From Semantic IoT-Service Descriptions to Executable Test Cases - Information Flow of an Implemented Test Framework

Daniel Kuemper, Eike Reetz, Marten Fischer and Ralf Toenjes

Lab for RF-Technology and Mobile Communications
University of Applied Sciences Osnabrueck
Osnabrueck, Germany
Email: {d.kuemper,e.reetz,m.fischer,r.toenjes}@hs-osnabrueck.de

Elke Pulvermueller

Institute of Computer Science
University of Osnabrueck
Osnabrueck, Germany
Email: elke.pulvermueller@uni-osnabrueck.de

Abstract—**Automated test derivation is expected to be one of the key drivers of a rapid creation of robust Internet of Things (IoT) applications. The paper describes a two-step approach how concepts for semantically described IoT services can be used to derive functional test cases to test services in a sandbox environment. In the first step, the description of the service is used to generate a state based model of the service behaviour and its interfaces. Therefore, a methodology to enrich service descriptions for (semi-) automated test derivation and the required IoT specific adaptations are discussed in detail. These descriptions are used to generate customised test data and to achieve full parameter combination coverage. In the second step, the generated extended finite state machine model is analysed to create test cases in a standardised testing notation. Utilising this two-step automation approach enables test developers to evaluate and influence resulting test cases. The implementation proves that the envisaged extension can amplify the usefulness of web services descriptions for the test derivation for IoT services by reducing the effort to create and execute test cases.**

Keywords–IoT; Model Based Testing; Test Derivation; Semantic Annotation; RESTful; TTCN-3; WADL.

I. INTRODUCTION

Distributed IoT services are becoming increasingly complex since the usage of sensors and actuators with atomic functionalities brings along a high variety of heterogeneous interfaces [1]. Therefore, it is crucial to employ functional tests to evaluate faultless service interaction. Manual test creation causes a high effort in analysing interfaces and service behaviour to find suitable test cases. This effort can be reduced by employing model-based test approaches [2]. This work illustrates how a two-step model-driven testing approach, which utilises explicit information representation at different abstraction levels, can be used to create test cases for semantically described Representational State Transfer based (RESTful) IoT services whilst regarding their stateful behaviour. A fully automated model based testing approach needs very extensive Input, Output, Precondition and Effect (IOPE) descriptions [3] and deprives the control of the test developer to evaluate tests dependent on the services usage. Therefore, this work tries to lower the effort for the service description by enabling a use case-based sequence description approach. Furthermore, it aims at reducing the effort for manually enhancing service descriptions compared to a full manual test case creation.

The remainder of the paper is structured as follows: After the discussion of the current state of the art in Section II, the overall project concept and implemented architecture is outlined in Section III. Detailed descriptions of the utilised annotation methods are shown in Section IV. Section V depicts an IoT example service, which is used in Section VI to discuss the test derivation process and its model transformation. The conclusion and outlook section completes the paper.

II. RELATED WORK

In recent years, a lot of research efforts have been invested in providing efficient ways to automate the process of testing software. Different strategies have been developed in generating test cases and providing them with adequate test data. Basically, three approaches can be identified. First, finding suspicious code through code analysis [4]. This approach requires access to the source code of the System Under Test (SUT) and is able to find unreachable code and other violations to coding rules. The second approach tries to find implementation faults by exploiting public interfaces with a large number of randomly generated data [5]. This fuzzing approach can find security relevant implementation errors (e.g., buffer overflow) but on the other hand it produces a very large number of test cases of rather poor quality. The employment of Equivalence Class Partitioning (ECP) can reduce the amount of test data and test cases by defining valid and invalid arguments for the interface invocation [6]. By employing more precise semantic service descriptions, the approach proposed in this work tries to overcome solely random generations by taking reusable parameter range definitions into account.

The third group of approaches uses abstract behaviour models of the application to generate meaningful test cases. These test models are created manually, generated from source code or derived from other models through model transformation [7]. Walkinshow et al. [8] trace the execution of software to infer a test model. Different modelling languages including state charts, Petri nets, message sequence charts or Finite State Machine (FSM) can be used [9]. While executing a test case an execution engine iterates through the elements of the model, e.g., a transition in a FSM, to trigger the SUT and validate its output. Since the efficiency of the test case highly depends on the test data, a lot of research has be done in the field of

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-370-4

test data generation. The challenge is to find boundaries of the valid input space. Tracey et al. [10] exploit search techniques to automate the generation of test data. Evolutionary algorithms, namely Genetic Algorithm (GA) are used to derive test data from an initial data pool in [11] to have a fully automated test data creation. Here, a new generation has close relations to the generation before, thus focusing on relevant test data. Fischer et al. [12] propose the use of a GA to enhance the quality of automatically generated test data. IoT-based services are often based on energy restricted (e.g., battery driven) sensors and actuators that have a limited number of usage cycles. This IoT limitation hinders GA usage in testing due to the high amounts of test cases, which are used to optimise the test data. To overcome this limitation the proposed approach realises the optimisation of test data and test cases by using service descriptions before the service is tested.

In recent years several works investigated model based testing approaches for services. Ramollari et al. [3] create functional conformance tests by utilising IOPE sets without detailed interface descriptions. A semantic parameter conformance validation can be found in [13] missing the abstraction of detailed interface parameters and following a stateless approach. The commercial available solution [14] enables functional test creation based on web service interface descriptions. The approach of Schanes et al. [15] concentrates on Extensible Markup Language (XML) as generic data format for test execution. They both do not consider stateful service behaviour of reactive systems by modelling conjunctions between various methods of the service.

III. Concept and Architecture

The presented concepts are part of the IoT.est [16] project, which aims at developing an IoT service creation environment whilst bridging the gap between various business services and the heterogeneity of networked sensors, actuators and objects. The approach employs semantic service descriptions to compose IoT services and derive corresponding functional conformance tests, semi-automatically. After the manual annotation of a service the service model is generated automatically. It can be altered manually before the automated generation of test cases begins. A consistent service concept is specified to enable this process.

A. *IoT.est Service Concept*

IoT.est utilises RESTful interfaces to encapsulate IoT services for enhanced re-usability. It defines two types of services to ensure direct consumption and composition of IoT services without dealing with heterogeneous interfaces:

The Atomic Service (AS) is a RESTful web service, accessing $0 - n$ IoT resources via their own individual interfaces and radio technologies. It enables access to these resources via standardised `Get, Post, Put, Delete` request methods, whose invocation is defined in a Web Application Description Language (WADL) document [17]. Input parameters as well as service responses are extensively semantically described in the Knowledge Management (KM). The implemented AS can be deployed to a Runtime (RT) for web services and is registered in the KM.

The Composite Service (CS) enables a business process-based composition of various AS and CS. It also provides a RESTful interface for service invocation and does not directly connect to IoT resources using their proprietary interfaces. It only uses AS and CS interfaces to acquire sensor information and to control actuators. The interfaces are also described in WADL and a semantic description is used to enable re-usability for composition and testing.

B. *Architecture*

The IoT.est project architecture specifies a Test Design Engine (TDE), which enables the generation of test data and derivation of test cases and flows for IoT services (see Figure 1). The derivation is driven by processing service descriptions and utilising domain knowledge. IoT.est uses a KM to store descriptions of IoT services. These services can be deployed and composed via a Service Composition Environment (SCE) in distributed RT environments of the framework. To support testing by the Test Execution Engine (TEE) prior to runtime deployment we employ a Sandbox Runtime (SRT) instance of the RT. The SRT supports emulation of IoT resources [18] to enable IoT service testing without communicating with resource constrained IoT sensors or altering IoT actuators during test execution.

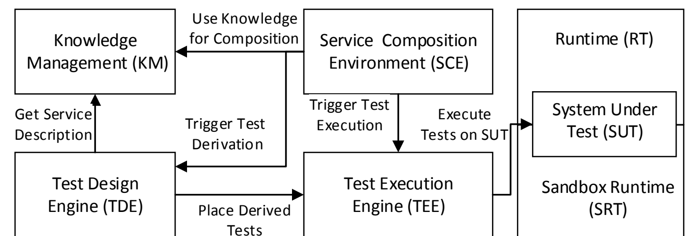


Figure 1. Simplified IoT.est Architecture.

To obtain a comprehensible test generation, the TDE utilises an explicit information representation approach, which can also be used to evaluate and alter the model and tests, which are automatically derived. During the first step the service model, which is generated from the semantic description, is represented as an Eclipse Modeling Framework (EMF)-model. It is editable with the Graphical Modeling Framework (GMF). During the second step, test cases are created in the ETSI standardised Testing and Test Control Notation Version 3 (TTCN-3) to obtain a readable and reusable representation.

IV. Service Interface Description

In this section, the different types of service descriptions are described. These descriptions are used in Section VI to build the EMF state machine model. The client–server communication of RESTful services is constrained by no client context being stored on the server between requests [19], although services can follow a stateful behaviour. Since the interfaces are implemented stateless there is a missing support of behavioural descriptions in established description notations like WADL. To enhance testability the proposed approach extensively describes service interfaces and also the service behaviour to get information for valid and invalid interface calls with test parameters depending on parameter values and current service states. The information is used to enable an ECP-based model generation.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-370-4

A. Precise Parameter Descriptions

The service model creation utilises service descriptions to find valid and invalid equivalence classes, which are used to model state-based transitions. The equivalence classes are processed by a boundary value analysis and random value generators to derive the test cases. Conventional service descriptions, based on WADL, describe resource parameters as implementation-specific technical parameters using well known data types like `string` and `double` (shown in Figure 2). This leads to a very simple equivalence class model, which accepts the whole data type as valid input although the application specific usage of the parameter can be restricted to a small range.

```
1 <resource path="/zoom/{id}/{value}">
2   <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="
       id"
3     style="template" type="xs:string" />
4   <param name="value" style="template" type="xs:double" />
5   <method id="setZoom" name="POST" />
6 </resource>
```

Figure 2. Basic WADL Parameter Descriptions.

The following paragraphs show examples of information, which is used to precisely define service parameters.

1) Simple Value Range Limitation: A fundamental approach of the enhanced service descriptions is to define the precise value ranges of parameters to gain an abstracted model of method parameters. This model is not only based on a technical data type that is used to transfer the information. It also specifies the defined value ranges processed by the service logic (e.g., a valve position between -25.0 and 15.5). A simple limitation of this parameter value in an XML-Schema is shown in Figure 3. Numeric data types can be restricted by value ranges and an enumeration of allowed values. Character data types can be restricted by the number of allowed characters, the length, an enumeration or a regular expression which could e.g., define an email-pattern. The mapping between a parameter of a method or resource in the WADL file is performed by namespaces, which do not require any extension of the existing WADL definition.

```
1 <?xml version="1.0"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3 <xs:element name="valve">
4   <xs:simpleType>
5     <xs:restriction base="xs:double">
6       <xs:minInclusive value="-25.0"/>
7       <xs:maxInclusive value="15.5"/>
8     </xs:restriction>
9   </xs:simpleType>
10 </xs:element>
```

Figure 3. Simple XML parameter restrictions.

The given example describes one valid (vP) and two invalid equivalence classes (iP). Since division by zero and switching between negative and positive values are typical code weaknesses, we divide the valid class into two using $-0, +0$ for boundary value analysis (see Figure 4). This methodology results in 4 disjoint classes: iP_1, vP_1, vP_2, iP_2

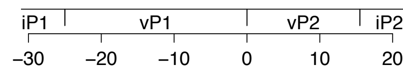


Figure 4. Equivalence Class Partitioning Example.

The definition of valid partitions is not only limited to a single value range it can describe various valid and invalid partitions for one parameter (see XML-Schema restrictions [20]). It also supports complex definitions of strings not only by enumerations but also with regular expressions. This way, e.g., an email address can be described as parameter input. The definition of regular expressions is then reused for the test data generation.

2) Semantic Parameter Description: The test data generation uses semantic annotations that can be linked to upper level ontologies like SUMO [21] for reusable test case derivation. Reusable parameter limitations can, e.g., restrict the range of a Celsius temperature and the possible temperature units or define e.g., sets of countries.

Semantic Parameter Interdependency: The description of service parameters has to take into account that they have interdependent connections to each other. Ontology documents take this into account by describing individuals using classes, relations and attributes. The value range of a parameter *Cityname* for example depends on the *Countryname* parameter since for example the city *Bologna* exists in *Italy* but not in *Germany*. The description of linking interdependent parameters on the predicate *geographicSubregion* is shown in Figure 5. The *owlType* definition is declared for each semantic parameter and linked to a class definition within the ontology by the *requestLink* tag (Figure 5:3,6). The *restriction* tag describes the predicate on which the interdependency is defined (Figure 5:7). Figure 6 shows the generated SPARQL Protocol and RDF Query Language (SPARQL) code that is used to find the matching entities in the ontology.

```
1 <owlTypeDefinition>
2  <owlType name="Countryname" type="base">
3   <requestLink uri="http://www.onto.org/SUMO.owl#Nation"/>
4  </owlType>
5  <owlType name="Cityname" type="restricted">
6   <requestLink uri="http://www.onto.org/SUMO.owl#City"/>
7   <restriction uri="http://www.onto.org/SUMO.owl#
        geographicSubregion" value="{Countryname}"/>
8  </owlType>
9 </owlTypeDefinition>
```

Figure 5. Semantic Parameter Interdependency.

```
1 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 prefix sumo: <http://www.ontologyportal.org/SUMO.owl#>
3 select ?city where {
4  ?city rdf:type sumo:City .
5  ?city sumo:geographicSubregion sumo:Germany .
6 }
```

Figure 6. SPARQL-Query.

B. Geospatial Testdata Derivation

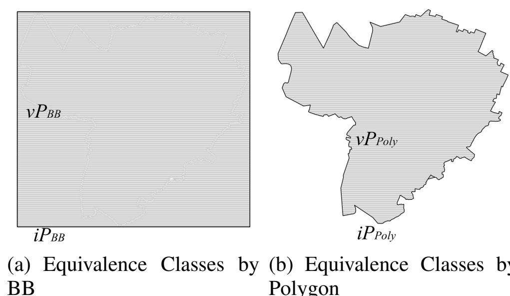
Since IoT-based sevices often cover specific areas, a geospatial description of services is very useful to determine functional conformance. A common description approach for geospatial areas is to describe a bounding box (rectangle)

defining the min. and max. latitude and longitude values that cover an area. This often leads to a very imprecise area description. A better way is to specify the precise geospatial coverage by defining a polygon(concatenation of a list of coordinates, surrounding an area), which defines the covered area. Since for a complex polygon like a city boundary it's a very long description it is not feasible that everybody annotates a precise polygon for every supported area. Therefore, we use an external knowledge base (OpenStreetMap (OSM) [22]) that can access those polygons just by annotating it with the city/country name. The following shows an example of the three annotation methods. You can see the resulting areas in Figure 7.

- Bounding Box (BB):
  longitude *min:11.2295654 max:11.4336305*,
  latitude *min:44.4211136 max:44.5566394*
- Polygon (544 coordinates): *(11.366030, 44.449526 11.363828, 44.450242 11.362467, 44.450953 11.362356, 44.451083 11.360538, 44.44932 . . .)*
- Country name and city name lookup in spatial data infrastructure (OSM): *Italy, Bologna*

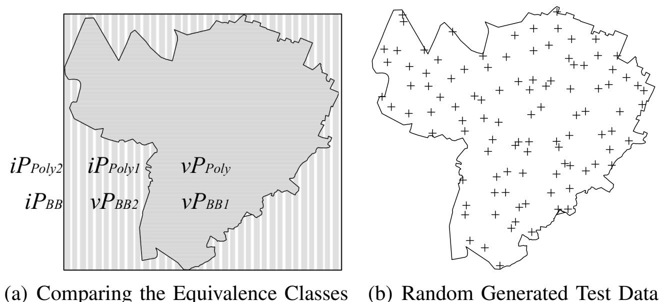
Figure 7 shows the created equivalence partitions of the bounding box (Figure 7(a)) and the polygon (Figure 7(b)). The precision improvement of the polygon is shown in Figure 8(a) since after comparison it shows that the bounding box describes a false positive valid equivalence class (see Equation 1):

$$vP_{BB2} \backsim iP_{Poly1} \quad (1)$$



(a) Equivalence Classes by BB (b) Equivalence Classes by Polygon

Figure 7. Equivalence Partitioning of the Bologna Area.



(a) Comparing the Equivalence Classes (b) Random Generated Test Data

Figure 8. Utilising OSM for ECP.

Figure 8(b) shows the test data generation creating 102 coordinates for randomly testing the service in the covered area. Using boundary value analysis on the buffered polygon it is also possible to test the service behaviour at the defined border with valid and invalid values.

C. Service Behaviour Description

By featuring clearly defined interface and parameter descriptions, documents such as WADL enable easy technical integration of RESTful services. The lack of standardised stateful service descriptions is the main challenge for the process of stateful testing. Therefore, a simple and easy to use description format has been developed which can be used to define typical use cases for the IoT service. In addition to the existing WADL document, this description format facilitates the definition of a sequence of resource and method executions including loops and concurrent calls. The XML-based sequence description document refers to method calls in the WADL document to gain compatibility. Figure 9 illustrates the main structure of a sequence description document.

```
1 <sequencespecification xmlns:ws1="application.wadl">
2  <vars><var name="cameraPan" type="double"/></vars>
3  <paramsets><paramset id="cameraPan">
4   <param name="id">10.11.127.6</param>
5   <param name="value"></param>
6  </paramset>...</paramsets>
7  <results><result name="testPosition" mediatype="application
       /xml">...</result>
8  </results>
9  <sequence mode="multiple" subSequenceType="All">
10  <subsequence mode="single" subSequenceType="
        MutualExclusive">
11   <subsequence mode="single">
12    <wsuri path="ws1:/Camera/pan/{id}/{value}" paramset="
        cameraPan"/>
13    <wsmethod name="ws1:setPan" returnCode="2xx"/>
14    <setvar var="cameraPan">{value}</setvar>
15  </subsequence>
16  <subsequence mode="single">...</subsequence>
17 </sequence>
18 </sequencespecification>
```

Figure 9. Service Sequence Description.

The sequence and subsequence elements are transformed into a state machine to enable the model based testing process. Each sequence and subsequence with a *wsmethod* and *wsuri* definition represents a single state and at least one transition to this state in the constructed state machine. The number of transitions depends on the number of values for the parameters and the combination of the same. The elements are structured into groups, which will be affecting the structure of the state machine directly (e.g., multiple paths, creation of sub state machines). Each sequence has a definition of a called WADL-resource (*wsuri*, Figure 9:16) and a method (*wsmethod*, Figure 9:18), which is provided by the IoT service. Furthermore, control commands are also a part of a sequence. In case of the control command *setvar* an actual value of a parameter from a method or a resource is saved into an internal variable. This procedure allows the implementation of stateful knowledge since the internal variable can be used over different transitions at any time and can also be part of a validation process. The content of a response message of an IoT service is mapped on a result definition to validate the content against previous inputs. With the sequence description, the values for each parameter (e.g., resource and method) can be predefined for this use case by the user. Missing values for each individual simple or complex (e.g., structures like XML) parameter in the sequence description are generated by the test data generation if a user provides only a portion of parameter values for the use case.

Copyright (c) IARIA, 2014.

V. Evaluation of Example Service

To demonstrate the algorithms and the process of the test framework this work employs a camera control example service. The service is used to control multiple CCTV Cameras at different locations that are adjustable in their pan and tilt via a RESTful interface. The following sections describe the transformation and test derivation aligned to this service. The sequence begins with an initialisation process of the camera (illustrated in Figure 10).

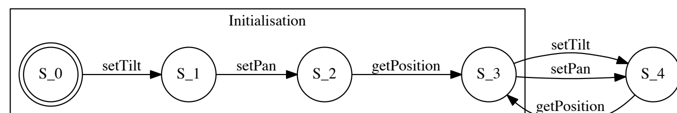


Figure 10. Camera Example Service.

Between S_3 and S_4, the values can be set with the `setTilt` and `setPan` method and evaluated with the `getPosition` method at the transition back to S_3.

VI. Test Case Derivation & Execution

The retrieval of the outlined service behaviour description of Section IV-C is the starting point for the test case derivation and execution, which is explained in this section. Whereby the main aim is to enable a fully automated test derivation process it also allows manual enhancements based on expert knowledge. The approach is divided into two translation steps: *i)* derive an EMF service model that represents an abstract behaviour of the SUT from testing perspective (e.g., detectable behaviour) and *ii)* derive executable test cases from this model based on TTCN-3. While the two steps are fully automated, the test developer can adapt the derived EMF service model with an Eclipse GMF editor and the created TTCN-3 test cases. For the model transformation, classical state machine concepts of states and transitions are re-used. In addition, the inclusion of concepts of TTCN-3 (e.g., Ports, Components, MessageTemplates) enables an easy model transformation. The basic model objects are shown in Figure 11 and can be described as follows:

States represent different logical conditions of the SUT and limit the number of correct functionality.

Events characterise the starting of an activity, which might result in actions or a state change. Events can be either from the type timer or input message.

Actions describe the reaction of the system to an event. An Action can be either a response message (output) or can result in a request sent to an IoT resource.

Transitions describe how the SUT reacts (action) to a certain event and a specific state. A Transition connects different states within the model.

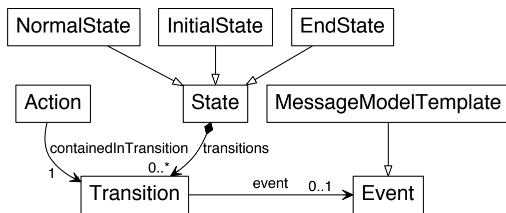


Figure 11. Simplified EMF Service Model.

The stages of the test derivation process and its information flow are shown in Figure 12. The process is initiated via a web interface during stage A. At this time, the required service behaviour descriptions are retrieved from the Knowledge Management. The service descriptions are analysed and transferred into the EMF model in stage B. At stage C this model is translated into executable test cases (TTCN-3). The final stage D executes these test cases and evaluates the results. The following paragraphs explain this process in more detail.

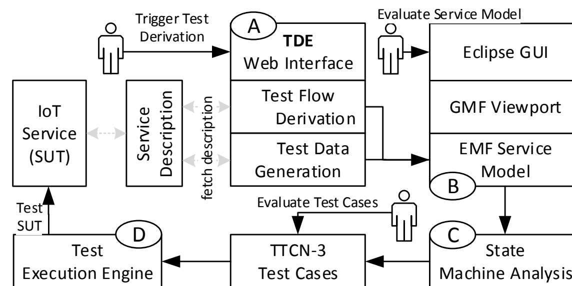


Figure 12. Information Flow.

A. Start of the Test Derivation Process

The test derivation process is triggered via a web interface, which accepts an ID identifying the service that was registered and has to be tested. The TDE fetches the needed service description documents from the KM and evaluates links of the semantic annotations to build the complete data model.

B. Building the EMF Service Model

The *Test Data Generation* analyses referenced data types and their interdependencies, which are described in the service descriptions. For the derivation of test data for each parameter ECP is used since it has been proven to provide high effectiveness in finding defects [23]. This technique divides the possible input data for each parameter in at least two disjunctive partitions (e.g., valid and invalid values). The partitions are created by parsing the parameter restrictions (see Section IV). The test data generation is designed to cover each partition with at least one test case. Due to the behaviour change of an IoT service between the boundary of two disjunctive partitions, the test data generation uses a boundary-value analysis to fully cover the boundaries of each partition. In this approach, valid parameter ranges are annotated with the use of XML Schema and with a semantic description. It generates code snippets for parameterised method invocations for the RESTful service and stores them in the EMF service model. Those snippets are based on generated random parameters for the used data types or enable code libraries based on lazy testing [24] to generate test data during runtime.

The *Test Sequence Analysis* is used to build the EMF service model based on the IoT Service WADL description and a sequence description. The model is implemented as a Extended Finite State Machine (EFSM), which has at least a unique InitialState and an EndState, as well as one NormalState definition. While the InitialState and the EndState(s) are generated automatically, the NormalStates will be created through the process shown in Table I. If a sequence has subsequence definitions multiple Normal states are created, followed by the creation of state transitions, which connect two

TABLE I. DOCUMENT TRANSFORMATION TO MODEL OBJECT NORMALSTATE.

Input:	Action:
<sequence mode="single"> ... </sequence>	Create NormalState for each sequence part.
<!-- WADL document --> <resources base="http://10.1.1.42:80/CameraService/iot/"> <resource path="/Camera"> <resource path="/pan/{id}/{value}"> <param name="id" style="template" type="xs:string"/> <param name="value" style="template" type="xmlschema:pan"/> <method id="setPan" name="POST"/> </resource> </resource> </resources> <!-- Sequence description --> <sequence mode="single"> <wsuri path="ws1:/Camera/pan/{id}/{value}" paramset="cameraPan"/> <wsmethod name="ws1:setPan" returnCode="2xx"/> </sequence>	Create Message-ModelTemplate (MMT) event for a request to a IoT service.
<paramset id="cameraPan"> <param name="id">10.11.127.6</param> <param name="value">12</param> </paramset>	Generate test data or use user defined values from sequence for each parameter.
<vars> <var name="cameraPan" type="double" schema="response:PositionResponse#pan"/> </vars> <sequence mode="single"> <wsuri path="ws1:/Camera/pan/{id}/{value}" paramset="cameraPan"/> ...<setvar var="cameraPan">{value}</setvar> </sequence>	Create variables which will represent the actual used value for a parameter. Add to MMT event.
<method id="getSensingData" name="GET"> <response> <representation mediaType="application/xml"/> </response> </method>	Create a MMT action for the response of a IoT service.
<wsmethod name="ws1:setPan" returnCode="2xx"/>	Create and add range of expected HTTP status code to MMT action.
<vars> <var name="cameraPan" type="double" schema="response:PositionResponse#pan"/> ...</vars> <results> <result name="testPosition" mediatype="application/xml" type="xml">{cameraTilt,cameraPan}</result> <results> <sequence mode="single"> <wsmethod name="ws1:getPosition" return="responseVar" result="testPosition"/> </sequence>	Add handling of possible return values to the MMT action (e.g., xml structure as response). Define variables to save the return values.

different states. These transitions represent an interaction with the IoT service or a timer Event. Therefore, the transaction needs a definition for both parts of the communication (e.g., request and response). The request is represented by a MMT event, which is the following step for the transformation. The resource and method information are part of the sequence definition and are extracted from the linked WADL document. If the sequence does not specify user defined parameter values, the test data generation will produce test values for each parameter.

A sequence definition is enabled to host control commands like the *setvar* tag listed in Table I/row 4. It is used to save the current value of a parameter for future operations and thereby allows the integration of stateful knowledge into the state machine representing a data model of the IoT service. The response template of the IoT service is modelled as a MMT action storing the method and response representation, which are defined in the WADL document that describes the service (I/row 5). Furthermore, the sequence description defines the expected HTTP status code for the response. The result attribute for a *wsmethod* tag in a sequence description (I/row 7) indicates another control command for the algorithm of the test case derivation and execution. This command produces a mechanism to compare the response of the IoT service against previous used parameter values, which can be used as a decision if the actual transition is valid or invalid. The MMT action as well as the event is be linked to a single transition in the resulting state machine model.

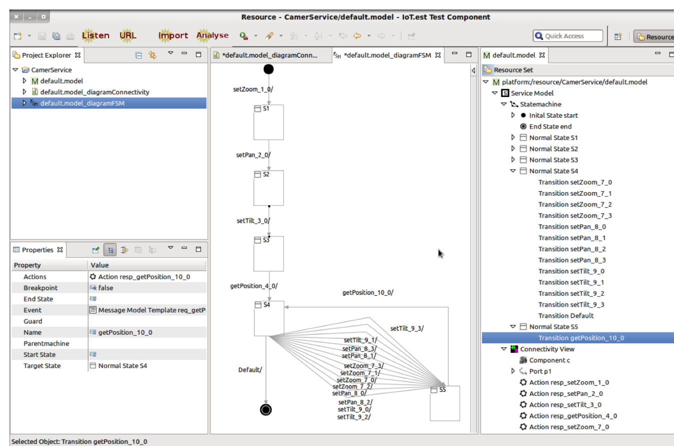


Figure 13. EMF Model Evaluation in Eclipse.

After creation the EMF model can be evaluated and altered in an Eclipse GMF - Model Editor (see Figure 13).

C. Creating Test Cases From the Service Model

The EMF model is used to analyse the resulting state machines of the previous stage. A configuration of the test case generation allows transition coverage or transition and state-coverage based on the W method [25]. The transition coverage is computed by identifying the *InitalState* and building a test tree based on a breadth-first visit of all transitions. Each transition in each state is inspected and if the transitions directs to a unvisited state a new branch path is created. Afterwards, the new branch end states are visited and their transitions are inspected. Each new inspected transition results in a new test path, which represents the test cases if only transition coverage is selected. For transition and state coverage it is further needed to identify a characterisation set (also called W set), which is a set of input sequences that can be utilised to distinguish every pair of existing states in the model. The resulting test cases are created by concatenating every sequence from the transition coverage set with every sequence in the characterisation set and apply them after the SUT is initiated.

The Model transformation from EMF to standardised TTCN-3 ensures explicit representation and reproducibility of test cases. As output of stage C test cases are derived from the EMF Service Model. A test case is a directed graph consisting of states and transitions and represents one possible path from the InitialState to another defined NormalState or

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-370-4

EndState (e.g., $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_3$, see Figure 10). During the model transformation each element is inspected and the required TTCN-3 elements are created. The actual writing of the TTCN-3 code is realised with a template engine. This enables the separation of syntactical details of the TTCN-3 language from the analysing logic thus reducing the complexity and enhancing the manageability. The followed approach uses the Java-based template engine Velocity [26]. In the following the transformation step is outlined with some detail. Table II depicts the first step while going through the model elements in the current test case. The model object *InitialState* is used to create the general test case structure and assures that the test case stops after a defined time by adding a timer. Afterwards, the TTCN-3 element *function* is created and added to the test case. TTCN-3 functions are utilised to separate different steps of the test execution. These reusable functions are used to represent the different states of the SUT.

TABLE II. TTCN-3 TRANSLATION OF MODEL OBJECT INITALSTATE.

Action:	TTCN-3 Output:
Add timer to enable timout	testcaseMaxExecutionTimer.start;
Create function	function start_1_0() runs on c { ...}
Add function to Test Case	testcase tc_1() runs on c system sys { start_1_0(); ... }

The next element Transition consists of an Event that can describe that an input is received by the SUT and an Action that describes the output reaction of the SUT to this input message. Table III sketches the transformation from the model object event to a send operation and the storage of the sent values for later usage. Since the EMF service model is created from the service point of view the translator inverts certain expressions for the purpose of testing. In this case the event of a transition becomes a send call.

TABLE III. TTCN-3 TRANSLATION OF MODEL OBJECT EVENT.

Action:	TTCN-3 Output:

Create send call and local variable	template HttpRequest req_setPan_1_0 := { postRequest := { url := "http://10.1.1.42:80/CameraService/iot/Camera/pan /10.11.127.6/19.27", ... } }
	v_PositionResponse_pan := 19.27;
	f_request(p1, req_setPan_1_0);
	v_req_setPan_1_0 := req_setPan_1_0;

Subsequently, the action part of the transition is utilised to derive TTCN-3 code. Initially a new function for the next state is created. Afterwards the defined response of the SUT is translated into TTNC-3. Then, the TTCN-3 element *alt* is used to form the possibilities of the SUT behaviour. At first, the failure case for delayed or unexpected service responses is modelled. After that the followed approach assumes deterministic service behaviour with only one possible valid reaction. This expected behaviour is included in the *alt* element of TTCN-3 including the jump to the next TTCN-3 function (state) created before. Table IV shows the discussed transformation process of the model object action.

While the link to the next function has been created during the action transformation, in the last step the function itself is created at the time the next element (*NormalState*) of the test case is inspected. Table V reveals the resulting TTCN-3 output.

At the final stage of the test case the model, object *EndState* is reached. This completes the TTCN-3 code creation by

TABLE IV. TTCN-3 TRANSLATION OF MODEL OBJECT ACTION.

Action:	TTCN-3 Output:
Create Target Call	S1_1_2();
Create expected response message	var template GETResponse resp_setPan_1_0 := { statusCode := (200 .. 299), content := {rawContent := omit, plainTextContent :=?}, headers := ? }
Form alt for Message	alt { [] testcaseMaxExecutionTimer.timeout { tcMaxExecutionTimeout_1();} [] any port.receive { unexcepctedStateReached_1(); } }
Create reply element in alt	alt { [ischosen(req_setPan_1_0.postRequest)] p1.getreply(POSTreq: {req_setPan_1_0.postRequest} value resp_setPan_1_0) −> value v_resp_setPan_1_0 { S1_1_2();} ...}

TABLE V. TRANSLATION OF MODEL OBJECT NORMALSTATE.

Action:	TTCN-3 Output:
Create function	function S1_1_2() runs on c {...}

setting the verdict to pass. If all functions, corresponding requests and response messages have been transmitted during the test case execution this final statement indicates that the SUT has the expected behaviour for this test case. Table VI shows the resulting TTCN-3 code.

TABLE VI. TTCN-3 TRANSLATION OF MODEL OBJECT ENDSTATE.

Action:	TTCN-3 Output:
Set verdict	setverdict(pass, "End−state reached");

D. Executing the Test Cases

After compilation of the TTCN-3 test cases the whole test flow can be executed by a web service interface or manually using the TTworkbench [27]. It enables a visualised logging of test execution in a log report, which can be used to evaluate the detailed test results (see Figure 14).

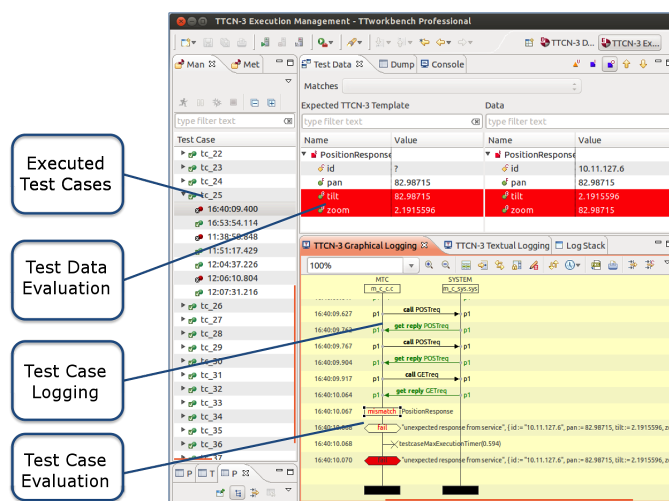


Figure 14. Execution of the Test Cases.

The TTworbench provides a comprehensible graphical view to easily identify the cause of an occurred error (e.g., protocol, encoding or data).

VII. Conclusion and Future Work

The complexity to describe IoT services for testing purposes in conjunction with missing domain specific knowledge for data types has prevented the utilisation of automated model-based testing for IoT services. The outlined framework tries to lower the gap by employing a sequence based modelling description which can be easily created, whereby the automated state machine analysis allows a transition and parameter combination coverage. Utilising semantically definitions in combination with ECP provides distinct test data pools enabling a more efficient and domain specific test case generation. The testing framework follows a two-step approach where the service description includes common utilisation information within a sequence description. The combination of standardised WADL interface description, semantic parameter descriptions and a sequence description empowers the transformation into a service model. Afterwards test cases can be derived and executed based on TTCN-3. This approach enables adjustments by developers at an early stage due to simple sequence descriptions and the standardised test notation TTCN-3. The key principles of the test framework are explained based on an example IoT service. The example is directly taken from our prototypical implementation and proves the applicability of our approach for IoT services. Although there is a high complexity in the initial implementation of the framework, the automated derivation allows the tester to take a systematic model driven approach to test IoT services though keeping possibilities to evaluate and modify the created test cases in a standardised test notation. The implemented sequence definition fills the gap between stateless interface descriptions and model-based testing and can be used for a more simplified and controllable test automation.

As a common approach IoT service compositions are utilising high level business modelling languages like Business Process Model and Notation (BPMN) [28]. Therefore, future work will include the integration of such languages and annotate them semantically to enable automated derivation of a service model. Besides functional behaviour, the influence of networking and service quality characteristics needs to be addressed for large scale IoT service testing.

Acknowledgment

The research leading to these results has received funding from the European Union FP7 for the IoT.est project under grant agreement n° 257521.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," Computer networks, vol. 54, no. 15, May 2010, pp. 2787–2805.

[2] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Software Testing, Verification and Reliability, vol. 22, no. 5, 2012, pp. 297–312.

[3] 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, Jun. 2009, pp. 593–607.

[4] D. Binkley, "Source code analysis: A road map," in 2007 Future of Software Engineering, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 104–119.

[5] A. Takanen, Fuzzing for software security testing and quality assurance, ser. Information security and privacy series. Artech House, 2008.

[6] W.-l. Huang and J. Peleska, "Exhaustive model-based equivalence class testing," in Testing Software and Systems, ser. Lecture Notes in Computer Science, H. Yenign, C. Yilmaz, and A. Ulrich, Eds. Springer Berlin Heidelberg, 2013, vol. 8254, pp. 49–64.

[7] E. G. Aydal and J. Woodcock, "Automation of model-based testing through model transformations," in Testing Conference - Practice and Research Techniques, 2009. TAIC PART '09. IEEE, Sep. 2009, pp. 63–71.

[8] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," in 2013 20th Working Conference on Reverse Engineering (WCRE), Oct. 2013, pp. 301–310.

[9] A. Pretschner, "Model-based testing," in 27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings, May 2005, pp. 722–723.

[10] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in Automated Software Engineering. 13th IEEE International Conference on, Oct. 1998, pp. 285–288.

[11] M. Deng, R. Chen, and Z. Du, "Automatic test data generation model by combining dataflow analysis with genetic algorithm," in Pervasive Computing (JCPC), 2009 Joint Conferences on, Dec. 2009, pp. 429–434.

[12] M. Fischer and R. Tonjes, "Generating test data for black-box testing using genetic algorithms," in 2012 IEEE 17th Conference on Emerging Technologies Factory Automation (ETFA), Sep. 2012, pp. 1–6.

[13] K. Belhajjame, S. Embury, and N. Paton, "Verification of semantic web service annotations using ontology-based partitioning," IEEE Transactions on Services Computing, vol. 99, no. PrePrints, 2013, p. 1.

[14] C. Kankanamge, Web Services Testing with SoapUI. Packt Publishing Ltd, 2012.

[15] C. Schanes, F. Fankhauser, S. Taber, and T. Grechenig, "Generic data format approach for generation of security test data," in VALID 2011, The Third International Conference on Advances in System Testing and Validation Lifecycle, Oct. 2011, pp. 103–108.

[16] 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 Future Network and Mobile Summit, Berlin, 2012, pp. 1–8.

[17] M. J. Hadley, "Web application description language (wadl)," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 2006.

[18] E. Reetz, D. Kuemper, K. Moessner, and R. Tönjes, "How to test iot-based services before deploying them into real world," in 19th European Wireless Conference (EW 2013), Guildford, United Kingdom, Apr. 2013, pp. 1–6.

[19] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[20] P. Biron and M. Ashok, "Xml schema part 2: Datatypes," W3C Recommendation, vol. 2, 2001.

[21] I. Niles and A. Pease, "Towards a standard upper ontology," in Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001, ser. FOIS '01. New York, NY, USA: ACM, Oct. 2001, pp. 2–9.

[22] A. Ballatore, M. Bertolotto, and D. C. Wilson, "Geographic knowledge extraction and semantic similarity in openstreetmap," Knowledge and information systems, vol. 37, no. 1, Oct. 2013, pp. 61–81.

[23] N. Juristo, S. Vegas, M. Solari, S. Abrahao, and I. Ramos, "Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects," in Software Testing, Verification and Validation (ICST), 2012 IEEE, Apr. 2012, pp. 330–339.

[24] M. Lin, Y. Chen, K. Yu, and G. Wu, "Lazy symbolic execution for test data generation," Software, IET, vol. 5, no. 2, Apr. 2011, pp. 132–141.

[25] A. Gargantini, "4 conformance testing," in Model-Based Testing of Reactive Systems. Springer, 2005, pp. 87–111.

[26] Apache Software Foundation, "The apache velocity project," Website, available online at http://velocity.apache.org/ retrieved: 2014-08-30.

[27] Testing Technologies, "TTworkbench," Website, available online at http://www.testingtech.com retrieved: 2014-08-30.

[28] S. Meyer, A. Ruppen, and C. Magerkurth, "Internet of things-aware process modeling: Integrating iot devices as business process resources," in Advanced Information Systems Engineering, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7908, pp. 84–98.