

Model-Based Testing as a Service for IoT Platforms

Abbas AHMAD^{1,2}, Fabrice BOUQUET², Elizabeta FOURNERET³, Franck LE GALL¹, and Bruno LEGEARD^{2,3}

¹ Easy Global Market, Sophia-Antipolis, France

² Université de Franche-Comté - Femto-ST, Besançon, France

³ Smartesting Solutions & Services, Besançon, France

Abstract. *The Internet of Things (IoT)* has increased its footprint becoming globally a 'must have' for today's most innovative companies. Applications extend to multitude of domains, such as smart cities, healthcare, logistics, manufacturing, etc. Gartner Group estimates an increase up to 21 billion connected things by 2020. To manage 'things' heterogeneity and data streams over large scale and secured deployments, IoT and data platforms are becoming a central part of the IoT. To respond to this fast growing demand we see more and more platforms being developed, requiring systematic testing. Combining Model-Based Testing (MBT) technique and a service-oriented solution, we present *Model-Based Testing As A Service (MBTAAS)* for testing data and IoT platforms. In this paper, we present a first step towards MBTAAS for data and IoT Platforms, with experimentation on FIWARE, one of the EU most emerging IoT enabled platforms.

Keywords: Model Based Testing, Testing As A Service, Internet of Things, Standard Compliance

1 Introduction

Internet of Things (IoT) applications can be found in almost all domains, with use cases spanning across areas such as healthcare, smart homes/buildings/cities, energy, agriculture, transportation, etc. **FIWARE** [4] is an emerging IoT platform, funded by the European Union (EU), which is pushing for an ecosystem providing APIs and open-source implementations for lightweight and simple means to gather, publish, query and subscribe context-based, real-time "things" heterogeneous information. This independent community includes 60 cities across Europe in the Open and Agile Smart Cities alliance, which adopted FIWARE standardised APIs to avoid vendor lock-in of proprietary solutions.

FIWARE provides an enhanced Open Stack-based cloud environment including a rich set of open standard APIs that make it easier to connect to the heterogeneous IoTs, process and analyse Big Data and real-time media or incorporate advanced features for user's interaction. These platforms, strongly dependent on the cloud, need to be properly tested to cover all necessary points to achieve

success in their adoption. Scalability, security, performance, conformance, interoperability are the main points that must be ensured, for instance through white, gray or blackbox testing [13]. Moreover, as IoT is an emerging technology re-tooling and changes are frequent, requiring reducing the cost of testing by rethink the way of testing. In this context, Model-Based Testing (MBT) offers tool and language independence thus aiming to lower the testing effort of IoT [3].

In this paper, we focus on Model-Based Testing in terms of conformity and interoperability. We demonstrate through the FIWARE [4] case study that for these purposes, MBT as a Service is a suitable and scalable approach for testing IoT Platforms. We introduce the basic concepts of MBT and how it applies for the testing of IoT systems as a Service. Indeed, most recent IoT platforms are using standardized protocols to communicate (MQTT, CoAP, HTTP). This makes MBT testing deployment very suited by enabling design of a generic model, based on these standards and producing test cases that can be used over multiple applications.

FIWARE has defined its own standard starting from the Open Mobile Alliance Next Generation Services Interface (*NGSI*) standard [2] in order to make the development of future internet applications easier and interoperable. This standard is used as part of general-purpose platform functions available through APIs, each being implemented on the platform, noted as GEi (Generic Enabler Implementation). We used our MBT solution to increase confidence in the development of the FIWARE platform applications. We show through our experiment on the FIWARE Orion Context Broker [4] how it indeed helps developers to create high quality applications.

The paper is organised as follows. Section 2 poses the challenges of testing IoT platforms and the context of our approach, MBT and FIWARE. Section 3 defines our approach for MBT as a Service for IoT platforms testing. Section 4 summarizes the results and lessons learnt on the case study. We discuss related works in Section 5. Finally we conclude and provide a roadmap for future works in Section 6.

2 Challenges and Context of Testing IoT Platforms through FIWARE

We identify the challenges and define the context for our approach based on the analysis of the FIWARE IoT Platform, which is a perfect representative for this domain due to its presence in the European market. We further applied MBT to FIWARE, which was already an ongoing project. We illustrate the MBT approach and the test generation later in Sections 2.2 and 2.3.

2.1 The FIWARE IoT Platform

The **FIWARE** [4] cloud and software platform is the perfect catalyst for an open ecosystem of entrepreneurs aiming at developing state-of-the-art data-driven applications. This ecosystem is formed by application developers, technology and

infrastructure providers and entities that aim to leverage the impact of developing new applications based on the produced data. Building applications based on FIWARE is intended to be quick and easy thanks to the use of pre-fabricated components in its cloud, sharing their own data as well as accessing "open" data. However, the challenge remains to build developers' trust and confidence into this FIWARE underlying platform. This is achieved by setting up quality assurance (QA) processes relying on effective testing of the platform. This raises questions such as balancing of test coverage with time and cost. However, several questions arise when testing IoT platforms with respect to the specificities of the communication protocols, devices and the heterogeneity of the data.

Connecting "things" as devices, requires to overcome a set of problems arising in the different layers of the communication model. Using devices' produced data or responding to device's requests requires interacting with an heterogeneous and distributed environment of devices running **several protocols** (such as HTTP, MQTT, COAP) , through multiple wireless technologies.

Devices have a lot of particularities so it is not feasible to provide a testing solution where one size fits all. Devices are resource constrained and cannot use full standard protocol stacks: they cannot transmit information too frequently due to battery drainage, they are not always reachable due to wireless connection based on low duty-cycles, their communication protocols are IoT specific and lack integrated approach [3] and use **different data encoding languages**, which makes global deployment hardly existing.

Developers face complex scenarios where **merging the information** is a real challenge. For this reason, an IoT platform must enable intermediation and data gathering functions to deal with devices variety and it must be configurable and adaptable to market needs.

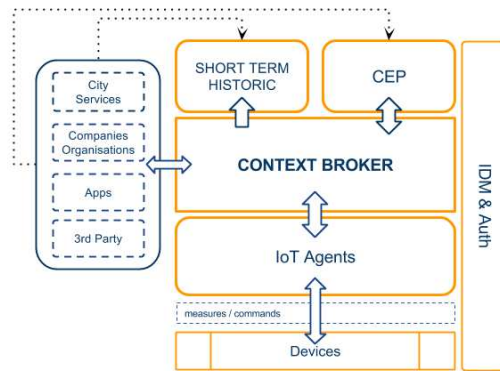


Fig. 1. FIWARE IoT platform Architecture

Figure 1 exposes the FIWARE architecture that deals with all these specific elements, which remain a challenge when testing them in a real situation. More specifically, the figure illustrates a connector (IoT Agent) solving the issues of heterogeneous environments where devices with different protocols are translated into to a common data format: NGSI. While several enablers are im-

proving the capacities of the platform to manage stored information (security tools, advanced data store models, historical retrieval of information, linkage to third party applications. . .), a core component known as Orion Context Broker allows to gather and manage context information between data producers and data consumers at large scale. This context broker is at the centre of the exposed MBT based evaluation.

2.2 Introducing MBT in FIWARE

FIWARE was an ongoing project facing testing problems when we introduced MBT to its community. The introduction required to adapt the existing FIWARE testing process. In Fig. 2, we illustrate this introduction through the FIWARE use-case: **Orion Context Broker**. The focus is to bring examples over the method `registerContext`. This method enables the registration of a new "thing" in the Context broker implementation.

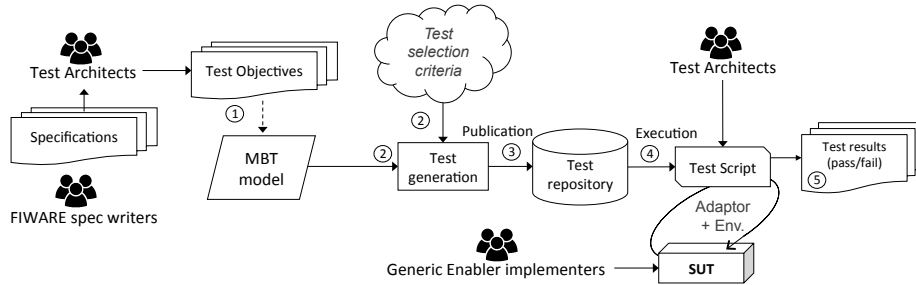


Fig. 2. FIWARE MBT Process

Classically an MBT process [17] includes activities such as test planning and controls, test analysis and design (which includes MBT modelling, choosing suitable test selection criteria), test generation, test implementation and execution [11]. Figure 2 illustrates the classical MBT process applied to FIWARE. The test analyst takes requirements and defines tests objectives as input to model the System Under Test (SUT) (step ①). This MBT model contains static and dynamic views of the system. Hence, to benefit as much as possible from the MBT technology, we consider an automated process, where the test model is sent as an input to the test generator that automatically produces abstract test cases and a coverage matrix, relating the tests to the covered model elements or according to another test selection criteria (step ②). These tests are further exported, automatically (step ③), in a test repository to which test scripts can be associated. The automated test scripts in combination with an adaptation layer link each step from the abstract test to a concrete command of the SUT and automate the test execution (step ④). In addition, after the test execution, tests results and metrics are collected (step ⑤) and feedback is sent to the user.

From one MBT model different test selection criteria exist to drive the test generation approach [17]. In Fig. 2 we illustrate the general FIWARE MBT process that is based on different test selection criteria, depending on the tool being used.

In our approach for compliance testing we used the Smartesting CertifyIt tool [12], as it has already shown its benefits in compliance testing [7]. The CertifyIt tool uses coverage-based test selection criteria (see Section 2.3) and it considers, among others, UML class and object diagrams to develop MBT models. Each type has a separate role in the test generation process. The class diagram describes the system's structure, namely the set of classes that represents the static view of the system: (i) Its entities, with their attributes, (ii) Operations that model the API of the SUT, (iii) Observations (usually denoted as *check* operations) that serve as oracles, for instance an observation returns the current state of the user's connection on a web site.

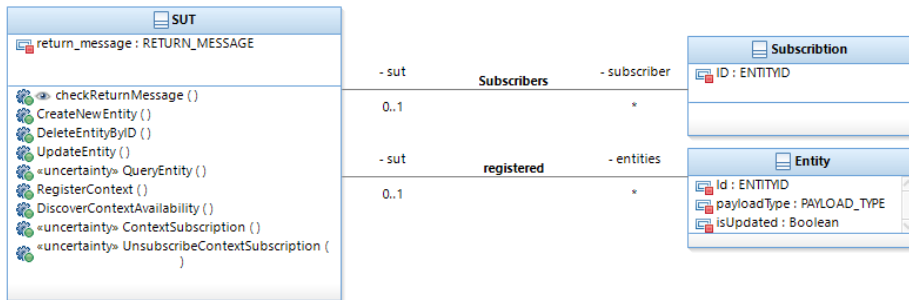


Fig. 3. MBT Orion Context Broker UML model - static view (Class diagram)

The MBT model given in Fig. 3 shows the architecture of the SUT - the Orion Context Broker. The classes have attributes and functions. For instance in the *SUT* class we model respectively `return_message`, the SUT response to the sent messages and the function `RegisterContext`, that registers the entities (for instance a sensor) .

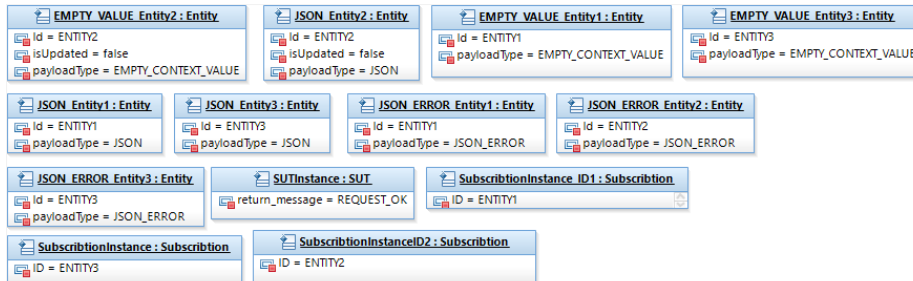


Fig. 4. MBT Orion Context Broker UML model - input data (Object Diagram)

Next, from the previous class diagram we instantiate an object diagram (Fig. 4). This data view provides the initial state of the system and also the objects that will be used for test generation as input data for the operation parameters.

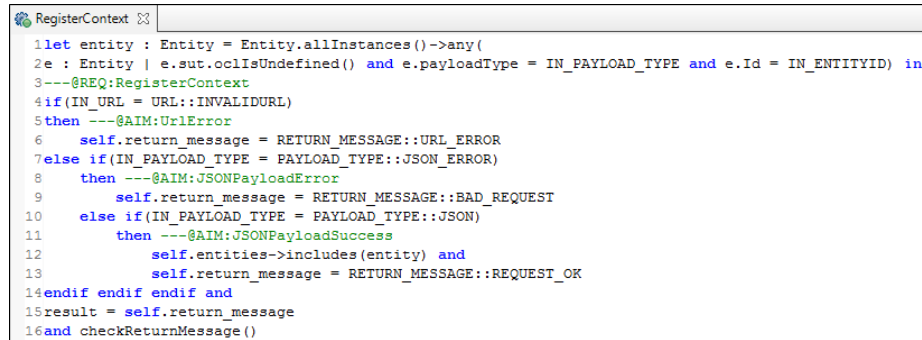
The dynamic view of the system or its behaviours are described by Object Constraint Language (OCL) constraints written as pre/postcondition in opera-

6

tions in a class (Fig. 5). The test generation engine sees these behaviour objects as test targets. The operations can have several behaviours, identified by the presence of the conditional operator if-then-else. The precondition is the union of the operation's precondition and the conditions of a path that is necessary to traverse for reaching the behaviour's postcondition. The postcondition corresponds to the behaviour described by the action in the **then** or **else** clause of the conditional operator. Finally, each behaviour is identified by a set of tags, which refers to a requirement covered by the behaviour. For each requirement, two types of tags exist:

- @REQ - a high-level requirement
- @AIM - its refinement

Both tags are followed by an identifier. Figure 5 shows the OCL code for the Register Context method. The high level requirement is to test the registration of an entity. Its refinement is the case where an error in the destination URL can be found, an error in the payload sent or its success.



```

1 let entity : Entity = Entity.allInstances()->any(
2 e : Entity | e.sut.oclisUndefined() and e.payloadType = IN_PAYLOAD_TYPE and e.Id = IN_ENTITYID) in
3 ---@REQ:RegisterContext
4 if (IN_URL = URL::INVALIDURL)
5 then ---@AIM:UrlError
6   self.return_message = RETURN_MESSAGE::URL_ERROR
7 else if (IN_PAYLOAD_TYPE = PAYLOAD_TYPE::JSON_ERROR)
8   then ---@AIM:JSONPayloadError
9     self.return_message = RETURN_MESSAGE::BAD_REQUEST
10  else if (IN_PAYLOAD_TYPE = PAYLOAD_TYPE::JSON)
11    then ---@AIM:JSONPayloadSuccess
12      self.entities->includes(entity) and
13      self.return_message = RETURN_MESSAGE::REQUEST_OK
14 endif endif endif and
15 result = self.return_message
16 and checkReturnMessage()

```

Fig. 5. Orion Context Broker "Register Context" method OCL - dynamic view

Deducing the test oracle from our model is a major advantage of the used tool (Smartesting CertifyIt). A specific type of operations, called observations are used to assign the test verdict. The tester with these special operations can define the system parts or variables to observe, for instance a function `checkReturnMessage()`. Thus, based on these observations, the test oracle is automatically generated for each test step, based on the `return_message` variable expected and actual (from the execution) result..

2.3 Test generation with CertifyIt

The CertifyIt tool uses the object diagram and the OCL constraints to extract automatically a set of test targets. The test targets are used to drive the test generation process. As discussed previously each test has a set of tags associated for which it will ensure the coverage.

Figure 6 illustrates a test target associated to the success behaviour depicted in Fig. 5 and its corresponding test in the CertifyIt tool. The tool lists all generated tests clustered per covered requirement. We can visualize the test case and

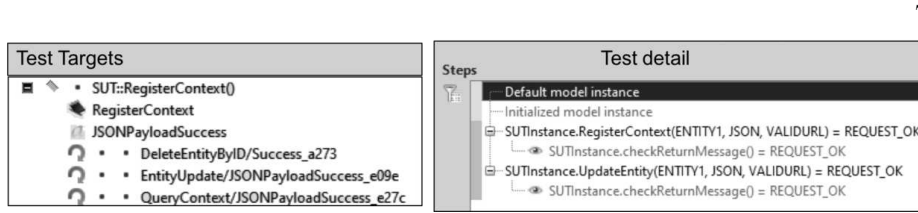


Fig. 6. Register Context test case

for each step a test oracle is generated. As discussed, the tester with the observation manually defines the system point to observe when calling any function. As we can see on Fig. 6, `checkReturnMessage()` observes the return code of each Orion Context Broker function with respect to the activated requirement. One test covers one or more test targets. Moreover, the tool's engine generates fewer tests than test targets, because it uses the *light merge* of tests method, which considers that one test is covering one or more test objectives (all test objectives that have been triggered by the test steps) [8]. The *light merge* of tests means that the generator will not produce separate tests for the previously reached test targets but will consider the test targets as covered by a specific test. The generated tests are exported as an XML file gathering the description of each test suite with its test cases containing parametrized abstract test steps. Generated tests are abstract and to execute them on the system an adaptation layer is required, as classically done in MBT.

3 MBTAAS for IoT platforms testing

IoT platforms offer services to applications users. The question of conformance testing and validation of IoT platforms can be tackled with the same *"as a service"* approach. This section presents the general architecture of our **Model Based Testing As a Service** (MBTAAS). We then present in more details, how each service works individually in order to publish, execute and present the tests/results.

3.1 Architecture

An overview of the general architecture can be found in Fig. 7. In this figure we find the four main steps of the MBT approach (MBT modeling, test generation, test implementation and execution) presented in Section 2.2.

However, to the difference of a classical MBT process, MBTAAS implements several webservices, which communicate with each other in order to realise testing steps. A webservice, uses web technology such as HTTP, for machine-to-machine communication, more specifically for transferring machine readable file formats such as XML⁴ and JSON⁵.

In addition to the classical MBT process, the central piece of the architecture is the database service ⑤ that is used by all the other services. We will see its

⁴ <https://www.w3.org/XML/>

⁵ <http://www.json.org>

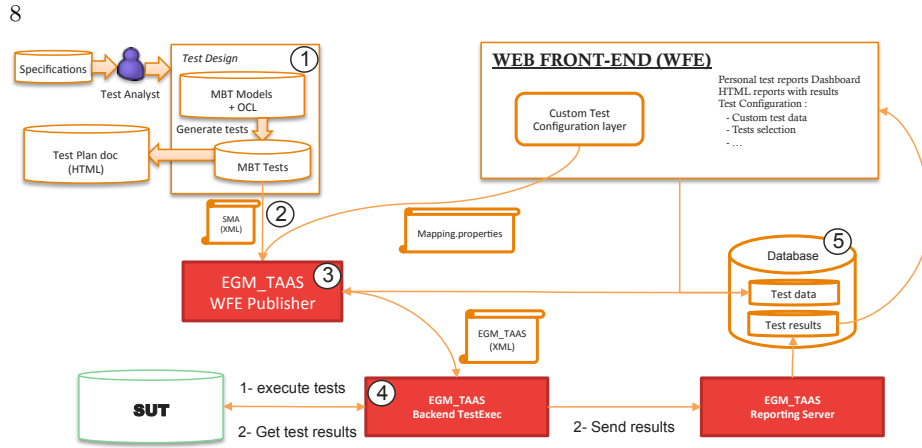


Fig. 7. MBTAAS architecture

involvement as we describe each service individually. Nevertheless, the database service can be separated from all the other services, it can be running in the cloud where it is accessible. The database stores important information such as test data (input data for test execution) and test results. The entry point of the system is the web-front end service (customization service). This service takes a user input to customize a testing session and it communicates it to a **Publication service** ③. The purpose of the publisher service is to gather the MBT results file and user custom data in order to produce a customized test description file (**EGM_TAAS** file). This file is then sent to an **Execution service** ④ which takes in charge the execution of the customized tests. It stimulates the SUT with input data in order to get response as SUT output data. The execution service then finally builds a result response and sends it to a last service, the **Reporting service**. The reporting service is configured to feed the database service with the test results. These test results are used by the web-front end service in order to submit them to the end-user. This testing architecture is taken to a modular level in order to respond to the heterogeneity of an IoT platform. In the following sections, a detailed description of each services is provided.

3.2 Customization Service

In order to provide a user friendly testing as a service environment, we created a graphical web-front end service to configure and launch test campaigns⁶. The customization service is a web site where identified users have a private dashboard. The service offers a pre-configured test environment. The user input can be reduced to the minimum, that is: **defining a SUT endpoint (URL)**. User specific test customization offers a wide range of adjustments to the test campaign. The web-service enables:

- **Test selection:** from the proposed test cases, a user can choose to execute only a part of them.

⁶ <http://193.48.247.210/egm-taas/users/login> (for reviewing purpose login:isola, password:isola)

- **Test Data:** pre-configured data are used for the tests. The user is able to add his own specific test data to the database and choose it for a test. It is a test per test configurable feature.
- **Reporting:** by default the reporting service will store the result report in the web-front end service database (more details on this in Sec. 3.5). The default behaviour can be changed to fit the user needs for example, having the results in an other database, tool, etc.

After completion of the test configuration and having the launch tests button pressed, a configuration file is constructed. The configuration file as can be seen in Fig. 8: *Configuration File excerpt*, defines a set of {key = value}. This file is constructed with default values that can be overloaded with user defined values.

```

14 #####
15 #####      REQUIRED PARAMETERS      #####
16 #####
17
18 #NAME OF THE OWNER OF THE REPORT
19 OWNER=EGM_TE_XML_PUBLISHER
20 #REPORT LOCATION AFTER TESTS (FOR EGM_TAAS_BACKEND)
21 REPORT_LOCATION=http://193.48.247.210:8081/report
22 #HOW TO REPORT (FOR EGM_TAAS_BACKEND)
23 REPORT_TYPE=POST_URL
24 #URL OF SUT TO TEST WITH THE PORT (FULL PATH)
25 ENDPOINT_URL=http://193.48.247.246:1026
26 #URL of EGM_TAAS backend that will execute the tests
27 EGM_TAAS_BACKEND = localhost:8080/executeTests
28 #Name of the Model file to be Used by EGM_TAAS_BACKEND
29 EGM_TAAS_MODEL = OrionCB_GE.xml
30 #Where to Output the results in the EGM_TAAS_BACKEND
31 EGM_TAAS_OUTPUT = tmp

```

Fig. 8. Configuration File excerpt

The configuration file is one of three components that the publisher service needs to generate the test campaign. The next section describes the publication process in more details.

3.3 Publication service

The publisher service, as its name states, publishes the abstract tests generated from the model into concrete test description file. It requires three different inputs (Fig. 7 step ②) for completion of its task: the model, the configuration file and test data. The model provides test suites containing abstract tests. The concretization of abstract tests is made with the help of the database and configuration file. For example, the abstract value `ENDPOINT_URL` taken from the model, is collected from the configuration file and `PAYLOAD_TYPE` (Fig. 5) parameter is gathered from the database service ⑤.

The concrete test description file is for our use case, an XML file that instantiates the abstract tests. The test description file has two main parts, general informations and at least one test suite (Fig. 9). A test suite is composed by one or more test cases and a test case itself is composed of one or more test steps. This

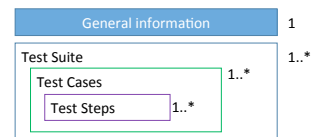


Fig. 9. Published file parts

hierarchy respects the IEEE 829-2008, Standard for Software and System Test Documentation.

The general information part of the file is useful to execute the tests (Sec. 3.4) and report the results. Here are some of the most important parameters that can be found in that part:

- **owner:** used for traceability purposes. It allows to know who is the detainer of the test in order to present it in his personal cloud dashboard.
- **sut_endpoint:** the hostname/ip address of the System Under Test. This address should be reachable from the execution service point of view.
- **location:** the hostname/ip address of the reporting service (Sec. 3.5).

The test suites contain all useful test information. In our case, for FIWARE, the applications are HTTP-based RESTful applications. The mandatory informations required to succeed a RESTful query are: the URL (SUT endpoint) and the HTTP method (GET, POST, PUT, DELETE). The Test suite and test cases purpose is the ordering and naming of the test but the test information and test data are stored in the test steps. Each test step have its own configuration. Once the file published, it is sent to the execution service in order to execute the tests.

3.4 Execution service

The execution service is the functional core of the MBTAAS architecture. The execution service will run the test and collect results depending on the configuration of the received test description file. FIWARE RESTful interface tests are executed with the REST execution module. Each test is run against the SUT and a result report (Listing 1.1) is constructed on test step completion. The result report contains information on date-time of execution, time spent executing the step and some other test specific values. The "endpoint" represent the URL value and validity of where the step should be run. An invalid endpoint would allow to skip the type check on the endpoint value and thus allowing to gather an execution error. The response of each step is validated within the "assertion_list" tags (test oracle). It validates each assertion depending on the assertion type and values with the response received.

Listing 1.1 – Test step result report

```
<teststep name=" UpdateEntity1">
  <executionResults>
    <timestamp>{TIMESTAMP}</timestamp>
    <executionTimeMs>22</executionTimeMs>
  </executionResults>
  <endpoint>
    <value>{IP}:{PORT}/upContext</value>
    <isinvalid>>false</isinvalid>
  </endpoint>
  <method>POST</method>
  <headers>{HEADERS}</headers>
  <payload>{PAYLOAD}</payload>
  <assertion_list>
    <assertion>
      <key>code</key>
      <value>404</value>
      <result>>false</result>
    </assertion>
  </assertion_list>
  <result>>false</result>
  <response>{
    "errorCode" : {
      "code" : "400",
      "reasonPhrase" : "Bad Request",
      "details" : "JSON Parse Error"
    }
  }
</response>
</teststep>
```

Figure 10 shows an excerpt of the execution service log. In order to execute one test step, the endpoint (URL) must be validated. Then a REST request is

created and executed. A response is expected for the assertions to be evaluated. At the end, an overall test result is computed. The overall assertion evaluation algorithm is as simple as: "All test assertions have to be true", that implies if one assertion is false, the step is marked as failed.

```
[egm.modelTools.HttpRequestExecutor] Validating url : http://[REDACTED]:1026/v1/updateContext
[egm.modelTools.HttpRequestExecutor] URL is VALID
[egm.modelTools.HttpRequestExecutor] Starting Jetty HTTP Client
[egm.modelTools.HttpRequestExecutor] Jetty HTTP Client Started with Success
[egm.modelTools.HttpRequestExecutor] Creating request : URL = http://[REDACTED]:1026/v1/updateContext, HTTPMETHOD = POST
[egm.modelTools.HttpRequestExecutor] Request created
[egm.modelTools.HttpRequestExecutor] Request status code: 200
[egm.modelTools.HttpRequestExecutor] Response content: {
  "errorCode" : {
    "code" : "400",
    "reasonPhrase" : "Bad Request",
    "details" : "JSON Parse Error"
  }
}
[egm.modelTools.HttpRequestExecutor] Stopping Jetty HTTP Client
[egm.modelTools.HttpRequestExecutor] Jetty HTTP Client Stopped
[egm.modelTools.HttpRequestExecutor]

[egm.model.Assertion] Asserting...expression to be assert: is key "code" contains value: "404"
[egm.modelTools.TestStepResponseParser] is JSON data key "code" contains value: "404"
[egm.modelTools.TestStepResponseParser] no value of : "404" has been found for id: "errorCode"
[egm.modelTools.TestStepResponseParser] no value of : "404" has been found for id: "code"
[egm.model.TestStep] Execution Result of step UpdateEntity148 : false
```

Fig. 10. Execution snapshot

Once the execution service has finished all test steps, their results are gathered within one file, we call this file **Test Results**, and it is sent to the reporting service.

3.5 Reporting service

After executing the test, a file containing the test description alongside their results are sent to and received by the reporting service. The reporting service configuration is made in the web front-end service. The configuration is passed with the test configuration file where the publisher service re-transcribes that information to the file sent to the execution service. The execution service then includes the reporting configuration in the report file where it is used in the reporting service once it receives it. By default the reporting service will save the results in the database service ⑤. For our use case, the database is implemented as a MySQL database. This database is afterwards used by the web front-end service to present the test results to the user.

4 Evaluation

We present results on requirements coverage, test execution and time spent to apply the approach in order to show the cost-benefits of MBTAAS.

4.1 Results

We have created two test suites for the Orion Context Broker. One test suite for automated test generation (to cover compliance requirements) and an other test suite with user defined scenarios to use the tester's experience (to cover specific functional requirements (as also available with CertifyIt [8]). For our FIWARE scope 22 requirements were manually extracted from the FIWARE

standard, traced into the MBT model using the tagging feature with REQ/AIM. This automatically produced a coverage matrix between the generated test cases and the requirements, which in our case is 100% (for more details consider the following report [1]. To further evaluate our approach we gathered information on the test case generation (number of generated tests) and test execution time. We compare these results with respect to a manual approach, a tester crafting the test cases to cover the same test objectives.

In total, the two CertifyIt test suites contain together 31 test cases and 172 test steps. The execution time including the response evaluation is accomplished in less than six seconds (5438 ms). Compared to a manual test step execution, where the execution and the evaluation can take up to approximately 1 min by test step in the best conditions (all testing environment pre-set up) we have a 1720 times improvement in time consumption. The test execution resulted in 165 successful test steps (25 tests) and 7 failed test steps (6 tests). Failed tests are due to a gap between specification and implementation. The model showed that some test result were noted as "success" in specification and does not state a clear result which we could match with actual implementation results. Applying our MBT approach on an enablement API made possible to clearly identify the benefits of applying a service oriented MBT approach in terms of APIs interoperability verification thus ensuring the respect of the specifications. In terms of project planning, it took us 26 person/hours to create the MBT model. More specifically, it took 10 person/hours to model the static view of the system (the class diagram) suitable for testing and 16 person/hours to model the dynamic view of the system (to write the OCL). These metrics abstract away the domain knowledge on FIWARE standards and the FIWARE NGSI specification itself. If the MBT approach is integrated within the project, the testing teams have already this knowledge. This is linked to the developers/testers experience and we consider the process of getting additional knowledge of the platform as negligible. The MBT part of our approach is transparent for the community, thus the community will simply submit their application for FIWARE compliance testing and use the MBT output artefacts i.e. the test cases produced by our MBT approach. Additionally, time spent on building the service approach is also given. And it is important to notice that the services are modular. They are only developed once for RESTful application and each new model comes as input to the MBTAAS system already in place. In case of an other IoT platform protocol, lets take MQTT (<http://mqtt.org/>) for example, adjustments to the services is required. The web front-end service should include the possibility to choose the new type of platform and a new execution module needs to be developed and integrated in the execution service. The same modifications are needed in the reporting service if we want to propose for example to export the results to a mongo-DB (<https://www.mongodb.org/>) database rather than MySQL.

4.2 Discussion

We are confident in our work and results following that the paper user case "Orion Context Broker" testing is a continuation of a preceding proof of con-

cept on FIWARE enablers testing. The last use case was on the *Espr4FastData* enabler: a complex event processing tool [5]. We demonstrated that a classical MBT approach is suited to test an IoT platform thus encouraging us move forward and bring the service layer to our proof of concept.

One major advantage we saw in applying the MBT approach on FIWARE is that the test repository remains stable, while the project requirements and specification may evolve within the time. MBT is a suitable approach for emerging technologies and especially on IoT platforms where the maturity level is still increasing while technology development is still on going. Being able to generate tests automatically and in a systematic way, as we did, makes possible to constitute a test suite covering the defined test objectives. The MBT further allows generating reports to justify the test objective coverage, which can be easily used for auditing for instance. These couple of examples show the usefulness of an automated and systematic use of an MBT approach on applications that should comply to specific standards. Combined with a user friendly and ease of access through service oriented solution, first experiments with MBTAAS show that it is a promising powerful tool.

5 Related Works

In this section, we review work related to our proposed approach in the areas of model-based testing (MBT) related to Internet of things (IoT) systems, more specifically mobile and cloud testing, and Model-Based Testing as a service. Model-based testing has been extensively studied in the literature [17] [16]. However, the majority of existing approaches in connexion to the IoT domain are mostly designed for mobile application. For instance, authors in [6] design a GUI (Graphical User Interface) ripping approach based on state machine models for testing Android applications. Other work concentrates on vulnerability testing of mobile application based on models, for instance authors in [15] propose an MBT approach to generate automatically test cases using vulnerability patterns, that target specifically the Android instant Messaging mechanism.

In addition, recent survey by Incki et al. [10] reports on the work done on testing in the cloud, including mobile, cloud applications and infrastructures, testing the migration of applications in the cloud. They realized a categorization of the literature on cloud testing based on several elements among which: test level and type, as well as contribution in terms of test execution, generation and testing framework. They underlined that testing as a service for the interoperability for cloud infrastructures today remains still a challenge. Authors in [9] propose a model-based testing approach based on graph modeling for system and functional testing in cloud computing. Hence, contrary to these approaches that refer to testing approaches of the different layers of the cloud: Software as a service, Platform as a service and Infrastructure as a service, our approach proposes Model-Based Testing as a service for compliance testing of IoT systems, were the cloud is one element of it.

Testing service can be provided to cloud consumers as well as cloud providers generally called Testing as a Service (TaaS)[14]. Previous work on testing as a

service, to the best of our knowledge, specifically relates to web services and cloud computing. Zech et al. in [18] propose a model-based approach using risk analysis to generate test cases to ensure the security of a Cloud computing environment when outsourcing IT landscapes. More recently Zech et al.[19] proposed a model-based approach to evaluate the security of the cloud environment by means of negative testing based on the Telling Test Stories Framework. Model-Based Testing provides the benefit of being implementation independent. In this paper, we propose MBT as service to the Internet of Things, making thus the test cases available for any platform implementation of the specification. Contrary to the existing works, our approach proposes an abstraction on model construction, it is configuration over development. Our model-based test generation does not take into account risk analysis elements neither security requirements, which can be one possible extension of this module of our Model-Based Testing as a Service approach.

6 Conclusion and future works

This paper presented a successful application of an MBT approach with a service oriented solution. We believe that this approach can be generally applied on a wide range of specifications defining APIs for FIWARE. Within the FIWARE context, the created MBT model, NGSI compliant, can be reused for testing any range of enablers respecting that specification. New developments focus on the test configuration layer which is made in the front-end service, in order to make the tests compatible with the System Under Test. The modularity, will be explored to be used in *integration testing* between IoT platform applications (Fig. 1). Furthermore, one of our concerns was to provide the IoT platform tests to the community in the easiest way possible, including the possibility to choose the version of standard compliance only by model selection. This is done with the service oriented approach, providing to all involved stakeholders (not only testers) the capacity to test their generic enabler installation remotely from an online webpage in a Plug and Test approach. IoT platforms can be used by third party entities (data consumers/providers) that connect to the platform to verify their compliance to the platform's standard. The next step of the research work is to explore to what extent the models of an IoT platform can be used to test those third party applications in order to validate their behaviour on the platform.

Acknowledgments This research was supported by the project FP7 FI-CORE.

References

1. FIWARE test repository and requirements matrix. <http://fiware.eglobalmark.com/html/>, [Online; accessed 29-april-2016]
2. Open Mobile Alliance. <http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/ngsi-v1-0>, [Online; accessed 18-april-2016]

3. Reinhart Richter, Xcerra Corporation, Does the Internet of Things force us to rethink our test strategies? http://xcerra.com/ep_doestheinternetofthingsforceustorethinkourteststrategies-vision
4. The FIWARE Project. <https://www.fiware.org/2015/03/27/build-your-own-iot-platform-with-fiware-enablers/>, [Online; accessed 8-april-2016]
5. AHMAD, A.: Iot interoperability model-based testing, a fiware case study (2015), poster at UCAAT, ETSI, Sophia-Antipolis, France
6. Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M.: Using GUI ripping for automated testing of android applications. In: 27th IEEE/ACM ICSE. pp. 258–261. ASE 2012, ACM, New York, NY, USA (2012), <http://doi.acm.org/10.1145/2351676.2351717>
7. Bernabeu, G., Jaffuel, E., Legeard, B., Peureux, F.: MBT for global platform compliance testing: Experience report and lessons learned. In: 25th IEEE ISSRE Workshops, Naples, Italy. pp. 66–70 (2014)
8. Botella, J., Bouquet, F., Capuron, J., Lebeau, F., Legeard, B., Schadle, F.: Model-based testing of cryptographic components - lessons learned from experience. In: 6th IEEE ICST, Luxembourg, Luxembourg. pp. 192–201 (2013)
9. Chan, W.K., Mei, L., Zhang, Z.: Modeling and testing of cloud applications. In: Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific. pp. 111–118 (Dec 2009)
10. Incki, K., Ari, I., Sozer, H.: A survey of software testing in the cloud. In: 6th IEEE International Conference SERE-C. pp. 18–23 (June 2012)
11. Kramer, A., Legeard, B.: Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester: Foundation Level. Wiley (May 2016)
12. Legeard, B., Bouzy, A.: Smartesting CertifyIt - model-based testing for enterprise IT. In: ICST'13, 6th IEEE Int. Conf. on Software Testing, Verification and Validation, Testing Tool Track. pp. 192–201. IEEE, Luxembourg (Mar 2013)
13. Nebut, C., Traon, Y.L., Jezequel, J.M.: Software Product Lines, chap. System Testing of Product Lines: From Requirements to Test Cases, pp. 447–478. Springer (2006), http://dx.doi.org/10.1007/978-3-540-33253-4_12
14. Riungu, L.M., Taipale, O., Smolander, K.: Research issues for software testing in the cloud. In: 2nd IEEE International Conference CloudCom., pp. 557–564 (Nov 2010)
15. Salva, S., Zafimiharisoa, S.R.: Data vulnerability detection by security testing for android applications. In: Information Security for South Africa, 2013. pp. 1–8. IEEE (2013)
16. Utting, M., Legeard, B., Bouquet, F., Fournier, E., Peureux, F., Vernotte, A.: Chapter 2 - recent advances in model-based testing. *Advances in Computers* 101, 53–120 (2016), <http://dx.doi.org/10.1016/bs.adcom.2015.11.004>
17. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. *STVR* 22(5), 297–312 (2012), <http://dx.doi.org/10.1002/stvr.456>
18. Zech, P., Felderer, M., Breu, R.: Towards a model based security testing approach of cloud computing environments. In: 6th International Conference SERE-C. pp. 47–56 (2012)
19. Zech, P., Kalb, P., Felderer, M., Breu, R.: Chapter 40 - threatening the cloud: Securing services and data by continuous, model-driven negative security testing. *Transportation Systems and Engineering: Concepts, Methodologies, Tools, and Applications* 3, 789–814 (2015)