Future Network and MobileSummit 2013 Conference Proceedings
Paul Cunningham and Miriam Cunningham (Eds)
IIMC International Information Management Corporation, 2013
ISBN: 978-1-905824-36-6

Test Derivation for Semantically Described IoT Services

Daniel KUEMPER[1], Eike REETZ[1,2], Ralf TÖNJES[1]
[1]*University of Applied Sciences Osnabrück, P.O Box 1940, 49009 Osnabrück, Germany*
Tel: +49 541 969-3899, Fax: + 49 541 969-13899
Email: {d.kuemper, e.reetz, r.toenjes}@hs-osnabrueck.de
[2]*University of Surrey, Guildford, GU2 7XH, UK*
Tel: +44 1483 300800, Fax: +44 1483 686011, Email: e.reetz@surrey.ac.uk

Abstract: Encapsulating IoT resources in services with comprehensive semantic descriptions of the involved components is a promising approach to overcome current silo architectures in the IoT domain. More important than the enablement of service composition, automated test derivation is expected to be on of the key drivers of rapid creation of robust IoT applications. The paper describes how concepts for semantically described web services can be transferred into the IoT domain. Therefore, methodology to enrich service descriptions for (semi-) automated test derivation and the required IoT specific adaptations are discussed in detail. First prototypical implementations prove that the envisaged lightweight extension can amplify the usefulness of web services descriptions for the test derivation for IoT-based services.

Keywords: IoT, Model-Based Testing, TTCN-3, Semantic WADL Extension, Test Derivation, Semantic Description

1. Introduction

Scalability and interoperability of Internet of Things (IoT) applications are key challenges of the next decade. One approach to overcome current silo architectures is the re-use of Service-Oriented Architecture (SOA) paradigms in the IoT domain. To realise services which are capable of communicating with heterogeneous IoT devices, the semantic gap between high-level description of services and the description of the IoT devices (e.g., actor and sensor) needs to be closed. The introduction of a service oriented architecture and a more advanced semantic description in the IoT domain offers two main benefits: i) composition of IoT services by re-using existing services and ii) facilitating (semi-) automated derivation of tests from the IoT service description.

The derivation of tests for IoT-based services requires extensive knowledge of data types, service behaviour and physical dependencies of resource constrained devices providing IoT data. In addition, the heterogeneous nature of sensors and actors, realised with a variety of radio technologies and communication protocols, depending on their purpose and manufacturer, needs to be taken into account by the service description and the test derivation process. The encapsulation of IoT into services eases the re-use of IoT resources by composite services: it enables a systematical approach to emulate the IoT resource behaviour during the test execution. Robustness tests of IoT-based services, require device emulation strategies to ensure the correct functionality of the service even if the underlying IoT resources might partly be unavailable (e.g, high delay, packet loss) or malfunctioning (logical failure, bad Quality of Information (QoI)). The test environment prevents extensive cost due to extensive resource emulation capabilities, whereby enabling a sufficient test coverage.

The proposed approach makes use of previous research from different areas. It utilises techniques from the domain of classical web services (service description, interface testing approaches), life cycle management, model-based testing and semantic knowledge modelling by adding IoT specific adaptations. In advance the combination of the different approaches will leverage the applicability of already investigated SOA approaches within the IoT domain. Depending on the available IoT resources, service encapsulation can be deployed in a service cloud, on specific IoT gateways or on the IoT device itself. Established protocols like Constrained Application Protocol (CoAP) [1] reveal easy integration of IoT resources as web services. This approach can even improve scalability of resource constrained devices and their networks by providing caching and load scheduling. The paper contributes to the issue if already available web service descriptions can be adopted to the IoT domain. The authors propose to modify existing lightweight solutions and add further knowledge only at identified test-related sections derived by the identified life cycle approach. First practical experience encourages to build a test derivation process based on the proposed knowledge extension.

The rest of the paper is structured as follows: in the next Section the progress beyond the state of the art is outlined. Afterwards, Section 3. explains in detail at which stage knowledge needs to be added to the service description in order to enable (semi-) automated test derivation. Section 4. explains our knowledge driven approach and the following Section 5. shows the resulting architectural approach. Section 6. discusses the approach based on an example service and the conclusion (Section 7.) completes the paper with an outlook.

2. State of The Art

While test environments (i.e., testbed) have been investigated by many projects (e.g., WISEBED [2], Smart Santander [3] etc.) systematic testing (i.e., software testing) in the domain of IoT has not been widely investigated so far. Nevertheless, some approaches, for example from the Probe-IT Project [4], have tested IoT protocols like 6LoWPAN using the Test Control Notation Version 3 (TTCN-3) Language [5].

Interface descriptions of web services (e.g., WSDL [6], WADL [7]) seam to be applicable to describe IoT-based services but require some adaptation to address the specific characteristics needed for testing(e.g., knowledge about valid and reasonable values of a specific physical unit). In advance, additional information about the non-functional description of the service needs also to be taken into account (e.g., QoI, QoS).

Model-based testing is one of the promising technologies to meet the challenges imposed on service testing. In model-based testing a System Under Test (SUT) is tested for compliance with a model that describes the behaviour of the implementation. Many different approaches of this behaviour model have been introduced (e.g., Extended Finite State Machine (EFSM) [8]) including standardised representations like the UML 2.0 Testing Profile (U2TP) [9] which can be automatically transformed into a test specific programming language like TTCN-3 [10]. Nevertheless, domain specific adaptation is needed to assure a flexible and scalable solution to provide testing capabilities for distributed IoT services, which are connected to heterogeneous IoT devices.

This work assumes that a common understanding of the service life cycle is crucial in order to build successful IoT services. To ensure a knowledge driven service composition and testing approach the annotation process of the service becomes eminent. While previous life cycle approaches like the classical V-Model or agile programming such as Extreme Programming [11] have already considered techniques like test-first [12] and test-driven development they did not explicitly describe the process of knowledge annotation.

3. IoT Service Life Cycle Concept

The proposed test derivation process of the IoT.est Project [13] is driven by a *Framework for IoT Service Life Cycle Management* (see Figure 1) [14] and accesses information of a semantic description framework, which is used for service search and composition, testing and deployment. The provided service descriptions deliver useful information for in- and output values as well as an important interface and deployment model for the SUT.

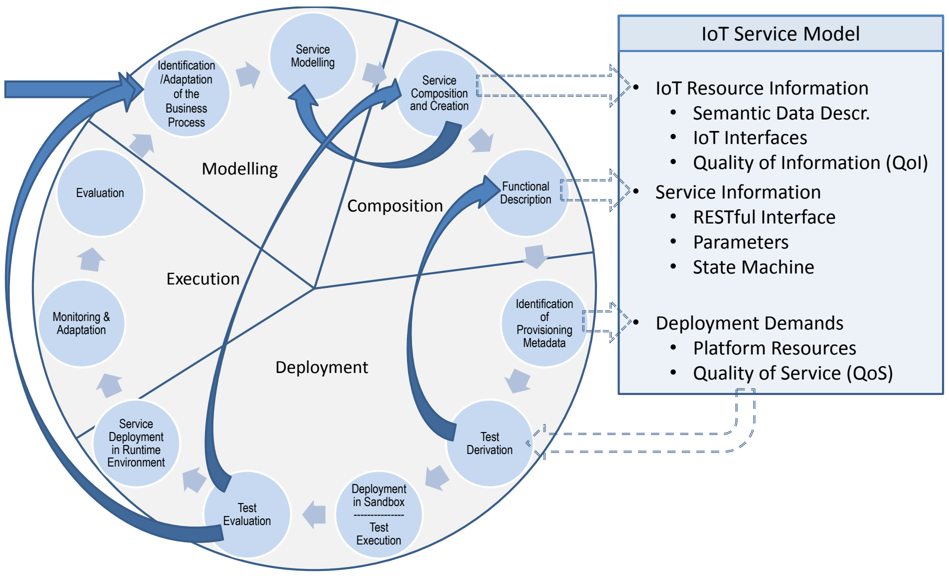


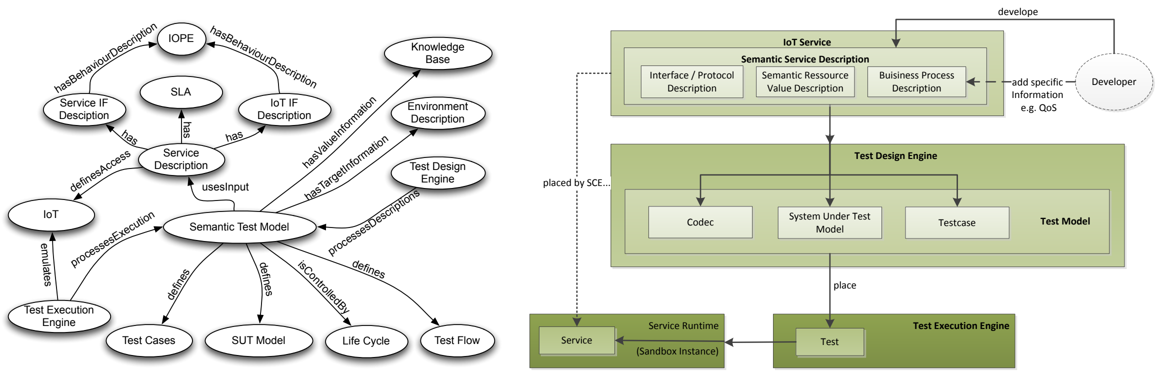
Figure 1: Service Life Cycle

The following steps have to be completed to ensure that sufficient information is gathered for the test derivation:

Service Composition and Creation: During the creation phase of the service, developers have to specify interfaces and semantic information for data types of the IoT resource access. Thus this information can be extracted during the derivation phase.

Functional Description: After the description of raw interfaces and data types that can be included during development via annotations, a semantic description of in- and output parameters as well as the stateful description of the service behaviour has to be added. A test oracle can be filled with test case data as well as being connected to data of the resource captured by the resource emulation interface. [14].

Identification of Provisioning Metadata: The provisioning metadata represents deployment information for the runtime environment (during testing: sandbox environment) as well as requirements to the Quality of Service (QoS) of the service. This information is needed for the non-functional test case derivation as well as the test case compilation.



(a) Semantic Test Model (b) Test Derivation Process

Figure 2: Semantic Test Model and Test Derivation Process

After these three steps in the service life cycle all required information for the test derivation is present. If one step can not be completed, for example, because wrong or insufficient information is present the service life cycle allows stepping back for adding more semantic description to the service.

4. Test Knowledge Modelling and Processing

The procedure for testing is closely related to service process modelling. The IoT knowledge models described in [15] are used to describe test cases, test data and the test flow. Concepts defined in the OWL-S [16] are used 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 test ontology is fundamental for automated test case creation and execution. The distinct definition of the service model across the test ontology enables the creation of reusable test cases for the SUT and the modelling of the test flow, which will be executed (see Figure 2(a)). The test derivation process is shown in Figure 2(b). For this purpose the Test Design Engine (TDE) utilises precise semantic service descriptions, containing state machine knowledge, semantic parameter descriptions and interface/protocol descriptions, which were added by the developer. These descriptions are utilised in order to select suitable test codecs, generate a SUT-Model with narrowed test data spaces and extract knowledge for the test case generation. The interface/protocol description enables the TDE to determine the access methods, input parameters (including data types) and the response format of services. An interface analysis is made in order to build an SUT model that later will be transformed into executable test cases described in TTCN-3. The model thereby also supports the interface description of external IoT resources used by the resource emulation interface. Attribute values of service descriptions are constrained. Moreover, events are defined to describe transitions of states and events based on the interface/protocol description. In the next steps the service model is enriched by further parameter constraints taken out of the semantic value descriptions. This description and ontology parsing exploits the knowledge to derive the behaviour model and constrain the test cases, i.e., behaviour model plus test data.

The defined test flow enables a Test Execution Engine (TEE) to process different tests and ensures the desired coverage regarding different states and paths of the finite

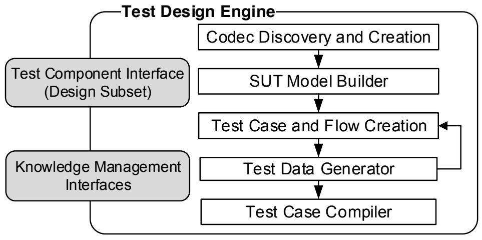

Test Design Engine
Codec Discovery and Creation
Test Component Interface (Design Subset)
SUT Model Builder
Test Case and Flow Creation
Knowledge Management Interfaces
Test Data Generator
Test Case Compiler

Figure 3: Test Design Engine Architecture

state machine of the service. The SUT will be tested within a service runtime, which is controllable form the TEE to ensure that the SUT can be tested without affecting the real world. The semantic test model is involved in different stages of the services and enables an automated test management. This test automation results in a consistent definition of the elements that are meant to test and a machine processable definition of the service life cycle including the test. Due to these definitions user interaction is only used to trigger a next step in the testing phase of the life cycle and thereby re-usability and traceability of test cases is ensured.

5. Test Derivation Architecture

The Test Design Engine (TDE) (see Figure 3) is responsible for generating test cases for new or changed services and preparing their execution. It is informed about a changed or newly developed service by receiving the location of a semantic description. In the first phase the Codec Discovery and Creation identifies which protocols are utilised by the service interfaces. If required it initiates the creation of a new codec in order to abstract the protocol flow. After the generation of a SUT model defining the internal behaviour of the service, based on a state machine, single test cases for functional and non-functional tests are created. Information for the generation of test cases will be derived out of the semantic description of the service that is stored in a knowledge base. This derivation is processed by the SUT Model Builder and the Codec Discovery and Creation component. By parsing the service description a SUT model is built and converted into test code utilising the knowledge described in the ontology and service description. The order of the test case execution is defined as a test flow. During creation the test cases will be enriched with generated test data based on the Input, Output, Precondition and Effect (IOPE) of the semantic service description. The Test Case Compiler produces executable code from the test cases and assures a native test case execution using the TTWorkbench [17].

Codec Discovery and Creation: In order to test services numerous codecs for an abstracted communication with the interfaces of the SUT have to be maintained whereby there is a high re-usability of the prepared codecs. The Codec Discovery and Creation component enables the TDE to obtain a suitable codec library that can be used for test case and flow execution. It fetches the protocol relevant semantic service description of a service and analyses the required codec capabilities.

SUT Model Builder: The SUT Model Builder is a software component, which is analysing the service description documents to build a TTCN-3 SUT model. The SUT model describes the interface and behavioural model of a service. The interface information contains used data types and parameter combinations for inputs and outputs. Therefore the SUT Model builder parses the service description and interprets the interface definition. The behavioural model describes states of the SUT that external stimuli have to be aware of, e.g. a two-way handshake, before you can access data from the service.

Test Case and Flow Creation: Utilising the Codec Pool and a SUT model derived by the TDE the Test Case and Flow Creation generates particular test cases to ensure test coverage of service states defined in the semantic service description. Due to the infinite number of test data based on the protocol and interface information, the Test Data Generation is used to improve test data based on its pertinence. A variety of test cases will be assembled to test flows in order to provide best possible coverage of services EFSM based on statement-, branch-, path- and condition coverage. The initial description of the generated test cases flow is accessible by the service creation in order to allow service developers the manual adaptation of test flows and cases.

Test Data Generator: The Test Data Generation utilises the information of the semantic service description regarding different, i.e. physical, value types as well as information to derive the EFSM of the SUT. It furthermore provides algorithms for test data reduction and optimisation, such as data fuzzing, regarding statement-, branch-, path- and condition coverage. The Test Data Generation also utilises a test oracle to use developer input for proper identification of the functional test results. This input is provided by annotations of the service descriptions.

Test Case Compiler: Utilising the standardised testing language TTCN-3, for service testing, the Test Case Compiler creates machine executable or interpretable code. It is invoked by the Test Case and Flow Creation during the test case creation. Compiled test cases can later be used by the Test Execution Engine for test case execution.

6. Test Concept Realisation Based on Example Service

6.1 Test derivation driven requirements to the service description

The proposed test derivation concept is driven by semantic service descriptions of services encapsulating IoT resources. The following line-up shows the utilisation of service descriptions by the TDE. It uses the semantic description of a stateless service which consists of a service interface (providing several operations) and accesses $1 - n$ IoT resource interfaces. The service is described by its IOPE from the perspective of the SUT and its interface. To achieve the derivation of test cases the following information has been identified as the minimal information set required to be modelled for one service interface and (0-n) IoT- resource interfaces accessed by the service:

Service Interface:

- A Service has a variety of interface operations that provide different types of input and output data.
- Representational State Transfer (REST) interfaces are based on URL, URI, HTTP commands, input parameters and return type.
- It has clearly to be defined how the mapping of REST resource URIs to parameters. Is handled e.g: `http://domain.eu/operation/resValue1/resValue2?parameter1=Value3&parameter2=Value4`

Precondition:

- Depending on the SUTs state machine the precondition describes the perquisite state of the described transitions. This stateless example does not change its preconditioned state.

Input Information:

- Data Types (e.g. Int32, Double or complex types based on XML or OWL).
- Data Constraints (e.g. positive values, $[1.0 - 10.0]$, distinct values $\{1.3, 4.4, 6.5, 8.3\}$).
- Semantic Data Description (e.g. location(countryName), location(zip)).
- Complex constraints of input parameter value dependency (e.g. $Par1 + Par2 < Int32.max$).

Timing information:

- Time after a service has to be completed from a functional view (not only on a QoS level).
- Definition of the duration after which the test-framework expects that there will be no more answer to the request sent.

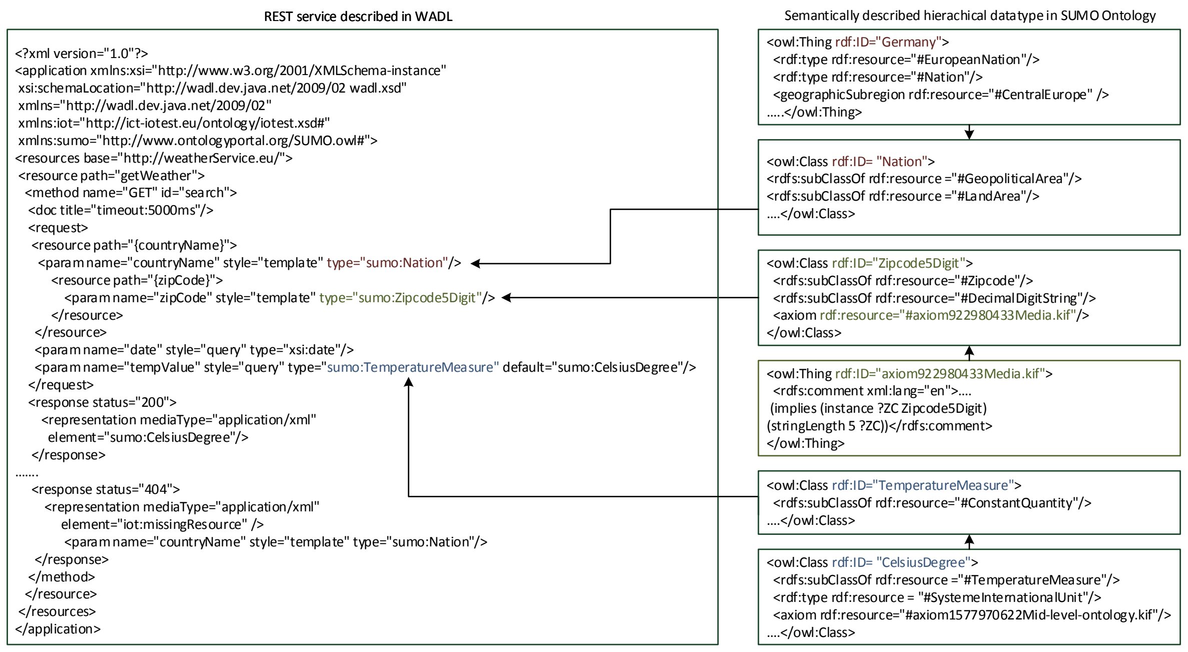
Output:

- Data Types and Constraints.
- Semantic Data Description (e.g. temperature value in Celsius, traffic value described in Owl, . . .).
- Test oracle data, which predicts distinct response data of the service.

Effect:

- Effect on the Service that depends on the Precondition and Inputs as well as behaviour model of the IoT resources. This has to be modelled as the resources IOPE to determine the new state of the service (if service is stateful). These IOPE also effect resources (e.g. actor states).

The example service in this paper provides temperature information by interpolating sensor information for a specific location. The service access is stateless and utilises a REST interface, which primarily will be used for service interaction in the IoT.est project. Based on the state machine, stateful services have to be described by an IOPE tuple for every possible transition in the state machine, whereby in this stateless example there is only one transition for every interface. At the moment different existing description technologies are utilised to realise which concepts can be reused and integrated in a definition of the service descriptions to achieve optimal description coverage. Figure 4 shows an exemplary implementation excerpt of the service description. It utilises the Web Application Description Language (WADL) for interface descriptions and links datatypes to the Suggested Upper Merged Ontology (SUMO) [18] for semantic description of in- and output parameters. This enables a remarkable reduction of the test data space. The service offers a getWeather-method. It requires the country name and a zip-code as REST resources in the URI and has two optional parameters to determine the measurement date (for history and forecast) and the intended type of temperature unit (with Celsius degree as default). An example call is: `http://weatherService.eu/getWeather/Germany/49078?date=2013-02-08&tempValue=CelsiusDegree`. The SUMO ontology used in this example ensures the consistency of parameters and their values. It supports a hierarchical description of the parameter meaning as well as lists of instances like nation names. Physical measurements are categorised and related units like Kelvin or Celsius are referenced by each other including their conversion rules (realised with Knowledge Interchange Format (KIF). WADL was used because of its full support of REST service descriptions and provision of an easy human readability and it is expected that the described concept can be easily transferred to other description standards. The SUMO ontology provides a wide spread formal domain knowledge and is easily expendable. The description technologies chosen in this example can be easily exchanged.


REST service described in WADL
Semantically described hierachical datatype in SUMO Ontology

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
 xmlns="http://wadl.dev.java.net/2009/02"
 xmlns:iot="http://ict-iotest.eu/ontology/iotest.xsd#"
 xmlns:sumo="http://www.ontologyportal.org/SUMO.owl#">
<resources base="http://weatherService.eu/">
 <resource path="getWeather">
   <method name="GET" id="search">
    <doc title="timeout:5000ms"/>
    <request>
     <resource path="{countryName}">
      <param name="countryName" style="template" type="sumo:Nation"/>
        <resource path="{zipCode}">
           <param name="zipCode" style="template" type="sumo:Zipcode5Digit"/>
        </resource>
     </resource>
     <param name="date" style="query" type="xsi:date"/>
     <param name="tempValue" style="query" type="sumo:TemperatureMeasure" default="sumo:CelsiusDegree"/>
    </request>
    <response status="200">
      <representation mediaType="application/xml"
       element="sumo:CelsiusDegree"/>
     </response>
.......
     <response status="404">
       <representation mediaType="application/xml"
          element="iot:missingResource" />
           <param name="countryName" style="template" type="sumo:Nation"/>
     </response>
    </method>
   </resource>
 </resources>
</application>
```

```
<owl:Thing rdf:ID="Germany">
 <rdf:type rdf:resource="#EuropeanNation"/>
 <rdf:type rdf:resource="#Nation"/>
 <geographicSubregion rdf:resource="#CentralEurope" />
.....</owl:Thing>
```

```
<owl:Class rdf:ID= "Nation">
<rdfs:subClassOf rdf:resource ="#GeopoliticalArea"/>
<rdfs:subClassOf rdf:resource ="#LandArea"/>
....</owl:Class>
```

```
<owl:Class rdf:ID="Zipcode5Digit">
 <rdfs:subClassOf rdf:resource="#Zipcode"/>
 <rdfs:subClassOf rdf:resource="#DecimalDigitString"/>
 <axiom rdf:resource="#axiom922980433Media.kif"/>
</owl:Class>
```

```
<owl:Thing rdf:ID="axiom922980433Media.kif">
 <rdfs:comment xml:lang="en">....
 (implies (instance ?ZC Zipcode5Digit)
(stringLength 5 ?ZC))</rdfs:comment>
</owl:Thing>
```

```
<owl:Class rdf:ID="TemperatureMeasure">
 <rdfs:subClassOf rdf:resource="#ConstantQuantity"/>
....</owl:Class>
```

```
<owl:Class rdf:ID= "CelsiusDegree">
 <rdfs:subClassOf rdf:resource ="#TemperatureMeasure"/>
 <rdf:type rdf:resource = "#SystemeInternationalUnit"/>
 <axiom rdf:resource="#axiom1577970622Mid-level-ontology.kif"/>
....</owl:Class>
```

Figure 4: RESTful Service partly described by WADL and the SUMO

6.2 *Test Derivation Process*

The implemented prototype is realised by following the subsequent steps to derive the SUT model and basic test cases:

Document Analysis: Analysing supported documentation language types. Accessing referenced `.xsd` and `.owl` descriptions to ensure integrity.

Interface Analysis: Parsing the documented interface of the service, extracting used protocols, URI and methods to build the basic SUT model. Determination of needed Codecs (e.g. HTTP/REST). The implementation is using jlibs [19] as parser.

Parameter Analysis: Analysing the in- and output parameter definitions and defining basic data types including their constraints for the SUT.

Semantic Analysis: Following ontology links and looking up instances and rules for defining complex parameters of the SUT-Model and setting constraints. Implemented using Apache Jena [20] for accessing ontologies. The following example SPARQL-query is based on the information of the WADL-description. It shows the request to get a list of worldwide nation names [1]:

```
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>
SELECT ?Nations WHERE {?Nations rdf:type sumo:Nation .}
```

Constraint Analysis: Intersecting derived and manual constraints to get the final model.

Testcase Generation: Utilising SUT model and manual data from test oracle to generate test cases.

[1]after the initialisation of the $34MB$ SUMO .owl file, the request needs $< 10ms$ on a Intel Core i7 1.7 GHz mobile processor to generate the country list

Test cases, created in the Test Case and Flow Creation, adapt the created TTCN-3 SUT model for injecting customised test data. The SUT model constrains the valid input data types by utilising the semantic parameter descriptions and enables the assessment of service response data. When creating test cases it is important not only to generate test cases with valid inputs but also to use invalid input values to check if it valid exception-responses are created by the service even if it is incorrectly used. Figure 5 shows a simplified TTCN-3 test code, derivated from the enriched service description.

```
module WeatherService {
  type enumerated sumo_Nation {"Denmark", "Ireland", "
        Norway", "Italy", "Germany" ....};
  type integer sumo_Zipcode5Digit (00000 .. 99999);
  type record xs_Date {
    integer year (0 .. 3000), integer month (1 .. 12),
          integer day (1 .. 31)
  }
  type charstring MissingResource;
  type charstring SystemException;
  type float CelsiusValue (−273.15, infinity);
  type float FahrenheitValue (−459.67, infinity);
  type float KelvinValue (0.0, infinity);
  type enumerated sumo_TemperatureMeasure {CelsiusDegree,
        KelvinDegree, FahrenheitDegree};
  signature getWeather(sumo_Nation countryName,
        sumo_Zipcode5Digit zipCode, xs_Date date optional,
        sumo_TemperatureMeasure tempValue := CelsiusDegree)
    return float exception (MissingResource,
          SystemException);
  type port WeatherPortType message {
    inout getWeather;
  }
  type component WeatherComponentType {
    timer localTimer := 5.0;
    port WeatherPortType weather;
  }
}
```

(a) SUT Model

```
module WeatherServiceTest {
  import from WeatherService all;
  testcase tcGetWeather() runs on WeatherComponentType
        system WeatherComponentType {
    map(mtc:weather, system:weather);
    localTimer.start;
    weather.send (getWeather: {"Germany", 49078, {year :=
          2013, month := 02, day := 08}, CelsiusDegree},
          5.0) {
      [] weather.receive (getWeather: {"Germany", 49078, {
            year := 2013, month := 02, day := 08},
            CelsiusDegree}
        value (CelsiusValue:?)) {
        setverdict (pass, "Valid value");
      }
      [] weather.catch (getWeather, MissingResource:?) {
        setverdict (fail, "Invalid request");
      }
      [] weather.catch (getWeather, SystemException:?) {
        setverdict (fail, "Communication exception");
      }
      [] weather.catch (timeout) {
        setverdict (fail, "Timeout occurred.");
      }
    }
    unmap(mtc:weather, system:weather);
  }
}
```

(b) Test Case

Figure 5: exemplary TTCN-3 Testcode

7. Conclusion and Future Work

The integration of semantic data models into formal service descriptions enables the reduction of manual test creation during the service life cycle. Detailed ontology knowledge of real world coherences can be utilised to gain extensive discrete test data sets of parameters and to narrow down valid value ranges for continuous datatypes. This is a valuable contribution for the (semi-) automated test derivation. The used SOA technologies show a good applicability in the IoT domain. The applied service approach lowers the obstacle for integrating IoT resources into existing service architectures. In combination with selected semantic description approaches it maintains the re-usability of services and facilitates (semi-) automatic test derivation and composition.

Next steps will be the formal specification for the combination of various description formats and ontologies, to combine interface definitions, semantic value specifications and service behaviour with a distinct and confined description set. For the improvement of test cases, further test data generation algorithms will be examined to narrow down the data space of continuous data types and generate adjusted test cases.

8. 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] G. Moritz, F. Golatowski, and D. Timmermann, “A lightweight SOAP over CoAP transport binding for resource constraint networks,” in *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pp. 861–866, IEEE, 2011.

[2] 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.

[3] J. Hernández-Muñoz, J. Vercher, L. Muñoz, J. Galache, M. Presser, L. Hernández Gómez, and J. Pettersson, “Smart cities at the forefront of the future internet,” *The future internet*, pp. 447–462, 2011.

[4] P. Cousine, Ed., “D3.4 - Guidline for IoT Validation and Deployment.” Probe-IT (Public Deliverabel), 2012.

[5] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, “An introduction to the testing and test control notation (TTCN-3),” *Computer Networks*, vol. 42, no. 3, pp. 375–403, 2003.

[6] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, *et al.*, “Web services description language (WSDL) 1.1.” W3C Note 15 March 2001. www.w3.org/TR/wsdl., 2001.

[7] M. J. Hadley, “Web application description language (WADL),” *Technical report, Sun Microsystems (November 2006), https://wadl.dev.java.net*, 2006.

[8] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, “A survey on model-based testing approaches: a systematic review,” in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: in conjunction with 22nd IEEE/ACM ASE 2007*, pp. 31–36, ACM, 2007.

[9] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch, “The UML 2.0 testing profile and its relation to TTCN-3,” in *Testing of Communicating Systems*, pp. 79–94, Springer, 2003.

[10] ETSI, *EG 201 873-1: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part1: TTCN-3 Core Language, Version 4.4.1*, April 2012.

[11] G. Mangalaraj, R. Mahapatra, and S. Nerur, “Acceptance of software process innovations–the case of extreme programming,” *European Journal of Information Systems*, vol. 18, no. 4, pp. 344–354, 2009.

[12] J. Andrea, “Envisioning the next-generation of functional testing tools,” *Software, IEEE*, vol. 24, no. 3, pp. 58–66, 2007.

[13] 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.

[14] 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.

[15] W. Wang, S. De, R. Toenjes, E. Reetz, and K. Moessner, “A Comprehensive Ontology for Knowledge Representation in the Internet of Things,” in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pp. 1793–1798, IEEE, 2012.

[16] D. L. McGuinness, F. Van Harmelen, *et al.*, “Owl web ontology language overview,” *W3C recommendation*, vol. 10, no. 2004-03, p. 10, 2004.

[17] Testing Technologies, “TTworkbench.” Website. Available online at http://www.testingtech.com/products/ttworkbench.php retrieved: April, 2013.

[18] I. Niles and A. Pease, “Towards a standard upper ontology,” in *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*, pp. 2–9, ACM, 2001.

[19] Santhosh Tekuri, “jlibs - common utilities for java.” Website. Available online at http://code.google.com/p/jlibs/ retrieved: April, 2013.

[20] Apache Foundation, “Apache Jena.” Website. Available online at http://jena.apache.org retrieved: April, 2013.