How to Test IoT-based Services before Deploying them into Real World

Eike Steffen Reetz*†, Daniel Kuemper†, Klaus Moessner* and Ralf Tönjes†
*Centre for Communication Systems Research, University of Surrey, Guildford, UK
†Faculty of Engineering and Computer Science, University of Applied Sciences Osnabrück, Osnabrück, Germany

***Abstract*—Efficient testing of Internet of Things (IoT)-based services suffers from the underlying heterogeneous nature of IoT resources and hinders the process of rapid service creation and deployment. Real world effects, based on the behaviour of IoT-based services, tend to prevent the straight forward execution within the productive environment. Current solutions for testbeds, involving physical or virtual IoT resources, appear to require intense capacities for time and resources. This paper describes a new approach for testing IoT-based service build on a code insertion methodology, which can be derived from the semantic description of the IoT-based service. The proposed IoT resource emulation interface is described from the semantic, architectural and implementation perspective. The paper compares its applicability and efficiency with classical approaches and expose high emulation capabilities while minimising the testing effort.**

***Index Terms*—functional testing; model-based testing; stream X-Machine; semantic test derivation**

I. Introduction

Real world interaction of services connected to Internet of Things (IoT) resources requires sophisticated capabilities for evaluation and testing. Functional tests and performance evaluation have to assess an uncritical deployment of the service in a productive environment (e.g. traffic control system not working properly). Contrary to classical service testing approaches, the strong interaction with the physical world needs to be addressed by the test environment. Furthermore, the scalability and time efficiency of the approach need to be estimated realistically. Previous testbed approaches handle IoT resources in different ways (e.g. WISEBED [1], Kansei [2] cf. Gluhak et al. [3] for a comprehensive comparison of testbed features):

- physical IoT resources in a testbed
- virtual IoT resources in a testbed
- a combination of physical and virutal IoT resources
- gateways to generically connect physical IoT resources

In the first category real hardware is used for testing the behaviour of the system. Therefore, hardware-specific problems can be identified during the test execution. The main drawbacks are the lack of scalability and the limited ability to test specific situations (e.g. room temperature $< -20\,^{\circ}\mathrm{C}$). Improved scalability as well as larger control of test situations can be achieved with virtual resources. Most approaches build a virtual machine that represents the IoT resource. The target platform can then be directly deployed in this virtual machine. Tests can therefore be run from the perspective of the IoT resources as well as from the connected services. Although, concepts combining virtual and physical resources in the testbed have a great coverage of hardware, IoT resource, and software-related failures, they show limitations with regard to scalability. While it is comparably easy to handle one specific IoT resource and build a virtual equivalent, what if there are hundreds of different IoT resources involved? In addition to these scalability issue, it is necessary to implement the virtual machines of the IoT resources as well as the test environment needs to know how to interact with the virtual resources (e.g. provide hardware interfaces like Bluetooth, ZigBee, and also knowledge of partly proprietary protocols at the application layer). One approach to overcoming this limitation was proposed by Diaz et. al [4]. The authors argue that a systematic implementation of gateways can lower the costs of testing by connecting the System Under Test (SUT) and testing tool. As a result, complexity of interaction between the SUT and the IoT resource is hidden from the test framework but still needs to be modelled within the gateway. Model-based testing is a well-investigated research area whilst the allowance of the integration of resource constrained devices is not solved. Especially concerning emulation and simulation due to the avoidance of device specific costs [5].

This paper investigates how to overcome the complexity and drawbacks of these approaches. We propose a (semi-) automated process of test code insertion in order to support fast prototyping with test integration of large-scale IoT-based services. We postulate that, if the logical interaction between the service and IoT resource is described in a machine readable way (e.g. semantically), the service knowledge can be utilised to build a generic resource emulation interface. This abstracts the heterogeneous interaction with the IoT resource and thus enables fast and scalable emulation of IoT resources from the logical perspective. With the approach of code insertion, the test environment is capable of controlling the interaction of the service with IoT/external resources (not limited to IoT resources) and can transparently (from service logic perspective) replace IoT/external resources with controllable resource emulation components. Our approach facilitates testing IoT-based services by insertion of emulated IoT resource requests and responses at a logical level. This approach can be utilised to test how the service acts when facing unexpected but well-formatted inputs and outputs originating from the IoT resources. Thus allows to rapidly and efficiently emulate critical environment situations.

The presented concepts have been conducted within the European research project "IoT.est". The overall project concept can be found in [6] and a general description of the automated test approach is described in [7].

The rest of the paper is structured as follows: the overall concept is outlined in Section II and detailed descriptions are given based on a stateful service example. Subsequently, the concept is discussed from architectural (Section III) and from implementation (Section IV) perspectives. Section V summarises the benefits of the proposed approach and discusses the differences with classical approaches. The conclusion and outlook section completes the paper.

II. Concept

Service-oriented paradigms have considerable potential to be one of the key drivers for IoT enabled applications. In the future, the concept of re-using existing services for composing richer services and applications may constitute a novel ecosystem for large scale IoT deployments that is still missing today.

In this domain, the concepts of the semantic web seems to fit perfectly. While semantics for pure web services have not prevailed so far, the IoT domain seems to be a much more suitable candidate. Due to the heterogeneous nature of IoT devices and expected abstraction services, it is required to exchange complex information about the possibly involved components and their interaction capabilities. From a testing perspective, this offers the opportunity to derive tests from the semantic service descriptions. Therefore, a commonly-used model-based testing approach with (semi-) automatised test derivation and execution is pursued. The involvement of IoT resources changes the way testing can be applied to these semantically described services: Risks of costs and possible real world effects result in the need for a controllable environment, capable of investigating the service behaviour under different behaviours of the IoT resources (including unexpected behaviour or specific values, which are hard to reconstruct in the real world). As outlined in the introduction we aim at testing IoT services based on virtual IoT resources that only emulate the IoT resources from the logical point of view, enabling efficient and automated test derivation and execution. Similar to the description of the service interface (input, output, precondition, effect), there is the need to describe these logical interfaces to the IoT resource semantically (class, method, parameters as input and output structure and precondition and effects).

Our approach attempts to decrease the effort for emulating IoT resources by focussing on logical failures, which can occur in the IoT-based service. Therefore, the approach is suitable for testing the service connected to IoT resources but cannot be used to investigate the IoT resource itself. In contrast to classical virtual resource emulation techniques our approach foresees inserting small pieces of code into the SUT itself. This code insertion technique enables capturing and creating messages, which originated from the IoT resources or are directed to it, at a logical level. This way significant simplification of the required emulation responses and requests can be achieved. The difficulty of a complex communication interaction (from physical interfaces like IEEE 802.15.4 to communication protocols like 6LoWPAN) can be kept away from the software, hardware, and virtual resources. Moreover, the test framework does not require protocol specific knowledge.

In the next subsection we discuss how interfaces need to be described in general and how the test model can be abstracted from this information.

A. Rules for Precondition and Effects

Input Output Precondition Effect (IOPE) are commonly known as the information required to describe the functional behaviour of a service interface. Ontologies may be utilised to describe information consumed by the service (Input) and corresponding transformed Output. One approach widely applied utilises rules for describing the preconditions, that need to be fulfilled to invoke the service. In addition, rules can also be utilised to describe the state changes after the invocation (Effect).(cp. [8], [9]) Ramollari et. al [10] propose utilising production rules. We adopt their approach to describe the precondition and effects of an IoT-based service example. The Rule Interchange Format Production Rule Dialect (RIF-PRD) is adopted to present the required changes for our approach to insert a test controller into the SUT. Production rules can be classified as reactive rules, which can specify one or more actions that are executed if their conditions are satisfied [11].

An Example of a Stateful IoT Service: The service under investigation performs elementary lookup functionalities. It allows for requesting values from embedded sensors via a service interface. The service interface offers one operation: *getValue* and provides two operations enabling communication with the IoT resource: i) *register* ii) *getResourceValue*. Furthermore, there is one internal timer event: *unregistered*. For simplicity, we assume that an IoT resource entity is created during the initialisation of the service and is marked as unregistered as long as the IoT resource does not use the register functionality. If an IoT resource has been registered previously, a consumer can request the IoT resource value (e.g. sensor data), by querying the service interface. If the IoT resource does not re-register itself during a specific time period, the IoT resource entity is marked as inactive and service consumers will no longer be able to request the IoT resource values. Listing 1 represents the encoded precondition and effect of the IoT-based service example. For clarity the RIF-PRD is depicted with an abstract syntax. An XML syntax with an associated semantics is available as well.

We assume a semantic model that describes the service. The model specifies input, output and state-related attributes of the services. Table I lists entities that need to be described by the ontology. The description of precondition and effects with RIF-PRD together with the semantic model for the input and output parameters can leverage building a model of the service interface. The next section explains how and why stream X-Machines are utilised to describe the model of the service.

```
(* wsdl:operation getResourceValue *)
Forall  ?resource ?status ?value ?request
And (   ?resource#Resource
        ?resource[hasStatus->?status]
        ?resource[hasValue->?value]
        ?request#GetResouceValueRequest)
If (    External (pred:matches(?status "S2")
Then Do (Retract (?request)
        (?response New (?response#ResourceValueResponse))
        Assert (?response[hasValue->?value]))
(* wsdl:operation register *)
Forall  ?resource ?status ?lastseen ?request
And (   ?resource#Resource
        ?resource[hasStatus->?status]
        ?resource[hasLastseen]->?lastseen]
        ?request#RegisterResourceRequest)
If Or(  External (pred:matches(?status "S1")
        External (pred:matches(?status "S2"))
Then Do (Retract (?resouce[hasStatus->?status])
        Assert (?resource[hasStatus-> "S2"])
        Retract (?resource[hasLastseen->?lastseen])
        Assert (?resource[hasLastseen->nowDateTime])
        Retract (?response)
        (?response New (?response#RegisterResouceResponse)
            )
        Assert (?response[hasMessage-> "Resource
            Registered"]))
(* wsdl:operation unregister *)
Forall  ?resource ?status ?lastseen ?event
And (   ?resource#Resource
        ?resource[hasStatus->?status]
        ?resource[hasLastseen->?lastseen]
        ?resource[hasEvent->?event]
        ?event#unregisterEvent)
If And( External (pred:matches(?status "S2"))
        External ( pred:dateTime-greather-or-equal(?
            nowDataTime (?lastseen+360))))
Then Do (Retract (?resource[hasStatus->?status])
        Assert (?resource[hasStatus->"S1"])
        Retract (?request)
        (?response New (?response#UnregisterResponse))
        Assert (?response[hasMessage->"Resource
            unregistered]))
```

Listing 1. Rule Description Example

TABLE I
ONTOLOGY ENTITIES

Type	Name
Classes	Resource, GetResouceRequest,ResourceValueResponse, registerResourceRequest, RegisterResouceResponse, UnregisterResponse
Datatype properties	hasStatus (domain: Resource, range: string enumeration "S1", "S2"), hasLastSeen (domain: Resource, range: dataTimeStamp), hasValue (domain: Resource, range: double), hasMessage (domain: RegisterResouceResponse, UnregisterResponse, range: string) hasEvent (domain: Resource, range: boolean)
Individuals	ri (resource individuals, hasStatus: "S1", hasLastseen: "1972-12-31T00:00:00")

B. Streaming X-Machine

A streaming X-Machine is capable of representing the data and the control model of a system. It can be seen as an extension of the Finite State Machine with an attached complex data model, which is forming a memory model. In addition, transitions are represented by processing functions instead of input symbols. The processing functions utilise input streams as well as read and write memory values by producing an output stream [12]. In the following sections we briefly outline how the rules shown in Listing 1 can be utilised to build a Stream X-machine representing the SUT.

State Variables: The data model of the Streaming X-Machine can be utilised to represent the state variables of the IoT-based service example. State variables contain the attribute that they can be influenced by the input and output transformation process. Therefore, the state variables appear in the action part of the rules. The reserved words for *?request (Input)* and *?response (Output)* are ignored. In the example the identified variables are: *hasStatus* and *hasLastSeen*. State variables are limited with regard to their range of validity. One limitation is defined by the *Datatype properties* (cf. Table I) of the ontology (e.g. enumeration, nonNegativeInteger). On the other hand, state variables may also be restricted by the precondition sections of the production rules. In the service example the state variables have the following partition:

- *hasStatus*: S1, S2
- *hasLastSeen*: $\geq$?nowTime - 360, $<$?nowTime - 360

Identifying Preliminary State: The preliminary states are the product of the partitions of the state variables. In our example this results in four pre-states, namely:

- *S1*$\geq$*timeM360, S1*$<$*360*,
- *S2*$\geq$*timeM360, and S2*$<$*360*.

where *timeM360* stands for *?nowTime - 360*.

Determining Input and Outputs: The next step is to identify the input and output functions. In the production rules the reserved words *?request* and *?response* allow for identifying these functions. In addition, our example has an event-based input, which is identified by the variable *?event*. This might not be a standard conform assumption but it is expected that event-based computation could be described based on extension of RIF-PRD. It is out of scope to discuss a fully valid model for event-based transactions. The inputs are defined as follows (including events as input):

- *GetResourceValueRequest*
- *GetRegisterRequest*
- *UnregisterEvent*

and the Outputs:

- *ResourceValueResponse*
- *RegisterResouceResponse*
- *UnregisterResponse*

Determining Transition Pre-States: The input results in the execution of a processing function if the preconditions are satisfied during the pre-state phase. With mathematical deduction the valid pre-state of the inputs are:

- *GetResourceValueRequest*: S2$\geq$360, S2$<$360
- *GetReqisterRequest*: S1$\geq$360, S1$<$360, S2$\geq$360, S2$<$360
- *UnregisterEvent*: S2$<$360

State Merging: If one input is acceptable for more than one pre-state, the states can be merged. In our IoT-based service example the pre-states *S1*$\geq$*360 S1*$<$*360* are both acceptable for any input. The resulting states are:

- *S1*: (hasStatus = S1, hasLastSeen = *)
- *S2*: (hasStatus = S2, hasLastSeen¡ nowDataTime -360)

TABLE II
TRANSITION PRE-STATES AND POST-STATES

Input	Pre-State	Effect	Post-State
GetResourceValueRequest	S2	–	S2
GetReqisterRequest	S1, S2	hasLastSeen ← 0	S2
UnregisterEvent	S2	hasStatus ← S1	S1

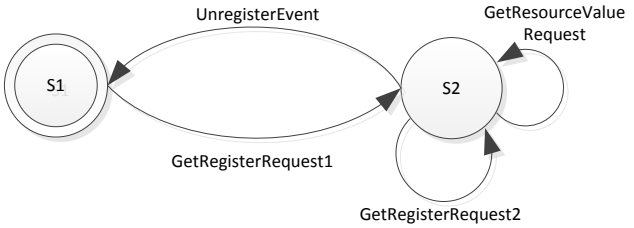

Fig. 1. State Diagram

Determining Transition Post-States: Similar to the approach of identifying the pre-states, post-states are identified by investigating the effect of the processing function execution. If more than one pre-state is acceptable for one input, but not for all inputs, two different processing functions are created (in the example *GetRegisterRequest1* and *GetRegisterRequest2*). The resulting transitions are outlined in Table II. Finally the resulting finite state machine of the Stream X-machine is shown in Figure 1.

C. Test Adaptation

The methodology previously described is expected to be applicable for any kind of software component interface tests as long as the ontology describes the interface types as well as input and output parameters specifically enough. However, our approach primarily aims at evaluating one interface (SOA) and emulating the logical communication over other interfaces to external/IoT resources. Therefore, additional information is required in case that requests received over the SOA interface result in communication with the external/IoT resource. In our example the production rules of the *GetResourceValue* operation needs to be extended at the action part by the following extension: *Execute(act:getValue(?resource))*

The function *act:getValue* is an action that is externally described. The external description needs to cover the knowledge about the executed method and class as well as the required parameters. Further the data types and expected return values need to be defined. For a more sophisticated test approach, semantic descriptions of non-functional behaviour like typical response durations or freshness of data (time between detection and transmission) can play an important role for a more realistic emulation of IoT resources.

III. ARCHITECTURE

The test environment comprises the following main components: i) Test Design Engine (TDE) capable of deriving tests and and test flows from the semantic description of the service; ii) Test Execution Engine (TEE) responsible for executing the tests in an controllable environment (sandbox); and iii) a Sandbox Environment which is capable of emulating the network behaviour, the system load and also the communication with IoT or external resources. The SUT is placed inside the sandbox and needs some adaptation before it is ready to be tested. Figure 2 illustrates a simplified IoT-based service (SUT). The dashed lines indicate new or changed components and interfaces of the SUT, which are inserted with the previously outlined insertion methodology. Our initial model consists of two interfaces: i) a *service interface* handling request from service requests and an ii) *IoT/external resource interface* capable of communicating with IoT/external resources. Without any test related changes the service has a service logic and it is expected that some classes (in this example entitled as *Resource Request*) provide access to the IoT/external resource interface. From the architectural point of view the SUT needs to be extended by one interface: IoT/external resource emulation interface, and a test controller component, which takes care of the current test status of the service (in test mode, logging mode, or regular mode). These test related changes are realised with the outlined insertion methodology. In addition, each component that is connected to the IoT/external resource interface needs to be encapsulated.

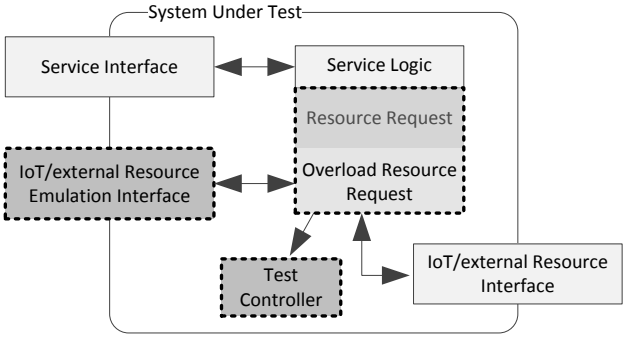

Fig. 2. System Under Test with Test Controller

If the test cases are ready to be executed the SUT is placed in the sandbox and the TEE starts to run the test flow. Figure 3 shows a simplified version of the TEE and highlights the components required to utilise the IoT/external resource emulation interface, inserted in the SUT within test preparation process. The TEE consists of three interfaces to communicate with the SUT: i) *Communication Interface* connected to the service interface of the SUT to send and response the test request, ii) the *Context Emulation Interface* connected to the IoT/ external Resource Emulation Interface, which receives and responses to Resource requests as described within the test cases, and iii) a *Log Interface* to enable logging of messages transmitted from or to the IoT/external resource in case real resources are connected.

IV. IMPLEMENTATION

This section outlines the prototypical implementation based one the discussed code insertion approach and highlights related issues. The insertion has been conducted manually in

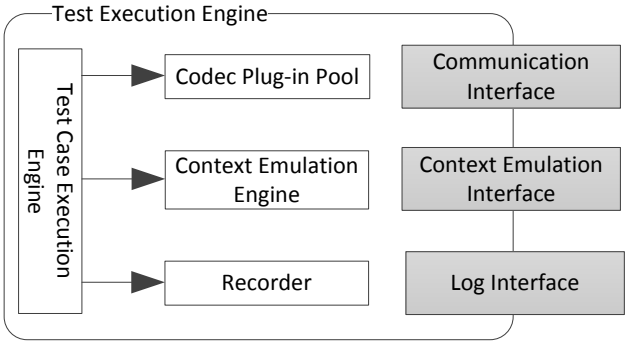

Fig. 3. Simplified Test Execution Engine

order to identify if the followed concept can be realised. As part of future work it is planned to investigate how far this process can be accomplished in an automated manner.

The SUT has been realised based on Java 1.6 and Jetty 8 for the HTTP server part. The service consists of a HTTP-Get interface that forwards the request to the IoT resource, if the resource has been registered previously. In this implementation JAVA Remote Method Invocation (RMI) has been used to communicate with the IoT resources but it is expected that other communication paradigms can be described in the same way. To makes sure that the TEE is capable to emulate the communication with the IoT resources, the *ResourceRequest* class is encapsulated by creating the subclass *ResourceEmulationRequest* and overloading the methods. In addition, this concept is extended by creating another subclass, which enables logging functionalities. In case the logging mode is activated, communication between ”real” IoT resources and the service is logged and stored into the TEE. This information can then be utilised to evaluate if the the emulation is realistic enough (e.g. typical message response times and value ranges). It is part of future work to further elaborate if this information can be utilised to enhance the emulation quality from real world experiments.

The service needs to be switch between *real emulation* and *logging* mode. Therefore, a test controller class is created. In order to change the mode during the deployment phase, the initialisation process of the service is altered. Before the interface to the IoT/external resources are initialised a waiting timer is included and therefore the TEE can set up the test controller with the planned mode. Due to simplicity, the mode is not changeable after the initialisation phase and needs to be re-deployed for mode change. In case the TEE does not send any message to the SUT the service is executed with the ”real” interface to the IoT/external resources. After receiving the required mode from the TEE either the original interface or the emulated ones are started. Afterwards the service will not be able to notice if he is executing methods from the original class *ResourceRequest* or the subclasses *ResourceEmulationRequest* or *ResouceLoggingRequest*.

The TEE has been realised based on the the testing framework TTworkbench [13]. Therefore, the service interface and the resource emulation interface are connected with the TTworkbench framework and the test cases as well as the model of the IoT resource emulation are described with the standardised Test Control Notation Version 3 (TTCN-3).

V. Discussions

The outlined approach of forming a controllable resource emulation interface, is compared with the approach of physical and virtual IoT resources involved in the testing process within this section. Table III summarises the concept comparison. The different concepts are assessed on their capabilities (*Cap*) and required efforts (*Eff*) to enable test cases of different test goals (test types). For this comparison four different test types are identified: *Hardware* test cases are capable of detecting if the wrong behaviour of the SUT is based on hardware failures (e.g. flash storage failures, internal bus errors); *Communication* test cases investigate if the protocols involved to establish a communication link are working as expected, while the *Logical* test cases cover possible failures resulting from wrong data exchange at a logical level. Test cases for *Scalability* evaluate if the service is still working correctly in case the number of attached IoT resource are increased.

Hardware test cases can only be realised in cooperation with *physical* IoT resources but come with high effort of either programming and designing a hardware debug interface and/or utilisation of hardware analysers. *Communication* test cases can be provided by either physical or *virtual* IoT resources. Both approaches are highly capable of investigating if the communication is working as expected. Although due to the visualisation it is much easier to investigate the behaviour.

While the test types *Hardware* and *Communication* are relevant to test the IoT resource functionality itself, these test are not required to test the IoT-based service. *Logical* test cases can be realised with all three approaches compared above. However, the capabilities of physical IoT resources to process *Logical* test cases are limited due to the limited possibility to manipulate the execution process. High effort is needed to manipulate data values (e.g. temperature values) either by manipulating the environment (e.g. heating the room) or implementing a strong manipulation interface. Virtual IoT resources have more potential for manipulation but still require high effort to build the required manipulation interfaces. The presented approach of a resource emulation (*Res. Emul.*) interface reveals the best characteristics for this comparison. Due to the concentration of the logical level, the simplicity of the interface enables a highly flexible solution with low implementation effort. This advantage is also visible if *Scalability* is under investigation. While physical IoT resources are not easily available and manageable at a large scale, virtual IoT resources as well as the resource emulation interface concept expose high capabilities for scalability since hardware components are not involved. Compared to the other solutions, the simple structure of the resource emulation interface decreases the effort. In conclusion, the test types relevant for IoT-based service testing are efficiently realised with the proposed resource emulation interface. The next paragraph discusses how the Stream X-Machine can be applied with

TABLE III
CONCEPT COMPARISON

Test Type	Physical		Virtual		Res. Emul.	
	Cap	Eff	Cap	Eff	Cap	Eff
Hardware	high	high	NA	NA	NA	NA
Communication	high	high	high	low	NA	NA
Logical	med	high	high	high	high	low
Scalability	med	high	high	med	high	low

the three different approaches and why our proposed solution contributes significantly to a simplified and easier automation of the test execution process.

The deduction of test cases from the identified stream X-Machine of Section II-B results in test cases for each processing function (cp. Figure 1). One of the major problems involving physical IoT resources is the requirement to control the resources (functionally but also timely). In order to execute the test case *GetResourceValue* the SUT needs to be put into the state "S2". Therefore, the following procedure is required: i) initialise SUT ii) try the input *GetResourceValueRequest* (failure expected) to assure that the current state is "S1", iii) invoke *GetRegisterRequest1*, for state change, and iv) send input *GetResourceValueRequest* (expect a positive response). Based on different test coverage criteria the procedure could be different. This approach empowers a state and transition fault coverage [14]. This test flow requires the synchronisation between the SUT and the test realisation of the IoT resource. In case of chosen a physical IoT resource, knowledge about the required time duration for initialising the IoT resource after power-on or reset is required. In addition, the power-on or reset capability need to be either triggered manually by pushing a button or need a strong debugging/controlling interface, capable of resetting the hardware by software. Both solutions tend to be complex in case that more than one IoT resource is involved. By choosing a solution with virtual resources realising this functionality is easier, but there is still the need to synchronise the virtual IoT resource with the test framework. Due to the complex model, this reset process will take time. It is very likely that it takes the same amount of time as required for utilising the physical hardware (realistic behaviour). It is obvious that the proposed solution of a resource emulation interface, directly controlled by the test framework, has large potential to simplify and accelerate the testing process due to timing issues of the competing solutions.

VI. CONCLUSION AND OUTLOOK

Testing of IoT-based services tends to be much more complex than testing of web services. Due to the distributed and heterogeneous nature of IoT resources controlling and timing of the involved components become difficult and resource intensive. To overcome current limitations of testing with physical IoT resources or virtual IoT resources, we propose a code insertion methodology, which enables a scalable and simplified solution for IoT-based service testing. The paper describes an example how a stream X-Machine model can be derived from service interface descriptions based on ontologies and rules. The approach is extended to empower the proposed resource emulation interface. Afterwards, the knowledge model is transferred into a prototypical realisation. Finally, the proposed approach is summarised and compared to classical approaches. Discussions about the IoT-based service example point out that the resource emulation interface enables a simplified, more flexible and scalable solution to test IoT-based services without the involvement of IoT resources. Future work will investigate whether the emulation model can be extended either by physical IoT resource measurements (e.g. typical response delays) or more advanced semantic descriptions of the IoT resource.

VII. ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Union Seventh Framework Programme for the IoT.est project under grant agreement n° 257521.

REFERENCES

[1] I. Chatzigiannakis, S. Fischer, C. Koninis, G. Mylonas, and D. Pfisterer, "Wisebed: an open large-scale wireless sensor network testbed," *Sensor Applications, Experimentation, and Logistics*, pp. 68–87, 2010.

[2] A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal, "Kansei: A high-fidelity sensing testbed," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 35–47, 2006.

[3] A. Gluhak, S. Krco, M. Nati, D. Pfisterer, N. Mitton, and T. Razafindralambo, "A survey on facilities for experimental internet of things research," *Communications Magazine, IEEE*, vol. 49, no. 11, pp. 58–67, 2011.

[4] J. Díaz, A. Yague, and J. Garbajosa, "A systematic process for implementing gateways for test tools," in *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*, pp. 58–66, IEEE, 2009.

[5] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, pp. 297–312, 2012.

[6] R. Tönjes, E. S. Reetz, K. Moessner, and P. M. Barnaghi, "A test-driven approach for life cycle management of internet of things enabled services," in *Procedings of Future Network and Mobile Summit, Berlin, Germany*, pp. 1–8, 2012.

[7] E. Reetz, D. Kümper, A. Lehmann, and R. Tönjes, "Test driven life cycle management for internet of things based services: a semantic approach," in *VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, pp. 21–27, 2012.

[8] A. Urbieta, E. Azketa, I. Gomez, J. Parra, and N. Arana, "Analysis of effects-and preconditions-based service representation in ubiquitous computing environments," in *Semantic Computing, 2008 IEEE International Conference on*, pp. 378–385, IEEE, 2008.

[9] H. Huang, W. Tsai, and R. Paul, "Automated model checking and testing for composite web services," in *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pp. 300–307, IEEE, 2005.

[10] E. Ramollari, D. Kourtesis, D. Dranidis, and A. Simons, "Leveraging semantic web service descriptions for validation by automated functional testing," *The Semantic Web: Research and Applications*, pp. 593–607, 2009.

[11] W3C, "RIF Production Rule Dialect (RIF-PRD)." Available online at http://www.w3.org/TR/rif-prd/ retrieved: January, 2013.

[12] F. Ipate and M. Holcombe, "Testing data processing-oriented systems from stream x-machine models," *Theoretical Computer Science*, vol. 403, no. 2, pp. 176–191, 2008.

[13] Testing Technologies, "TTworkbench." Website. Available online at http://www.testingtech.com/products/ttworkbench.php retrieved: January, 2013.

[14] A. Simao, A. Petrenko, and J. Maldonado, "Comparing finite state machine test," *Software, IET*, vol. 3, no. 2, pp. 91–105, 2009.