Test Driven Life Cycle Management for Internet of Things based Services: a Semantic Approach

Eike S. Reetz, Daniel Kümper and Ralf Tönjes
Faculty of Engineering and Computer Sciences
University of Applied Sciences Osnabruck
Osnabrueck, Germany
Email: {e.reetz, d.kuemper, r.toenjes}@hs-osnabrueck.de

Anders Lehmann
Institute for Business and Technology
Århus University
Århus, Denmark
Email: anders@hih.au.dk

***Abstract*—Concepts for Internet of Things (IoT) are currently limited to particular domains and are tailored to meet only limited requirements of their narrow applications. To overcome current silo architectures we propose a business oriented service composition of IoT enabled services with (semi-) automated model based testing capabilities. Explicit description of services as well as the target environment allows for automated design and execution of tests, hence enabling fast and robust IoT based service provision. This work proposes a semantic description of the test design and execution process to enable reasoning of test behaviour and suitability in the different phases of a service life cycle. The proposed work describes a test model and an appropriate test architecture. A first testbed implementation demonstrates their applicability. The proposed approach enriches current views of IoT architectures with knowledge from the field of service oriented architectures and makes them usable in distributed environments with partial unreliable resources by introducing a formalised integration of automated testing into the life cycle management.**

***Keywords*-model based testing; Internet of Things; life cycle management; semantic test description**

I. Introduction

Seamless and transparent integration of smart objects into the environment is an open research topic for almost 20 years. Several aspects of a smart interaction between the virtual and the physical world from low level resource constraint sensors and actuators to high level description and interaction capabilities based on context-awareness have been proposed so far, resulting in several isolated solutions for the Internet of Things (IoT). Nevertheless, most of the proposed approaches address only the open issues of a specific application domain. Moreover, the limited interoperability between different silo solutions tends to prevent mass market services and service composition. To overcome these technological limitations, we propose a flexible service creation environment for IoT in order to dynamically design and integrate new types of services and therefore enable new business opportunities.

Due to interactions with the real world and a large variety of involved technologies these services have to be very flexible and robust. This requires enhanced testing capabilities already included in the service creation process. Our approach is not only to pursue a test-driven service life-cycle management but also to automate the testing process appropriately. Testing in the IoT domain is a challenging task: The diversity and distribution of involved components as well as their unreliability raise current research issues. Furthermore, the need for realistic conditions results in a complex testing environment. Moreover, the dynamics of the IoT environment make the development and maintenance of services an error prone challenge.

The presented concepts have been conducted in the scope of the European research project "IoT.est" (the overall project concept can be found at [1]). This paper focuses on the aspects related to automated testing by describing the current status of the approaches and their limitations.

The overall contribution of this paper can be summarises as follows: we propose a test-driven life cycle management which can be utilised for rapid IoT based service creation and deployment based on automated testing. Therefore our overall concepts of a test-driven life cycle management, semantic descriptions of service and tests, and the derived test architecture are envisaged. We are following an Service Oriented Architecture (SOA) and therefore we are extending classical approaches by adding enhanced testing capabilities (e.g, test automation, emulation of network, resources and real world context), which we believe enables the applicability of SOA to the IoT domain.

The rest of the paper is structured as follows: after giving a brief overview of the State of the Art in Section II the IoT service concept will be briefly explained in section III. Afterwards the model based testing approach is introduced in section IV and the test architecture is described in section V. First prototype implementations and testing principles are then presented in section VI. Conclusions and future work finally conclude this paper.

II. Related Work and Open Research Issues

Current solutions do not consider test-friendly automated development of services. Some approaches like the UML 2.0 Testing Profile [2] provide concepts for designing and

developing black-box tests but do not provide guidance how to utilise it. The abstract UML 2.0 Testing Profile (U2TP) notation has to be transformed into a test specific programming language like TTCN-3 [3] or JUNIT [4]. TTCN-3 specifies tests and how they have to be executed. Several tools provide an environment for test creation and execution. Nevertheless, a drawback of TTCN-3 is the lack of simplicity in the generation of tests since the users need to learn a new language. Our approach tries to overcome this drawback by automated test case design and execution during service development.

Initial concepts of software engineering processes applied phase-models like the waterfall model and assume software can be split into sequential phases. Iterative models such as the spiral model or V-model change this paradigm by considering more iterative approaches. The V-model introduces testing to the phases of the waterfall model. In recent years, agile development processes such as Extreme Programming have been introduced. One important outcome of of the agile development approach is the test-first method, which aims at a test-driven target orientated service development process [5]. A Pattern Oriented Software Development (POSE) approach for web service development has been proposed by Chengjun [6]. A pattern represents components and relationships amongst them. For each identified POSE process a pattern is built; it comprises the activities, goals, architecture definition and validation. Adapting these software development processes to IoT service development rises some challenges: the support for services at different levels of granularity, the support for service composition, consistency checking, the identification of dependencies and the service development as contentious process. For automation purposes detailed descriptions of processes and resources are crucial to control the IoT resource effects in the service life cycle, especially during the testing phase.

III. IoT Service Concept

The investigated approach of IoT enabled services shows similarities to classical service oriented architectures (cf. [7]). Our model for IoT enabled business services extends the classical consumer/provider role concept by connecting IoT resources (Inspired by the EU Project Sensei [8]) to the service component and differentiate between Atomic Service (AS) and Composite Service (CS). The AS is the smallest separable, which could be either a classical web service or an IoT service. IoT services access sensors and actuators by abstracting their interfaces and capabilities. They provide an interface based on SOA interfaces (e.g., RESTful), which enables reusability by other entities. A CS is a conjunction of atomic or composed services and integrates the business flow perspective into the service. The CSs are modelled with a Business Description Language like Business Process Model and Notation (BPMN) or Business Process Execution Language (BPEL), thus providing an abstract way to design the service and accelerating evaluation of services and integration of business logic.

IV. Model based Testing

Due to dynamic and unreliable components involved in IoT based services efficient planning of resources for testing is required. In order to address test levels with a certain coverage it is necessary to simplify and automate the test design and execution process. Therefore, model based testing is a promising approach to improve the IoT based services by generating test cases from models extracted from the service description. Figure 1 highlights the well known concept of model based testing. Abstract test cases can be derived from a service model, which is a partial description of the service. The figure shows an extension to this classical approach by introducing an explicit model of the environment in which the service is executed. The explicit description of an environmental model can be utilised to build components for emulating the environment in an abstract and executable way. Modelling the environment appears crucial for convincing test results due to the distribution and unreliability of IoT components as well as due to the interaction with the real world. Different to the classical SOA domain of web services, IoT services require a novel paradigm to model the expected interaction with physical and virtual objects. Enhanced models of the environment assure more sophisticated testing capabilities of resources, network and real world effects to the System Under Test (SUT).

The next subsection explains how model based testing is integrated into the life cycle management and highlights the different scopes of the testing process within various phases of the life cycle. Afterwards, a semantic test model is proposed, enabling reasoning of test behaviour and suitability in the different phases of a service life cycle with self-explanatory derivation and execution of test cases.

A. Test driven life cycle management

In order to understand why life cycle management is important, we need to keep the paradigm shift of SOA in mind. A SOA enables an improved alignment of business and IT needs. Due to the composition of services logic can be separated from the implementation. Classical approaches tend to decide how to achieve quality of service, security, or combinations of functions at design time and thus reduce the flexibility of the business processes. Another improvement is the reduction of the time to market since the decomposition of applications into services increases sharing and reusability of services. Different stakeholders as well as the integration of IoT enabled business services require coordination and collaboration in terms of service design, execution and testing. This results in the need of a a common understanding of the service life cycle process. In advance to these classical outcomes of a well defined

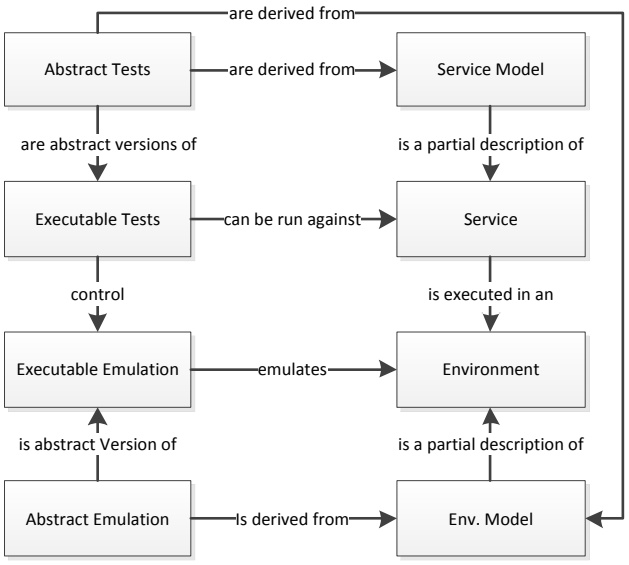

Figure 1. Extended model based testing approach

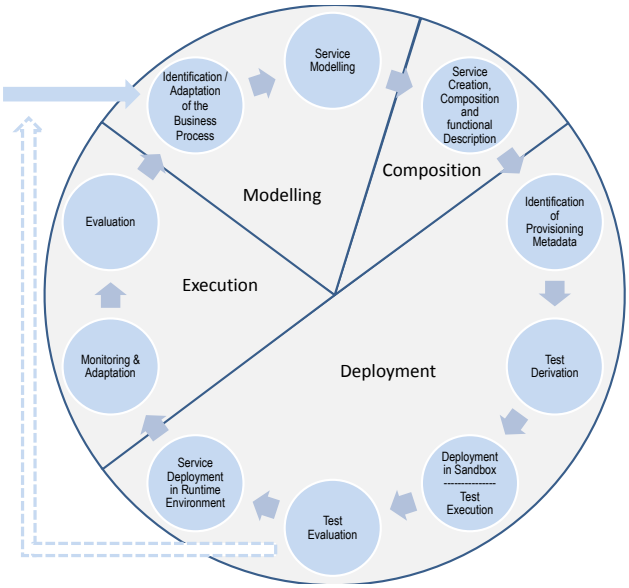

Figure 2. Test driven Life Cycle Management

life cycle a management process enables to (semi-) automatise the test process based on the semantically described process and the service itself. Therefore knowledge about the functional and non-functional behaviour of the service as well as the test modelling and execution are explicitly described by utilising a machine and human interpretable semantic description. The proposed life cycle approach takes advantage of well known phase models. Nevertheless we believe that the explicit integration can significantly enhance the applicability for IoT based services. This test description enables reasoning of test behaviour and suitability in the different phases of an IoT service life cycle. In these life cycle phases (cf. Figure 2), the focus of interests is different.

In the *modelling* phase the focus is on making the service perform according to the functional specifications. Thus the focus is on functional testing, i.e. unit tests and integration tests.

In the *composition* phase the focus is on building complex services by composing other services. In this phase it becomes more important to discover the atomic services needed to achieve the goals of the composite service. Likewise it is important to be able to discover the tests that effectively represent the composite service. These composite service tests need to make sure that the underlying services are available or can be emulated adequately. Therefore there is a need to gather this specific service composition information and add it to the description of the tests.

In the *deployment* phase a number of services is typically deployed. In order to be able to evaluate the success of the deployment, the semantic description of the services to be deployed can be checked for inconsistencies, contradictions and overlaps. The focus of the deployment is to prove that the deployed services will be able to deliver the services as promised in the Service Level Agreement (SLA). In the deployment phase the concrete environment for running the service is chosen. By adding information of the concrete environment the expected load can be determined by load tests.

In the *execution* phase the service provider measures relevant parameters of the services being executed in order to prove compliance with the SLA. By using the information added to the load tests in the deployment phase the service provider can predict when the services are close to reach the maximum capacity. These results can also be used to set up alarms, which will be triggered if the measured parameters indicates imminent breach of the SLA. The effect of the alarm can then lead to a dynamic re-selection of the atomic services in utilisation.

The detailed steps of the test driven life cycle management are shown in Figure 2. Its original purpose is to identify the business process requirements and goals and categorise them into different life cycle phases (short term and long term requirements). The categorisation assures a fast ability to demonstrate first results and helps to adjust requirements during the life cycle process. The next step, *Service Modelling*, decomposes the business process and tries to identify possible service components, taken into account that already available services should be reused if possible. As in all steps, requirements from previous steps are evaluated (in terms of feasibility) and new requirements are identified. Modelling goals ensures the proper identification of the required service components.

Contrary to the first steps requiring rather manual actions, the *Service Creation and Composition* phase is supported with tools to discover and compose services. The outcome

of this step is a deployable service including a semantic service description. The next phase takes care of required meta data for service provisioning. This includes semantic descriptions of the service contract and the service run-time where the service is be to deployed. With this information the *Test Deviation* phase can reason about the semantic descriptions in order to build test cases as well as the test execution flow. Afterwards the service is executed within a sandbox environment. The sandbox is controlled from the test execution engine (to be discussed in section V) and intends to act as the service run-time environment under realistic conditions in terms of load, traffic and competing services running in parallel. The execution of test cases results in the test evaluation. The test outcome is finally compared to the expected outcome and if the tests pass successfully the service is available for deployment.

The *Service Monitoring and Adaptation* phase takes place if the service is deployed successfully. Service monitoring based on current behaviour can result in dynamic re-selection of utilised atomic services. In addition, the monitoring phase discovers if the service consumption is as expected, for example if the number of request per minute fit to the expectations. From the discoverer information further needs for the next life cycle as well as adjustments for the sandbox are detected within the *Evaluation* phase.

Contrary to classical life cycle approaches this life cycle is driven by the semantic description of the service, the service run-time environment and the test environment. Therefore, it is possible to automatise and integrate the test design and execution into the service design-time. As a benefit there is a clear separation between developing and testing, i.e. the developer does not create the test cases explicitly, which might result in a non-optimal test coverage. Moreover, there is a kind of integrated testing since the test cases are automatically built and executed in a controllable sandbox. This results in fast feedback and rapid service improvement during design time.

B. Semantic Test Model

The procedure for testing is closely related to service process modelling; the service models described in [9] are used to describe test cases, test data and the test flow. This work employs the concepts defined in the OWL-S [10] to specify the service test semantics – including inputs, outputs, preconditions and effects (named IOPE) used for behavioural description of the service interface and the resource interface connecting to various IoT resources.

The description of the test model within a test ontology is the basis for automated test case creation and execution. The test ontology enables the creation of reusable test cases for an SUT and the modelling of the test flow which will be executed (cf. Figure 3). A Test Design Engine (TDE) utilises precise service descriptions and a knowledge base containing business expert knowledge, test data generation

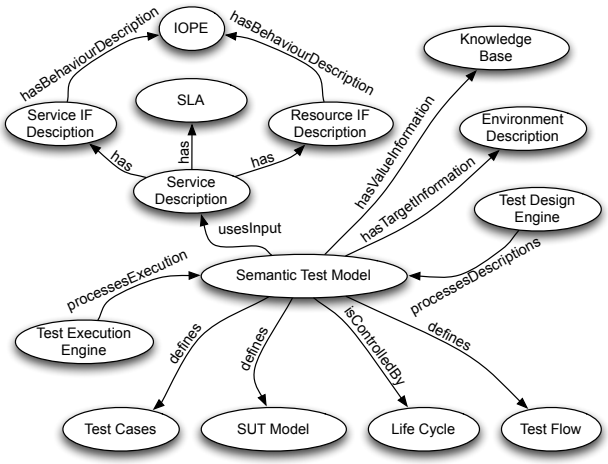

Figure 3. Semantic Test Model

and a test oracle for deriving tests, distinguishing different types, e.g., functional or reliability, and levels of tests, i.e., unit, interface, integration, or collaboration test. The service description thereby also supports the interface description of external IoT resources used by the generic emulation interface (cf. Section V-C). Attribute values of service descriptions are constrained by a min and max value or a value list and an optional default value. Moreover, events are defined to describe transition of states and events. Reasoning engines, e.g., rule based systems, can exploit the knowledge to derive the behaviour model and constrain the test cases, i.e., behaviour model plus test data. The defined test flow enables the Test Execution Engine (TEE) to process different tests and ensures the desired coverage regarding different states and paths of the finite state machine of the service.

The semantic test model is involved in different stages of the services and enables a highly automated test management.

V. Test Architecture

As mentioned in the previous section, due to the real world interaction and the lack of control of components involved in atomic and composite services, tests can not be executed in a productive environment. Our approach integrates systematic testing into the life cycle management. Therefore each service that is designed will be tested in a (semi-) automated way before being deployed. The SUT will be placed in a so called sandbox, which emulates the target environment as realistically as possible – not only functionally but also in from a real world, e.g., network and resource oriented, point of view. In order to achieve automated test case creation and execution each SUT needs to be described semantically. Although tests based on the semantic description can only detect whether the service acts as described and not as it was imagined by the developer, the test automation promises to overcome current limitations

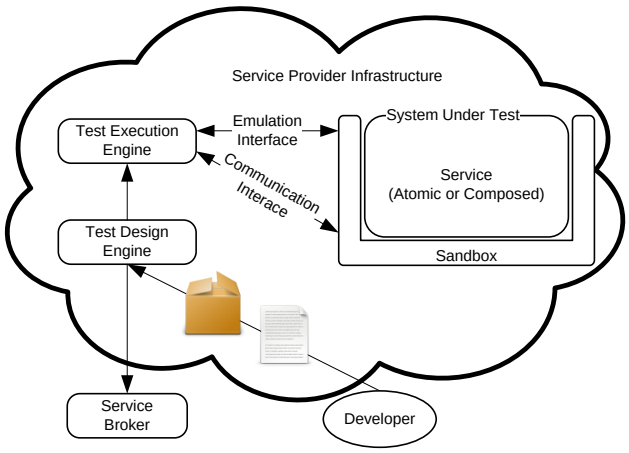

Figure 4. Test Environment for IoT enabled Composite Service

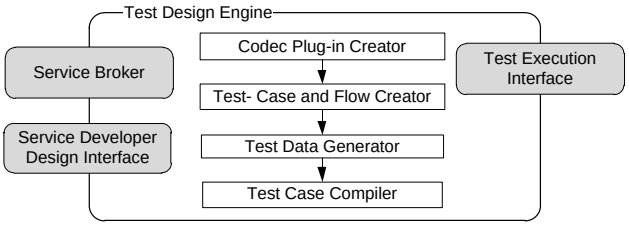

Figure 5. Test Design Engine for IoT based Composite Service

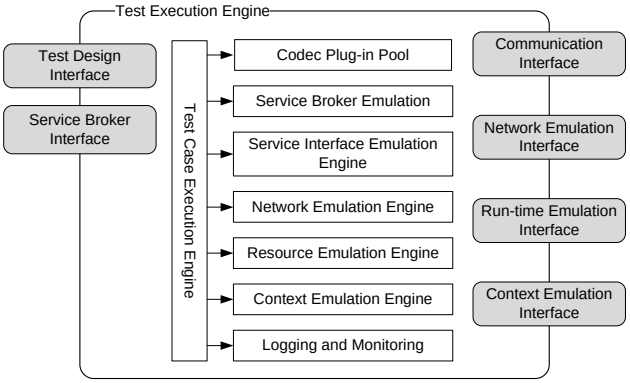

Figure 6. Test Execution Engine for IoT based Composite Service

as far as complex and distributed IoT enabled composite services are concerned and can improve the service quality significantly. Figure 4 depicts the main components of the test architecture for IoT enabled services. The SUT can be either a AS or a CS. The *sandbox* ensures that the behaviour can be emulated according to the test cases. This includes the emulation of network, hardware resources and IoT resource related parameters and characteristics. The Test Execution Engine (TEE) controls the environment and executes the test cases. The TDE is responsible for deriving the test cases from the semantic description of the SUT and generates test data. The test creation process is triggered either by the upload of a new or updated service from the Service Developer (including semantic description) or by the detection of new or changed service elements in the Service Broker lookup, which might be selected from a CS at run-time. The proposed test environment is located at the service provider infrastructure of the SUT and does not consider white-box unit tests from the service developer perspective.

A. Test Design Engine

The Test Design Engine (TDE) is responsible for create test cases for new and changed services and takes care of preparing their execution. The main functions and interfaces are shown in Figure 5). The TDE is trigged via the Service Developer Design Interface by transmitting the SUT together with a semantic description. In the first phase a *Codec Plug-in Creator* will identify which protocols are utilised by the service interfaces and, if required, create a new codec in order to abstract the protocol flow. Afterwards, the test cases for functional and non-functional tests are created and the order of execution is defined based on a extended finite state machine of the service. The test cases are described with the standardised Test Control Notation Version 3 (TTCN-3) language. Afterwards, the test cases will be enriched with generated test data based on the IOPE conditions of the semantic service description. The Test Case Compiler produces executable code from the test cases and assures a native test case execution by sharing test case executable code via the Test Execution Interface with the TEE.

B. Test Execution Engine

The Test Execution Engine (TEE) is the central component to coordinate the test flow. Figure 6 depicts the main components and interfaces of it. The execution is triggered by the Test Design Interface. While the *Test Case Execution Engine* takes care of the test execution it accesses emulation components in order to execute the SUT under controlled and emulated conditions of the target environment. This includes also the emulation of the Service Broker (in case of composite services) in order to control the run-time service selection. In case of a service request from the SUT the emulated Service Broker answers the request with the binding address of the *Service Interface Emulation Engine*. This assures that no external resources are involved in the execution and allows for testing all possibly correct and partly incorrect behaviours of the external service with fully controllable emulated components. Therefore, the test execution of composite service is capable of testing the interoperability of the connected services without directly executing the involved components.

C. Sandbox

The sandbox ensures that the SUT can be executed in a test environment and can be manipulated during the test execution (shown in Figure 7). In addition, the separation between the TEE, and the sandbox offers the ability to execute

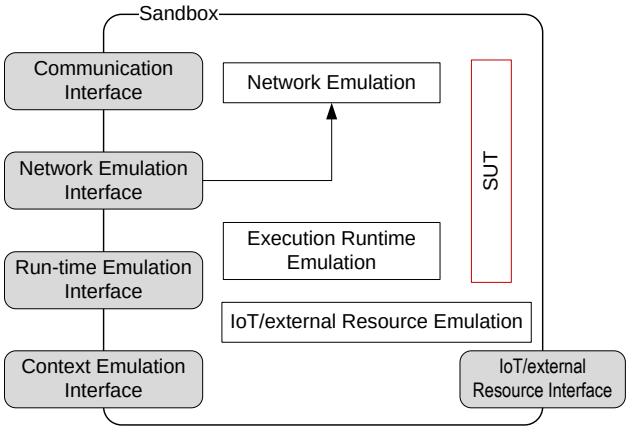

Figure 7. System Under Test Encapsulation

the tests in a distributed manner. The SUT service interfaces are connected with the Test Encapsulation Communication interface via the *Network Emulator*. Each message from or to the SUT can be manipulated in terms of delay and packet loss for evaluating the robustness. The network emulation is controlled via the Network Emulation Interface from the Network Emulation Engine of the TEE. Run-time behaviour changes are made by the Execution Runtime Emulation and assure the identification of potential SLA violations. The strict isolation of the SUT within the sandbox is realised by encapsulating interfaces to external web services or IoT resources. Therefore, the service description mandatorily includes the IOPE description to all external services or IoT resources. Nevertheless functional descriptions from interfaces only partly describe the utilisation flow of the interface (e.g, typically time deltas between request, typical response delays). To overcome this limitation the inclusion of a capture and reply mechanism is intended in order to reuse real communication with IoT resources and inject the traffic back in the test mode.

VI. Prototype Implementation

The requirements of IoT enabled business services raise a lot of open test issues as mentioned earlier. In order to explain some of our concepts by example we have implemented parts of our proposed architecture to highlight the complexity of a (semi-) automated test case creation and execution approach. In the outlined example test cases are designed for our Context Provisioning Middleware [11]. The test case execution validates if a lookup service interface is working properly (integration tests). The Context Broker is requested via a HTTPS/Get interface. The correct answer is expected to be encoded in an Extensible Markup Language (XML) based language called ContextML [12] described within a provided XML Schema Definition (XSD) file.

As outlined in the previous section the Test Design Engine (TDE) prepares the test cases for execution. After recognising the new service the TDE identifies an appropriate codec for HTTP and XML. Hence, knowledge from the defined structure of the XML data is transferred into a codec and the constraints of the provided XSD file are utilised for building the expected data structures in the TTCN-3 format (cf. [13]). An element of the related structures is shown in Listing 1. Basically it enables casting the received data stream into the expected XML structure. Based on the XSD description the possible data structure consists of a *scopeEl* element that consists of a list of *par* elements structured in a *parA* element. In addition, the expected data types and data items are described. The next step is to build the test cases and the test execution flow. Even a very simple example like this can illustrate the required complexity of an automated process. Like many other data types, the XSD description allows interlaced structures without limitations of the length. Hence, it has to be tested against dynamic data structures. The templates depicted in Listing 2 are utilised to test the interface against the expected data input. As shown in Listing 2 the template restricts the data item ‘n‘ to the string ‘scope‘ and allows only letters from a-Z and numbers of 1 to 15. But where does this knowledge originate from? Data constraints like these are not included in the XSD description. Therefore the services needs an additional semantic description as proposed in IV-B.

```
type record scopeEl {
  record {
    XSDAUX.string n,
    record of record {
      XSDAUX.string n,
      XSDAUX.string content
    } par optional
  } parA
}
```

Listing 1. XML Structure Described with TTCN-3

The templates are utilised by a test execution function. A control structure defines the flow of the test execution. The test execution function sends a response to the SUT and

```
template scopeEl.parA.par[-] parTest := {n := "
    scope", content := pattern "[a-zA-Z]#(1,15)"};
template contextML responseCheck(template scopeEl.
    parA.par p_list) := {
  content := {
    scopeEls := {
      scopeEl := {
        {
          parA := {
            n := "scopes",
            par :=  p_list
...
}
template GETInfo getInfoAuth := {...}
```

Listing 2. TTCN-3 Templates

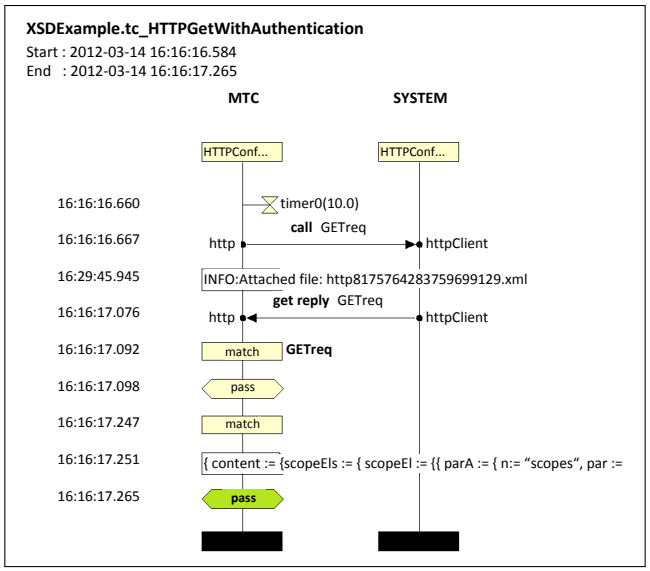

Figure 8. Test Execution Screenshot

validates the response with the shown template. Figure 8 shows the resulting graphical view of the the test execution realised with a tool called TTworkbench [14]. It shows the interaction flow between the Main Test Component (MTC) and the System to test. After receiving the *get reply* the test case is marked as *pass*.

VII. Conclusion and Future Work

In this paper we discussed that the domain and application boundaries for IoT can be overcome with a business oriented service composition. Our life cycle management takes into account that IoT enabled services need to cope with the abstraction of heterogeneity, reliability and robustness by integrating (semi-) automated self-testing capabilities. Our model based testing approach addresses these issues and identifies a semantic test description enabling reasoning of test behaviour and suitability in the different phases of a service life cycle. An appropriate test architecture has been presented. Practical problems are discussed based on an IoT based service with a typical web interfaces. Due the broad scope of the paper the discussions are rather concentrated on a high level and future work work will include a more detailed description of the proposed approach, which appears to be a promising way utilising SOA in the IoT domain.

VIII. Acknowledgement

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

References

[1] 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 2012, Berlin, Germany*.

[2] I. Schieferdecker, Z. Dai, J. Grabowski, and A. Rennoch, "The uml 2.0 testing profile and its relation to ttcn-3," *Testing of Communicating Systems*, pp. 609–609, 2003.

[3] ETSI, "The testing and test control notation version 3 (ttcn-3)." European Standard 201 874, 2002/2003.

[4] Y. Cheon and G. Leavens, "A simple and practical approach to unit testing: The jml and junit way," *ECOOP 2002 Object-Oriented Programming*, pp. 1789–1901, 2006.

[5] M. Huo, J. Verner, L. Zhu, and M. Babar, "Software quality and agile methods," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pp. 520–525, IEEE, 2004.

[6] W. Chengjun, "Applying pattern oriented software engineering to web service development," in *Future Information Technology and Management Engineering, 2008. FITME'08. International Seminar on*, pp. 214–217, IEEE, 2008.

[7] G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," *Software Engineering*, pp. 78–105, 2009.

[8] M. Presser, P. Barnaghi, M. Eurich, and C. Villalonga, "The sensei project: integrating the physical world with the digital world of the network of the future," *Communications Magazine, IEEE*, vol. 47, no. 4, pp. 1–4, 2009.

[9] W. Wang, S. De, T. Toenjes, E. Reetz, P. Barnaghi, and K. Moessner, "A comprehensive ontology for knowledge representation in the internet of things," *accepted for publication at KAMIoT 2012 in conjunction with IEEE IUCC-2012*, 2012.

[10] W3C, "Owl-s: Semantic markup for web services." W3C Member Submission 2004.

[11] M. Knappmeyer, N. Baker, S. Liaquat, and R. Tönjes, "A context provisioning framework to support pervasive and ubiquitous applications," in *Proceedings of the 4th European Conference on Smart Sensing and Context (EuroSSC)*, (Berlin, Heidelberg), pp. 93–106, Springer-Verlag, 2009.

[12] M. Knappmeyer, S. Kiani, C. Frà, B. Moltchanov, and N. Baker, "Contextml: a light-weight context representation and context management schema," in *Wireless Pervasive Computing (ISWPC), 2010 5th IEEE International Symposium on*, pp. 367–372, IEEE, 2010.

[13] I. Schieferdecker and B. Stepien, "Automated testing of xml/soap based web services," in *Kommunikation in Verteilten Systemen*, pp. 43–54, 2003.

[14] Testing Technologies, "Ttworkbench." Website. Available online at http://www.testingtech.com/products/ttworkbench.php visited on Sempteber 21th 2012.