrator microservice and (2) the validation agent microservice. In addition, it is possible to need specific tools for the execution of tests. For instance, Orona, our industrial partner providing one of the use-cases to the ADEPTNESS project, required the Elevate tool to execute tests. Our microservices-based architecture was sought to support such domain-specific tools too.

In this deliverable we provide the internal details of the developed validation orchestrator and validation agents microservices that will allow for an agile verification and validation framework for the automated test execution of test cases for simulation-based testing of CPSoS. In addition, we provide the STELLIO data model of a validation plan to allow for Testing as a Service of CPSoS. Lastly, we give details of how the agile verification and validation framework has been instantiated within the use-cases of the project. Specifically, for Orona's case, we have instantiated it by developing the external tool microservice for the integration of Elevate, i.e., a domain-specific tool, with the ADEPTNESS ecosystem. On the other hand, for Bombardier's case, we have experimented with a specialized model-based test generation framework, both as a microservice accessible through cloud-based RESTful API as well as a stand-alone tool.

3. Background

3.1. Testing standards for CPSs and CPSoS

3.1.1. UTP standard

Object Management Group, Inc. (OMG) was established in 1989 and it is an open membership system, non-profit computer Industry Standards Association, responsible for formulating and maintaining interoperable computer industry specifications, portable and reusable enterprise applications in distributed heterogeneous environments. Members include information technology suppliers, end users, government agencies and academia [8].

In 2001, OMG's working group began to develop a UML configuration file dedicated to model-based testing, called UML Test Configuration File (UTP). It is a standardized language based on OMG's Unified Modeling Language (UML), which is used to design, visualize, specify, analyze, construct and record common and required artifacts in various test methods (especially model-based testing). The UML is a graphical language to support the design and development of complex object-oriented systems. UTP has the same important role for model-based testing methods as UML for model-driven system engineering.

UTP is part of the UML ecosystem, so it can be combined with other configuration files of the ecosystem to associate test-related artifacts with other related system artifacts, requirements, risks, use cases, business processes and system specifications. This enables requirements engineers, system engineers, and test engineers to bridge the communication gap between different engineering disciplines. Model-based test specifications expressed with the UML Testing Profile are independent of any methodology, domain, environment or type of system.

Later, the UTP Working Group agreed to consolidate UTP's experience and achievements to justify the transition from UTP 1.2 to subsequent specifications. These efforts led to the Request for Information of UML Test Profile 2 (UTP 2), whose purpose is to find and collect the shortcomings of the current UTP and the most urgent requirements of the OMG and model-based testing communities for subsequent specifications.

Some of the main issues in the Request For Information response are that UTP 2 should be the ability to design test models of different test levels, testing to address non-functional requirements, integrate with SysML to achieve requirement traceability and test generation, comply with relevant industry standards and the ability to reuse test logs for further test evaluation.

The target audience of UML Testing Profile is users who can read model-based test specifications Represented in the UML test profile model, including testing tools, system/software engineers, stakeholder, certification authorities test engineers, requirements engineers and domain experts.

There are diverse reasons why UTP should be used. Build a model-based test plan on the basis of the existing system model to reuse existing components, such as requirements, interface definitions, type definitions, etc. Requirements traceability within test specifications are very important for recording and evaluating test ranges and calculating other indicators, such as progress reports. The inherent traceability is provided by the underlying UML function. Provide necessary information related to test plan optimization because test plan optimization can be based on priority, risk-related information, cost, etc. The test environment includes hardware, instruments, simulators, and software tools and other supporting elements required for testing. Moreover, the specification of test cases is the basic task of each test process, the purpose is to evaluate the test quality of the project and verify whether the test project meets its specifications. Model the deployment specification of specific test artifacts to rely on UML deployment standardized functions can complete the actual deployment of the test system in a model-based manner. Additionally, specify the design and configuration of the test system because the design of the test system includes identifying the test item (also called the system under test or abbreviated as SUT), its boundary, derivation of test components and identification of communication channels between interconnections that can exchange data. Data modeling includes data values, which are used both as incentives for test items and as incentives for responses such as test oracles. The expected response of the test item, this is, test oracle. Finally, record the execution results of the test cases to correlate the test

cases with the actual results of the execution, perform further analysis in exactly the same model, calculate specific indicators, etc [8].

3.1.2. ASAM-XiL

ASAM-XIL is an API standard that allows the interoperable execution of tests among test automation tools, simulation tools, test benches and test levels. For instance, the idea is that no manual intervention is required when sharing tests among different simulation tools (e.g., Labview and simulink), test levels (e.g., MiL and HiL) and test benches (e.g., speedgoat and CompactRIO). One such advantage is that users can freely choose testing products based on their needs and requirements, with little effort to share the tests among them. Such advantages are the results of two components within the ASAM-XIL standards:

1. The framework, which aims at providing data measuring and mapping functionality that is independent from the used test bench hardware and software
2. The test bench, which has as a goal to provide port-based communication mechanisms to the simulation model, the ECU, the electrical subsystem and the ECU network.

The standard was intended originally for the automotive domain, although it can be used in any domain. To demonstrate how to implement and use the XIL interfaces and the XIL-API, the standard comes with a C# implementation and application examples.

3.1.1. TTCN-3

TTCN-3 *(Testing and Test Control Notation version 3)* is an, ETSI standardized, modular testing language specifically designed for testing. TTCN-3 addresses not only the language for specifying tests but also the interfaces that control and adapt a test to any given environment. It provides a syntax for the definition of tests independent of any application domain. The abstract nature of a TTCN-3 test system makes it possible to adapt a test system to any test environment. This separation significantly reduces the effort required for maintenance allows experts to concentrate on what to test and not on how. The standardization of TTCN-3 means that users are not forced to rely on the use of one proprietary tool.

The standardized interfaces make it possible for a suite of TTCN-3 tests to be adapted to execute in a range of different environments, platforms or operating systems including HTTP API tests. Writing HTTP API tests requires the development of a System Adaptor (SA) for HTTP protocol which may be complex along with the corresponding Coding and Decoding (CD) for adapting message-based communication to and from the SUT, and SUT action operations and encoding and decoding data associated with message-based communication within the TE according to the following TTCN3 system architecture:

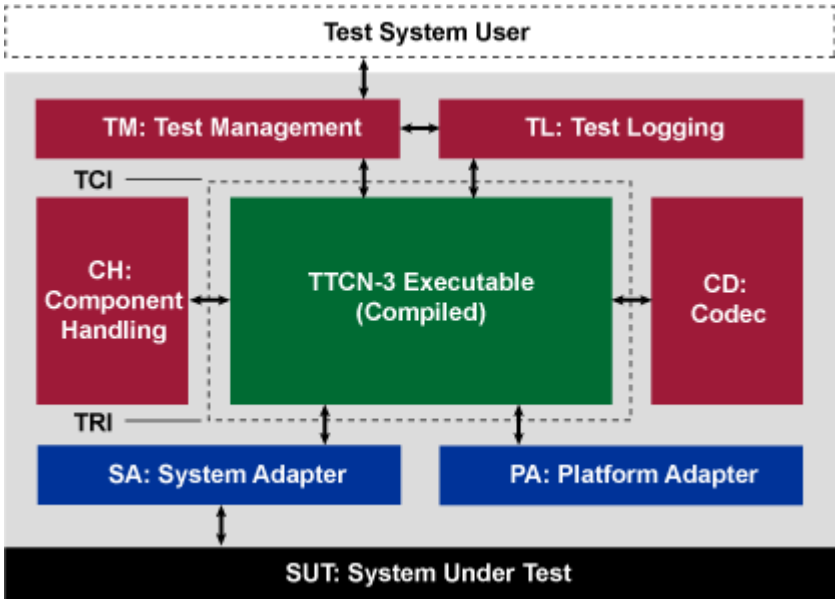

FIGURE 1 TTCN-3 ARCHITECTURE

The main interfaces are :

- TTCN-3 Control Interface (TCI): includes Test Management (TM), Test Logging (TL), Coding and Decoding (CD) and Component Handling (CH)

- TTCN-3 Runtime Interface (TRI): includes System Adaptor (SA) and Platform Adaptor (PA)

The TTCN-3 language is strongly adapted to test systems set-up. However, it brings limited developer experience as it requires for developers to gain a specific TTCN-3 coding capability. In addition, the software support is limited with mostly TT Workbench (a commercial product from Spirent) and Titan an Open Source tools which does not cover the whole TTCN-3 TRI &TCI specification.

3.2.Testing of embedded systems at different levels

Embedded systems are widely used to facilitate the real life operations in complex systems such as controlling the operational functions in vehicles, trains and planes etc. An embedded system is a collection of subsystems composed of hardware and software components. It communicates with sensors to collect data in form of input signals and perform the operations in real environment using actuators by sending output signals. All the hardware in an embedded system are controlled by an embedded software stored in the Non-Volatile Memory (NVM), flash card or hard disk [14]. Embedded systems also contain some safety critical systems which can cause catastrophic events in case of unexpected behaviour. Hence, testing an embedded system rigorously to ensure proper working of the system is of great importance to avoid such incidents.

3.2.1. Model-based testing (MBT) and GraphWalker (GW)

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 SUT. It supports multiple algorithm and coverage criteria to traverse through the model elements (i.e. states and transition) and generates test sequences known as abstract test cases. The abstract test cases contain the abstract input and output operations as well as their corresponding values. The abstract test cases are then transformed into concrete and precise actions to validate a system. MBT has shown to be an efficient and effective technique in finding faults in a system [23].

Multiple model-based tools are available for commercial and academic purposes based on modelling notations and test case generation methods. GraphWalker is one of such MBT tools based on a finite state machine. It generates abstract test cases in the form of test sequences by traversing through nodes and edges of FSM. 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.

3.2.2. Behaviour driven development and Gherkin

Behaviour Driven Development (BDD) is an agile methodology to practice automated testing. It has been used to design and execute test cases in order to verify and validate a System Under Test (SUT). It provides a common view of domain and collaborative work between business partners, domain experts, testers, and developers to ratify a concrete understanding of the system and its behavior [16]. By following a BDD approach, each functionality of the system as well as designed test cases are documented in natural language in an iterative way. Based on BDD, test cases can be developed at initial level to test the system and each test case can be represented as a set of instructions which are independent of the testing tool. One way of documenting BDD requirements and test cases is Gherkin (*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* [17].

3.2.3. Automating MBT using Gherkin

Towards the automation of the testing process, these test cases can be generated using Model based Testing (MBT). MBT is an automated testing technique which generates the test artefacts based on a model representing the SUT. After specifying the requirements in Gherkin format, a tester can model the system using certain modelling notations i.e. Unified Modelling Language (UML), Finite State Machine (FSM) etc. and can generate the abstract test cases by traversing through the model elements (i.e. states and transitions). To facilitate the modelling efforts, a Domain Specific Language (DSL) could be used to define the requirements in a well-defined format and to extract certain information (i.e. states, transitions, guard variables and their values) from it. A DSL contains an abstract syntax based on 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 [19].

3.3. Test execution frameworks and test levels

At design-time, the execution of test cases in CPSs is typically carried out at different test levels, coined as Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Processor-in-the-Loop (PiL) and Hardware-in-the-Loop (HiL),

- MiL is the first simulation stage test level; when engineers build the model of the CPS, this test level is employed, where the both the software as well as the physical layer (also called as the plant model) are models [20]. The execution of tests at this level are performed using high precision, as the computations use floating-point arithmetic to obtain the simulation results as reference for the following testing stages. Functional requirements along with floating-point arithmetic computations are tested at this stage.
- SiL is the second simulation stage test level; when the test results at MiL are considered satisfactory, the software model at MiL is replaced with an executable object code [20]. At the SiL test level, fixed-point computations are employed, which represent the computations done in the final target system [20]. Functional requirements along with fixed-point arithmetic computations are tested at this stage.
- PiL is the third simulation stage test level, and it is the first test level where hardware of the real CPS is used. It uses real object code, cross-compiled and executed on the real target processor, which communicates with the simulation tool in the computer to obtain data from the simulated physical part of the CPS [20]. At this test level, the main objective is to detect faults introduced by the code compilation tool.
- HiL is the last simulation stage test level. The embedded software is integrated with the Electronic Control Unit (ECU) as well as the real-time infrastructure (e.g., drivers, Real-time Operating system) [20]. On the other hand, the physical

layer of the CPS is encapsulated in an embedded device (e.g., FPGA) [20]. The simulation at this stage is performed at real-time. Both functional as well as non-functional requirements (e.g., performance and timing constraints) are tested at this test level.

Domain-specific frameworks

For testing the dispatching algorithms of elevators, Orona does not use the MiL test level because their controllers are directly programmed in C/C++. Subsequently, the verification and validation phases begin at the SiL test level.At this level, a domain-specific tool named Elevate [ref] is used. This tool permits the traffic analysis of a system of elevators. It allows users to specify the characteristics of a system of elevators, including, but not limited to, number of elevators of the system, dynamic characteristics of each of the elevators (i.e., speed, acceleration yerk), maximum number of passengers and load that it can handle, number of floors, etc. All this information is named as the building installation installation, it is stored as *.xml file and is typically provided as part of a test case. The other part of a test case is related to the passenger list. This is stored in a *.txt file and provides information of the passenger flow in a building. For each passenger, information such as the arrival time, arrival floor, destination floor, and weight of the passenger is provided in a file. The simulator is connected with Orona's traffic algorith through a wrapper that implements the interface between both tools. With such information (i.e., the building information and the passenger file), a test is executed through simulation-based testing. When the simulation has finished, elevate returns a set of metrics that can be used to measure the performance of the dispatching algorithm. For instance, in our preliminary experiment [ref], we used the following metrics:

- Average Waiting Time (AWT), which refers to the average time the passengers waited from the time they arrive to the calling floor until the elevator arrived and opened the door. According to the study of Barney and Lutfi, for elevator passengers AWT is the most important measure to determine whether a system of elevators performs well or not [21].
- Total Distance (TD), which refers to the sum of the distances travelled by all the elevators of the building, measured in floors [22]. This metric is relevant because an unexpected value may reveal faulty behaviours like consistently not assigning elevators close to the landing calls or unnecessarily dispatching multiple elevators to a single call.
- Total Movements (TM), which refers to the total number of engine start-ups of all the elevators of the building. Similar to the previous metric, this QoS measure may reveal inefficient dispatching or faulty behaviours.

When the SiL phase has finished, the following level refers to the HiL. At this level, Orona has an hybrid real/virtual infrastructure composed by the following elements:

- Controllers: Real software controllers of the Elevators. The controllers coordinate the vertical movement of the lift and the horizontal (doors opening and closing)
- Real user interfaces: call devices and floor signaling
- Communication buses: real Horizontal and Vertical CAN buses. Horizontal CAN bus communicates the controllers and other elements shared between controllers (for example, the traffic master, call buttons shared between controllers, etc). Vertical CAN bus the controller with the rest of devices of a lift (for instance call buttons associated to a single lift.
- Virtual simulation of the elevators. Movement of the lifts are simulated in an proprietary simulator

FIGURE 2 HIL TEST BENCH IN ORONA

3.3.1. Test levels at TCMS

There are three distinct test levels of TCMS at Alstom Transport: TCMS SW Test (at SW application level), TCMS System Integration Test and TCMS System Test.

3.3.1.1. *TCMS SW Test at SW application level*

The SW applications are composed from SW components with a clearly defined function and widely independent from other SW components. This means that they are sufficiently structured to adequately check the specific behavior of a component even if it is done at SW application level.

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

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

- integration of the SW components into the SW application
- SW application regarding the SW Requirement Specification

For that purpose, the TCMS SW test shall check at least the following:

- correct implementation of the functions and their intent as described by the SW requirements, including robustness regarding possible faults even if not directly described by the SW requirements (e.g. incorrect sequence or combinations of signals, incorrect pulse duration)
- disabled SW parts due to configurable SW don't influence the remaining SW parts
- correct implementation of the interfaces as described by the Interface Signal List (logical interface signals)
- correct handling of signals between SW components integrated into the SW application, at least for signals with complex sequences (timing of one or multiple signal) or complex compound signals (build by multiple elementary signals)

The TCMS SW Test at SW application level shall be executed preferably at the Virtual CCU, which provides an emulated environment which runs at a PC and which is compliant to the HW and firmware of the corresponding CCU type. Building of a Virtual CCU is done by utilizing the original SW of the SW application without any modification. If it is not possible to use a virtual CCU due to a strong dependency with the real firmware or HW, the test shall be done using the real TCMS device.

3.3.1.2. TCMS System Integration Test

The main target of TCMS System Integration Testing is to ensure the correct of

- integration of the SW application onto the target TCMS devices
- configuration of the TCMS devices
- interworking of the TCMS devices within the TCMS System
- behaviour in case of outage or malfunction of TCMS devices and interfaces
- detection of resource issues
- principal correctness of the functions of the TCMS system after changes

For that purpose, the TCMS System Integration Test shall check at least the following as part of the SW/HW and system integration:

- correct startup of the TCMS devices without any anomaly
- all interfaces of the TCMS devices (TCMS internal and external) are up and running (physical and logical availability, e.g. life sign, SDT)
- correct behavior of all fundamental functions of the TCMS System (integration regression test)

The TCMS Integration Test shall be executed at the test rack by utilizing the target TCMS devices and bus systems. The test rack also provides the environment for the test execution (stimulation, observation of signals) and the Environment Model which simulates the environment of the TCMS System at the vehicle to the extent as necessary by the test.

3.3.1.3. TCMS System Test

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

For that purpose, the TCMS System Test shall check at least the following:

- correct implementation of the functions and their intent as described by the system requirements, including robustness regarding possible faults even if not directly described by the system requirements (e.g. incorrect sequence or combinations of signals, incorrect pulse duration)
- disabled functions due to configurable SW don't influence the remaining functions
- correct implementation of the interfaces as described by the Interface Signal List (logical interface signals) and technical detailed by the Interface Control Documents (physical interface signals)
- correct implementation of the handling of safety relevant input and output according to the TCMS System Architecture & Design Specification (redundant/anti-valent input, validity handling of signals)
- response time of the TCMS System regarding the fulfillment of the performance demands of the system requirements
- correct interworking of TCMS Systems between different vehicles (multiple-unit operation)

The TCMS System Test shall be executed at the test rack by utilizing the target TCMS devices and bus systems.

3.3.2. Generalistic testing frameworks for CPSoS

Typically, CPSs are tested by using generalistic testing frameworks. MATLAB/Simulink is the predominant CPSs modeling tool in industry, especially due to its capability for easy integration between embedded software and physical processes. Another advantage of this tool includes the possibility to automatically generate C/C++ code compliant with several standards (e.g ISO26262), which makes it attractive for those industries using it. Due to its wide adoption by industrial practitioners, and the importance of testing in such contexts, Simulink is equipped with add-ons targeting the testing of Simulation models (e.g., Simulink Design Verifier, Simulink Test Manager, etc.). The use of such add-ons allows for different testing options, such as measuring the achieved white-box coverage or capability to automatically generate test cases.

Robot Framework is a generic open source automation framework. It can be used for test automation and robotic process automation (RPA). It is an open source software released under Apache License 2.0, and most of the libraries and tools in the ecosystem are also open source.

The Robot Framework allows testers to easily write automated test scripts using a Human-readable, keyword-driven syntax; also supporting Gherkin. Test cases are created using test case tables in test case files. Such a file automatically creates a test suite from all the test cases it contains. Test case files can be organized into directories, and these directories create higher-level test suites. These directories can then be placed into other directories creating an even higher-level suite.

Robot Framework has its own editor (RIDE) but there are plugins/extension for most common editors as well (Visual Studio Code, Eclipse, IntelliJ, etc.):

- IntelliJ
- Eclipse:
- VS Code

Since Robot Framework is built in Python language, it can be used and further extended using Python or Java by most of the developers. It can also easily be launched on the command line or via some shell scripting.

As interfaces, these can be extended via plugins. As example Robot provides some libraries that provide a DSL for testing HTTP APIs:

- RequestsLibrary
- RESTinstance
- HTTPRequestLibrary

And has an MQTT library. It allows to subscribe to a topic and validate a message is received (at least). However, it needs an MQTT broker to be launched as part of the test suite.

Robot Framework generates an HTML report. It has statistics based on tags and executed test suites, as well as a list of all executed test cases. It allows the possibility to navigate to more detailed information by adding links to the log files. The overall test execution status is seen by the report's background color: green if all critical tests pass, and bright red otherwise. Results can be presented in a Grafana dashboard.

The tool execution can be configured in CI tools such as Gitlab and Jenkins. In Jenkins, there is a plugin to collect and publish Robot Framework execution results. Robot Framework is actively supported, with many industry-leading companies using it in their software development.

4. Verification and Validation Test Benches for CPSoS

4.1. Reused concepts

UTP standard has been used to define the concepts necessary for developing JSON files that verify and validate the testing plan for CPSs. The elements needed are the following:

- ValidationPlan: The validation is usually the document that contains the planning for the validation system. The main element is ValidationPlan for this reason.
- ValidationPlan_Id: each ValidationPlan contains a unique identifier.
- TestCases: The UTP 2 document defines Test Case as a procedure that includes a set of preconditions, inputs and expected results, developed to drive the examination of a test item with respect to some test objectives. In the JSON each test case contains this information. The validation might contain more than one Test Case.
- TestCase_Id: every single test case is identified by a unique identifier.
- TestCase_Priority: each test case has a different priority and this must be included in the JSON.
- TestCase_TestLevel: there are different levels in each test case (HiL, SiL in our case)
- SUT: It provides information about the version of the system that is being tested.
- Version_Id: every single SUT is identified by an unique identifier.
- Configuration_Id: every single SUT contains a document that provides information about the initial configuration.
- TestArtifacts: is defined as an object produced or modified during the execution of a process. It is composed of different test inputs.
- TestDesingInput_Id: each Test Input contains a unique identifier that provides information about the name of the document that will be used with the input values.
- TestDesingInput_Type: each test input file could be in a file with different formats. This value provides information about the format of the document.
- Oracles: is a mechanism, different from the program itself, that can be used to test verify the correct functionality of a system. We may have more than one oracle.
- Oracle_Id: every single oracle is identified by an unique identifier.
- Oracle_Type: In the ADEPTNESS framework, we will have three types of oracles: Assertion, Metamorphic and Machine Learning-based test oracles
- Oracle_Criticity: The aim of this field is to inform the importance of the test failure. If the test is important, the criticity will be high and if it fails, all the test cases fail.
- Oracle_Inputs: it contains the information about the input values needed to execute the oracle.
- Oracle_Verdict: it informs about the final decision about the test, if it fails or passes.

4.2. Data model for CPSs

The ADEPTNESS data layer has been chosen to be based on NGSI-LD (see D1.1), an ETSI specification for cross-domain data exchanges based on linked data principles and serialized using JSON-LD.

JSON-LD is an extension of JSON, it is a standard data representation format to solve ambiguity when expressing data related to JSON format. The data is structured in a machine-readable format. This way, it is easy to compare all data attributes even if they come from different data sources. For example, if two data entities have the attribute name, computers cannot know that both may refer to a "name of something" in the same sense (rather than a user name or a last name or something else). Therefore, URLs and data models are used to remove ambiguity by allowing attributes to have both a short form (such as name) and a fully specified long form (such as a URL like http://schema.org/name) that easily provide the ability for machines to discover which attributes have a common meaning in a data structure.

In addition, JSON-LD introduces the concept of the @context element. The @context element provides additional information to entities, which allows context information to be interpreted by machines (e.g., unit of measure, concept definition, mandatory, optional attributes, etc.). Finally, the JSON-LD specification allows us to define a unique @type. The type allows the association of a well-defined data model with the data itself. In Linked Data, it is common to specify the type of a graph node. This can be done based on the properties used in a given object or the property for which a node is a value. For example, in the schema vocabulary, the diameter property is associated with a valve. Therefore, one might think that if a node object contains the property diameter, that type is a valve; making this explicit with @type helps clarify the association.

NGSI-LD is the evolution of the NGSI v2 information model, which has been modified to improve support for linked data (entity relationships), property graphs, and semantics (exploiting the capabilities offered by JSON-LD). NGSI-LD was standardized as part of the ETSI ISG CIM initiative and the updated specification was labelled NGSI-LD. The main constructs of NGSI-LD are entities, properties and relationships. NGSI-LD entities can be the subject of properties or relationships. Relationships allow for associations between entities using linked data. In addition, these entities relationships include a more complex data model and more rigid usage definitions that lead to a navigable knowledge graph.

Again, entities are the core elements. Every entity must use a unique ID, represented in the form of a URI (often a URN). A URI is also a type and is used to define the structure of the data to be held. This URI must correspond to a well-defined data model.

An entity can have a relationship with a property. Ideally, the name of each property should also be a well-defined URI. Properties have a value that reflects the state of the property. Finally, a property itself may have further properties (a.k.a. properties of properties). In such a case, the property reflects more information about itself. Properties and relations may also contain linked embedded structures (properties of properties, properties of relations, relations of properties, and even relations of relations).

Using these concepts, it is thus possible to build a data model able to connect the different steps of the ADEPTNESS workflow so operations get data driven (e.g. the existence of a testing plan trigger its execution). The central component in any related NGSI-LD deployment is the NGSI-LD broker. The STELLIO open-source broker, under development by EGM and hosted as a FIWARE enabler has been retained as central broker to deploy ADEPTNESS as a Test as a Service (TaaS) environment. This broker connects data producing and data consuming micro services over RESTful http or MQTT. As represented schematically in the next figure, a front-end (web GUI) allows a tesing team to easily define all parameters related to a test deployment. These include :

- The deployment information, to be used by the deployment orchestrator, contains all information to deploy software containers or scripts over the execution environment
- The monitoring information contains all configuration parameters of the monitoring agent, connected to the execution environment, so appropriate metrics get collected
- The validation information contains the list of test cases, grouped into test suite. These test suites also include the test assertions as well as oracles definitions.
- Together with the definition of the SUT this allows to create a test campaign which can be either manually triggered from the TaaS front-end or though back-end events such as a new SUT software update being committed into a Git repository.
- During the test execution, test results are being fed back to the STELLIO broker which then acts as a logger whose information can be consumed by e.g. TaaS monitoring interface but other plugins such as an OSLC one.

This approach allows dynamic update of the configuration information and is thus compatible with both offline and online testing approaches.

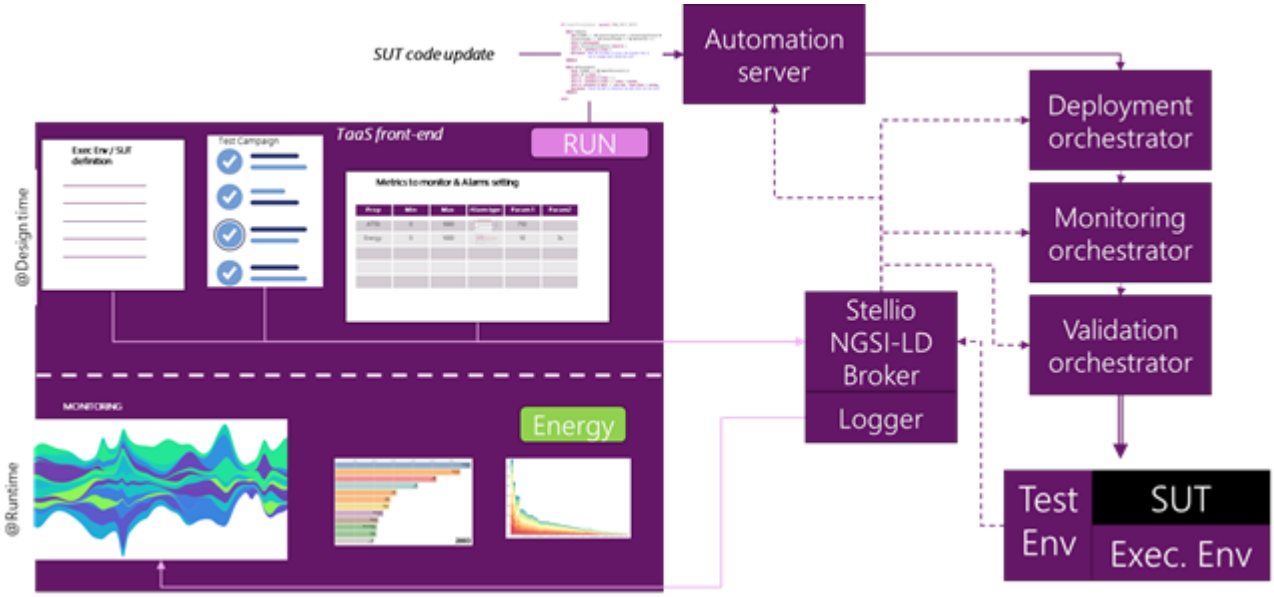

FIGURE 3 DATA FLOW WITHIN ADEPTNESS

Defining a complete NGSI-LD data model is a long task which needs to be improved over time. This activity has been initiated within WP2 and will be improved and generalised within WP5 and WP6.

To illustrate the modelling, the next figure describes the information elements required for a Monitoring Agent in charge of collecting information from different sensors. A test campaign can engage several monitoring agents. Each monitoring agent, modelled as a NGSI-LD entity, is connected to one specific protocol (CANbus, Modbus, OPC-UA, UART, etc.) also modelled as an entity with its own connectivity parameters. Each MonitorAgent can collect information from several sensors connected to that protocol bus. Each sensor has its own information : connection parameters, a sampling rate, the unit code of the measured metrics as well as a basename. The basename is used by a component: the MQTT bridge which is collecting data send by the MonitoringAgent over its MQTT client using the senML format. Finally, it is a possibility to define some alerting rules over the measured value.

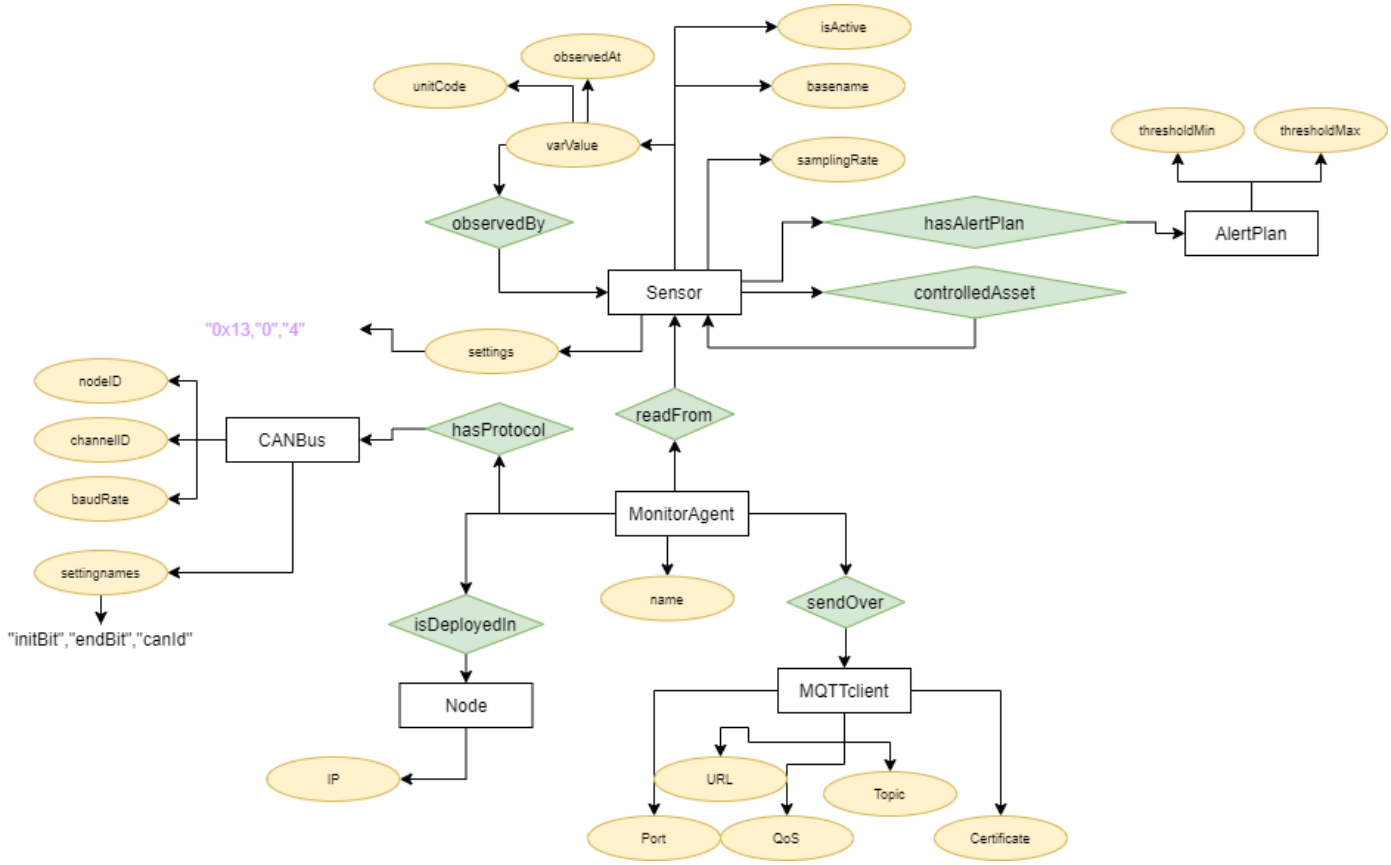

FIGURE 4 MONITORING AGENT MODEL

The next sections apply the principles over the different stages of validation.

4.3. Approaches to the automated testing of Cyber-Physical Systems of Systems

While analyzing the use-cases provided by the industrial partners from ADEPTNESS we figured out that two main approaches can be followed when validating the software of CPSoS. On the one hand, component level validation, where the test bench needs to interact directly with the component/software under test without considering neither the physical layer of the CPS nor the environment at which this operates. The typical test cases at this level are reactive ones, where the test bench stimulates the component and waits for its response, to further stimulate it based on the response given by the component. Given the nature of this kind of validation, the technique of Model-Based Testing (MBT) is appropriate to automatically generate test cases. Furthermore, script-based test cases are common when testing such CPSoS at such level (e.g., C# scripts). This is the case used in the railway use-case in ADEPTNESS. An example of such a test case can be found in Figure 5.

```
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 5 AN EXAMPLE OF GENERATED TEST STEPS IN C# SCRIPT

An example of a generated test script is shown in Figure 5 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.

On the other hand, CPS-level validation, where the software components are already integrated both with the physical layer of the CPSoS as well as with the environment at which the CPSoS operates. The typical test cases at this level are already predefined test scenarios, which are obtained by the test bench and convert them into certain stimulations, usually without considering the CPSoS status. This is the case for the elevation use-case in ADEPTNESS. In this context, a test case is a passenger file traveling over a building and a file indicating the building characteristics (e.g., number of floors, number of elevators, etc.).

5. Development of the Agile Testing Framework for the ADEPTNESS Ecosystem

In this section we explain a generic Agile Testing Framework developed for the ADEPTNESS framework. The framework implements two microservices of the ADEPTNESS Architecture (validation orchestrator and validation agent) proposed in the first stage of the project and which can be found in Deliverable 1.2.

5.1. Validation orchestrator

5.1.1. Introduction

The validation orchestrator microservice is in charge of managing the whole lifecycle of a validation plan in the context of the ADEPTNESS ecosystem. Developed in Python, the validation orchestrator is capable of running in the major platforms where the Python interpreter or Docker is supported. All the communications are made exclusively through APIs defined on the ADEPTNESS architecture, meaning that the orchestrator is also agnostic to the use-case. The microservice exposes a REST API that enables the management of validation plans and their execution, and publishes the validation plan status and verdicts through MQTT. The validation orchestrator interacts with validation agents deployed in edge validation platforms though their REST API, and listens for their verdict and status changes through MQTT.

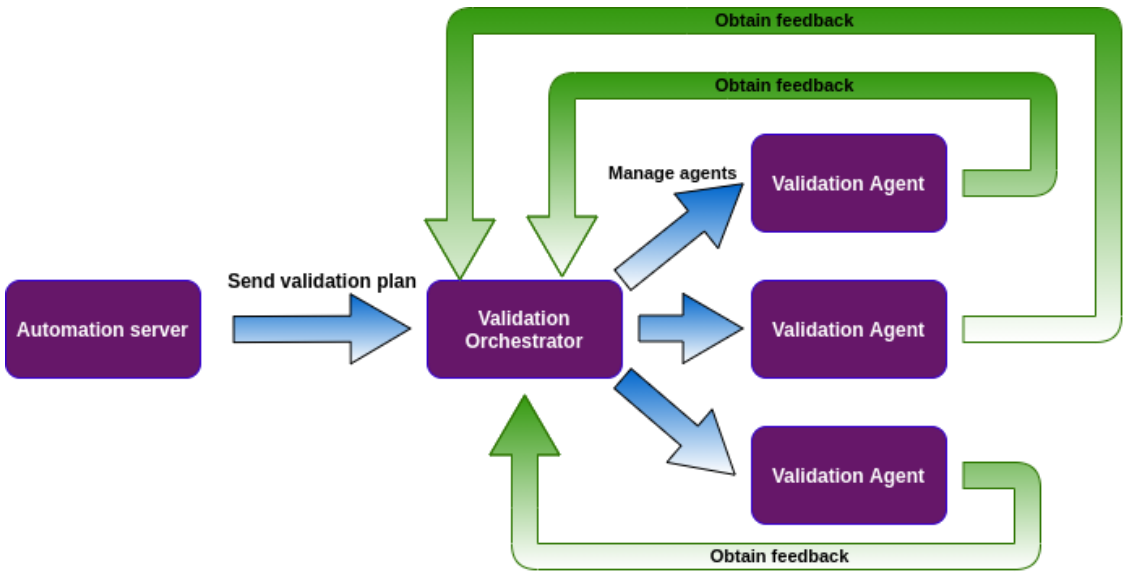

FIGURE 6 VALIDATION ORCHESTRATOR SCHEMA

On request by the automation server, the validation orchestrator receives a validation plan containing all the required information for performing a validation procedure via a REST API call. The validation orchestrator interprets the validation plan, requesting additional information of the components involved in the validation if necessary, and generates sub plans that contain the validation procedures to be delegated to each of the validation agents.

Upon an evaluation start request sent by the automation server, the validation orchestrator sends the generated subplans to the corresponding validation agents through their REST API, delegating the coordination of the evaluation procedure specified in the sub plan to the validation agents.

After starting the evaluation on each of the validation agents, the orchestrator awaits for the completion of the evaluation, by awaiting for verdicts generated by the validation agents and published through MQTT, When verdicts are received, the validation orchestrator integrates these verdicts on the validation plan, running aggregation rules defined on the plan to determine the global verdict of the validation plan.

When a global verdict is achieved for the validation plan, the generated verdict is published through MQTT, and the validation plan status is marked as finished.

The automation agent periodically polls the validation orchestrator through the REST API for validation plan status changes. When the automation agent notices that the validation plan evaluation has finished, it collects the generated verdict through the REST API and depending on the verdict, determines if the pipeline has succeeded or not.

5.1.2 Architecture

The validation orchestrator consists in a Python microservice developed on top of the official ADEPTNESS Python microservice template. The microservice exposes REST and MQTT endpoints for managing the validation plan execution, publication of verdicts and execution status of the microservice.

5.1.2.1 REST and MQTT APIs

In addition to the standard APIs provided by all ADEPTNESS Microservices, the validation orchestrator provides REST endpoints for managing validation plans and their verdicts. The validation orchestrator exposes the "*adms/v1/validation-orchestrator/plan*" for obtaining or setting the validation plan via GET or POST calls respectively. The validation plan sent or obtained by the REST endpoints follows the JSON format specified for the interpreted validation plan, specified in section 4.

When generating verdicts for the validation plan, the orchestrator publishes the generated verdict via MQTT, and makes it available through a dedicated REST API endpoint. The orchestrator publishes the verdict through the "adms/v1/validation-orchestrator/<orchestrator_id>/verdict topic, and exposes the verdict on the "adms/v1/validation-orchestrator/plan/verdict" REST endpoint, both following the payload specification for the verdict defined on the ADEPTNESS interface specifications.

The validation orchestrator makes use of MQTT subscriptions to receive status and verdict updates from child validation agents.

5.1.2.2 Interaction with other components

The validation orchestrator has a hierarchical relationship with the automation server and the validation agent, being managed and reporting back to the automation server, and managing and receiving reports from validation agents.

When a validation pipeline is executed on the automation server, the automation server sends the corresponding validation plan to the validation orchestrator through the *"adms/v1/validation-orchestrator/plan"* endpoint, triggering the execution of the provided plan. The validation agent then periodically checks the verdict REST endpoint for changes in the verdict for the validation plan.

The received validation plan triggers the configuration of the required child validation agents to perform the validation procedure specified in the validation plan. For each of the validation agents defined in the validation plan, their respective configuration is sent through each agent's REST API endpoints and verdicts and status changes from the validation agent are received through MQTT subscriptions.

5.1.2.3 Internal architecture

The validation orchestrator is divided into four main blocks: The microservice core, the parser, the agent manager, and the aggregation rules package.

The microservice core, developed on top of the ADEPTNESS microservice template, is responsible for managing the interaction of the orchestrator with the other components in the ADEPTNESS ecosystem. The microservice core implements the REST API endpoints exposed by the orchestrator, and serves as the entrypoint for accessing the functionality provided by the orchestrator. As mentioned in sections 5.1.2.1 REST and MQTT APIs and 5.1.2.2 calls for managing the validation plan and the microservices status have been developed.

When a call is received on the validation plan setting endpoint, the microservice core extracts the JSON payload from the HTTP request, and passes it to the parser component of the microservice. The parser component is responsible for creating an internal representation of the elements represented in the validation plan, creating a hierarchy of Python classes that maintain the state of each of the elements in the validation plan. As the structure and elements of the validation plan are shared among the validation agent and orchestrator, the parser and the Python class hierarchy has been extracted to an external library, that is shared and used between the validation agent, orchestrator, or any other component that needs to make use of components represented in the validation plan. The extraction of the shared components to an external library reduces the risk of inconsistencies that could arise in the independent development of the different validation microservices. The library also simplified the development of components that could make use of the validation plan.

After parsing the plan, the parser lends the parsed plan to the validation agent manager. The validation agent manager is responsible for managing the validation agents that are required for the execution of the validation plan. Each of the validation agents is responsible for managing a test level on the edge.

From the main validation plan, the validation agent manager will extract the subset of elements that need to be managed by each of the validation agents, and create sub-plans to be sent to each of the validation agents. The validation agent manager will then perform the required REST API calls to configure and start the validation on the validation agents, and subscribe to the topics where the verdicts generated by each of the validation agents will be published.

When receiving a verdict on the MQTT subscriptions, the validation orchestrator determines the source agent for the verdict, stores the received verdict and performs an aggregation procedure for the verdicts. The aggregation procedures allow to group verdicts raised by lower level components, and generate verdicts for higher level validation elements. In the ADEPTNESS ecosystem, the evaluation starts from oracles, which generate verdicts on their own, and then are aggregated to generate a verdict common to the test case, continuing with the validation level, and finally a global verdict for the validation plan. Several strategies have been developed for aggregation:

- Default Aggregation: The default aggregation strategy consists in marking the global verdict as passed when all the aggregated verdicts have passed. Any failed verdict would make the aggregated verdict to fail.
- Percentage Aggregation: Marks the aggregated verdict as passed if a certain percentage of the aggregated verdicts have passed.
- Criticity Aggregation: Marks the aggregated verdict as passed if all the critical oracles have passed.

5.1.2.4 Deployment

The validation orchestrator is shipped as a docker container that is deployed on the cloud. A new docker image is automatically generated each time a new commit is uploaded to the code repository via Gitlab CI/CD pipelines. As part of this process, the validation orchestrator passes several QA procedures, by performing unit tests and automatically generating the documentation for the project with the Sphinx documentation generation tool.

A single instance of the validation orchestrator manages all the validation agents that are running on the edge. The validation orchestrator shares the same requirements as any base ADEPTNESS microservice, being able to be executed on any platform supporting Docker.

5.1.3 Test execution workflow

For the execution of all validation plans, the validation orchestrator follows a well defined execution workflow that includes the interpretation of the validation plan, the configuration of child components and recollection of verdicts raised by the validation agents. The following diagram shows a quick insight on this execution process:

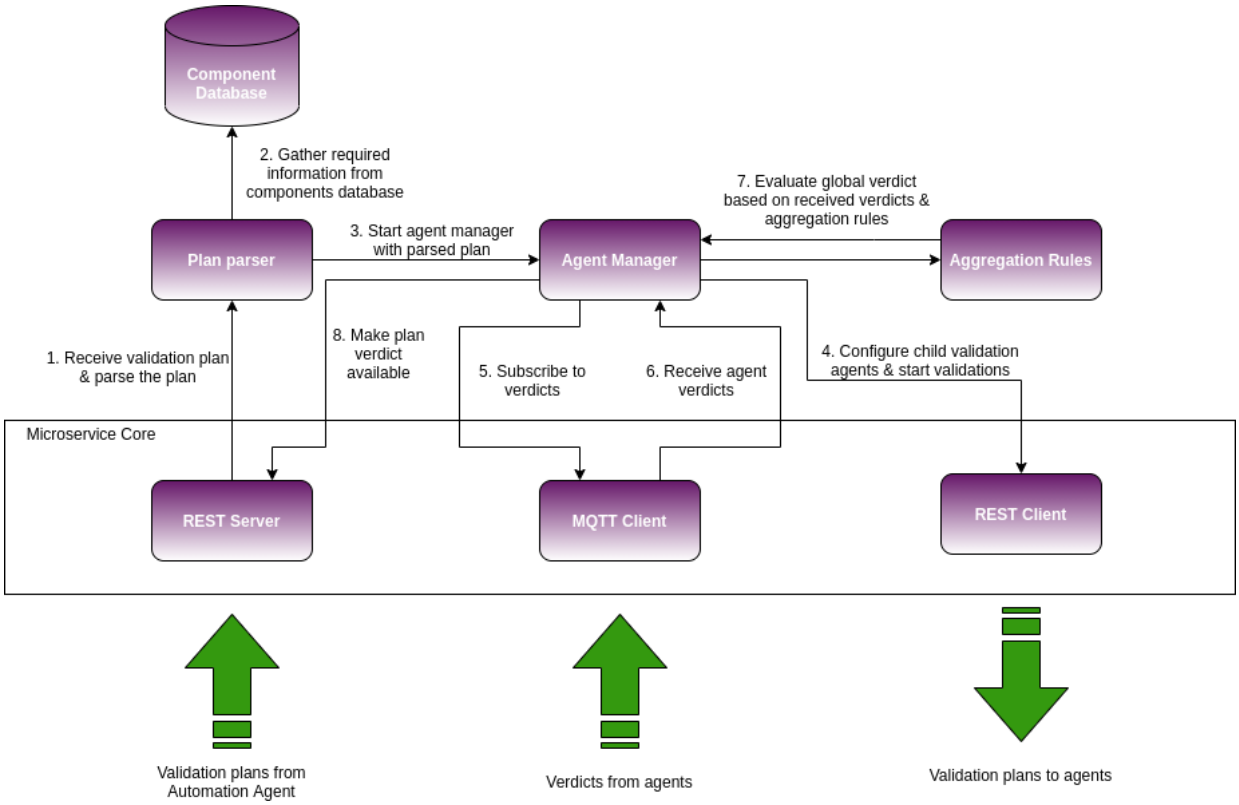

FIGURE 7 VALIDATION ORCHESTRATOR VALIDATION PLAN EXECUTION WORKFLOW

5.1.3.1 Parsing of the input validation plan

The validation orchestrator exposes a REST endpoint for receiving validation plans on the '/adms/v1/validation-orchestrator/plan' path. This endpoint receives a JSON formatted validation plan, and performs a parsing procedure for interpreting the contained validation elements.

The parser performs an initial interpretation of the validation plan and its contents. In the case of the validation orchestrator, this parsing involves determining which are the validation agents to be configured, and which is the aggregation procedure to be followed for combining the received verdicts and raising a global verdict for the validation plan. Given that the validation procedure is delegated to the validation agents, the validation orchestrator does not interpret the lower-level validation elements, leaving that interpretation to the parser on the validation agent. Thanks to this abstraction, the validation orchestrator is agnostic to changes or additions in the lower layers of the validation plan.

5.1.3.2 Obtaining additional configuration from external data sources

In some cases, it is possible that during the interpretation of the validation plan, additional information about the validation agents could be required. This information could include dynamically configured MQTT or REST endpoints or other configuration and runtime variables required to configure or communicate with the validation agents. The validation orchestrator has been developed with such cases in mind, and provides extendable abstract implementations in the validation plan parser for communication with additional data sources.

5.1.3.3 Agent configuration and plan execution

When all the information from the validation plan has been parsed and understood, the validation orchestrator determines the validation agents that will be used to run the validation, and starts the process of configuring the agents for the validation procedure to be managed by each of them. The configuration of the validation agents is made through the REST API endpoints exposed by the agents on the 'adms/v1/validation-agent/plan' path.

First, the subset of the validation plan corresponding to the agent is rendered as a JSON, by using similar procedures as the parser. as each of the classes representing a validation plan element provides methods to represent the data as a JSON object. Once the validation plan to be sent to the validation agent has been generated, the validation plan is sent to the agent through a REST API call to the 'adms/v1/validation-agent/plan' path. When receiving the plan, the validation agent answers with a 200 OK response to the validation orchestrator. Additionally, the validation orchestrator subscribes to the status changes and verdict publications of the validation agents published through the '/status' and '/verdict' topics of each of the agents.

This process is repeated for each of the validation agents, pushing the corresponding validation plan subset for each of the agents.

When all the validation agents have been configured, the process of starting the validation in each of the validation agents begins. Each of the validation agents exposes a REST API endpoint on the '/adms/v1/validation-agent/status' path for managing the execution of the validation plan. A call to the endpoint with a status change request, represented as {"status": "Running"} in JSON format is issued to the agents. When receiving the call, the agents start the management of the validation process in each of the levels.

5.1.3.4 Verdict arrival, aggregation and publication

The validation orchestrator awaits verdicts published by validation agents through MQTT. Upon reception of a verdict, the validation orchestrator checks the aggregation rules defined on the validation plan, and computes a global verdict according to the aggregation rules. If a global verdict is obtained from the aggregation, the global verdict is published through MQTT, on the '/verdict' topic of the orchestrator. Additionally, the verdict is also made available on the 'adms/v1/validation-orchestrator/plan/verdict' REST API endpoint.

5.1.4 Execution example in the Elevation use-case

In the elevation use case, a validation plan can extend over the SiL, HiL and operation test levels. The validation orchestrator will receive a JSON from the automation server containing all the aspects to be validated.

```
{
   "id": "urn:ngsi-ld:ValidationPlan:ValidationPlan01",
   "type":"ValidationPlan",
   "agents": {
       "urn:ngsi-ld:ValidationAgent:ValidationAgentSil01": {
           "endpoints": {
```

```
                "rest_endpoint": "http://127.0.0.1:5000",
                "mqtt_endpoint": "adms/v1/validation-agent/urn:ngsi-
ld:ValidationAgent:ValidationAgentSil01"
            },
            "tests": {
                        ...
                    },
                    "result": {
                        "aggregation_type": "DefaultAggregator"
                    }
                }
            },
            "result": {
                "aggregation_type": "DefaultAggregator"
            }
        },
        "urn:ngsi-ld:ValidationAgent:ValidationAgentHiL01": {
            "endpoints": {
                "rest_endpoint": "http://172.17.10.123:8201",
                "mqtt_endpoint": "adms/v1/validation-agent/urn:ngsi-
ld:ValidationAgent:ValidationAgentHiL01"
            },
            "tests" : {
                     ...
                    },
                    "result": {
                        "aggregation_type": "DefaultAggregator"
                    }
                }
            },
            "result": {
                "aggregation_type": "DefaultAggregator"
            }
        }
    },
    "result": {
        "aggregation_type": "DefaultAggregator"
    }
}
```


From the point of view of the validation orchestrator, three sections are included on the validation plan JSON:

- URN: An identifier that will uniquely identify the validation plan.
- Agents: The definition of each of the validation agents the validation orchestrator will manage. For each of the agents, a section containing the MQTT and REST endpoints required to communicate with the agent, and a tests section with the tests to be executed with the agent will be provided. The tests section will be forwarded to the validation agent as-is, abstracting the validation orchestrator for lower level validation structures.
- Result: Section that will determine the aggregation rules for the verdicts and their configuration, required to calculate a global verdict for the validation plan.

In the JSON example, definitions for two validation agents exist. The "urn:ngsi-ld:ValidationAgent:ValidationAgentSiL01" agent represents an example validation agent that is in charge of managing

the execution of the validation plan at SiL level, while the "urn:ngsi-ld:ValidationAgent:ValidationAgentHiL01" is the validation agent managing the execution of the validation plan in Orona's HiL test level.

During the configuration phase, the validation orchestrator will determine the address and port of each of the validation agents through the endpoints defined on the JSON and perform the configuration of each of the agents sequentially. After the configuration is performed, the same procedure will be followed for starting the validation in each of the agents.

As mentioned in previous sections, the validation orchestrator will then await for publication of verdicts and aggregate them to form a global verdict for the validation plan. The "DefaultAggregator" that is shown in the example JSON performs a simple aggregation that marks the global verdict as passed if none of the evaluated verdict has been marked as failed.

5.2. Validation agent

5.2.1. Introduction

The validation agent microservice is responsible for coordinating the execution of a validation plan and components in an evaluation platform (HiL, SiL, an installation in operation, etc.). As it happens with the validation orchestrator, the validation agent is also developed in Python, sharing the ability to be run on the majority of platforms. All the communications are also done through standard APIs defined on the ADEPTNESS architecture.

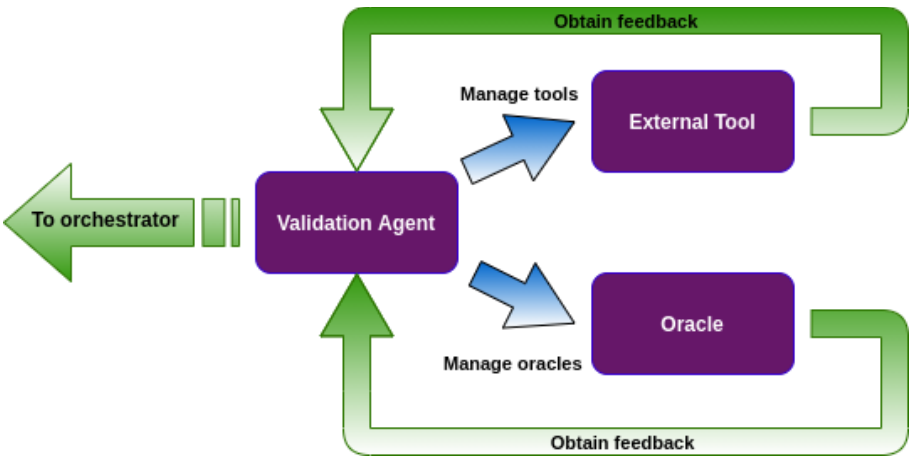

FIGURE 8 VALIDATION AGENT MICROSERVICE INTERACTION OVERVIEW

The agent will coordinate the Oracles, and external tools required to perform an evaluation, configuring them, managing the execution and collecting the generated test results. The generated results will then be aggregated and published, so that the validation orchestrator is able to collect the results.

Upon receiving a validation procedure to be executed on the evaluation platform managed by the validation agent, the agent interprets the validation plan and determines the components involved in the validation procedure. As mentioned before, these components can be Oracles, in charge of evaluating a set of conditions or aspects of the system and raising verdicts for these evaluations, or External Tools, which provide support to certain aspects of the validation process.

The agent configures each of the components with the settings specified in the validation plan through their REST APIs, and prepares them for running the validation procedure. These configurations include aspects such as mapping the inputs of the oracle to topics where the operational data will be published by monitors.

When receiving a start signal through their REST API, the agents forward this start signal to the managed components, and the evaluation procedure starts. The agent subscribes to the MQTT topics related to the status and verdict topics of the managed components, and awaits for publications in those topics.

When verdicts are received, the validation agent integrates them into the validation procedures through the defined verdict aggregation rules, publishing these aggregated verdicts through MQTT if necessary.

When the validation procedure is completed, the validation agent changes the validation status to complete, and publishes a verdict for the whole validation platform, which is collected and evaluated by the validation orchestrator.

5.2.2. Architecture

The validation agent is being developed on top of the ADEPTNESS Python Microservice template. This microservice exposes REST and MQTT APIs to communicate with the higher level validation orchestrator, and to manage child validation components: external tools and oracles. Internally, the validation agent consists in a parser component, responsible of parsing the validation plan subset to be executed in the test level managed by the validation agent, managers for both oracles and external tools, and aggregators for managing the aggregation of the collected verdicts into a single global verdict for the test level. All the components make use of the REST server and client and MQTT client provided by the microservice core to communicate with the rest of the components.

5.2.2.1. *REST and MQTT APIs*

Similarly as to the validation orchestrator, the validation agent exposes several endpoints for managing the validation plan and its execution, and in the case of the validation agent, communications over MQTT and REST are being used to manage child oracles and external tools.

The validation agent exposes a REST API endpoint at the 'adms/v1/validation-agent/plan' path for receiving validation plan subsets sent by the validation orchestrator or getting the current validation plan, through POST or GET calls respectively. These validation plan subsets are received in JSON format, and parsed by the parser component of the validation agent.

Communications with child elements are done through different API endpoints, both by REST and MQTT. The oracles exposes the 'adms/v2/oracle/inputs' REST API endpoint for binding the inputVariationPoints that the oracle supports with topics and sensors at runtime, the 'adms/v2/oracle/setup' REST API endpoint for additional configuration, and the 'adms/v2/oracle/status' endpoint for managing the execution status of the oracle. The validation agent issues POST calls to these endpoints to set up the oracle and start the validation procedure. The payloads sent to the REST API endpoints are in JSON format and follow the interface definition specified in deliverable D1.2. The validation agent subscribes to the 'adms/v2/oracle/<oracle_id>/verdict' MQTT topics to receive verdicts from child oracles, which are published by the oracles in JSON format following the interface specifications.

For managing the external tools, the validation agent makes use of the API endpoints defined in the external tool specification. The external tool microservice exposes the 'adms/v1/external-tool/process/<tool_type>/status' REST API endpoint for managing the execution of the tool. The validation agent performs POST requests to the status endpoint in order to control the execution of the process managed by the tool. In the case of the external tool microservice adapted to run the Elevate simulation tool, POST calls to the 'adms/v1/external-tool/process/elevate/status' endpoint manage the execution of Elevate, by commanding the external tool microservice to start the simulation procedure or

to stop it. Additionally, the validation agent subscribes to the 'adms/v1/external-tool/process/<tool_type>/status' MQTT topic to receive execution status updates from the external tool and to act accordingly.

When a global verdict for the validation plan in execution is obtained, the verdict is published through the 'adms/v1/validation-agent/<agent-id>/verdict' MQTT topic, and made available on the 'adms/v1/validation-agent/plan/verdict' REST API endpoint over GET requests. The execution status is also made available through the 'adms/v1/validation-agent/status' REST API endpoint. The validation orchestrator makes use of these endpoints for collecting the verdicts for the different test levels that will be used to raise a global verdict for the validation plan.

5.2.2.2. Internal architecture

The validation agent is divided into five main blocks: (1) the microservice core, (2) the parser, (3) the oracle manager, (4) the external tool manager and (5) the aggregation rules package.

The ***microservice core*** is developed by making use of the APIs exposed by the microservice template, building the REST API endpoints mentioned on section REST and MQTT APIs on top of the microservice template. As mentioned in the previous section, calls for managing the execution of the validation plan and the validation agent and retrieval of plan verdicts have been developed. The callbacks for each of the REST API endpoints interact with the parser and managers to implement the evaluation and execution logic.

The ***parser*** is responsible for interpreting the validation plans received in JSON format from the REST API endpoint, transforming the input payload to Python classes that maintain all the attributes specified in the validation plan and contain logic to transform the structures to and from JSON format, to be transferred to the child components. To maintain consistency between the structures and JSON format in the validation orchestrator and the validation agent, all the structures are defined in an external library shared between the validation orchestrator and validation agent. Thanks to the definition of a shared library, the validation orchestrator and agent are less prone to integration issues.

The parser expects a JSON structure containing the URN for the parent validation plan, aggregation section and a tests section containing each of the test cases to be run. For each of the test cases, a System Under Test (SUT) section, an oracles section, external tools section, and aggregation sections will exist.

```
{
   "id": "urn:ngsi-ld:ValidationPlan:ValidationPlan01",
   "type": "ValidationPlan"
   "tests": {
     "urn:ngsi-ld:TestCase:TestCase01": {
       "sut": {
       },
       "external_tools": {
           ...
       },
       "oracles": {
           ...
       },
       "result": {
         "aggregation_type": "DefaultAggregator"
       }
     },
```

```
      "urn:ngsi-ld:TestCase:TestCase02": {
          ...
      }
    },
    "result": {
      "aggregation_type": "DefaultAggregator"
    }
  }
```

The *oracle manager* is responsible for managing the child oracles, their execution and the recollection of verdicts raised during the evaluation. Upon plan receival, the oracle manager extracts the oracle structures from the validation plan. Each of the oracle structures contain a section defining the MQTT and REST API endpoints of the oracle, a inputs structure that contains the binding of each of the inputVariationPoints (supported inputs) on the oracle to monitors and sensors and an additional oracle settings structure containing additional settings for the oracle. For each of the inputVariationPoints, the oracle structure contains the monitor, sensorGroup and sensor where the data will be published.

```
"oracles": {
    "urn:ngsi-ld:Oracle:LiftPositionOracle": {
        "endpoints": {
            "rest_endpoint": "http://liftpositionoracle01:49981",
            "mqtt_endpoint": "mqtt://adms/v2/oracles/urn:ngsi-
ld:Oracle:LiftPositionOracle"
        },
        "inputs": {
            "LiftPosition": {
                "monitor": "urn:ngsi-ld:MonitorAgent:MonitorCAN01",
                "sensor_group": "urn:ngis-ld:SensorGroup:MonitorCAN01_Group01h",
                "sensor": "urn:ngsi-ld:Sensor:Lift1Position"
             }
        },
        "oracle_settings": {
        }
    }
}
```

For each of the oracles, the oracle manager performs POST calls to the 'adms/v2/oracle/inputs' to configure the inputVariationPoints of the oracle and POST calls to the 'adms/v2/oracle/setup' endpoint if the oracle contains the oracle_settings structure, passing its contents as-is. The manager then starts the validation process on the oracle, changing the execution status of the oracle by calling the 'adms/v2/oracle/status' through a POST call.

Additionally, the oracle manager subscribes to the verdict MQTT topic of each of the oracles and awaits verdicts raised by the oracles. When a verdict is received, the verdict is stored on the internal structure for each of the oracles and the aggregator is notified for performing the recalculation of the global verdict for the agent.

The external tool manager is responsible for managing the external tools defined on the validation plan. For each of the external tools, calls to set up the tool and manage its execution are issued. The internal structure of each external tool contains an endpoints section that contains the REST and MQTT endpoints for the external tool to be configured, the tool type, and an additional tool_settings section containing specific configuration for each of the external tools.

```
"external_tools": {
       "urn:ngsi-ld:ExternalTool:ExternalToolElevateSIL": {
         "tool_type": "elevate",
         "endpoints": {
           "rest_endpoint": "http://172.17.10.123:8890",
           "mqtt_endpoint": "mqtt://adms/v1/external-tool/urn:ngsi-
ld:ExternalTool:ExternalToolElevateSIL"
         },
         "tool_settings": {
           "building": "ShortDense4.elvx",
           "passengers": "ShortDense4.txt"
         }
       }
     },
```

When a validation plan is received, the external tool manager configures the child external tools through their REST API, by forwarding the tool_settings structure to the 'adms/v1/external-tool/<tool-type>/setup' REST API endpoint through a POST call. When the oracles are started, the external tool manager starts the external tools by performing a POST call to the 'adms/v1/external-tool/<tool-type>/status' endpoint.

The external tool manager then subscribes to the status MQTT topic of each of the external tools in order to receive status changes and update the internal status of the tool accordingly.

Finally, the aggregation rules module is responsible for aggregating the received oracle verdicts and generating a global verdict for the validation plan being run in the validation agent. The aggregation is made according to the rules defined in the validation plan, with the same aggregation strategies as in the validation orchestrator available.

5.2.2.3. Deployment

The validation agent is also shipped as a docker container. A new docker image is automatically generated each time a new commit is uploaded to the code repository via Gitlab CI/CD pipelines. As part of this process, as it happens with the validation orchestrator, the validation agent also passes several QA procedures, by performing unit tests and automatically generating the documentation for the project with Sphinx.

The validation agent is deployed on the edge, having one validation agent per test level. The validation agent manages all the validation components of that test level, with the ability to manage components that are deployed in the same node or in other nodes that also conform the same test level, as long as the agent is able to reach the REST endpoints of the managed components.
The reason for separating the validation process between agent and orchestrator is twofold. First, thanks to this

responsibility separation, the validation orchestrator is agnostic to the validation components being run in the agent, simplifying the orchestrator overall, and focusing the orchestrator in the coordination of high level components. Secondly, having the agent on the edge allows for improved communication latencies between the managed components and the agent, reducing the issues that could arise from incorrect timing in validation components, reduced traffic to the cloud, having the need to only forward global verdicts to the orchestrator, and finally, reducing the need for the exposure of REST endpoints to components outside of the edge, improving the security posture of the test level.

5.2.3. *Execution flow*

In the following section, the execution flow followed by the validation agent for carrying out a validation plan and coordinating child oracles and external tools will be described. This execution starts with the reception of a validation plan from the orchestrator continues with the configuration of child validation components and finishes with the recollection of verdicts, aggregation and forwarding to the validation orchestrator. On the following figure, an overview of the mentioned execution flow is shown:

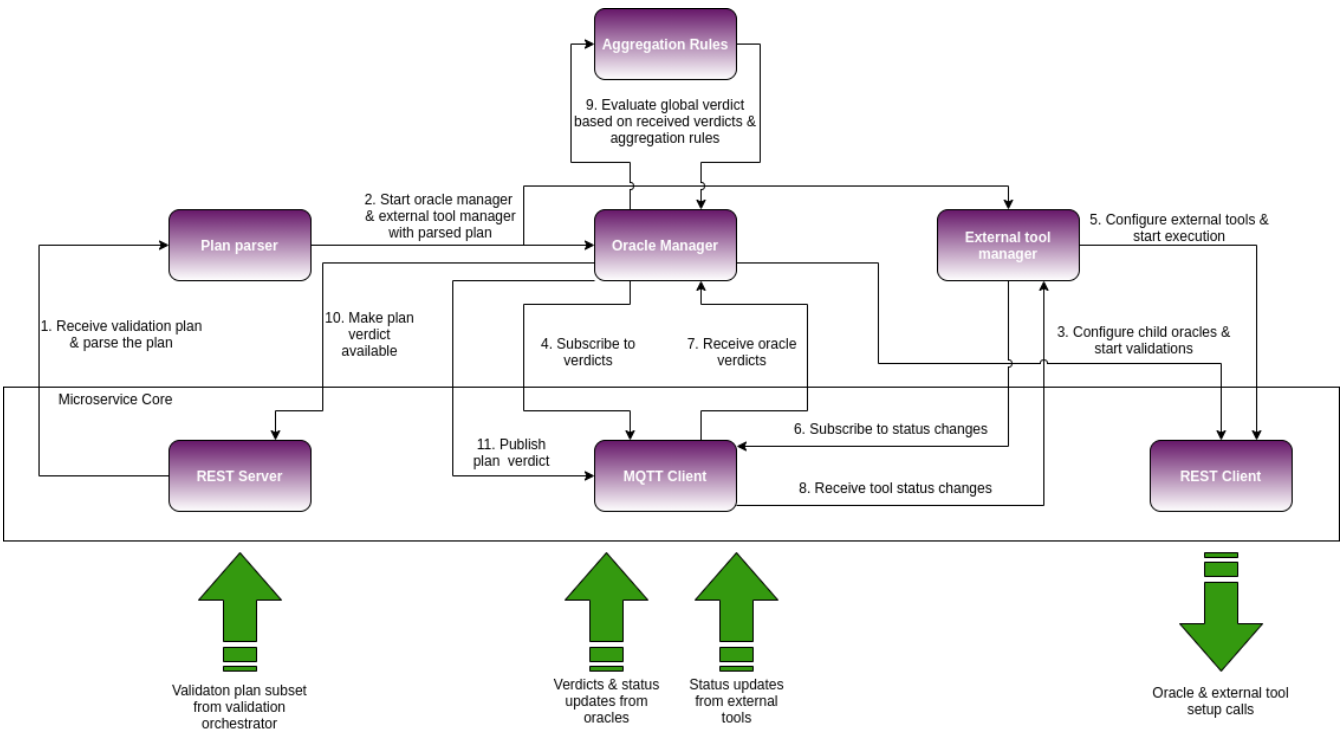

FIGURE 9 EXECUTION FLOW FOR THE VALIDATION AGENT

5.2.4. *Reception of validation plan*

Initially, the validation agent is in idle state, awaiting for reception of a validation plan from the orchestrator to be executed. When the orchestrator sends a JSON formatted validation plan to the 'adms/v1/validation-agent/plan' REST endpoint through a POST call, the validation agent starts the interpretation and execution of the received plan.

Initially, the validation agent will forward the validation agent to the parser component, and the parser will interpret the plan and store it as an internal representation of Python classes. This representation will include the test to be run,

with the structures for each of the child oracles and external tools, their configuration and verdict aggregation strategies.

If the parsing process is successful, the validation agent will send a 200 OK response to the orchestrator and continue with the next phases in the execution of the validation plan.

5.2.5. *Child component configuration*

After the interpretation of the validation plan, the agent and oracle managers will be notified, and the configuration of each of the child components will start.

The oracle manager will extract the oracle structures from the tests in the parsed validation plan, and for each of them, the oracle manager will send the respective REST API calls to set up the oracle. As mentioned in Internal architecture, the oracle manager will perform POST calls to the 'adms/v2/oracle/inputs', 'adms/v2/oracle/setup' and endpoints, for setting the inputs and sending additional configuration, if applicable.

The external tool manager follows a similar approach as the oracle manager does for configuring oracles. When a new validation plan is received, the external tool manager will be notified, and the configuration process for the required external tools will begin. The external tool manager will extract the required external tools from the validation plan, and send configuration requests over POST calls to the 'adms/v1/external-tool/setup' endpoint if the tools required configuration, as shown in the Internal architecture section.

5.2.6. *Validation plan execution*

After all the child elements have been configured, the validation procedure will start. For this, both the oracle manager and the external tool manager will request status changes for oracles and external tools by issuing POST calls to their '/status' API endpoints. Each of the components will answer with a 200 OK response if the status change has been successful. The oracles will be the first components to be started, as the external tools might cause the publication of values to be evaluated. This is the case with the Elevate external tool, as the executed simulations trigger monitors that publish operational values to be evaluated by oracles.

After all the oracles have been started, the external tool manager will be requested to send status change requests to the external tools. The status change requests will be performed through the 'adms/v1/external-tool/<tool-type>/status' endpoint, through POST calls that will be answered with 200 OK responses if the status change has been successful.

5.2.7. *Verdict arrivals & aggregation*

As part of the configuration procedure, the oracle manager will subscribe to verdicts to be raised by child oracles, published through the 'adms/v2/oracle/<oracle-id>/verdict' topic of each of the oracles. When an oracle verdict is received, the origin of the verdict will be determined, and stored in the respective oracle internal structure. After the verdict is stored, the corresponding verdict aggregators will be called. First, the aggregator for each of the test structures will be called with all the verdicts for the oracles in that test case. If a verdict for the test case is obtained, the test case structure will be updated with the updated verdict, and the aggregator for the validation plan being run in the agent will be called.

5.2.8. Submission of verdicts

If the top level aggregator reaches a verdict for all the test cases, the global verdict will be stored in the validation plan internal structure, and then forwarded to the validation orchestrator, which will run the corresponding aggregation for the parent validation plan.

5.2.9. Execution example in the Elevation use-case

5.2.9.1. Reception of validation plan by SiL validation agent

As it was mentioned in 5.1.4 Execution example in the Elevation use-case, in the Elevation use case, we would have a validation agent per test level. The validation orchestrator will forward the subset of the validation plan corresponding to each of the validation agents, and delegate the execution of the validation to each of them. The JSON example below corresponds to the subset of the validation plan shown in the validation orchestrator that the validation agent in SiL (urn:ngsi-ld:ValidationAgent:ValidationAgentSiL01) would receive:


```
{
 "id": "urn:ngsi-ld:ValidationPlan:ValidationPlan01",
 "type": "ValidationPlan",
 "tests": {
   "urn:ngsi-ld:TestCase:TestCase01": {
     "sut": {
       "type": "ElevateDLL",
       "configured_on": "urn:ngsi-ld:ExternalTool:ExternalToolElevateSIL"
     },
     "external_tools": {
       "urn:ngsi-ld:ExternalTool:ExternalToolElevateSIL": {
         "tool_type": "elevate",
         "endpoints": {
           "rest_endpoint": "http://172.17.10.123:8890",
           "mqtt_endpoint": "mqtt://adms/v1/external-tool/urn:ngsi-
ld:ExternalTool:ExternalToolElevateSIL"
         },
         "tool_settings": {
           "building": "OronaTestBuilding.elvx",
           "passengers": "ShortDense4.txt"
         }
       }
     },
     "oracles": {
       "urn:ngsi-ld:Oracle:AdditonalElevatorsAWT": {
         "endpoints": {
           "rest_endpoint": "http://testoracle00:49980",
           "mqtt_endpoint": "mqtt://adms/v1/oracles/urn:ngsi-ld:Oracle:AdditonalElevatorsAWT"
         },
         "inputs": {
           "AWT": {
             "monitor": "urn:ngsi-ld:MonitorAgent:MonitorSIL01",
```

```
              "sensor_group": "urn:ngis-ld:SensorGroup:MonitorSIL01_Group01h",
              "sensor": "urn:ngsi-ld:Sensor:Lift1AWT"
            },
          }
          "oracle_settings": {
            "type": "AdditionalElevators",
            "metric": "urn:ngsi-ld:Sensor:Lift1AWT"
          }
        },
        "urn:ngsi-ld:Oracle:AWTAssertionOracleExample": {
          "endpoints": {
            "rest_endpoint": "http://testoracle01:49981",
            "mqtt_endpoint": "mqtt://adms/v1/oracles/urn:ngsi-
ld:Oracle:AWTAssertionOracleExample"
          },
          "inputs": {
            "AWT": {
              "monitor": "urn:ngsi-ld:MonitorAgent:MonitorSIL01",
              "sensor_group": "urn:ngis-ld:SensorGroup:MonitorSIL01_Group01h",
              "sensor": "urn:ngsi-ld:Sensor:Lift1AWT"
            },
            "LiftSpeed": {
              "monitor": "urn:ngsi-ld:MonitorAgent:MonitorSIL01",
              "sensor_group": "urn:ngis-ld:SensorGroup:MonitorSIL01_Group01h",
              "sensor": "urn:ngsi-ld:Sensor:Lift1Speed"
            }
          }
        }
      },
      "result": {
        "aggregation_type": "DefaultAggregator"
      }
    }
  },
  "result": {
    "aggregation_type": "DefaultAggregator"
  }
}
```


In this example, we can see two different oracles that will perform an evaluation using the average waiting time (AWT) metric and the lift speed as inputs. The validation will be supported by the Elevate simulator external tool, executing the simulation of a traffic profile commonly used for validation and algorithm benchmarking at Orona, The target of the evaluation will be an ElevateDLL SUT, deployed in the same node as the external tool.

The example validation plan contains the following sections:

- URN: The URN of the parent validation plan. (urn:ngsi-ld:ValidationPlan:ValidationPlan01)
- Tests: Test cases to be run by the validation agent. Each test case consists in a unique identifier and sections for the SUT, oracles, external tools and verdict aggregation. (urn:ngsi-ld:TestCase:TestCase01)
    - SUT: Evaluated SUT type and node where the SUT is deployed. Currently informative. (ElevateDLL)
    - Oracles: Contains the oracles used in the test case evaluation and their configuration. (urn:ngsi-ld:Oracle:AdditonalElevatorsAWT & urn:ngsi-ld:Oracle:AWTAssertionOracleExample)
    - External Tools: Contains the external tools being used in the test case and their configuration. (urn:ngsi-ld:ExternalTool:ExternalToolElevateSIL)
    - Result: Contains the specification of aggregation rules used to determine the result for the test case.
- Result: Contains the specification of aggregation rules and their settings to be used to determine the result for the test level (validation plan subset run by the agent).

For this plan, the validation agent will parse the JSON input file, and determine the components to be configured. In this example, the plan will require the configuration of an Elevate simulator external tool instance and two different oracles that will evaluate different aspects of the Lift1Speed and Lift1AWT sensors published by monitors, that in this case will extract values from the running simulation in Elevate by a monitor with the urn:ngsi-ld:MonitorAgent:MonitorSIL01 URN.


```
    "external_tools": {
      "urn:ngsi-ld:ExternalTool:ExternalToolElevateSIL": {
        "tool_type": "elevate",
        "endpoints": {
          "rest_endpoint": "http://172.17.10.123:8890",
          "mqtt_endpoint": "mqtt://adms/v1/external-tool/urn:ngsi-
ld:ExternalTool:ExternalToolElevateSIL"
        },
        "tool_settings": {
          "building": "OronaTestBuilding.elvx",
          "passengers": "ShortDense4.txt"
        }
      }
    },
```


The example external tool configuration includes the configuration of the urn:ngsi-ld:ExternalTool:ExternalToolElevateSIL external tool, that in this case is of tool type elevate, meaning that the external tool is serving as a wrapper for the Elevate simulator, where a simulation using the SUT to be evaluated will be performed. The external tool section includes settings for defining the building settings file(OronaTestBuilding.elvx) to be used by Elevate, and the passengers traffic profile(ShortDense4.txt), that contains a series of passenger elevator calls to be used as a test input profile for the simulation. Each line of the passengers file will represent a passenger or group of passengers calling to an elevator at a certain time and building level.

```
    "oracles": {
       "urn:ngsi-ld:Oracle:AdditonalElevatorsAWT": {
         "endpoints": {
           "rest_endpoint": "http://testoracle00:49980",
           "mqtt_endpoint": "mqtt://adms/v1/oracles/urn:ngsi-ld:Oracle:AdditonalElevatorsAWT"
         },
         "inputs": {
           "AWT": {
             "monitor": "urn:ngsi-ld:MonitorAgent:MonitorSIL01",
             "sensor_group": "urn:ngis-ld:SensorGroup:MonitorSIL01_Group01h",
             "sensor": "urn:ngsi-ld:Sensor:Lift1AWT"
           },
         }
         "oracle_settings": {
           "type": "AdditionalElevators",
           "metric": "urn:ngsi-ld:Sensor:Lift1AWT"
         }
       },
       "urn:ngsi-ld:Oracle:AWTAssertionOracleExample": {
         "endpoints": {
           "rest_endpoint": "http://testoracle01:49981",
           "mqtt_endpoint": "mqtt://adms/v1/oracles/urn:ngsi-
ld:Oracle:AWTAssertionOracleExample"
         },
         "inputs": {
           "AWT": {
             "monitor": "urn:ngsi-ld:MonitorAgent:MonitorSIL01",
             "sensor_group": "urn:ngis-ld:SensorGroup:MonitorSIL01_Group01h",
             "sensor": "urn:ngsi-ld:Sensor:Lift1AWT"
           },
           "LiftSpeed": {
             "monitor": "urn:ngsi-ld:MonitorAgent:MonitorSIL01",
             "sensor_group": "urn:ngis-ld:SensorGroup:MonitorSIL01_Group01h",
             "sensor": "urn:ngsi-ld:Sensor:Lift1Speed"
           }
         }
       }
     },
```


Following with the oracles section, the example defines two different oracles named as urn:ngsi-ld:Oracle:AdditonalElevatorsAWT and urn:ngsi-ld:Oracle:AWTAssertionOracleExample.

The AdditionalElevatorsAWT exposes a single inputVariationPoint, AWT, that is mapped to the urn:ngsi-ld:Sensor:Lift1AWT sensor in the configuration. The oracle exposes the type and metric configuration settings.

On runtime, the validation agent interacts with the oracle on the defined REST endpoint, on the 'adms/v2/oracle/inputs' and 'adms/v2/oracle/setup' paths to configure the AWT inputVaritionPoint and to pass the oracle_settings section, respectively.

The second oracle shows an example oracle evaluating the AWT and LiftSpeed inputVariationPoints. In this case, the oracle does not require any additional settings apart from the inputs, and therefore, only calls to the 'adms/v2/oracle/inputs' REST endpoint are required to be issued.

After the oracles have been configured, the validation agent will issue status change calls to the urn:ngsi-ld:Oracle:AdditonalElevatorsAWT and urn:ngsi-ld:Oracle:AWTAssertionOracleExample oracles, by sending a POST request to their 'adms/v2/oracle/status' endpoint. The oracles will return a 200 OK on success.

When the validation agent has determined that the oracles have changed their execution status, it will issue a status change call to the external tool microservice defined in the plan, in order to start the execution of the Elevate simulation tool. Upon start, the monitors attached to the Elevate simulator will start publishing operational data that the oracles will collect, and the validation will take place.

During the validation process, the validation agent will listen for verdicts coming from the oracles, and perform aggregations accordingly. In the case of the example, the DefaultAggregator aggregation strategy will be used for both test case and test level aggregation. This strategy marks the aggregated verdict as passed if the verdicts published by the two oracles used in the validation are marked as passed. A failed verdict for any of the two oracles will cause the test case verdict to be marked as failed. The aggregation will continue to the top level, performing the aggregation for the test level when the verdict for the test case has been obtained.

When a top level verdict is obtained, the validation agent will publish the verdict through the 'adms/v1/validation-agent/urn:ngsi-ld:ValidationAgent:ValidationAgentSil01/verdict' MQTT topic, where the validation orchestrator will collect the obtained verdict for the SiL test level.

6. Instantiation of the Agile Test Execution Framework on the Elevation Use Case at MiL, SiL and HiL Test Benches

With the goal of demonstrating the feasibility and scalability of the approach with our industrial use-cases, this section explains the instantiation of the ADEPTNESS ecosystem within the SiL test level of Orona's use-case (i.e., the elevation use-case).

6.1. Introduction

The external tool microservice is a convenience microservice developed with the integration of legacy, unsupported or out of scope tools into the ADEPTNESS ecosystem. The external tool microservice exposes a basic API and means of managing the execution of a subprocess or external application. This API can be extended to provide additional functionality and means to manage the tool being commanded by the external tool microservice. For each tool that wants to be integrated on the ADEPTNESS ecosystem, an ad-hoc implementation of the external tool microservice is performed, creating external tool subtypes for each of the integrated tools.

6.2.Architecture

To instantiate the Agile Test Execution framework within one of the industrial use-cases used in ADEPTNESS, i.e., the elevation use-case, we have developed its external tool to execute tests at the SiL test level. The base external tool microservice is built on top of the ADEPTNESS Python microservice template, and it exposes basic API endpoints for managing the execution of a command, launched by the Python subprocess library, and receiving feedback about the execution status. The following figure shows a high-level overview of the external tool.

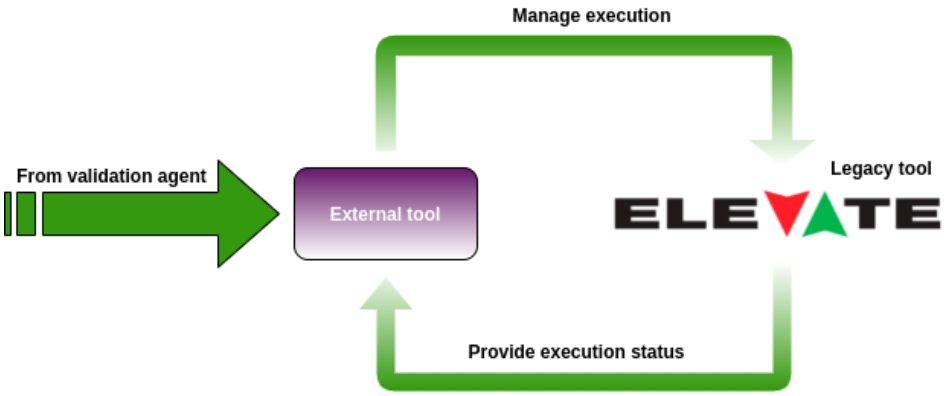

FIGURE 10 INTERACTION OF THE EXTERNAL TOOL MICROSERVICE WITH OTHER VALIDATION COMPONENTS

6.2.1. REST and MQTT API endpoints

The external tool microservice provides a very basic API to manage the execution of the tool being managed by the microservice. Over REST, the external tool microservice provides the 'adms/v1/external-tool/process/<tool_type>/status' endpoint for managing and getting the execution status of the subprocess. The management of the execution status is performed through a POST method call to the REST endpoint, with a basic JSON payload determining the status, which can be {"status": "running"} or {"status": "stopped"}. The execution status can be consulted following the same process, but by issuing a GET call to the same endpoint.

6.2.2. Internal architecture

The external tool is divided into the microservice core, where the REST API calls are implemented, and a subprocess manager component, that is in charge of managing the execution of the subprocess launched by the external tool. When receiving a status change request, the request is forwarded to the subprocess manager, which launches a predefined shell command in a separate thread, and waits for completion of the execution. Depending on the tool to be implemented, additional care has to be taken, as some legacy applications tend to run in background and return the control to the parent process, causing the standard python subprocess library to mark the process as terminated. For such cases, logic for monitoring the PID or the existence of the process has been implemented.

6.3.Execution flow

The external tool execution flow starts with the reception of a status change request from the validation agent, and continues with the execution of the contained tool and the reporting of the status changes. The following diagram shows an insight of this execution process, while the following sections will describe this process in depth:

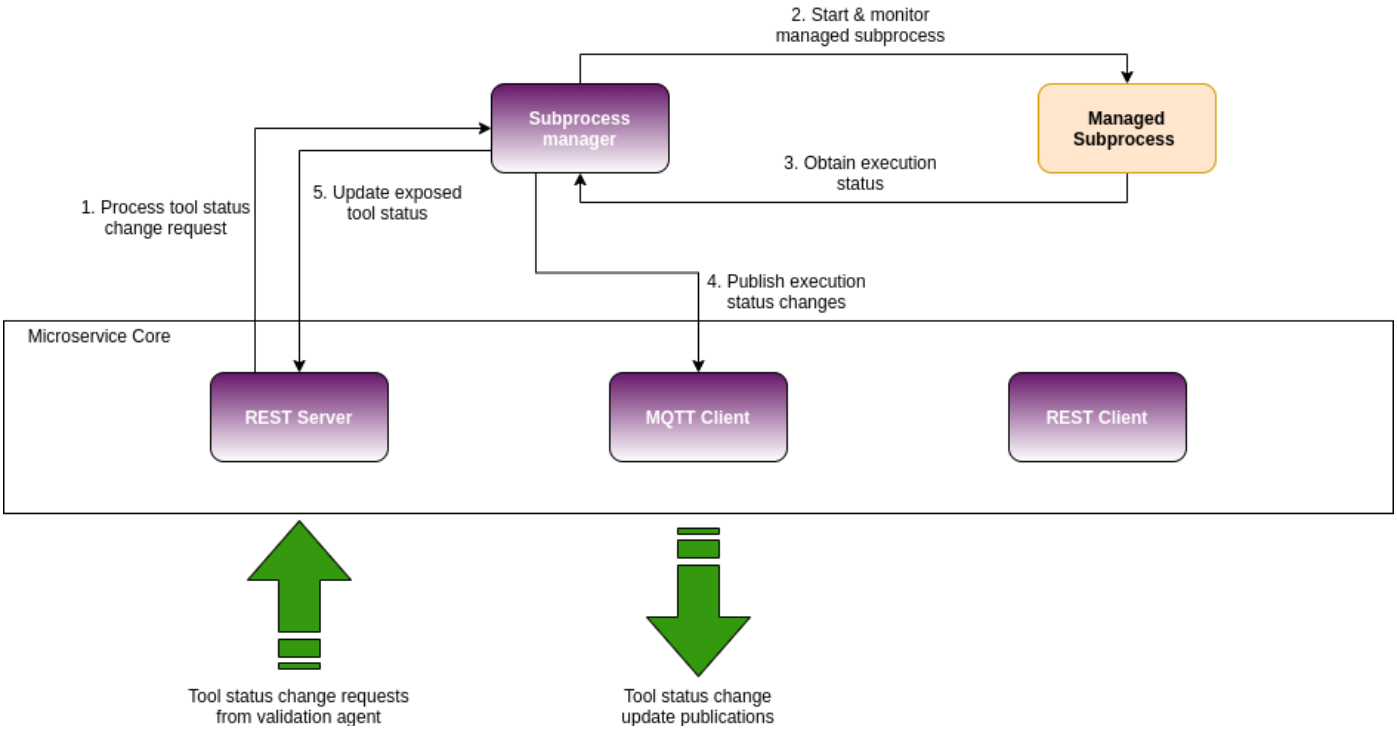

FIGURE 11 EXECUTION FLOW FOR THE VALIDATION AGENT

The external tool microservice awaits for status change requests made through the REST API endpoint. When receiving a status change, the external tool starts its execution flow.

When the status change request is received, the external tool microservice checks the current status of the subprocess. If the subprocess is idle, the external tool will proceed and request the subprocess manager to change the tool status.

6.3.1. Subtool execution

Upon request, the subprocess manager will start a new thread to manage the subprocess, and launch the predefined command through a call to the Popen function of the Python subprocess library. The execution of the called command is synchronous, so the thread launching the command will await until the termination of the tool. In these cases, the tool will be marked as running before launching the command, and marked as stopped after returning from the call to Popen.

In the cases that the launched command immediately returns control to the launching application and runs in the background, a different approach has to be followed. In these cases, the subprocess manager provides the ability to launch the process and monitor the process PID in a loop for termination. When the process PID is no longer valid, the tool will be marked as stopped.

As mentioned in the previous section, the external-tool microservice has ad-hoc implementations for the integrated tools, use-case specific implementations provide more sophisticated management of the launched applications and additional functionalities, not included in the provided reference implementation for the external-tool microservice.

6.3.2. Status reporting

The thread launching the tool will be in charge of updating the tool execution status. Upon launch, the thread launching the external command will update the internal structure storing the tool execution status, and perform a publication using the MQTT client provided by the microservice core. On termination of the launched command, the same approach will be followed,

the internal status structure will be updated, and a publication will be made through MQTT on the 'adms/v1/external-tool/<tool-type>/status' topic, indicating the status change.

6.4. Application to the Elevation use-case

In the Elevation use-case, Orona makes use of a legacy tool named Elevate. Elevate is an elevator installation simulator, used by Orona as a platform for validating developed traffic algorithms at SiL level. Elevate receives a configuration file that defines the installation characteristics (e.g., building level quantity, number of elevators, dynamic information of the elevators (e.g., speed, acceleration, jerk)) and a passenger list file, containing a list of "passengers", represented as calls to the elevators at different building levels and times. The passenger list can be built based on traffic captured from a real installation or based on simulated statistical models.

In Orona's use case, the SUT is the traffic algorithm being developed for Orona's elevators. This algorithm is tested at the SiL test level using Elevate, which supports loading an external traffic algorithm as a DLL library. As Elevate is a key part of the validation workflow at Orona, it was desired to be able to automate the execution of tests with Elevate on the ADEPTNESS ecosystem, so an external tool wrapper for the Elevate simulator has been developed.

6.4.1. Elevate external tool

The Elevate external tool is an ad-hoc implementation of the external tool microservice, adapted to the Elevate simulator application. Based on the reference implementation for the external tool, it provides an improved wrapper over the Elevate simulator.

Elevate is a Windows GUI application running on 32bits. Developed in 2008, the 8.19 version used in Orona provides a GUI and a basic command line interface to run simulations in batch, which has proven to require some additional system requirements to run the simulations. On launch, the application creates a background thread and returns control back to the terminal, meaning that additional checks need to be made to check the execution status of the tool. It makes use of a USB license dongle, license limiting Elevate to a single executing instance. The license dongle also limits the application to be run locally, disallowing the usage of RDP for connecting to the machine. Because of the mentioned limitations, Elevate is a perfect candidate for being wrapped in a microservice layer that ideally will serve as a Simulation-as-a-Service tool for Orona in the context of the ADEPTNESS project.

For managing the tool, the developed subprocess manager makes use of a GUI automation library, called PyWinAuto. This library allows not only to launch the application, but to also interact and read the status of GUI elements. Thanks to this, a more complete automation can be achieved, where the microservice is even able to perform automation of processes where otherwise user interaction would be needed.

6.4.2. Exposed capabilities for tool

The external tool microservice is capable of running batch evaluations of different traffic profiles automatically, through a status change in the tool execution status performed over the 'adms/v1/tool/process/elevate/status' REST API endpoint. Running batch simulations with Elevate is achieved by launching Elevate with the '/rb' command line option. However, Elevate is only capable of running simulations from the root folders of disk drives. This is a limitation the 8.19 version in use in Orona has. To overcome this limitation, the subprocess manager is capable of spawning virtual drives that will contain configuration and passenger files, and to manage their lifecycle in concordance with the execution of the tool. The creation of virtual drives is achieved by interacting with native Win32 APIs exposed by Windows, integrating them thanks to the Python's ctypes library. The PyWinAuto library used for launching Elevate also helps to overcome the background process launching issue exposed in previous sections. The library provides a wait_for_process_exit method that waits for the application exit even in cases that the control is returned to the calling application.

6.4.3. *Execution of the tool in validation example*

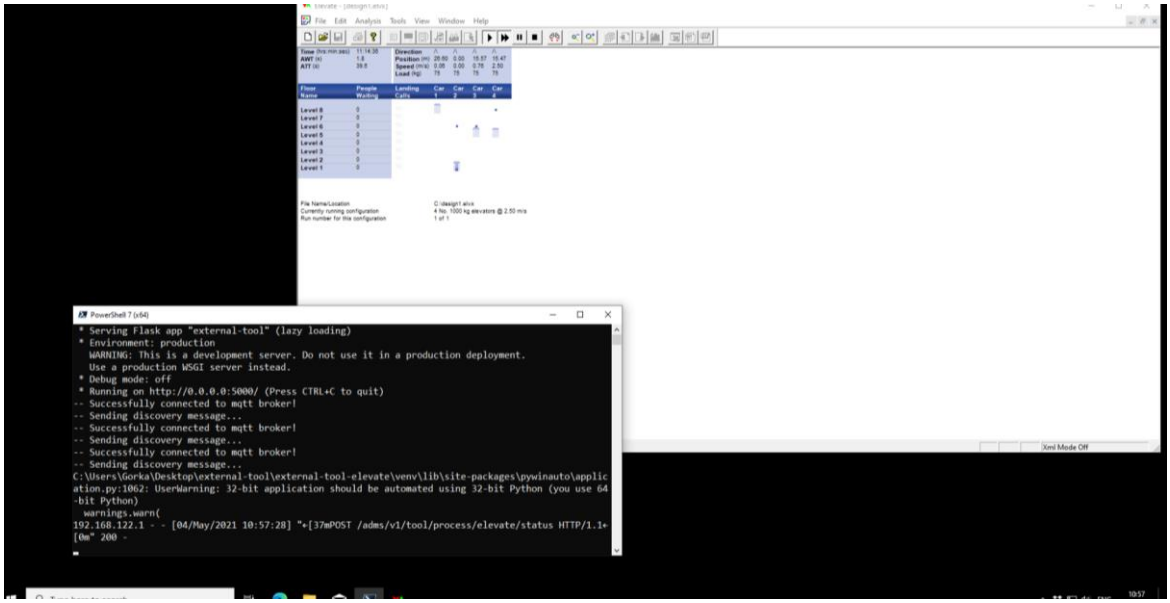

FIGURE 12 EXECUTION OF THE ELEVATE SIMULATOR WITH THE EXTERNAL TOOL

In the image above, we can see an example execution of the Elevate application in an ADEPTNESS validation context.

A physical Windows machine where Elevate is installed is used for launching the external tool microservice, where it awaits connections on the 5000 port. After the external tool microservice has received a POST request from the validation agent on the 'adms/v1/tool/process/elevate/status', we can see how the microservice answers the validation agent with a 200 OK response, after which the Elevate simulator is launched. In this case, the building configuration file and passenger list file are already placed on the target launch directory, and the target traffic algorithm SUT placed on the Elevate installation folder as a DLL file. While the application is running, operational values will be posted through MQTT, and collected by the oracles, that will perform the evaluation and raise a verdict for the tested conditions.

7. Instantiation of the Agile Test Execution Framework on the Railway Use Case at MiL, SiL and HiL Test Benches

Here we provide details on the methodology and tool support for describing domain specific requirements and using it for modelling and generating the concrete and executable test scripts to validate an embedded system. The proposed MBT workflow that covers requirements specification to test verdict assessment has been divided into three phases as shown in figure 13.

- 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) [12].
- Phase 3: Execution of generated test scripts on simulation levels and test verdict assessment.

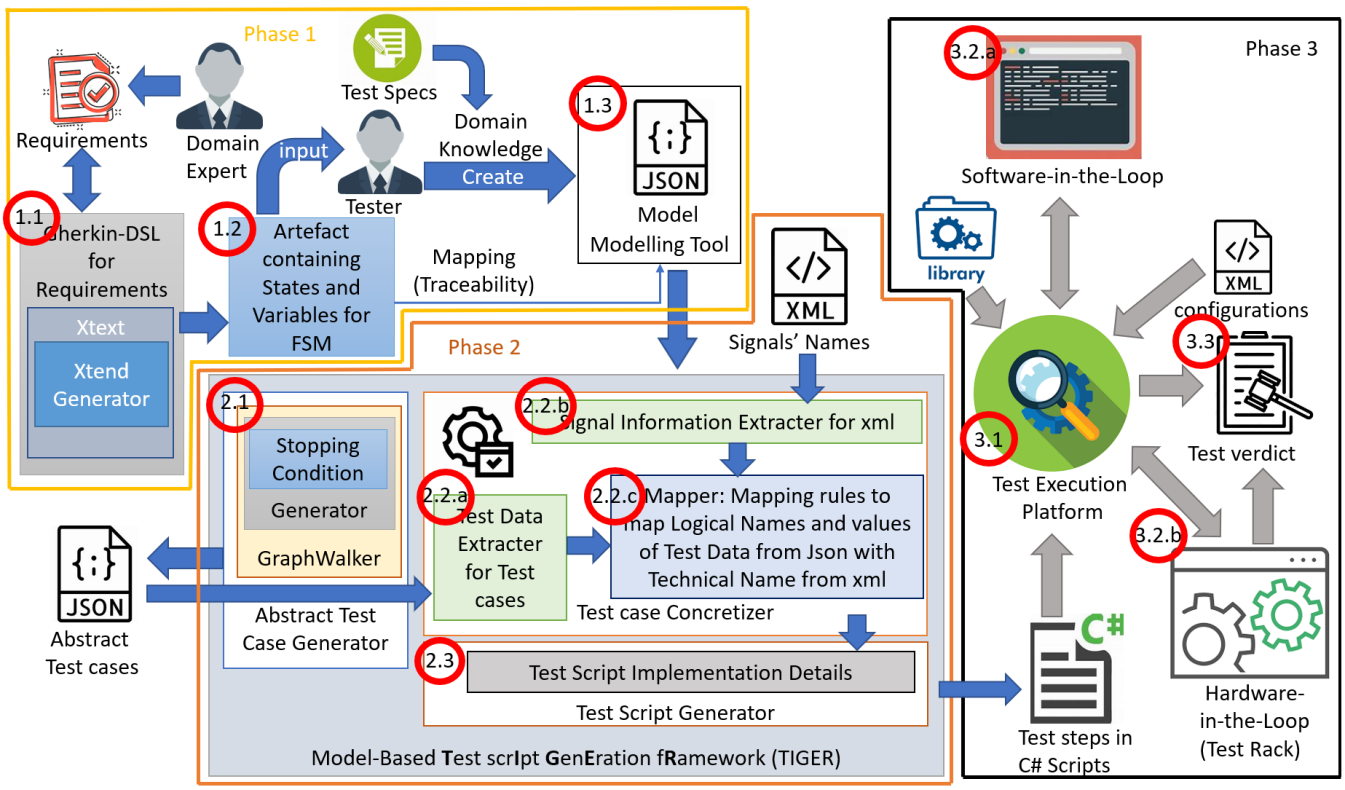

FIGURE 13 A MODEL-BASED TESTING WORKFLOW FOR AN EMBEDDED SYSTEM, FROM REQUIREMENTS SPECIFICATION TO TEST VERDICT ASSESSMENT, IN SIL AND HIL SIMULATION PLATFORMS

Phase 1: Requirements Description and Automatic Generation of Supporting Artefact for Modelling

The first phase deals with the modelling aspect of the SUT. To model a SUT, data sources (such as code or specification documents) are required to identify the domain specific entities. 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. These requirements are specified by a domain expert using the Gherkin-like DSL and are used to extract model entities, such as states and transitions, along with variables and their corresponding values involved in the transitions. 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.

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. figure 13figure 14 represents the meta-model for describing the concepts of requirements specification in a Gherkin-like format. In the standard 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. Hence, in the meta-model (figure 14), 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.

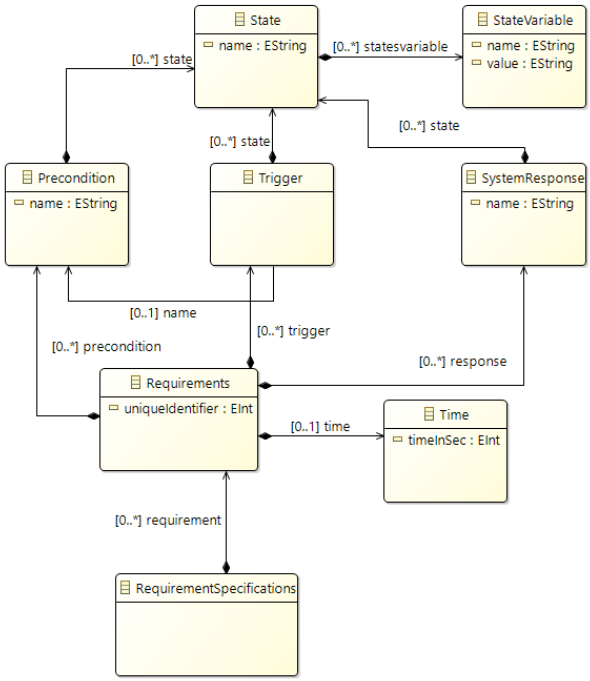

FIGURE 14 META-MODEL FOR GHERKIN-LIKE DSL SHOWN AS AN UML CLASS DIAGRAM

Figure 15 shows the implementation of the Gherkin-like DSL in concrete syntax using Xtext grammar.

```
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 15 CONCRETE SYNTAX OF THE GHERKIN-LIKE DSL USING XTEXT GRAMMAR

Automatic Generation of Supporting Artefact for Modelling. We have also implemented the Xtend generator as shown in figure 16 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 [9]. The Xtend generator contains the mapping between each meta-model element of defined DSL and model elements of FSM (i.e. states, transitions etc.). It extracts the information such as model name, state name, transitions, transition variables and their corresponding values from each requirement specified by the requirement engineer in the editor. The validation checks to validate each requirement's atomicity, completeness and unambiguity can be included as an advanced feature in Xtend, which is currently done manually in our case.

```
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 16 IMPLEMENTATION OF XTEND GENERATOR

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. Figure 17 shows a sample of the extracted information from the Xtend-generated artefact.

```
Requirement# 01: Given a FDCU is Master (varibale1 = 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 17 An example of a requirement specified in Xtext following the Gherkin-like DSL grammar

Modelling of the SUT. Our experience with modelling the SUT [10] 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 is created based on the requirements specification, our experience suggests that it evolves by including the tester's perspective and viewpoint. 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 [10].

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.

Phase 2: Generation of Executable Test Script using Model-Based Test scrIpt GenEration fRamework (TIGER)

The TIGER framework is proposed and developed by us to provide a cost and time efficient solution to generate an executable test script using the transformation approach [11] of MBT. Following the transformation approach, we have defined the mapping rules to transform the abstract test cases into concrete ones. 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 [12], here we briefly summarize these parts.

Abstract Test case Generator. The abstract test case generator is based on GraphWalker (GW) (https://github.com/GraphWalker/graphwalker-project/wiki), which is a freely available tool for generating abstract test cases from the model of SUT. 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.).

Test Case Concretizer. After the generation of abstract test cases, 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 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 [12].

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.

Phase 3: Execution of Generated Test Scripts on Simulation Levels and Test Verdict Assessment

The test scripts generated by TIGER can be executed on SiL and HiL levels after including specifically designed libraries and integrating a configuration file in our case company's C# generic test framework. The configuration files and libraries provide an adaption layer for the execution of generated test scripts by configuring the environment and simulations at SiL and HiL level. 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.

The test of the TCMS System uses test setups utilizing the following test equipment: Virtual CCU, Virtual Train and Test rack. The Virtual CCU is a tool which runs at a test PC, based on the Soft-TCMS technology and exposes a virtualized environment for the SW applications at an CCU. It provides the same functional environment as the real CCU (process scheduling, OS calls, communication means) including the capability for signal stimulation, observation and debugging. The Virtual CCU can be built directly from the TCMS project files. The test rack is a setup, which hosts the TCMS System of one vehicle comprising the real TCMS devices. In addition, the test rack hosts the test execution environment and supporting tools (e.g. remote controllable switch to power TCMS devices). Like the Virtual CCU, the Virtual Train is a tool,

which runs at a test PC, based on the Soft-TCMS technology and exposes a virtualized environment for the SW applications at a CCU as well as the configuration of the additional TCMS devices (switches, gateways) including the TCMS bus systems and the test execution environment.

7.1. Case study

This section deals with the evaluation of the proposed MBT workflow through an industrial case study. We validate the three phases of the proposed workflow, starting from the information generated from the Gherkin-like DSL in XText, the Xtend generator and the model created using it. We have used the fire detection subsystem, on-board the driver's cabin in one of Alstom Transport's metro train projects, as the SUT. The fire detection subsystem interacts with the Train Control Management System (TCMS) to detect possible fire in the driver's cabin. Using an iterative approach, we created the first version of the SUT model using identified model elements (such as states and transitions) and then included the tester's perspective by examining the test specification in the final version of the model. Further, using TIGER, we concretized the abstract test cases generated from GW and finally, we executed the generated test scripts on the SiL level to generate the test verdicts. The implementation details regarding these phases in the industrial case study are presented in the following subsections.

7.1.1. SUT

The case selected for evaluation is the part of an ongoing 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. This control and management of important subsystems is critical to the functioning of modern trains. 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 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.

7.1.2. Generated Artefact

The requirements specification related to the fire detection subsystem has been specified in natural language and resembles the Gherkin-like format. In contrast to a typical `Given-When-Then' Gherkin format, Alstom Transport's domain expert used `Given-Then-Within' format where the `Within' clause specified the timing constraint of each requirement.

After specifying the requirements in Xtext-based Eclipse editor as shown in Figure 18 the Xtend generator generated a text file containing the information about model elements as shown in Figure 17. 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 before, incorporating the tester's perspective in the model makes behavioral aspects more complete 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.

```
Requirement# 1

                Model Name: FDCU
                            State Name:FDCU
                            State Name: Master
                            Transition/Action Variables:
                            Variable Name: varibale1 = 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 18 A SAMPLE OF EXTRACTED INFORMATION FROM THE XTEND GENERATED ARTEFACT

7.1.3. Modelling of the SUT

Based on the information in the Xtend-generated artefact alone, we have created the initial models representing FDCU and TCMS in GW, as shown in figure 19. On both sides of the models, we have also provided an outline view of elements generated by Xtend generator and mapping of generated elements with initial model elements using arrows. After refining and including tester's domain knowledge in the modelling process, the final model of the system is shown in figure 20, 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 figure 20 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.

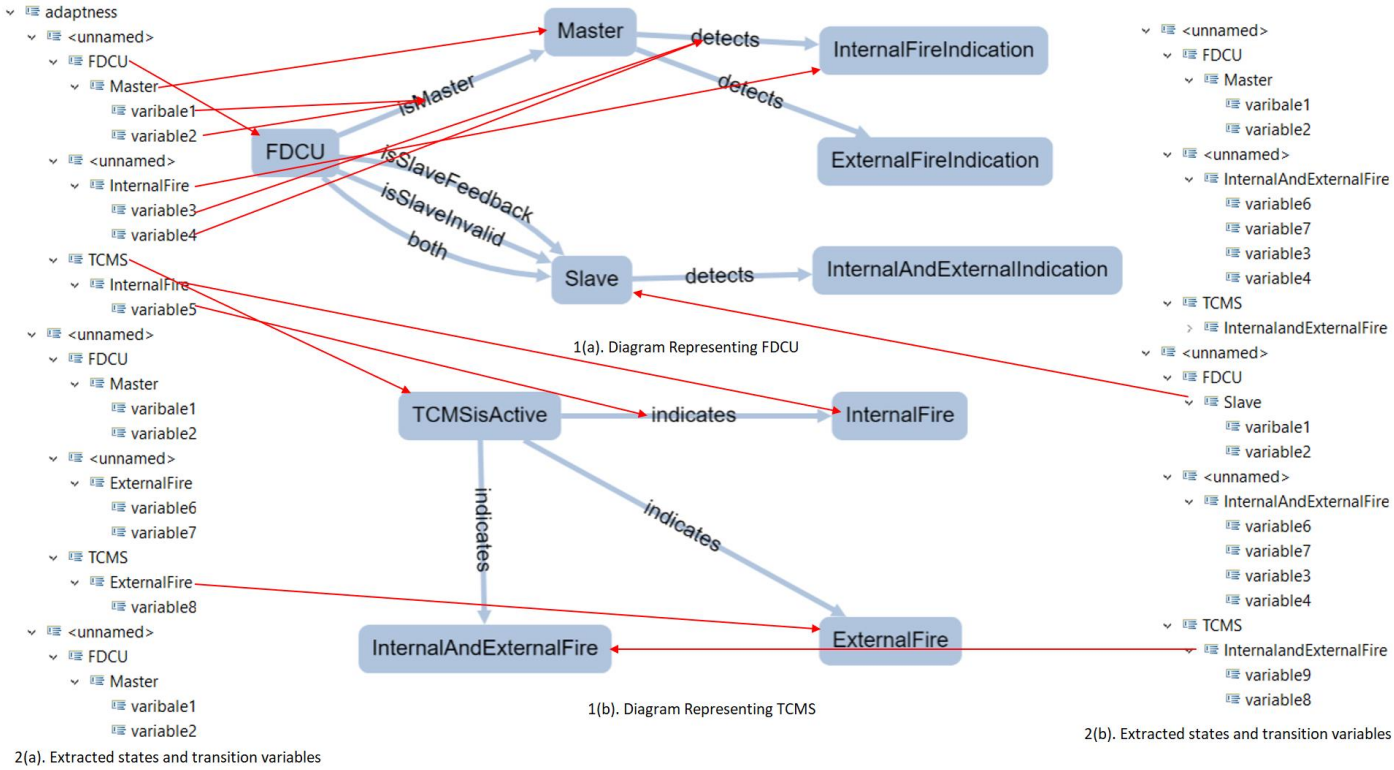

FIGURE 19 INITIAL MODEL REPRESENTING FDCU AND TCMS AS A BLACKBOX

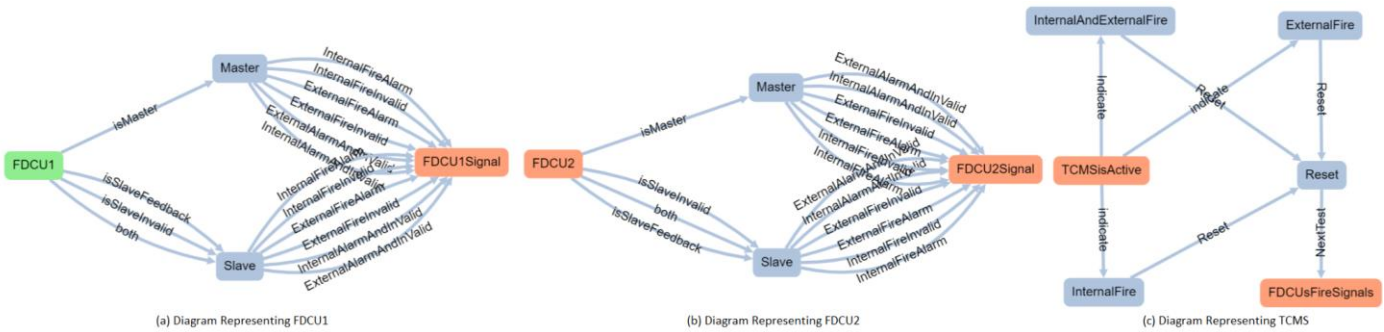

FIGURE 20 FINAL MODEL REPRESENTING FDCUS AND TCMS AS A BLACKBOX

7.1.4. 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 as shown in figure 21. 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.

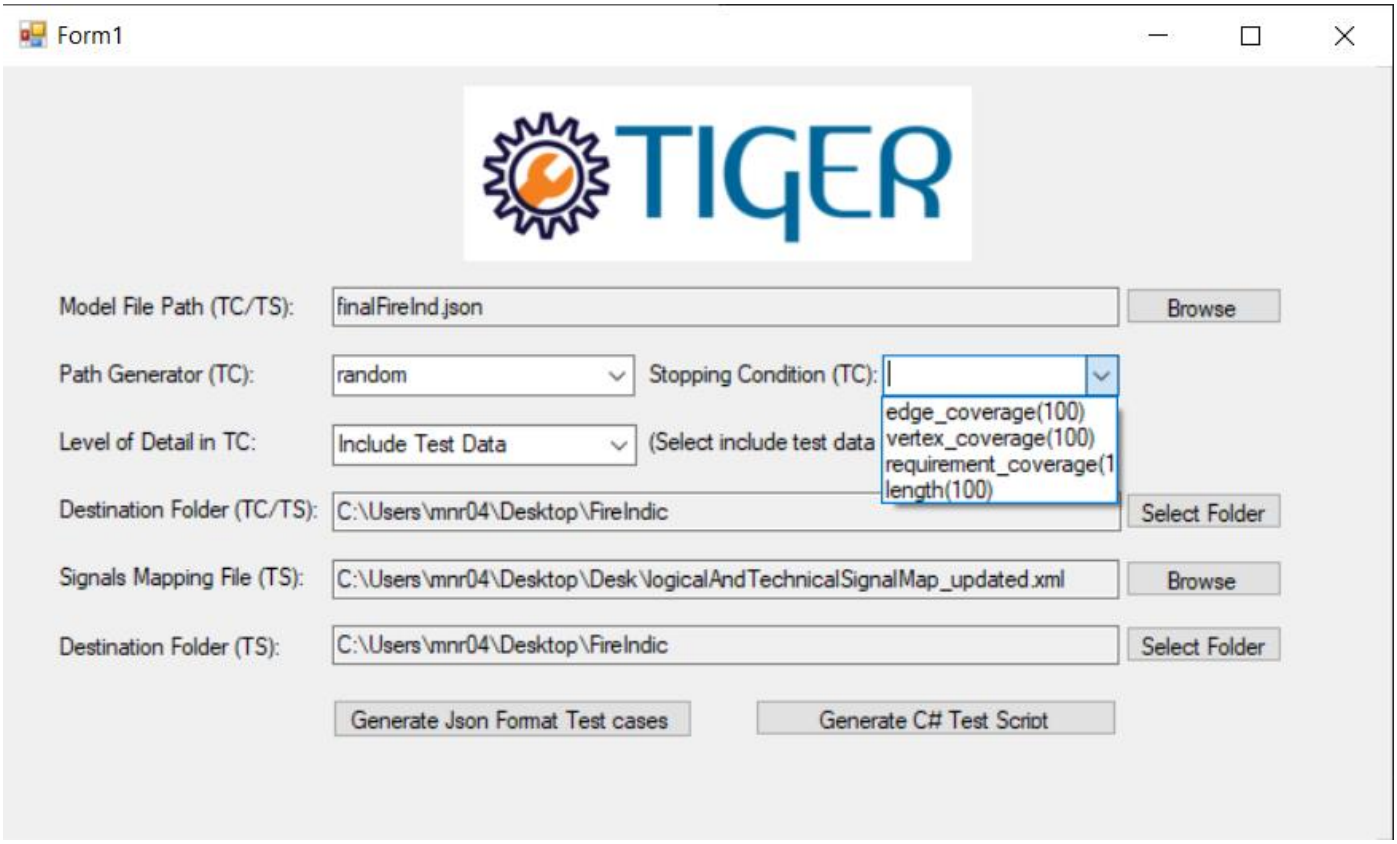

FIGURE 21 GRAPHICAL USER INTERFACE OF TIGER

Execution of generated test scripts on Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) level

We have also executed the generated test script on SiL and HiL level by creating a generic C\# project in Visual Studio and by importing TCMS libraries and configuration file in it. After the execution, the test execution platform generated a test verdict in html format. The test verdict contains the configuration steps, test steps along with their pass or fail status and a summary of the whole execution.

7.2. A Cloud & Microservice-Based Architecture and Realization for MiL Simulation Test Bench in the Railway Domain

The objective of this section is to propose a general-purpose cloud-based architecture of model-based simulation testing of embedded software where the model-based testing tool acts as a web service fulfilling requests for information from clients. The section further demonstrates different models of implementation of this general-purpose architecture by taking Amazon Web Services (AWS) as the example cloud service provider. The MBT tool of choice is GraphWalker (GW) while the implementation is validated on an embedded software case study from Alstom Transport, one of the world leading manufacturers of trains.

7.2.1. GraphWalker as a RESTful Web Service

REST (Representational State Transfer) is a software architectural style that applies to web services and allows for accessing and manipulating web resources through predefined operations. GW can be run as a RESTful web service. It has a number of useful REST API commands: load, hasNext, getNext, getData, setData, restart, fail and getStatistics:

- A call to **load** uploads model(s) in JSON format and resets GW.
- A call to **hasNext** returns true if stop condition has not been reached and that there are more nodes and transitions to cover.
- A call to **getNext** will return the name of the next element from the path generation.
- A call to **getData** will return the current data from the path generation.
- A call to **setData** will return true if the data is set correctly in the current model.
- A call to **restart** will reset the model to its initial state.
- A call to **fail** stops the path generation.
- A call to **getStatistics** will return basic numbers on edge coverage, vertex coverage, total number of visited edges and nodes, among other things.

7.2.2. GraphWalker's RESTful API Commands Flow

When GW is run as a Restful service, there is an API that is used to interact with it. There exists eight REST API commands to interact with the GW service: load, hasNext, getNext, getData, setData, restart, fail and getStatistics (figure 22).

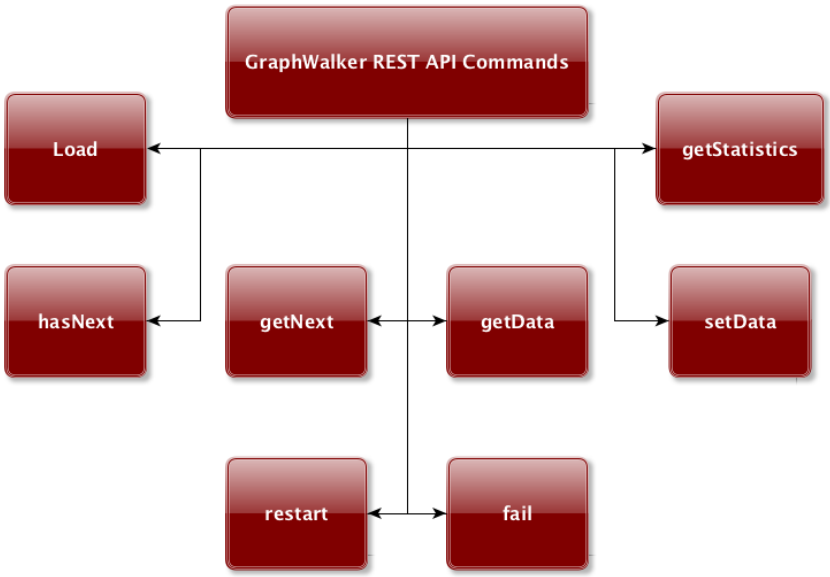

FIGURE 22 GW'S REST API COMMANDS

Below is the brief description of each of these commands:

- **load**: Uploads the model to the GW service.
- **hasNext**: Returns true if there is an element to get from the path generation.
- **getNext**: Returns the next element name in the response from the path generation.
- **getData**: Returns the current data values as key value elements.
- **setData**: Sets the value of a key to the user-defined value.
- **restart**: Resets the loaded model to its initial states.
- **fail**: Stops the test session.
- **getStatistics**: Returns a number of statistics for the current test session, including edge and vertex coverage.

7.2.3. A General Purpose Architecture for Model-Based Testing as a Web Service

Cloud computing has been around for several years, promising simplified access to a wide range of services based on a demand-driven resource availability. As GW generates a large number of test sequences for testing many interactions,

leveraging cloud-computing technology to enable the generation and execution of such a large number of test sequences seems like a good match to fulfil time and resource constraints. It is especially true for behavioural testing of the system under test in a black-box manner when the tests are generated through MBT and when the *online* MBT needs to connect directly to the SUT for dynamic testing of the SUT, powered through the RESTful API commands on offering from the MBT tool. Even in the case of the more traditional *offline* MBT, RESTful API commands can return useful information on the current state of test generation as well as enable controlling MBT as a service such as through restarts and updating the model elements.

In order to leverage upon cloud computing for MBT of embedded software, it is necessary to understand the usage of cloud offerings, generally divided into three categories of Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [13]. In IaaS, computational resources, such as servers, network, storage etc. are offered by the cloud service providers. In PaaS, developers design, develop and test their applications on cloud provider's infrastructure and then deliver them to the end user from cloud servers. In SaaS, applications are hosted in the cloud and accessed by the users through the internet.

In the current architecture of executing MBT as a web service for testing embedded software, we have explored the possibilities of IaaS and SaaS are applicable cloud offerings.

The generic MBT web service architecture (figure 23) is simple and applicable for any cloud service provider. There are four main entities in this generic architecture: RestServer, RestClient, Jenkins automation server and the database. The RestServer is the virtual machine that is configured to run an operating system that also supports the installation and running of a MBT tool such as GW. The interaction with this RestServer happens through the Rest API commands, such as POST, PUT and GET. The RestClient takes care of the interaction of the web service from the outside world. Here in this architecture, the Jenkins automation server sends a POST upload request to the RestClient with the purpose of loading the GW model file. The Jenkins automation server keeps the latest version of the model file and whenever there is a change in the model, the POST request is initiated such that GW always works on the latest version of the model. The RestClient is accessible to the Jenkins automation server through its public IP that is configurable through the cloud-service provider as well as the authorization mechanisms that support only valid and secure connection. The RestClient loads the model to the RestServer using the Rest POST command. The database keeps logs of test execution as well as metadata regarding test cases such as date and time of execution and even values of the variables.

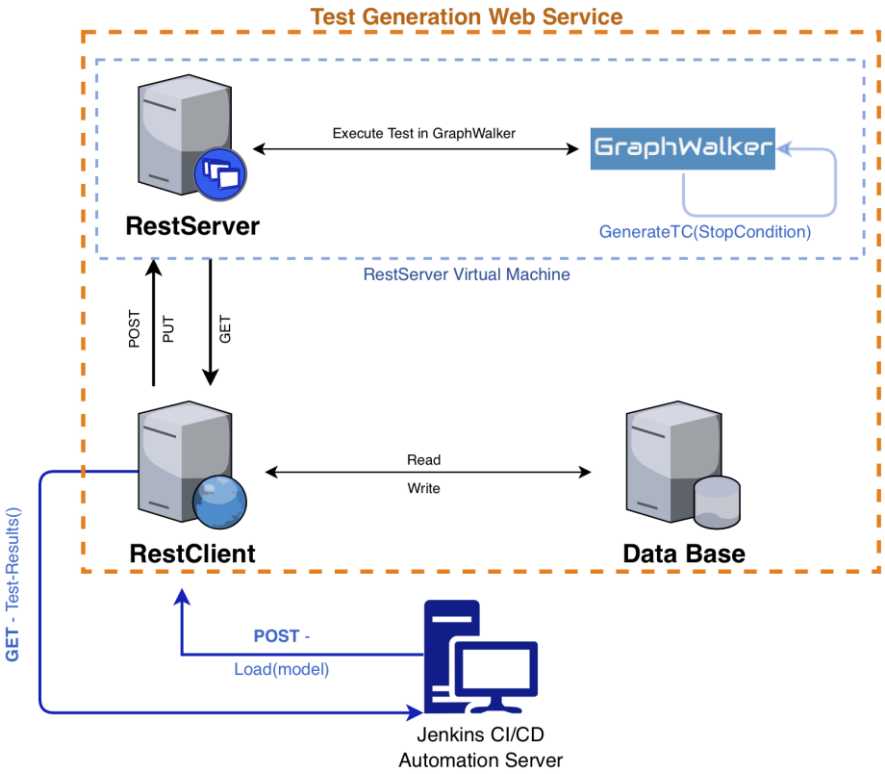

FIGURE 23 A GENERAL-PURPOSE ARCHITECTURE FOR MBT AS A WEB SERVICE

This generic MBT web service powered through cloud services is tested on four models of its implementation in Amazon Web Services (AWS) as the cloud service provider. These four models represent different options of implementation of the proposed architecture, each one adding some additional capabilities to the previous one. AWS was selected as a matter of convenience due to our previous knowledge about it, however, as mentioned earlier, the generic architecture can be instantiated using any cloud service provider. The four models of implementation of MBT as a web service are:

1. Variant 1 - The simple architecture: The external client access to the web service via RestClient (implemented through AWS Lambda).
2. Variant 2 - The dynamic architecture: The external client access to the web service via direct access to RestServer (implemented through AWS EC2).
3. Variant 3 - The advanced dynamic architecture: The external client access to the web service via RestClient with a database (implemented through AWS Dynamo DB).
4. Variant 4 - The HTML server enhanced dynamic architecture: The external client access to the web service via RestClient with a database and HTML server.

7.2.4. Variant 1 - The Simple Architecture

The first simplest model of implementation (Figure 24) makes use of certain services provided through AWS. S3 bucket is a storage service that might be needed to store certain files during and after the test generation. EC2 is the virtual machine, where GW is run, also represented as RestServer. The access to GW is through Lambda, which is a service that allows running of back end code in response to certain events. The external client access to the web service is through the API Gateway that allows configuration of an IP address to communicate with the external client, e.g., the Jenkins automation server. Many clients can access the web service through the IP at the same time. The API Gateway triggers execution of the code written in Lambda as a result of the Rest commands from external clients. There is also an admin role in the architecture that allows for necessary maintenance of the applications running in EC2.

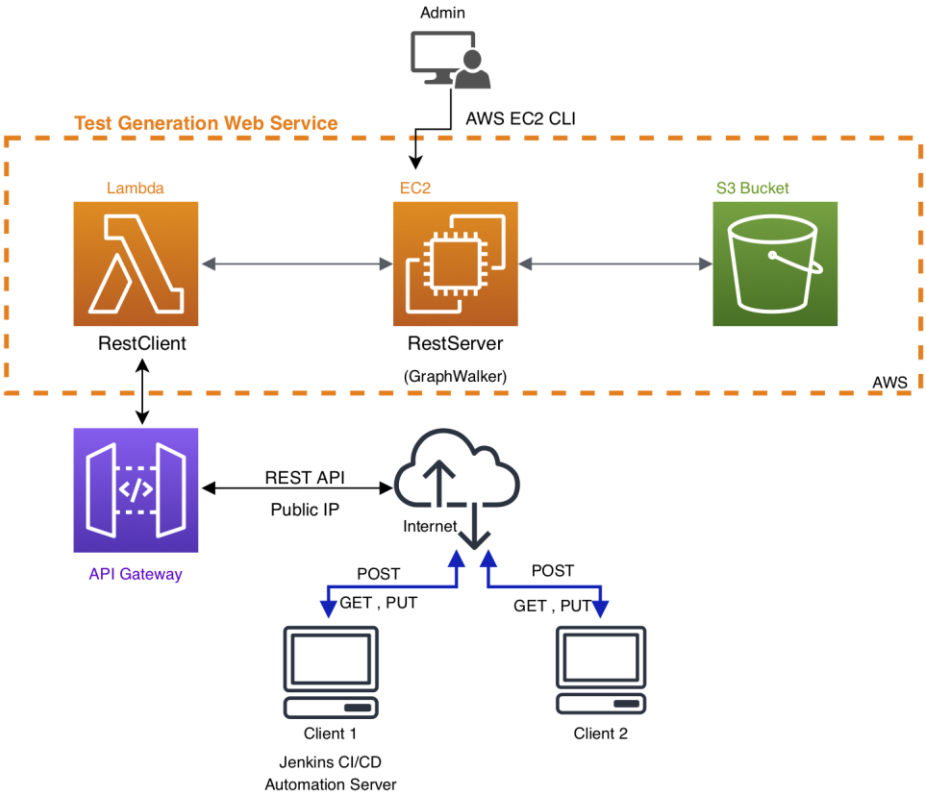

FIGURE 24 VARIANT 1: THE SIMPLE ARCHITECTURE FOR MBT AS A WEB SERVICE

7.2.4.1. Setting up AWS and Realization of the Simple Architecture

`AWS Management Console' is the starting point to access and manage AWS cloud. Here we begin with setting up the virtual machine, an instance of EC2. We installed the Ubunto operating system and assigned certain RAM. We can then access it through the public IP address from, let's say, the local host. Among the different ways to connect to EC2, one is through the SSH client. Using a unique private key provided by AWS, we can connect to EC2. One can now install or run different applications in EC2. We successfully ran GW CLI as a RESTful service controlled through the REST API and after running, GW begins listening to an open port.

In the first architectural variant, as we know, the client is accessible through the AWS Lambda function. In AWS Lambda, we can set our own configurations and settings through a broad selection of languages. We used Python. For communication through HTTP APIs and execution of code in AWS Lambda, we need an API Gateway (an AWS service for creating, deploying and managing RESTful API). For connecting AWS Lambda to EC2, we have used the public IP address. We can do different authorisations in AWS Lambda such as checking names of the client and the authorisation key for identifying a valid user as well as authorisations of GW REST API calls (figure 25).

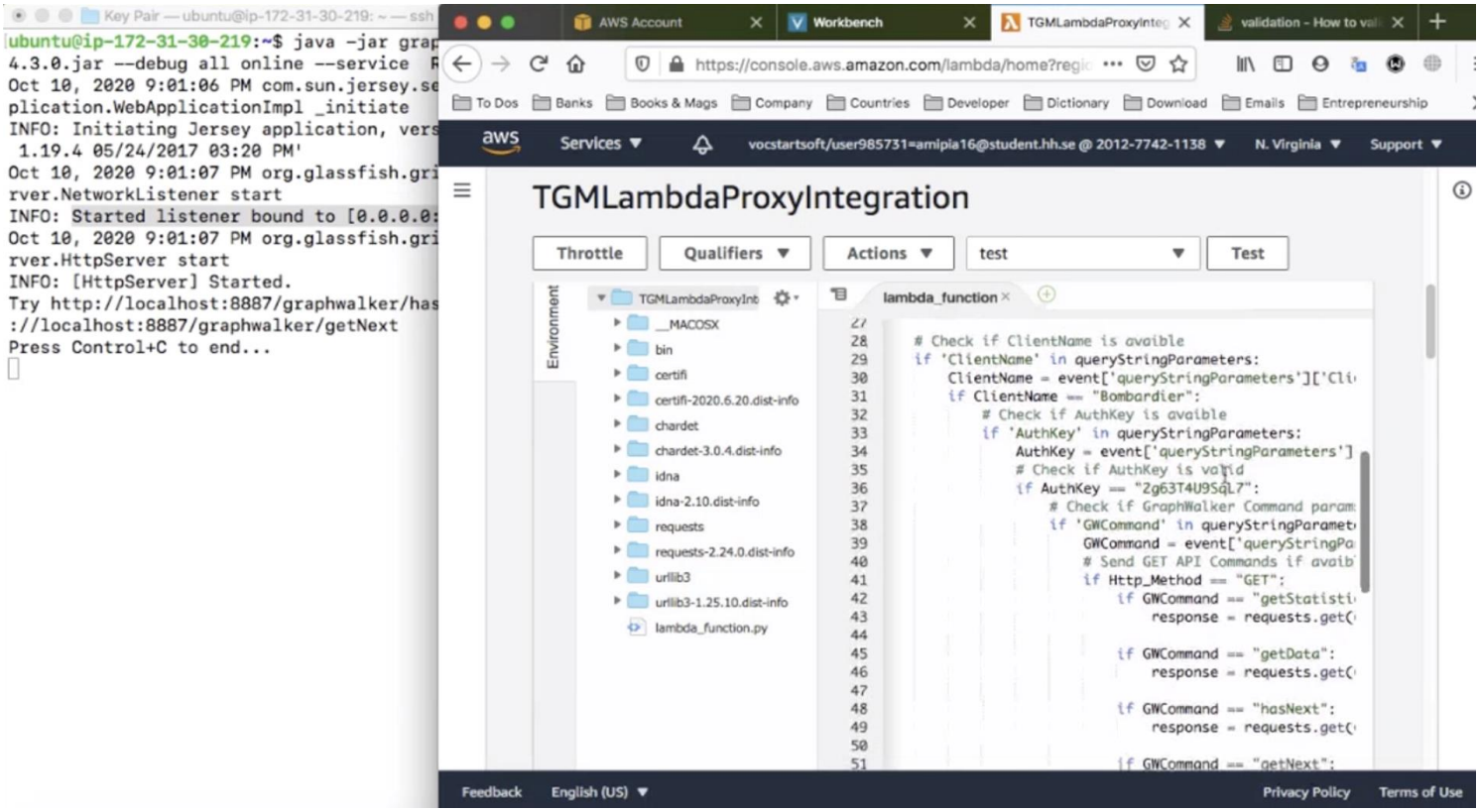

FIGURE 25 AWS LAMBDA CODE FOR AUTHORIZATIONS

Through Postman (https://www.postman.com/), one can make requests to APIs to retrieve data from a data source, or to send data. In Postman, we get a URL that connects the AWS API Gateway to AWS Lambda. We can send requests to AWS Lambda through the Postman and AWS Lambda then responds with a payload that appears inside the Postman user interface. For example, using a load command, we need a post request, authorisation key and the client name. In the Body field, we define the model of the fire indication system that needs to be loaded (figure 26).

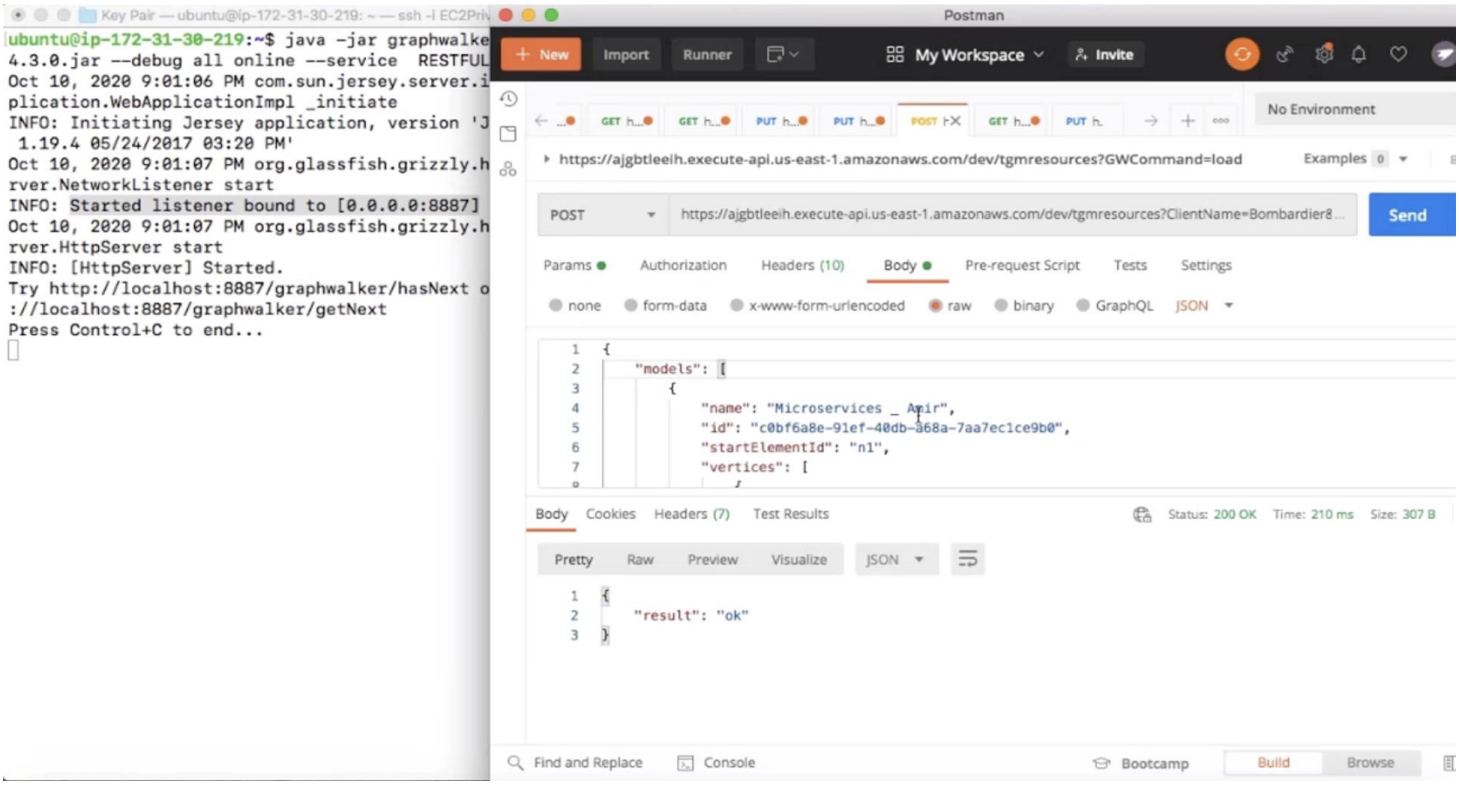

FIGURE 26 THE BODY PARAMETER IN POSTMAN HAVING THE GW-EXPORTED JSON FILE OF THE FIRE INDICATION SYSTEM'S MODEL

This model is in a JSON exported format from GW. Postman allows us to send this post request through a custom built URL. The model gets loaded to EC2 via AWS Lambda and Postman shows the sent payload from GW with a result: ok (figure 27).

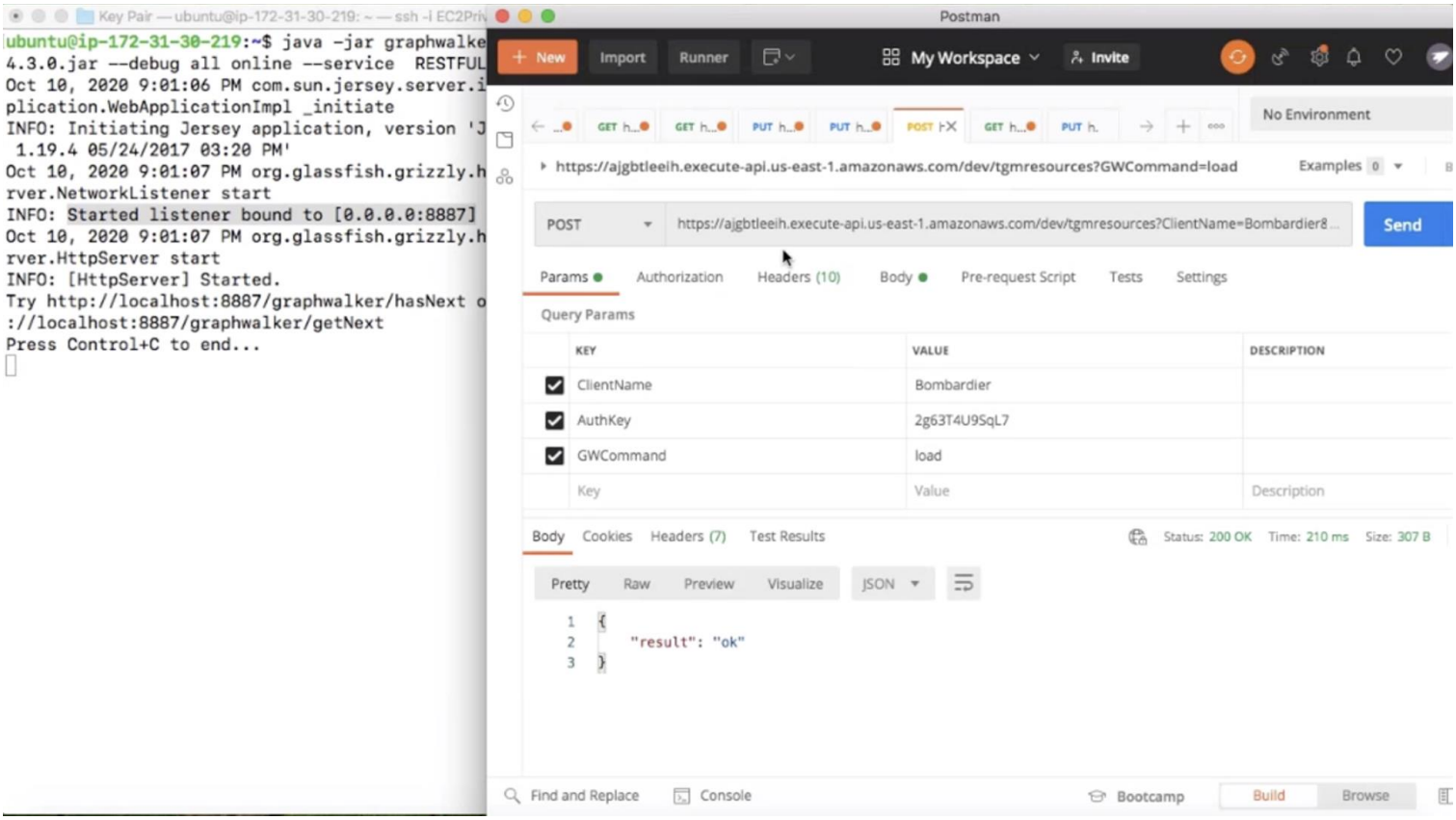

FIGURE 27 THE LOAD REQUEST SENT FROM POSTMAN AND THE RECEIVED PAYLOAD SHOWN IN POSTMAN'S USER INTERFACE

Similarly, through the GET command, we can use the getStatistics and getData commands, send these requests and get back the payloads. Using the PUT request, one can update the requirements, e.g. if we want to update a variable num_of_inputs to a particular value of 32455, we can do that (figure 28). We can then check the updated value using getData.

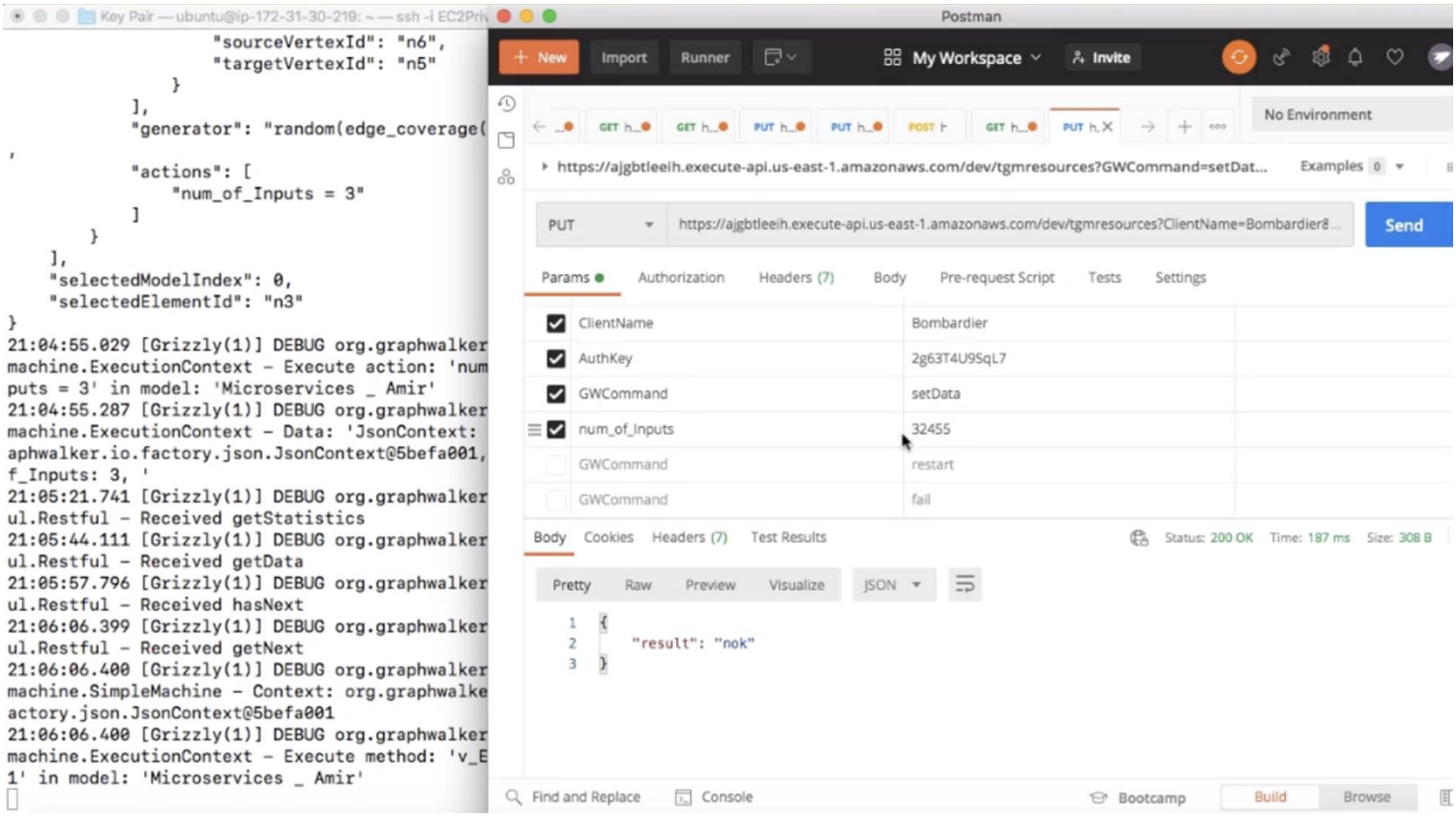

FIGURE 28 THE *PUT* REQUEST SENT FROM THE POSTMAN TO UPDATE *NUM OF INPUTS* TO THE VALUE OF 32455

7.2.5. Variant 2 - The Dynamic Architecture

In the dynamic model (figure 29), while we keep access to RestServer through Lambda, the external client can have direct access to RestServer through a public IP. In this additional communication channel, no coding in Lambda is desired and is suitable when the external client needs to configure settings directly in the virtual machine (RestServer), e.g., to load a model of the SUT without going through Lambda. The public IP is configured in EC2, through which GraphWalker's RESTful commands can be given.

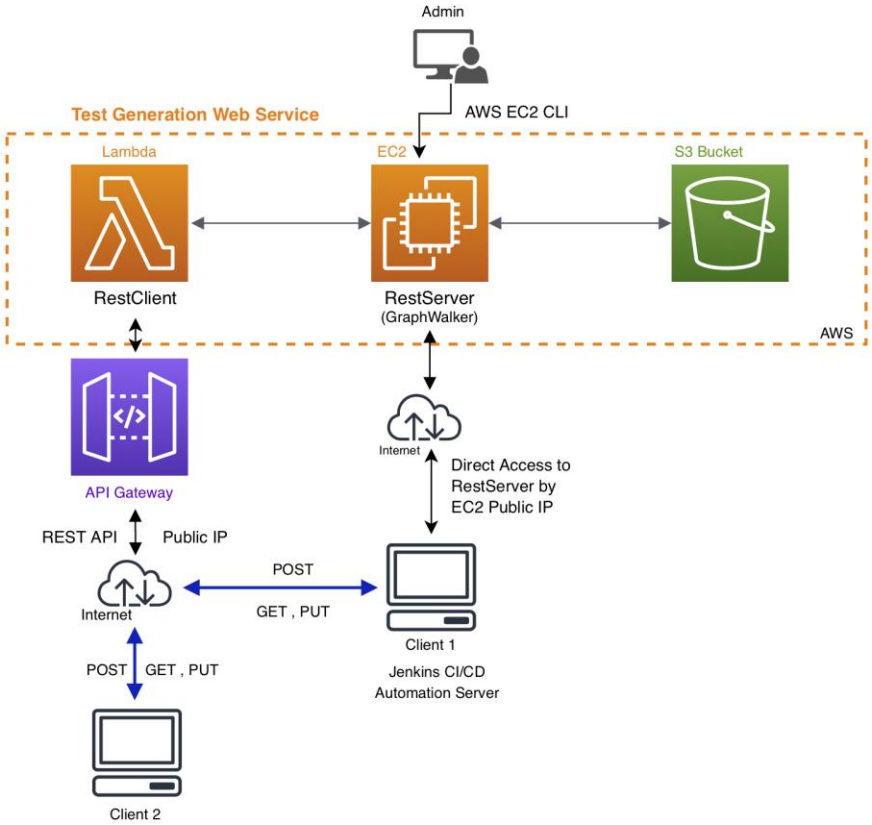

FIGURE 29 VARIANT 2 THE DYNAMIC ARCHITECTURE FOR MBT AS A WEB SERVICE

7.2.6. Realization of Variant 2 - The Dynamic Architecture

In the second dynamic architecture, we can connect the external client with the EC2 server directly, bypassing the need for the REST client (AWS Lambda). This direct access is useful when the external client wants to change certain administrative configurations or settings in the virtual machine directly. We can access EC2 using its public IP, thus bypassing the AWS Lambda function. Using Postman, we can use the public IP to create the URL to e.g. load the model (figure 30). The same is true for other REST API calls such as *getStatistics*.

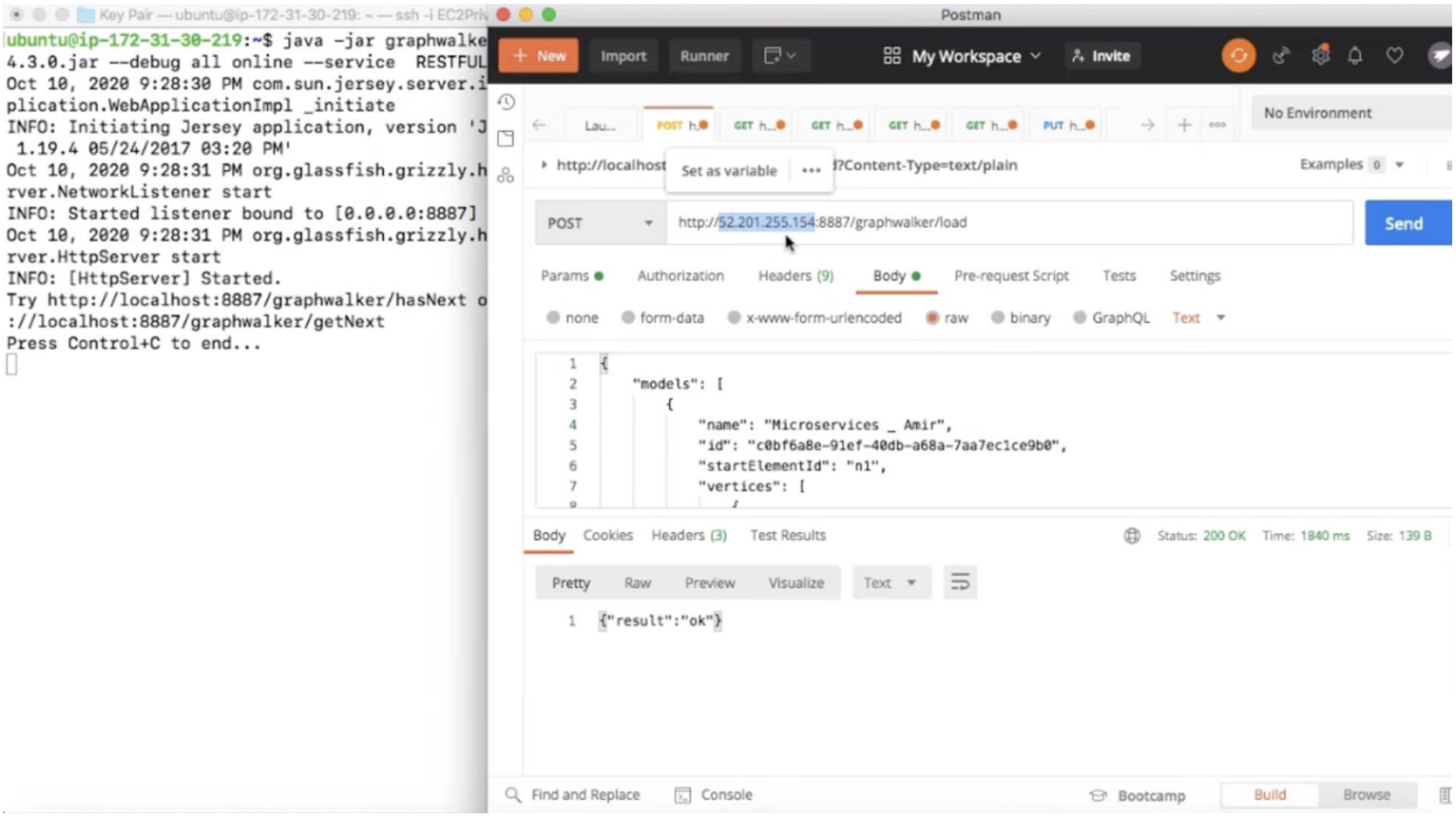

FIGURE 30 LOADING THE JSON FILE USING EC2'S PUBLIC IP IN VARIANT TWO ARCHITECTURES

7.2.7. *Variant 3 - The Advanced Dynamic Architecture*

The advanced dynamic architecture (figure 31) is enhanced with a database service provided by AWS, the DynamoDB. The Lambda function is configured to save test execution statistics to DynamoDB. For example, the RESTful command of *getStatistics* returns numbers on edge coverage, vertex coverage, total number of visited edges and nodes, among other things; these data can be stored in DynamoDB, along with the date, time and the client name. Once the model is uploaded, a call to *getStatistics* can return the desired data that can be stored in DynamoDB, even if the test generation is in progress. This data can then be analyzed in different ways to assess the test generation process. DynamoDB allows different functions such as export as an excel file or to query the table with different conditions and sorting capabilities. Thus, the test engineer has complete control over the test generation data to analyze.

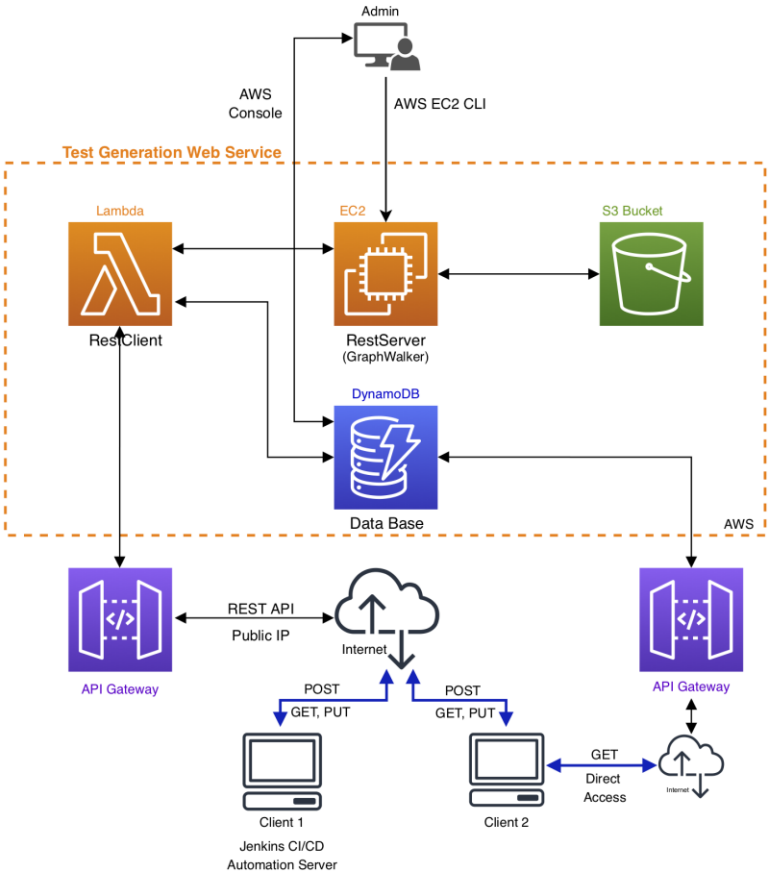

FIGURE 31 VARIANT 3 THE ADVANCED DYNAMIC ARCHITECTURE FOR MBT AS A WEB SERVICE

7.2.8. Realization of Variant 3 - The Advanced Dynamic Architecture

In the advanced dynamic architecture, we added a database, realized through Amazon DynamoDB. The results of the test execution (e.g. requested through *getStatistics*) AWS Lambda can save data to DynamoDB with the date, time and client's name. We can then access the recorded data from DynamoDB and perform different analyses. A table in DynamoDB can be created using the console or through code in the AWS Lambda function (figure 32).

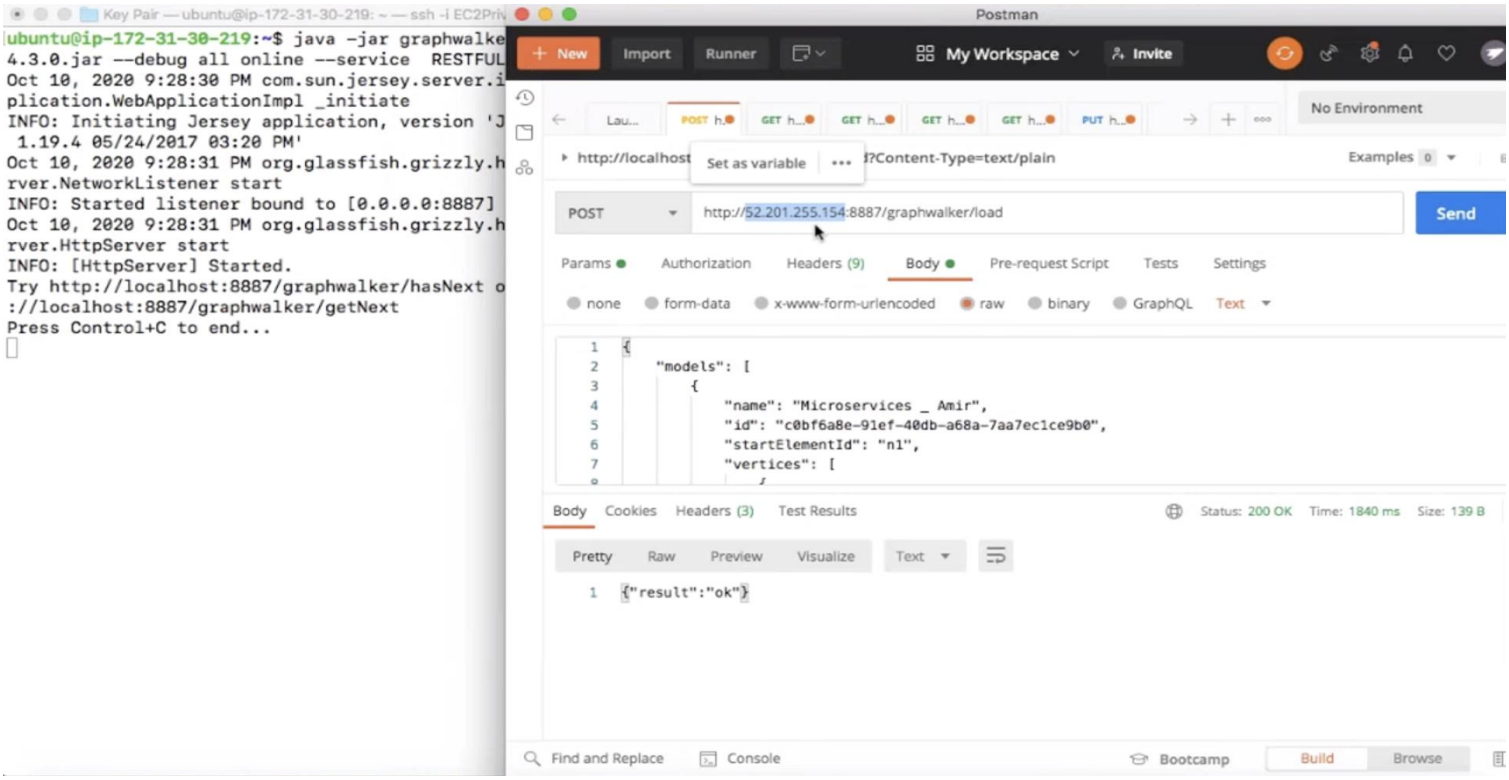

FIGURE 32 CREATING AN AMAZON DYNAMODB TABLE THROUGH AWS LAMBDA

Now after loading the JSON model file to EC2, we can request getStatistics at any point in the test execution. The corresponding data of all the variables get saved in the table (figure 33).

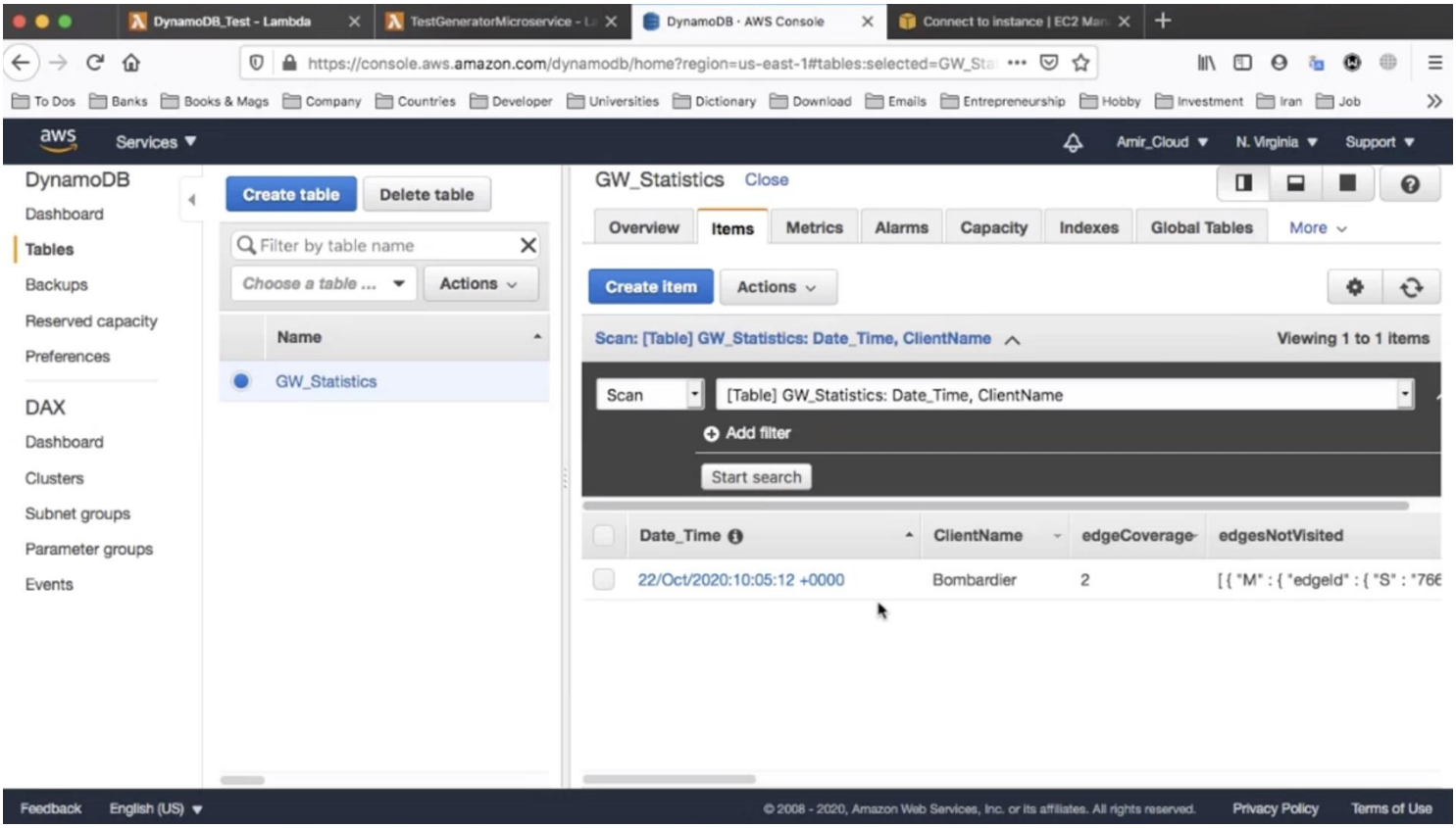

FIGURE 33 RECORDING OF THE RESULT OF *GETSTATISTICS* IN THE AMAZON DYNAMODB TABLE

7.2.9. Variant 4 - The HTML Server Enhanced Dynamic Architecture

The HTML server enhanced dynamic architecture represents the last variation of MBT as a web service architecture (figure 34). The HTML server acts as a front-end for the external client. The Lambda function is enhanced with HTML code, with some user interface elements such as buttons. Login credentials can also be given through the same HTML interface. The loading of the model of the SUT can also be done through the interface along with triggering of RESTful commands such as *hasNext*, *getNext*, *getData*, among others. The response of each of these commands also be viewed in the implemented interface. The test engineer is further given the option to use the RESTful command *setData* to change a particular variable's value. After having set the value, a call to RESTful command *restart* will reset the model to its initial state and a call to *fail* stops the path generation.

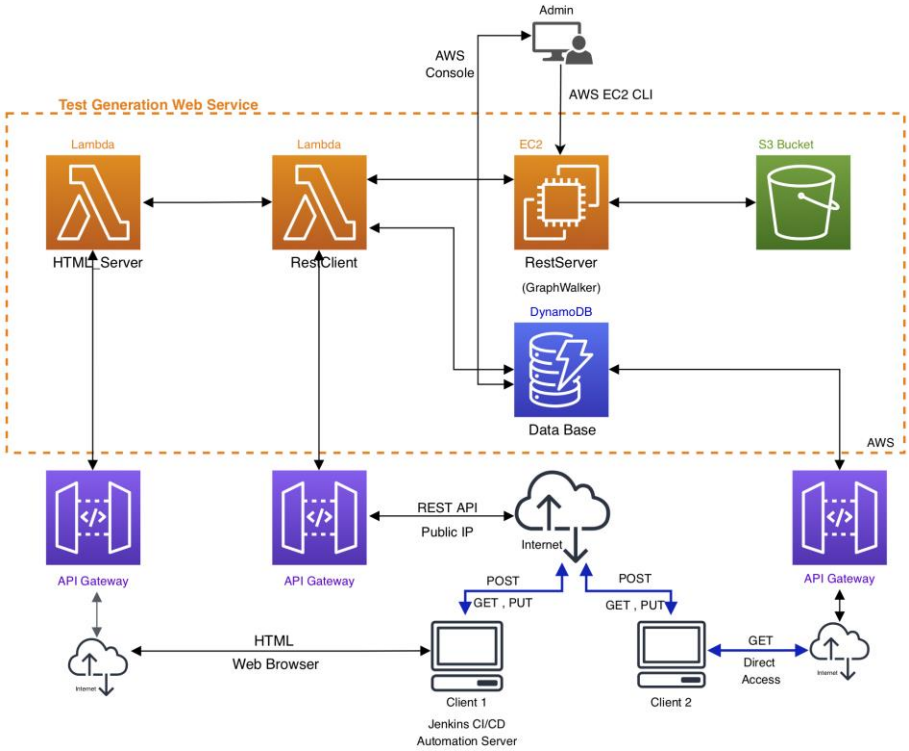

FIGURE 34 VARIANT 4 THE HTML SERVER ENHANCED DYNAMIC ARCHITECTURE FOR MBT AS A WEB SERVICE

7.2.9.1. Realization of Variant 4 - The HTML Server Enhanced Dynamic Architecture

The last variant is advanced with a HTML server, acting as a front end to the end user. If, e.g., the user wants to upload different models, HTML server can assist with a web interface, making the operation easy and user friendly. The html code for the user interface is written in AWS Lambda (figure 35), e.g., buttons, text fields, tabs etc.

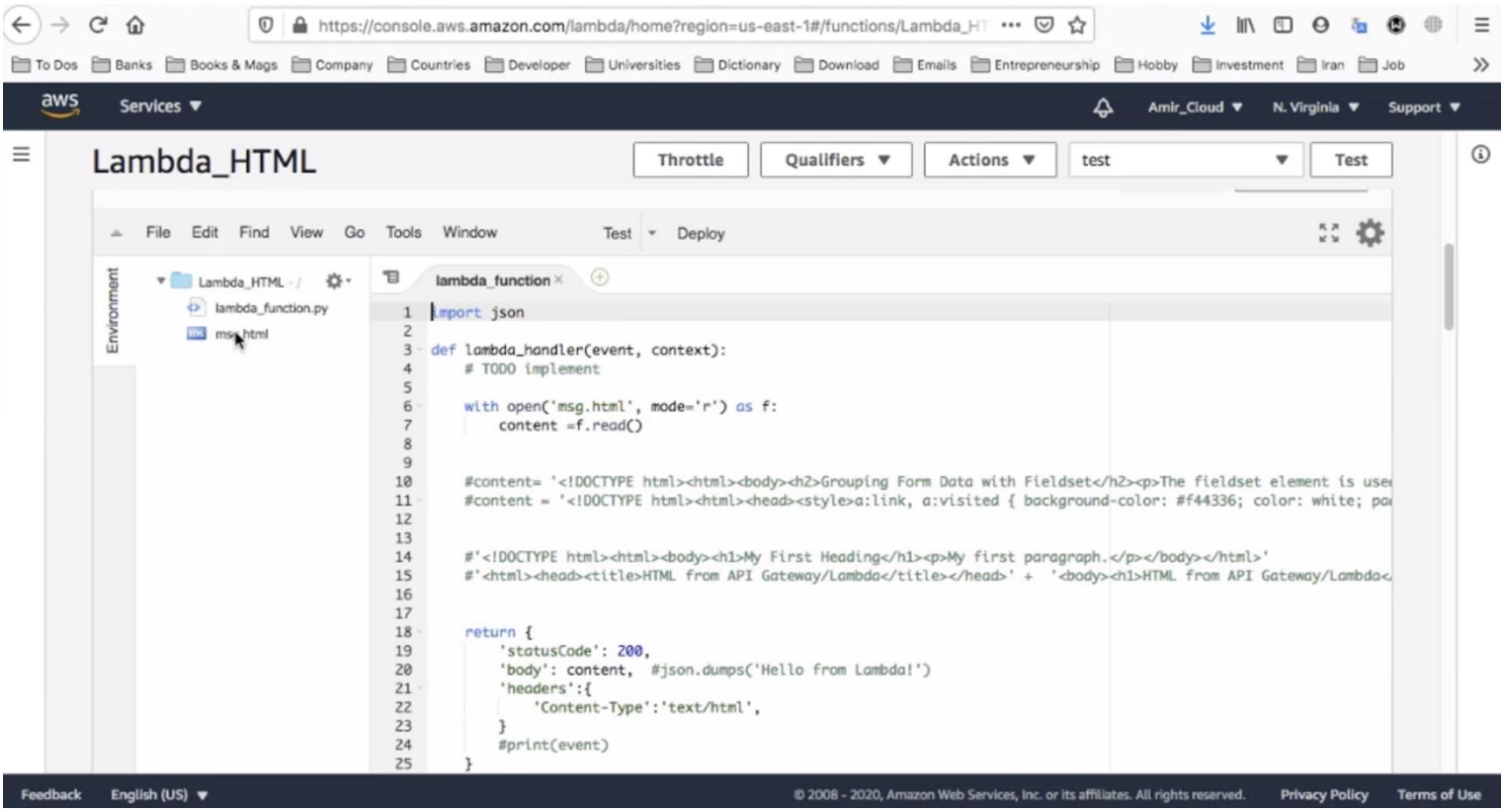

FIGURE 35 AWS LAMBDA CODE FOR INCLUDING THE HTML FRONT-END

One can put different information in the tab while also adding fields for user login to access the web page. Once can upload a model, while can also place requests through REST commands by simply clicking buttons, e.g., for *hasNext*, *getNext*, *getData* and *getStatistics* (FIGURE 36). The resulting payload is displayed on the web page as well as recorded in DynamoDB.

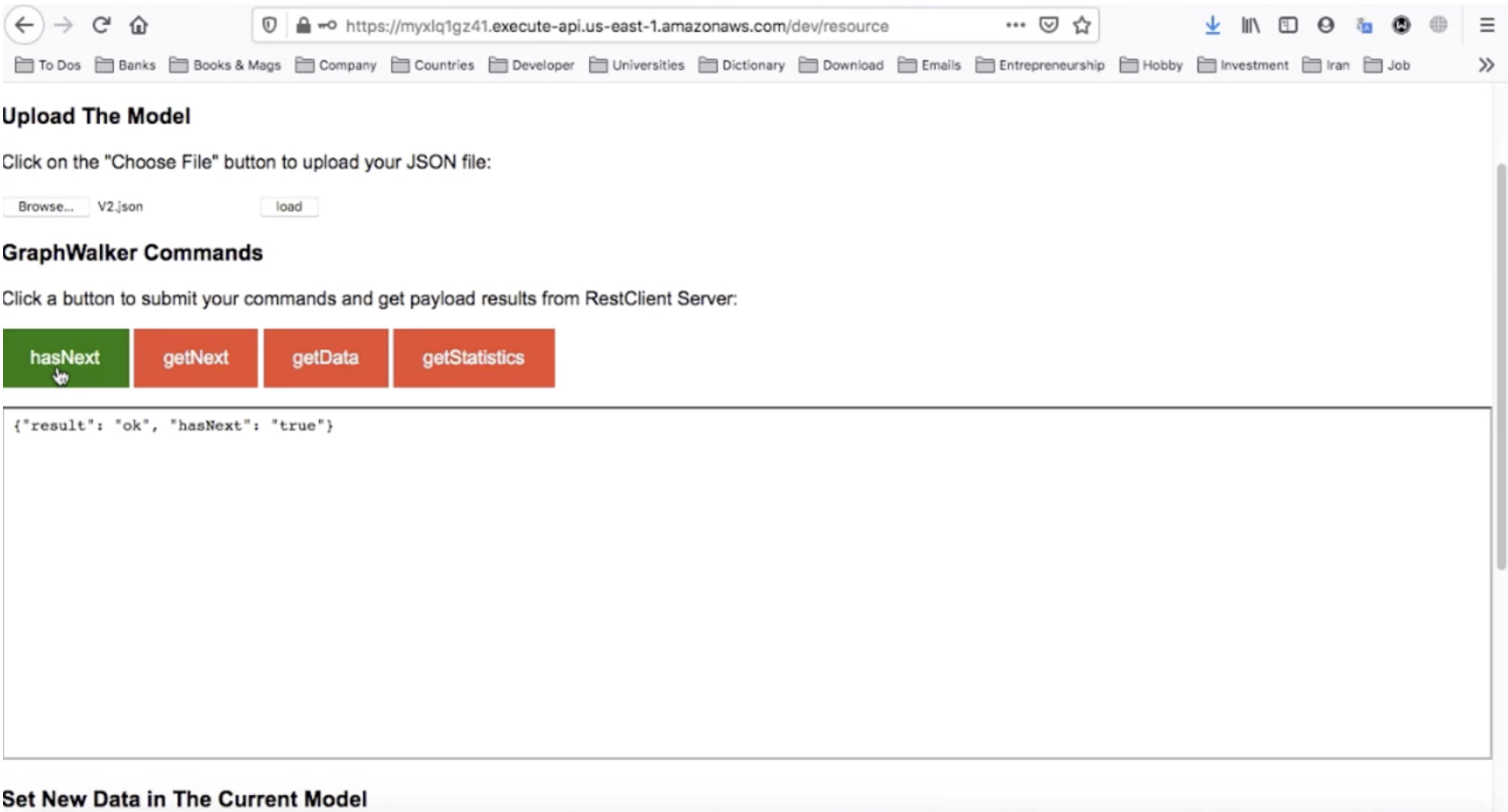

FIGURE 36 THE HTML FRONT-END OF VARIANT 4.

One can also set the value of a certain variable through the interface by calling *setData*. For example, in figure 37, the user can set the value of *FDCU2Valid* to true. Similarly, the user interface also provides for restarting and terminating the execution.

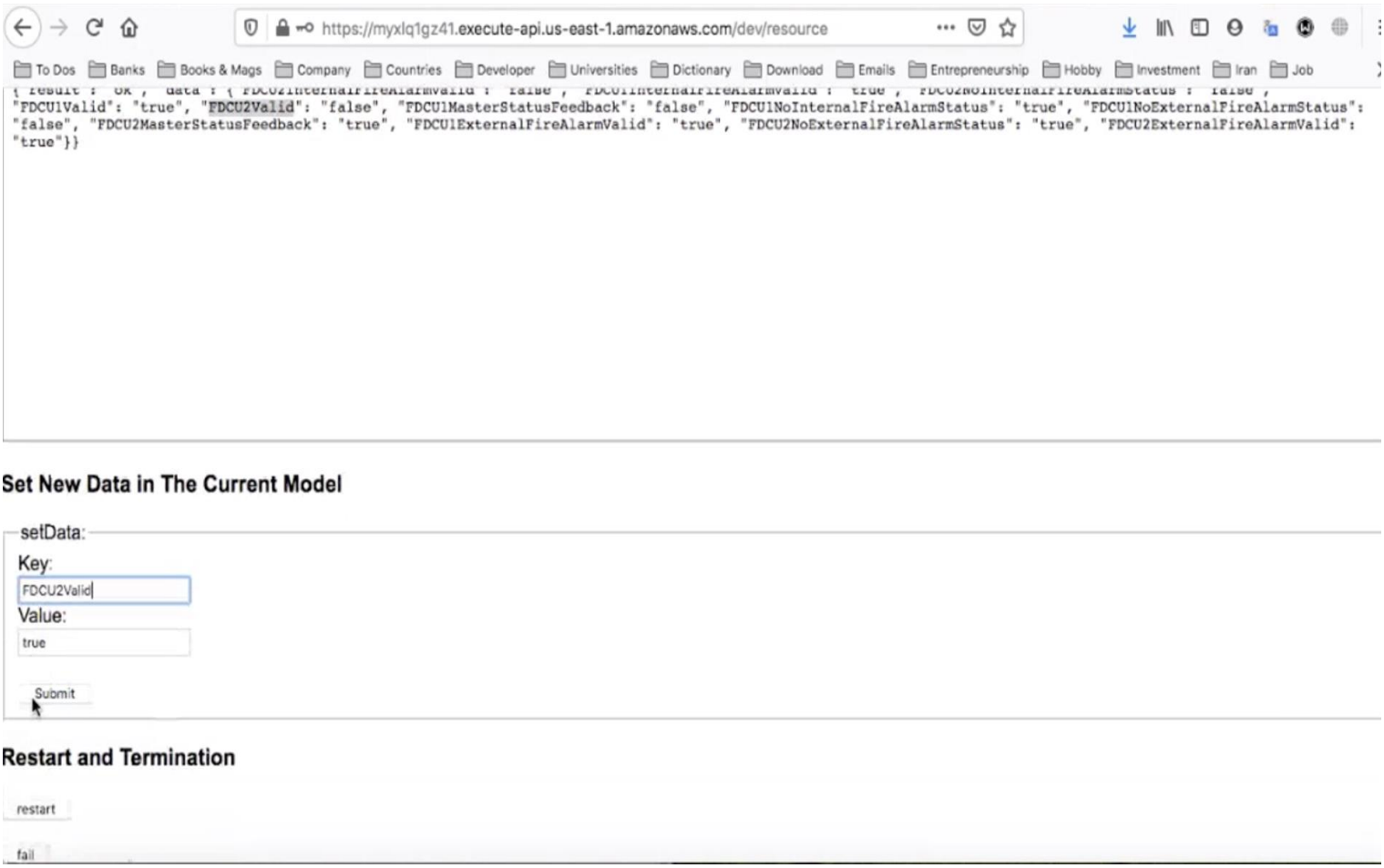

FIGURE 37 SETTING THE VALUE OF *FDCU2VALID* TO TRUE THROUGH THE HTML USER INTERFACE

8. Conclusion and Future Work

In this deliverable we report the Agile Testing Framework compatible with the ADEPTNESS infrastructure in order to perform the multi-level test execution. A prototypical tool for integrating the ecosystem with the use-cases has been also developed. For instance, we assessed the feasibility of executing multi-level tests within the ADEPTNESS ecosystem by developing the external tool compatible with Elevate. For orona's use-case, the external tool for the execution of tests at the HiL test level will also be developed within WP6, allowing this way the multi-level test execution of tests.

Report on test benches for simulation-based multi-level testing as a service of CPSoS

Acknowledgments

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

Disclaimer

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

Bibliography

[1] C. Richardson, "Pattern: Health Check API," 2020. [Online]. Available: https://microservices.io/patterns/observability/health-check-api.html.

[2] V. Authors, "What is REST," 2020. [Online]. Available: https://restfulapi.net/. [Accessed 02 06 2020].

[3] DZone, Inc. dba Devada, "REST API versioning strategies," 18 12 2019. [Online]. Available: https://dzone.com/articles/rest-api-versioning-strategies. [Accessed 02 06 2020].

[4] SmartBear Software, "OpenAPI Specification," 2020. [Online]. Available: https://swagger.io/specification/. [Accessed 19 05 2020].

[5] AsyncAPI Initiative, "AsyncAPI - Getting Started," [Online]. Available: https://asyncapi.io/docs/getting-started/. [Accessed 19 05 2020].

[6] M. Fowler, "Microservices Guide," 21 08 2019. [Online]. Available: https://martinfowler.com/microservices/.

[7] Adeptness Consortium, "Adeptness D1.1 - Requirements & Ethics," 2020.

[8] OMG, UML Testing Profile 2 (UTP 2), 2018. [Online]. Available: https://www.omg.org/spec/UTP2/2.0/PDF

[9] A. Rahman and D. Amyot, "A DSL for importing models in a requirements management system," in 2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE). IEEE, 2014.

[10] 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 (ISEC), 2021. [Online]. Available: http://www.es.mdh.se/publications/6101-

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

[12] M. N. Zafar, W. Afzal, E. P. Enoiu, A. Stratis, and O. Sellin, “A model-based test script generation framework for embedded software,” in The 17th Workshop on Advances in Model Based Testing (A-MOST), 2021. [Online]. Available: http://www.es.mdh.se/publications/6172-

[13] D. Mane, “Testbed platform: Amazon web services for library,” in Cloud Technology: Concepts, Methodologies, Tools, and Applications, M. Khosrow-Pour, Ed. IGI Global, 2014, ch. 34, pp. 730-748.

[14] Broekman, B., & Notenboom, E. (2003). Testing embedded software. Pearson Education.

[15] Utting, M., & Legeard, B. (2010). Practical model-based testing: a tools approach. Elsevier.

[16] Bussenot, R., Leblanc, H., & Percebois, C. (2018, June). Orchestration of Domain Specific Test Languages with a Behavior Driven Development approach. In 2018 13th Annual Conference on System of Systems Engineering (SoSE) (pp. 431-437). IEEE.

[17] Micallef, M., & Colombo, C. (2015, April). Lessons learnt from using DSLs for automated software testing. In 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 1-6). IEEE.

[18] Völter, M. (2006). Model-Driven Development with openArchitectureWare.

[19] Olajubu, O. (2015, August). A textual domain specific language for requirement modelling. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp. 1060-1062).

[20] Shokry, H., & Hinchey, M. (2009). Model-based verification of embedded software.

[21] Barney, G., & Al-Sharif, L. (2015). Advanced analysis of down peak. In Elevator Traffic Handbook (pp. 348-358). Routledge.

[22] Ayerdi, J., Segura, S., Arrieta, A., Arratibel, G. S., & Arratibel, M. (2020, October). QoS-aware Metamorphic Testing: An Elevation Case Study. In 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE) (pp. 104-114). IEEE.

[23] Schulze, C., Ganesan, D., Lindvall, M., Cleaveland, R., & Goldman, D. (2014, May). Assessing model-based testing: an empirical study conducted in industry. In Companion Proceedings of the 36th International Conference on Software Engineering (pp. 135-144).