Small or medium-scale focused research project (STREP)
ICT Call 7
FP7-ICT-2011-7

Internet of Things Environment for Service Creation and Testing

IoT.est

Grant Agreement 288385

Specification of Goal-oriented IoT Service Composition and Testing

Document Ref.	D4.2
Document Type	Report
Workpackage	WP4
Lead Contractor	ATOS
Author(s)	Lara López, Ömer Ozdemir (ATOS), Daniel Kuemper (UASO), Bogdan Stanca-Kaposta (TT), Paulo Chainho (PTIN), Suparna De (US), Lucian-Mircea Sasu (SIE)
Contributing Contractors	ATOS, UASO, TT, PTIN, US, SIE
Planned Delivery Date	31.03.2014
Actual Delivery Date	21.07.2014
Dissemination Level	PU
Status	Final
Version	0.3
Reviewed by	All

I. Executive Summary

WP4 designs and constructs a Service Composite Environment (SCE), agnostic of Internet of Things domains, which facilitates the design of IoT services that easily operate in a variable environment. As part of the IoT.est project this SCE has been developed, taking advantage of already existing tools but going one step further, by considering IoT specific service features. This means that the IoT.est SCE is offering not only capabilities for designing business goal oriented workflows, but constructing composite services and driving tests and all in the same action. For this reason, the SCE incorporates semantic capabilities that have been defined and specified in WP3, as well as deployment and execution options that have been investigated and implemented in WP5.

The IoT.est SCE allows the user to deconstruct their business goals into a workflow, search for an atomic service that fulfills the required needs, test this atomic service before adding it to the service composition to check if it is doing what it is expected to do. The SCE then creates the composite service adding the different chosen atomic services; it deploys the composition then in the RTE (Run Time Environment) and tests the composition to check if it is working correctly, before executing it. All of this happens in the same screen without the need to use any additional tool. The interactions between the different components as defined in WP4 and with the client and/or interfaces to interact with the components developed in WP3 and WP5 are also described in this deliverable.

II. Contents

III. List of abbreviations

AS	Atomic Service
BPMN	Business Process Model and Notation
CS	Composite Service
ECP	Equivalence Class Partitioning
EFSM	Extended Finite State Machine
EMF	Eclipse Modelling Framework
FSM	Finite State Machine
GA	Genetic Algorithm
GMF	Graphical Modelling Framework
GUI	Graphical User Interface
IOPE	Input, Output, Precondition and Effect
IoT	Internet of Things
KIF	Knowledge Interchange Format
KM	Knowledge Management
MMT	Message Model Template
QoI	Quality of Information
QoS	Quality of Service
RESTful	Representational State Transfer based
RIF-PRD	Rule Interchange Format Production Rule Dialect
RMI	Remote Method Invocation
RT	Runtime
RTE	Runtime Environment
SAT	Semantic Annotation Tool
SCE	Service Composition Environment
SOA	Service-Oriented Architecture
SRT	Sandbox Runtime
SUMO	Suggested Upper Merged Ontology
SUT	System Under Test
TDE	Test Design Engine
TEE	Test Execution Engine
TTCN-3	Testing and Test Control Notation Version 3
WADL	Web Application Description Language

1. Introduction

1.1 Motivation and Scope

As it was stated in D4.1 Concepts of Goal-oriented IoT Service Composition and Testing v2 [IOTESTD41], “the IoT Service Creation Environment aims to bridge the gap between various business services and the heterogeneity of networked sensors, actuator and objects”. In that deliverable, the concepts to reach this goal were clearly defined, and in the present document it is described how these concepts were implemented.

1.2 Purpose of the Document

The purpose of this document is to present the results of WP4, with the latest updates that have appeared during the integration process. It also included the milestones to be achieved within this WP: M4.3 Methods to derive SUT model from semantic service description defined, M4.4 Developed a test composition out of reusable building blocks, M4.5 Methods to derive abstract test suite from SUT model defined and M4.6 Integration of testing into IoT Service Creation Environment specified.

1.3 Structure of the Document

This document is structured in 4 main sections, going from the progress beyond the state of the art on related works, to the specification of the constructed SCE and its graphical representation, to the specification of the testing capabilities integrated with the SCE.

These sections are the following ones:

- *Section 2 Progress Beyond State of the Art:* reviews related work to the concepts previously specified and defined the final approach taken within the project.
- *Section 3 Specification of knowledge-based SCE:* final design of WP4 architecture and definition of the main components and their interaction.
- *Section 4 Graphical User Interface:* Graphical representations of WP4 architecture, as all actions are performed with human interaction.
- *Section 5 Test Derivation Specification:* definition of the already implemented test derivation approach.

2. IoT.est: Progress Beyond State of the Art

This section is divided into 3 main subsections, Goal-driven Service Composition, Reusable Services for IoT Business Modelling and Automated Test Derivation.

In subsection 2.1 it is defined how the Goal-oriented IoT Service Composition has been approached as implemented, in order to emphasize the results of the work package filling the gap of the Goal-oriented composition definition.

In subsection 2.2 it is specified how the reusable services defined in WP3 are used by the Service Composition Environment and the benefits brought for business modelling.

Finally, in subsection 2.3 challenges and solutions for automating test derivation within the scope of the IoT.est project are specified.

2.1 Goal-driven Service Composition

As defined in D4.1 [IOTESTD41], one of the main novelties of this project is the goal-driven (oriented) service composition. In order to define a goal for composing a service, it needs to be first decomposed in different steps, like it happens in real life. I.e. if the goal is to turn on the fan when a room is at 25º there are some previous steps to follow, like measuring the temperature of the room. This is the same while designing a workflow; each step must be represented in order to determine the path to achieve a goal. This goal-oriented composition can be translated in a more comprehensive way of composing service as the generated workflows are clearly human readable, what brings the gap between the development perspective and the design of a business goal, providing an easy and semi-automated way of translating a high level idea into an executable service. For research on inclusion of business knowledge, proposed architectures and service composition, one may consult [Rosenberg2005], [Zigin2012], [Charfi2004] and [Pu2006]. Note that at the time of writing there is no large consensus/framework on composition automation.

In order to support this goal-oriented modelling, all the services are semantically annotated in order to determine if they are suitable or not with each of the designed steps. This action is performed adding some reasoning on top of a custom semantic search, as there is a need of data flow mapping in order to create a consistent composition, producing high-level goal-oriented semantic searches. The semantic search process also supports approximate matches. This is particularly relevant for IoT services which are location-specific. Approximate searches for location are made possible with support from the semantic modelling of indoor locations capturing relative positioning of modelled locations, which help to return results pertaining to the “nearest” service if one is not found in the specified location. The returned services are also ranked in order of match (exact matches followed by approximate ones, with different prioritising weights attached to the different search fields).

2.2 Reusable Services for IoT Business Modelling

Reusable services were provided within the context of the IoT.est project to be applied on different domains. The Service Composition Environment (SCE) designed and developed for this project allows the identification of these reusable services to be used for composing the goal-oriented composite services defined in the previous step.

There are two main reasons for including reusable services for modelling business goal, one is that they are agnostic of the domain and the other one is to provide a set of services that can be used in different compositions avoiding the duplication of services with the same functionalities. For this reason, there can be different instances of the same services. This is also useful for adaptation purposes, allowing the redirection to a different instance of a service that is causing malfunctioning issues.

2.3 Automated Test Derivation

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 [Binkley2007]. 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 [Takanen2008]. This approach can find security relevant implementation errors (e.g. buffer overflow) but on the other hand 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 [Huang2013]. By employing more precise semantic service descriptions, our approach tries to overcome solely random generations and takes 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 [Aydal2009]. The authors in [Walkinshaw2013] 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 [Pretschner2005]. 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 test data generation. The challenge is to find boundaries of the valid input space. [Tracey1998] exploits search techniques to automate the generation of test data. Evolutionary algorithms, namely Genetic Algorithm (GA) to derive test data from an initial data pool are used in [Deng2009] 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. [Fischer2012] proposes 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 limitation hinders GA usage due to the high amounts of test cases, which are used to optimise the test data. To overcome this limitation the optimisation of test data and test cases is realised by service descriptions before the service is tested.

3. Specification of knowledge-based SCE

The SCE is the tool for composing IoT services in a semi-automated way, as the selection of each atomic service to be added to a composition is performed using different criteria getting as a result a list of atomic services that best match with the user needs. This action is not fully automated as it gives to the user the possibility of deciding whether to add or not an atomic service based on test results.

Once a service is composed, it will be also tested after deploying it so the user will also know if his composition is properly working before executing it.

As it was indicated in D4.1 [IOTESTD41], the SCE is domain independent, this means that it can be used for composing services that apply to any IoT specific domain without having any restriction.

3.1 Architecture Overview

The following diagram depicts the main components interacting within the SCE and the workflow followed to perform this interactions:

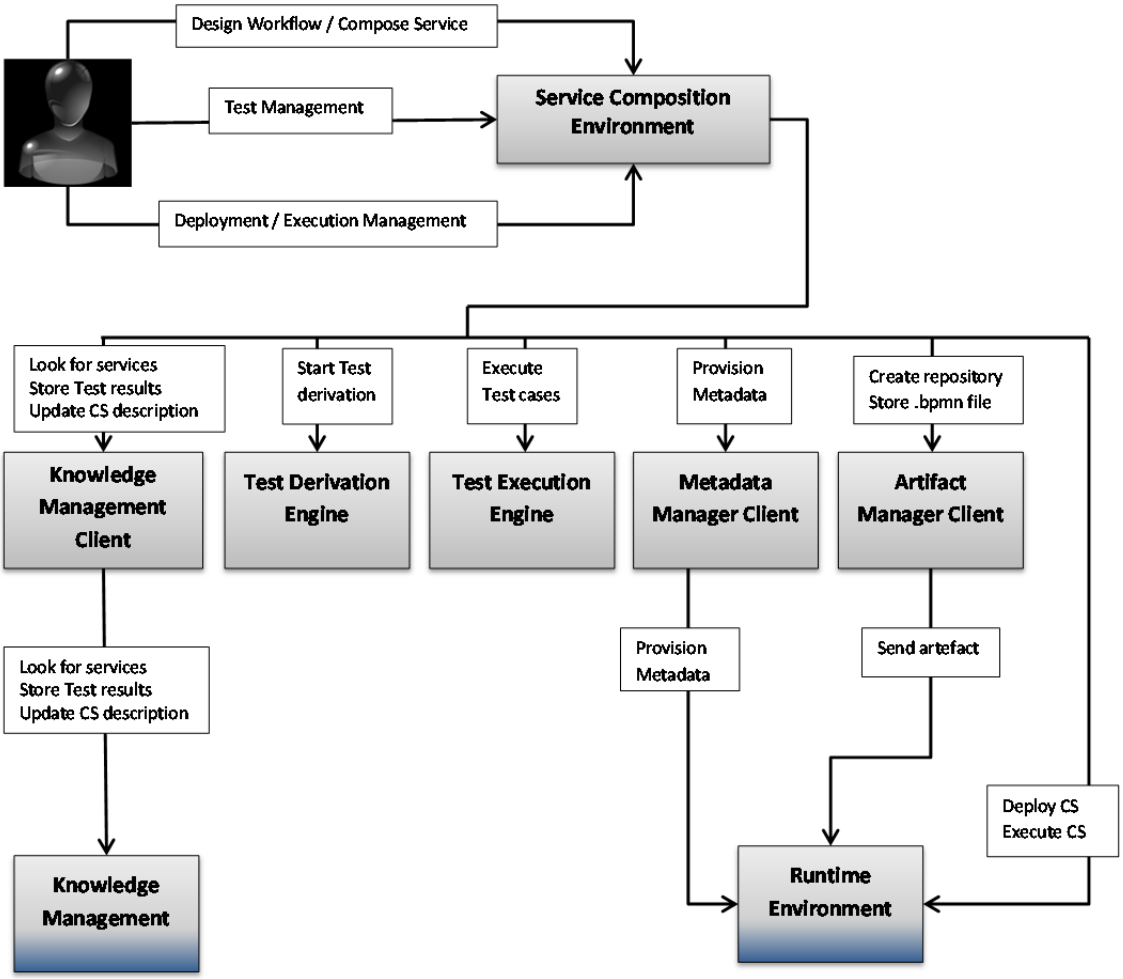

Figure 1: Updated Architecture – Interaction between the components of the SCE

As shown, the final architecture of the SCE and the interaction between its components has been updated from the previous version. In this new diagram, there is a component called Service Composition Environment, as it is the core of the SCE as it allows workflow designs and service compositions, while the other components remain the same.

Blue boxes represent components designed and developed in other work packages, Knowledge Management in WP3 and Runtime Environment in WP5.

3.2 Components

Due to the update on the architecture diagram and the interactions between WP4 components, in this section the components involved in the Service Composition Environment are described in terms of the functionalities that they provide, the limitations, dependences and technologies used.

3.2.1 Service Composition Environment

The Service Composition Environment component has the same name as the whole SCE developed within the IoT.est project, as it contains the key features for designing goal-oriented business workflows and create service compositions. As shown in Figure 1, the user starts any action within the IoT.est framework. These actions are:

- Design of a business goal (workflow).
- Create a service composition. This is a generic action that involves different actions at the same time:
    - Search for services given specific criteria.
    - Trigger atomic services testing and collect results.
    - Bind services into the composition.
    - Update the annotations of a composite service.
    - Provisioning Metadata and store the resultant .bpm file.
    - Trigger test derivation for composite services and collect results.
    - Update the Knowledge Management with tests results.
    - Deploy composite services in the Runtime Environment.
    - Execute composite services in the Runtime Environment.

The user can create workflows, search and discover services, bind services to service tasks, test, deploy and execute them by using this platform. As it was defined in D4.1 [IOTESTD41], different components composed the SCE, although there is no interface provided by this component as it attacks the already given ones by other components, providing the requested information in each case and analysing results.

Inside the GUI, the Editor is the main component for developing business workflows, as it provides the graphical user interfaces to drag&drop BPMN 2.0 predefined components.

This component is based on an Open Source BPMN engine, called Activiti, which is licensed under Apache v2.0. The main reason to choose this engine is that fully complies with the standard, this means that any workflow designed in any other engine using BPMN2.0 can be load and visualise on the platform, as well as any .bpm file generated can be used in different engines.

3.2.2 Knowledge Management

The Knowledge Management component provides a number of functionalities to the SCE through defined interfaces to achieve goal-driven service composition. Crucially, the Knowledge Management component assists the designer during the service composition by providing reasoning support for

matching services to the different workflow blocks. Semantic matchmaking and ranking is employed to match services to each workflow item.

The Knowledge Management functionalities relevant to the SCE include the creation of the composite services, the discovery of the atomic services that could be candidates for composition and the semantic annotation of composite services.

3.2.2.1 Create Service

The creation of the services is done through the Service Annotation Tool (SAT). This tool has been provided for the purpose of simplifying the process of creating semantic service descriptions. Since available ontology editors will require users to familiarise themselves with the editing tools, which in turn requires a learning curve, the composite service SAT presents a simple web form (see Figure 2) that the user can fill in with the relevant information.

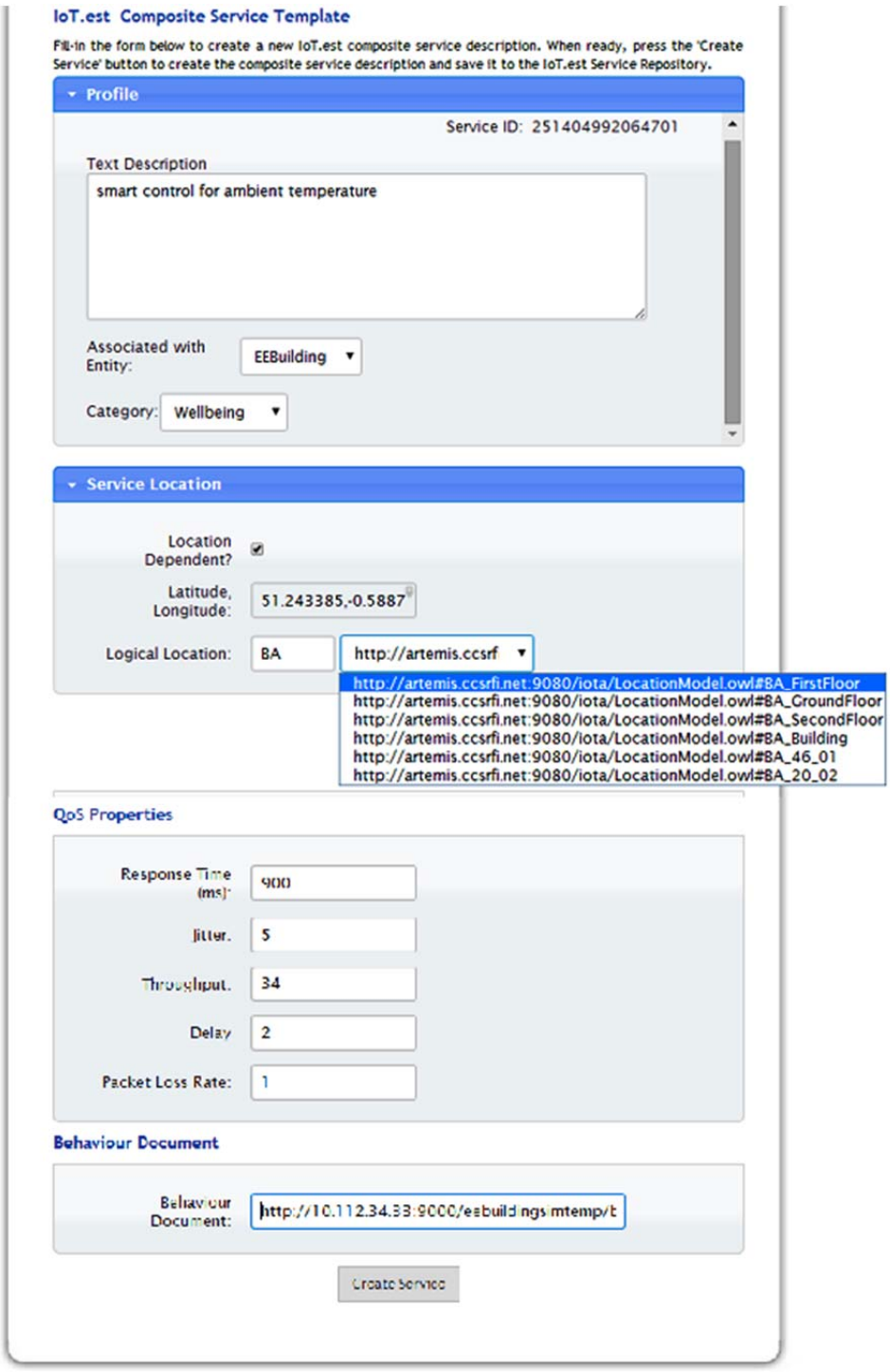

Figure 2: SAT tool for automated semantic annotation of composite services

The fields in the web form are:

- Service ID. It is automatically generated by the KM and copied across to the SCE during workflow design. The same ID is used to update the composite service's description to the KM when the composition process is completed.
- Text description. It provides a textual description of the composite service
- Associated with entity. This is a dropdown menu with the entities already stored in the knowledge base
- Category. It provides the category of the composite service in a dropdown menu, which currently has the following items:
    - Wellbeing
    - Business
    - Sensors
    - Commerce
- Location. This feature is optional and is enabled by selecting the checkbox. The relevant location fields are:
    - Geolocation. The user could provide the latitude and longitude of the composite service, or the user can move the landmark icon on the Google maps embedded window to automatically fill in the geolocation
    - Logical location. This is a dropdown menu which is populated with different logical (indoor) locations drawn from the locations modelled in the indoor location domain model when a keyword (e.g. room/building name) is supplied in the first text box. Currently, one of the partner (University of Surrey) premises has been modelled, including different buildings, floors/corridors as well as types of rooms on each floor. (i.e. Building BA at University of Surrey). The indoor location ontology also models semantic relations between the locations, depicting relative positioning, e.g. room1 *isAdjacentTo* room2, floor1 *contains* room1, building1 *contains* floor1 etc.
- Required Quality of services. The Quality of service is composed of the 5 traditional QoS features for network communications. They could be filling in with integers, float, double and ranges of the same. The features are:
    - Response time (ms)
    - Jitter
    - Delay
    - Throughput
    - Packet loss

Once the form is submitted, semantic service description for that service is automatically created and the Knowledge Management component will store this service description in a triple store (RDF database). Not all the fields in the form need to be provided and an incomplete service description can be stored in the triple store. Once a service is created, the user needs to copy the ID provided by the Create Service interface in order to add it to the SCE when creating a business workflow. This is the only way to keep the relation between the information stored in the Knowledge Management and the composite service developed in the SCE.

3.2.2.2 Discovery Service

When the SCE needs to create the composition, first it calls the discovery service, in order to search for (discover) atomic services that can match the different tasks of the designed workflow. For that purpose the RESTful interface to query the ontology is composed of the following parameters that specify the search criteria:

REST URL: http://10.112.34.35:8080/KnowledgeManagement/getServiceByQueryParams

POST body attributes:

Search attribute	Description
serviceID	The ID of the service
serviceName	The name of the service
serviceWADLURL	The URL of the WADL descriptor of the service
serviceCategory	One of the predefined categories: Wellbeing/Business/Sensors/Commerce
serviceLatitude	Geo-coordinates (latitude) of the service coverage area
serviceLongitude	Geo-coordinates (longitude) of the service coverage area
serviceLogicalLocation	Indoor location specification for the service coverage
serviceRelatedEntity	Real-world entity related to the service, which could be an instance of a building or person; selected from concepts already stored in the knowledge base
resourceName	Name of IoT resource whose functionalities are exposed by the service
resourceType	Type of the IoT resource
resourceLongitude	Geo-coordinates (longitude) of the IoT resource location
resourceLatitude	Geo-coordinates (latitude) of the IoT resource location
resourceLogicalLocation	Indoor location specification for the IoT resource
featureName	The relevant feature of the IoT resource, e.g. 'state'
featureType	Type of the IoT resource feature
operationName	Name of the operation possible on the IoT resource feature, e.g. getValue
operationType	Type of the operation
endpointURL	Access URL for the operation
inputParameterType	Type of the service input parameter
outputParameterType	Type of the service output parameter
maximumResults	Integer value to limit the number of returned service descriptions

RESPONSE:

The response body contains a JSON string that mirrors the search criteria, with the values of the various parameters specified.

None of the search criteria are mandatory, so if the SCE queries the Discovery Service with an empty JSON it will retrieve another one contain a list with all available services. The JSON string also includes a set of Boolean flags for each corresponding request attribute, specifying if the attribute needs to be matched exactly. By default, the flags are set to false. Since each attribute functions as a filtering criterion, missing request attributes allow all services to be matched. The flags can also be set to an enumerated list of approximations to be allowed in the search. For example, for the logical location criterion, an approximation of 2 levels means that the search will look at the next two levels of location containment, i.e. if a room is specified as the attribute and no service is found with the specified room as the coverage area, the search process will continue to look at the next higher level of containment, the floor and then search for services in the same building. This is enabled by the relative positioning annotations as well as the transitive closure of the 'contains' property in the indoor location model.

If there is no service matching the query, the Discovery Service will return an empty JSON. This file is read by the SCE that will interpret that there are not available services, showing the corresponding message on the screen.

3.2.2.3 Annotate Service

This service updates the KM with the composite service fields once the composition process is complete. This is done through a RESTful interface with the following parameters:

REST URL: http://10.112.34.35:8080/KnowledgeManagement/UpdateCompositeService

POST body format:

```
{
  "compositeServiceID": "ID",
  "composingServiceIDs": ["atomicserviceID1",
  " atomicserviceID2"],
  "usedAtomicOperationEndpoints": ["atomicservice1endpoint",
  " atomicservice2endpoint"],
  "compositeServiceTasks": [{
    "taskID": "StartTask",
    "operationName": "operationoftheTask",
    "operationURL": "http://UseThisURLToStartTheTask",
    "inputParameters": ["taskInputParam1",
    " taskInputParam2"]
  }]
}
```

3.2.3 Test Component

The Test Component has two main components, the Test Derivation Engine (TDE) in charge of creating test campaigns that contain all needed test cases for both atomic and composite services, and the Test Execution Engine (TEE), which executes the test cases given for a specific service and returns the results of the functional conformance tests to the SCE so it can update the Knowledge Management with the overall results.

3.2.3.1 Test Derivation Engine

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 3). The derivation is driven by processing service descriptions and utilising domain knowledge. IoT.est uses a KM to store descriptions of IoT services. These descriptions store knowledge of the services which is used to find matching services for compositions and derive functional test cases. The 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 [Reetz2013] to enable IoT service testing without communicating with resource constrained IoT sensors or altering IoT actuators during test execution. 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 Modelling Framework (EMF)-model. It is editable with the Graphical Modelling 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.

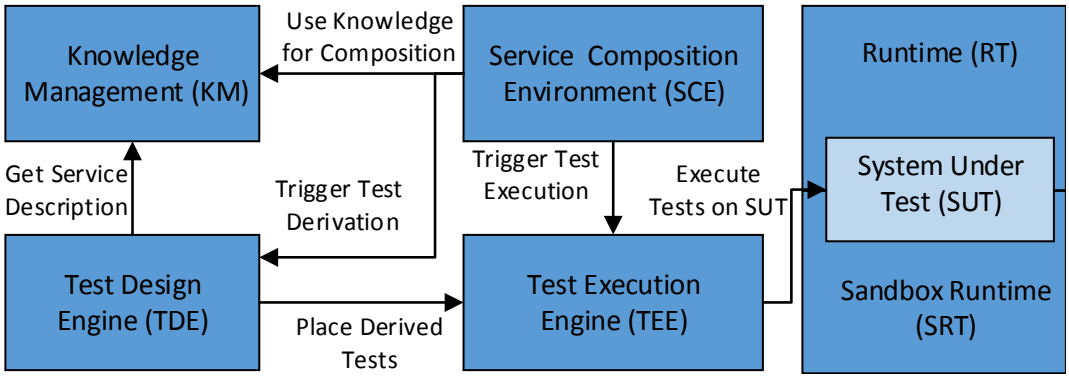

Figure 3a: Test-based Component Interaction

The test derivation engine is triggered via a web service interface. It creates an EMF service model that can be evaluated in an Eclipse GUI (GMF Editor). The test cases are generated in the standardised TTCN-3 [IOTESTD41]. A Brief Overview of the Test derivation can be found in Figure3.

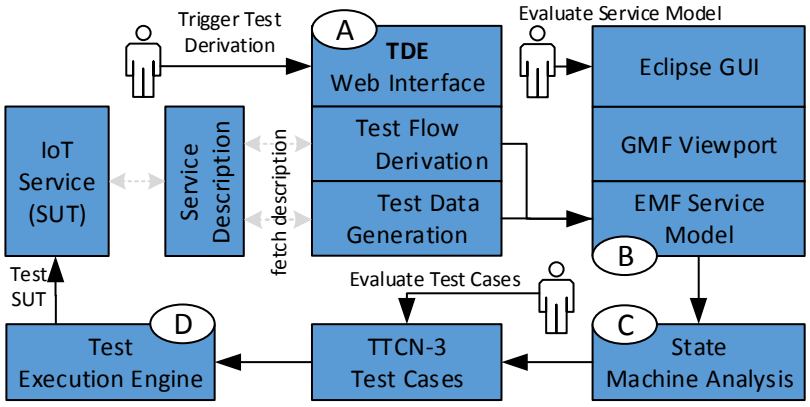

Figure 4b: Test Information Flow

3.2.3.1.1 API of the Test Derivation Engine

The Test Derivation Engine has an external HTTP-interface, which is utilised to derive automatically derive the test cases and provide the test campaign. The following paragraphs describe (simplified) the methods that are called from the SCE to control the process and show possible answers from the service:

Method DeriveTests (PUT):

Triggers the test derivation of a specified serviceID and creates an individual TestCampaignID that is used to access the test cases.

Example:

http://10.112.34.41:8081/TDE/deriveTests
?serviceID=<ID of the service to test>
&testCreationModifier=abc
&actionHandler=<http-url to send an event when test derivation is ready>

Response:

	Status	**Body**
Started Test Derivation	200	<testCampaignId as a long value>
Could not access serviceID or WADL	404	error message

Method DerivationStatus(GET):

Check the status of the test derivation process.

Example:

http://10.112.34.41:8081/TDE/derivationStatus
?testCampaignID=<TestCampaignID>

Response:

	Status	Body
List Job Queues (for validation)	200	<testCampaignId as a long value>

Method TestCases(GET):

Get the location of the CLF file which links the test cases as a test campaign.

Example:

http://10.112.34.41:8081/TDE/testCases
?testCampaignID=<TestCampaignID>
&info=<remotefie|localfile|projectname>

- &info=remotefile gives you the http url of the clf (like without parameter)
- &info=localfile gives you the local filename of the clf
- &info=projectname gives you the project name

Response:

	Status	**Body**
TestCampaign is ready, link to the clf	200	http link to the .clf or defined information by the info Parameter
Test Campaign not ready yet	423	error message
Not a valid test campaign	404	error message
Unknown Exception	403	error message

Method TestCampaign(GET):

Download the test campaign(s) as a file.

Example:

http://10.112.34.41:8081/TDE/TestCampaigns/1700561251.clf

Response:

	Status	**Body**
CLF file Download	200	application/octet-stream; charset=UTF-8
Test Campaign not ready yet	423	error message
Not a valid test campaign	404	error message
Unknown Exception	403	error message

Method ttWorkbench(GET)

GET):

Get the link of the Test Execution Engine (TTWorkbench), which can access the test cases

Example:

http://10.112.34.41:8081/TDE/ttWorkbenchUrl?testCampaignID=<TestCampaignID>

Response:

	Status	**Body**
TestCampaign is ready, you can get the tt-WB	200	<IP:PORT>
Test Campaign not ready yet	423	error message
Not a valid test campaign	404	error message
Unknown Exception	403	error message

3.2.3.2 Test Execution Engine

The Service Composition Environment (SCE) interacts with the Test Execution Engine (TEE) just after finishing the test derivation step. This component is responsible of executing the test cases that are defined in the Test Campaign (.clf file) provided by the Test Derivation Engine (TDE).

The connection between SCE and TEE is established via an RPC server, using TCP port 10280 for the client (SCE) and 10279 for the server (TEE).

In order to connect to the TEE, it is necessary to have the IP address and the port where it is running.

Regarding the operations that can be performed, there are several callback methods that may be implemented by the client to retrieve the status of the different actions. These actions are:

testCaseStarted

callback method used for the TEE to notify the client when a test case has just started.

testCaseFinished

calleback method used for the TEE to notify the client when a test case has finished.

serverShutDown

RPC server connection is shut down.

After the SCE connects to the TEE, it calls the ***loadTestSuiteFromFile()*** method in order to load the given test campaign. The SCE can receive a list including test cases contained in this test campaign (specified by the .clf file) by calling ***listAvailableTestCases()***. The test cases are selectively submitted for execution using ***executeTestCase()***, which returns a handle of kind Job. The test case execution status (or job status) is propagated from the server to the clients. ***IExecutionHandler*** interface methods, ***testCaseStarted*** and ***testCaseFinished***. Independently of these call backs, the

complete list of submitted jobs (both finished and running) is obtained from the execution server using ***listExecutionJobs()***.

There are 5 different states for a test case execution job: *NONE, WAITING, RUNNING, FINISHED* and *CANCELLED.*

After all the tests from the given test campaign were executed, the SCE stores the execution process log for later analysis calling ***saveLog()***.

A detailed view of the test execution is given in section “Sandbox Testing” of [IOTESTD52].

3.2.4 Runtime Platform

The Service Composition Environment interacts with the Runtime Environment through the Metadata Manager client, to save the service provisioning metadata, and with the Artefact Manager client, to store all service files needed, notably the BPMN file containing the composition workflow logic.

3.2.4.1 Metadata Manager Client

The Metadata Manager client interacts with the RTE Metadata Manager to provide data needed to provision the service in the most suitable runtime platform available. The Service Provisioning Metadata provided by the Metadata Manager client includes (Figure 4) information about the service implementation, the Interfaces that can be used to access it and the service components comprising the structure of a composite service. A detailed description of the Service Provisioning Metadata Model is provided in D5.1 [IOTESTD51] and D5.2 [IOTESTD52].

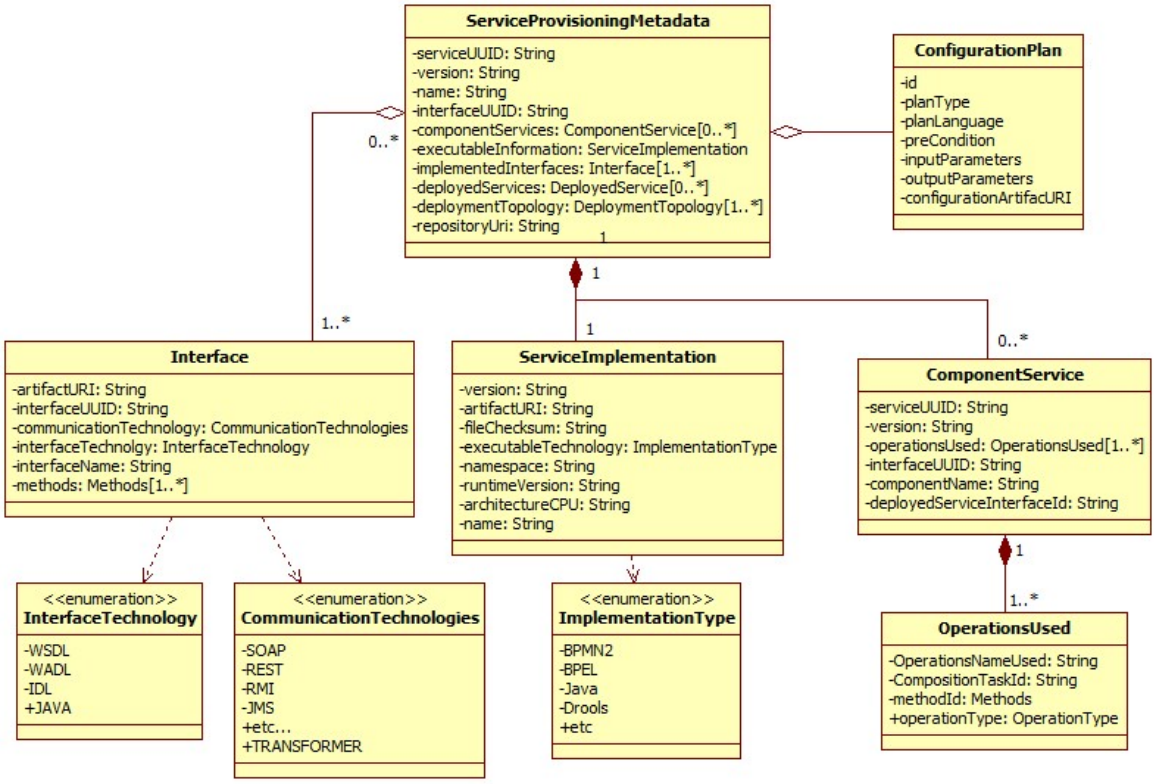

Figure 5: Service Provisioning Metadata instantiated by the SCE Metadata Manager client

The Metadata Manager RESTful resource used is:

/metadatamanagerservice/{serviceUuid}

where the full ServiceProvisioningMetadata object is passed in the payload of a POST operation.

3.2.4.2 Artefact Manager Client

The Artefact Manager client interacts with the Artefact Manager to create the Service Artefact Repository and store the BPMN file in the created repository. The following RESTful operations are used:

/artefactmanager/repository/{serviceuuid}

POST Operation

Creates a new Service Artefact Repository for the deployment of a new composite service. It returns the RepositoryUri that is used later to save artefacts.

/artefactmanager/repository/{serviceuuid}/{artefactid}

POST Operation

Stores a new artefact in the repository. The artefact may be passed in the body of the operation or it may be downloaded from the URI location when it is provided as an input parameter.

3.3 Interaction between the different components

In this subsection the interaction between the different components of the Service Composition Environment, among the interfaces described in the previous section is specified.

These relations have been listed according to the updated architecture shown in Figure 1.

3.3.1 Interaction between KM and SCE

The interaction between the KM and the SCE is divided into 3 steps, as there are different actions that can be performed involving these 2 components. More details are given in the following subsections.

3.3.1.1 Service Discovery (Search)

In order to create a composition, it is needed to first look for services that can provide the functionality required to fulfil the needs of each task designed in the workflow. This interaction is shown in Figure 5.

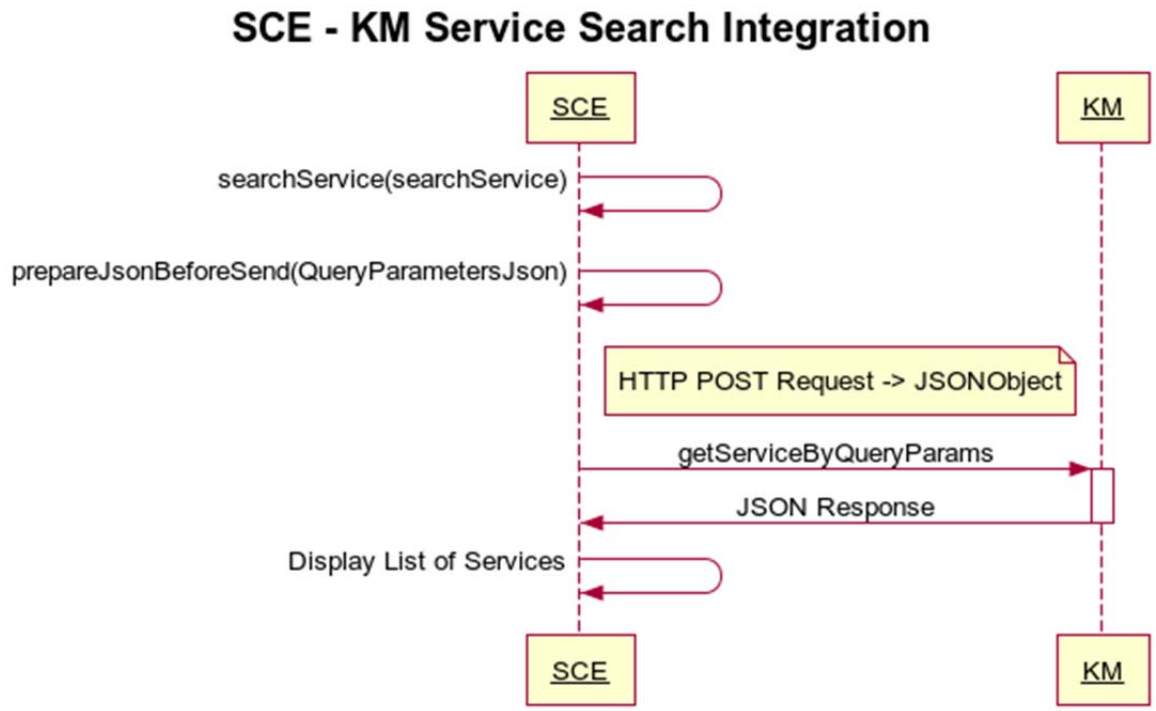

Figure 6: Interaction between SCE and KM for searching services

Step 1: before querying the Knowledge Management client, the SCE collects the criteria given by the user in the GUI's search panel in order to send it to the Knowledge Management component.

Step 2: the criteria is added to a JSON file, according to the format specified by the Knowledge Management component.

Step 3: the SCE sends the JSON file to the Knowledge Management client via POST request and retrieves another JSON as a response. If there were no criteria specified, the JSON contains a list with all the available services annotated in the Knowledge Management. If there are specific criteria, the list will contain only the matching services if there is any, if there isn't any service that matches the list will be empty.

Step 4: the SCE reads the list of services provided by the Knowledge Management client and display them on the screen. If the list is empty, it shows a message on the screen alerting the user that there is no service that fulfils the request.

3.3.1.2 Update CS annotations

Once the atomic services are bound to the service tasks (graphical representations of atomic services), and the composition is finished. The SCE updates the annotation of this composite service stored in the Knowledge Management with the deployment information.

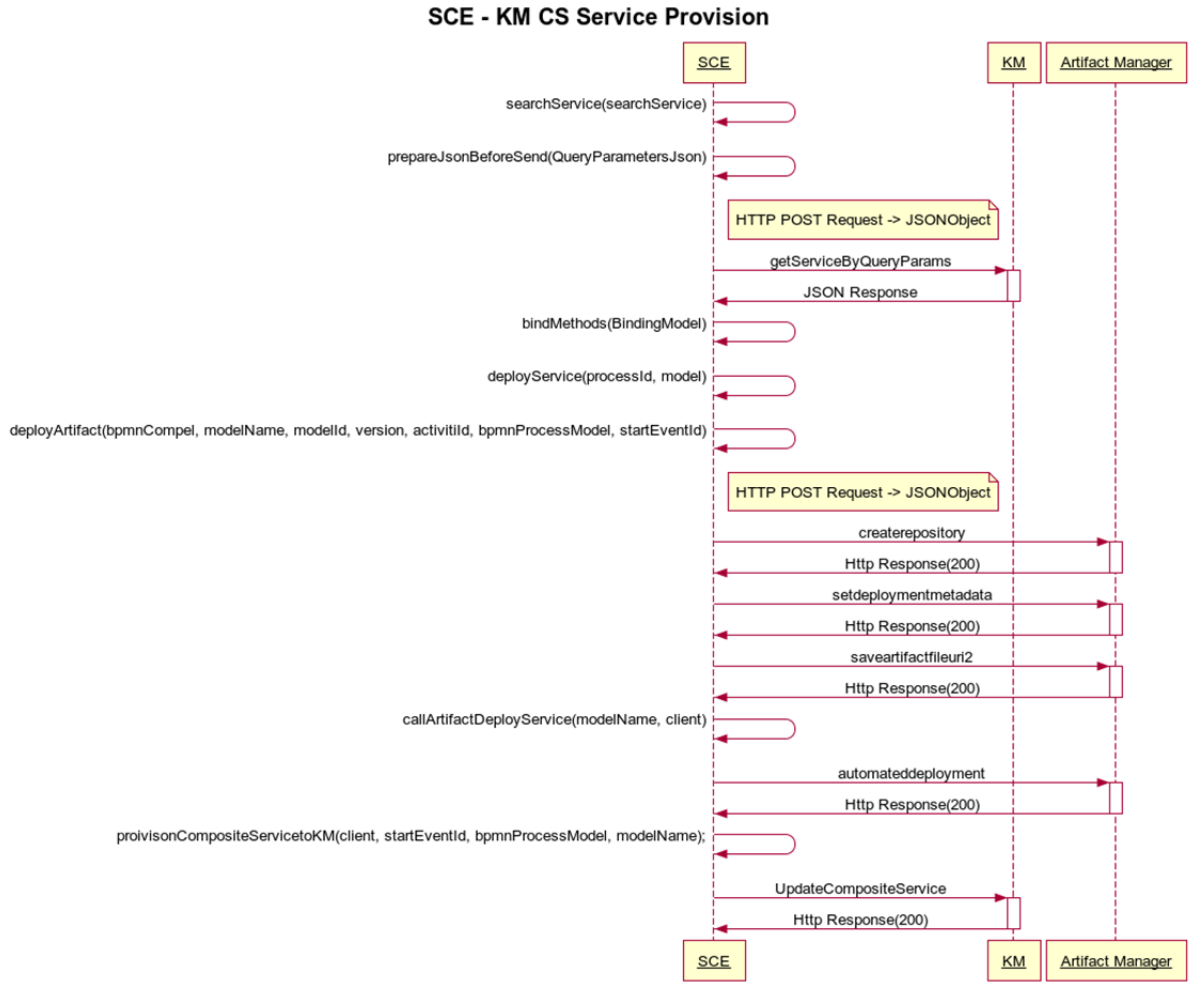

Figure 7: Interaction between SCE and KM for updating composite services annotations

Although there are many steps on this interaction, most of them refers to the storage and collection of information due to the interaction with the Artefact Manager client in order to update the Knowledge Management as there is no direct relation between Artefact Manager and Knowledge Management.

Steps 1 to 3: they are the same as for looking for atomic services as they were described in section 3.3.1.1.

Step 4: once the user has selected an atomic service from the list, he binds it to a specific service task. This means that a specific method of an atomic service is associated to an action designed in the workflow.

Step 5: once all tasks have a method associated, the SCE collects the information stored in memory and generates a .bpm file with the given specifications.

Step 6: the SCE converts the .bpm file in another version fully compatible with RTE prerequisites. This file cannot be executed outside the RTE, what can seems as a loss of functionalities due to the incompatibilities with other engines, but in fact it gains more relevance, as it is providing more additional information than expected in order to control the execution of a composite service and its performance.

Step 7: the SCE invokes the Artefact Manager client in order to create a repository where to store the already created .bpm file.

Step 8: after the creation of the repository, the SCE sends all the needed information to the Metadata Manager client, so it is accessible when the user decides to deploy the composite service.

Step 9: once the metadata is provisioned, the SCE stores the .bpm file in the created repository at the Artefact Manager.

Step 10: the SCE collects the information needed to deploy a composite service at the RTE.

Step 11: the SCE deploys the composite service at the RTE.

Step 12: once the SCE receives the response from the RTE confirming that the composite service has been deployed without any problem, it collects all the information needed to update the Knowledge Management with the deployment information.

Step 13: the SCE updates the annotations of the composite service in the Knowledge Management.

3.3.1.3 Update test results

The last interaction between the Knowledge Management and the SCE is regarding the update of the semantic annotations of a composite service with tests results. Regarding these results, as it can be a variable number depending on how many test cases are derived for the composition and that not all of them can be executed, as it depends on the user preferences because none of them is mandatory, what it is stored in the Knowledge Management is the average result. This means that if a test case fails, no matter how many test cases are executed, the status will be FAILED, otherwise it will be PASSED.

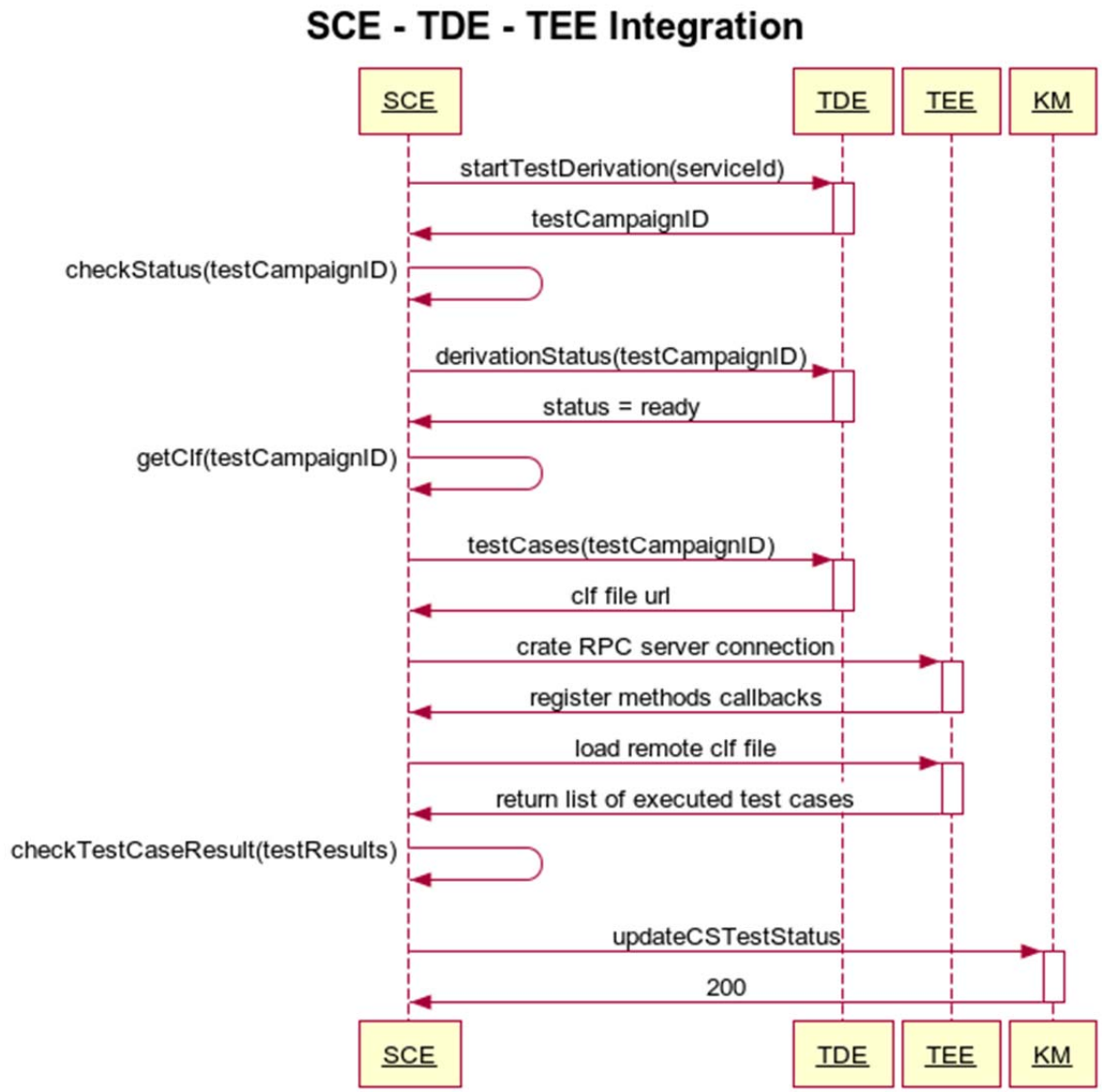

Figure 8: Interaction between SCE and KM for updating test results

Step 1: when a composite service is deployed, this is done in both environments: RTE and Sandbox. Once the service is deployed in the sandbox, the SCE sends the composite service ID to the TDE to start test derivation. The TDE gives back an ID for the test campaign (list of test cases that can be executed).

Step 2: the SCE check the status before sending back the test campaign ID to the TDE.

Step 3: once all the test cases are ready, the TDE sends the generated .clf file to the SCE.

Step 4: the SCE asks for the physical location of the generated file, as it is needed in order to execute it.

Step 5: once the SCE has all the necessary information to start the test execution, it creates a connection to the TEE. It is mandatory to register on the TEE before executing the test cases, and this has to be done for every test campaign execution.

Step 6: if there hadn't been any problem establishing the connection, the SCE sends the URL with the .clf file location to the TEE in order to start executing tests.

Step 7: the SCE collects the results that the TEE is sending once each test case is executed. If one test fails, the SCE shows it on the screen so the user is aware of it.

Step 8: once all tests are executed, the SCE transforms tests results in one single result (PASSED or FAILED) to be sent to the Knowledge Management component.

Step 9: the SCE sends the global result to the Knowledge Management in order to update the annotation of the composite service.

3.3.2 Interaction between SCE and RTE

As it was shown in the Architecture diagram (Figure 1), once the service composition is finished, the SCE interacts with the RTE by using three services:

- The Metadata Manager Service is used to provision metadata through the Metadata Manager client, this means sending all the needed information for the deployment and execution of the composite service.
- The Artefact Manager Service is used to store the generated artefact, in this case the .bpm file, after creating a specific repository for it.
- The Deployment Manager Service is used to deploy the service in the sandbox, from where tests are executed (see section 3.3.3) and to deploy and execute the service in the Runtime Production Platform as soon as the service is fully tested.

Main interactions between the SCE and the RTE are depicted in Figure 8, and each main message flow is described below.

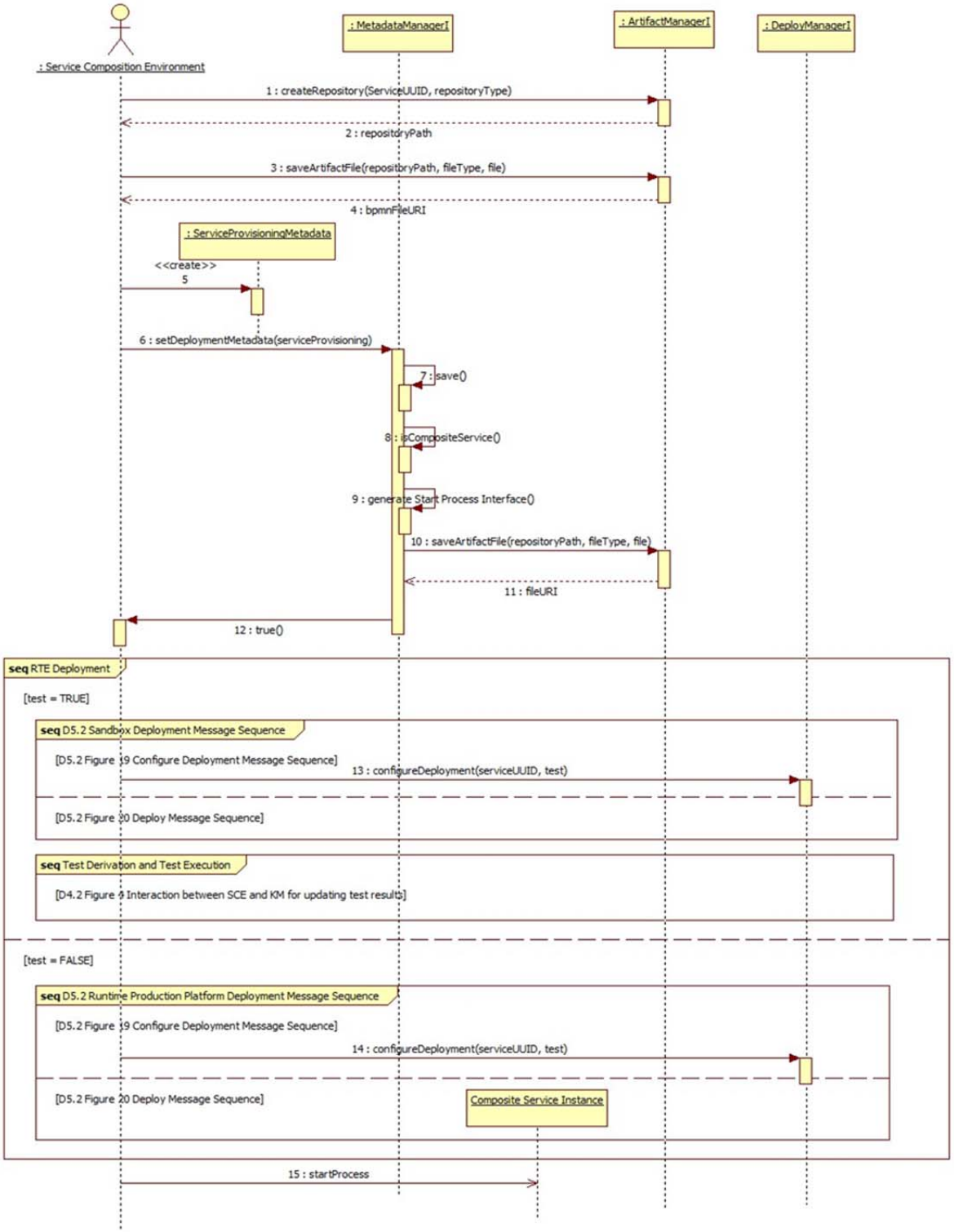

Figure 9: Interaction between SCE and RTE for provisioning, deployment and execution

Steps 1 and 2: once the composite service is finished, the SCE asks the Artefact Manager client to create a new repository where to store the artefacts for its deployment.

Steps 3 and 4: once the Artefact Manager client gives back the path where to store the .bpm file (aka artefact), the SCE stores it there and the Metadata Manager client returns the URI of the file.

Steps 5 and 6: the SCE instantiates a new ServiceProvisioningMetadata object containing the metadata needed to provision the new Composite Service and asks the RTE Metadata Manager to provision it. As this operation takes some time, depending on the size of the composite service, a handling event has been also implemented at the SCE in order not to allow the user to continue with the deployment until this operation has finished, otherwise it will cause an error.

Steps 7 to 8: according to the SCE request, the Metadata Manager saves the new composite service and checks if it is a new one.

Steps 9 to 11: when it is confirmed that it is a new composite service, the Metadata Manager generates the Start process Java interface needed to execute it and saves it in the Service Artefact Repository.

Step 12: once all previous steps were successfully executed, the Metadata Manager returns true, otherwise it will return false if fails.

The final interaction between the SCE and the RTE is to deploy composite services in the runtime production platform.

Step 13: once the provisioning metadata operation is finished without any error, the deployment of a composite service in the sandbox is performed. More details about this operation are provided in D5.2 section 3.3.3 [IOTESTD52].

Once the service is deployed in the sandbox, tests are executed and the Knowledge Management is updated. More details are provided in section 3.3.3.

Step 14: as soon as the service is fully tested and accepted, the deployment of the composite service in the Runtime Production Platform is performed by using the same operation that was used to deploy the service in the sandbox, only the “test” input parameter is different (“false”).

Step 15: once the service is successfully deployed and instantiated the user is able to execute the composite service through the SCE GUI.

3.3.3 Interaction between SCE and Testing Components

The interaction between the SCE and the Testing components is performed in two steps, one for the derivation of test cases and the other one for their execution, for both atomic and composite service. Steps for the two types of services are the same.

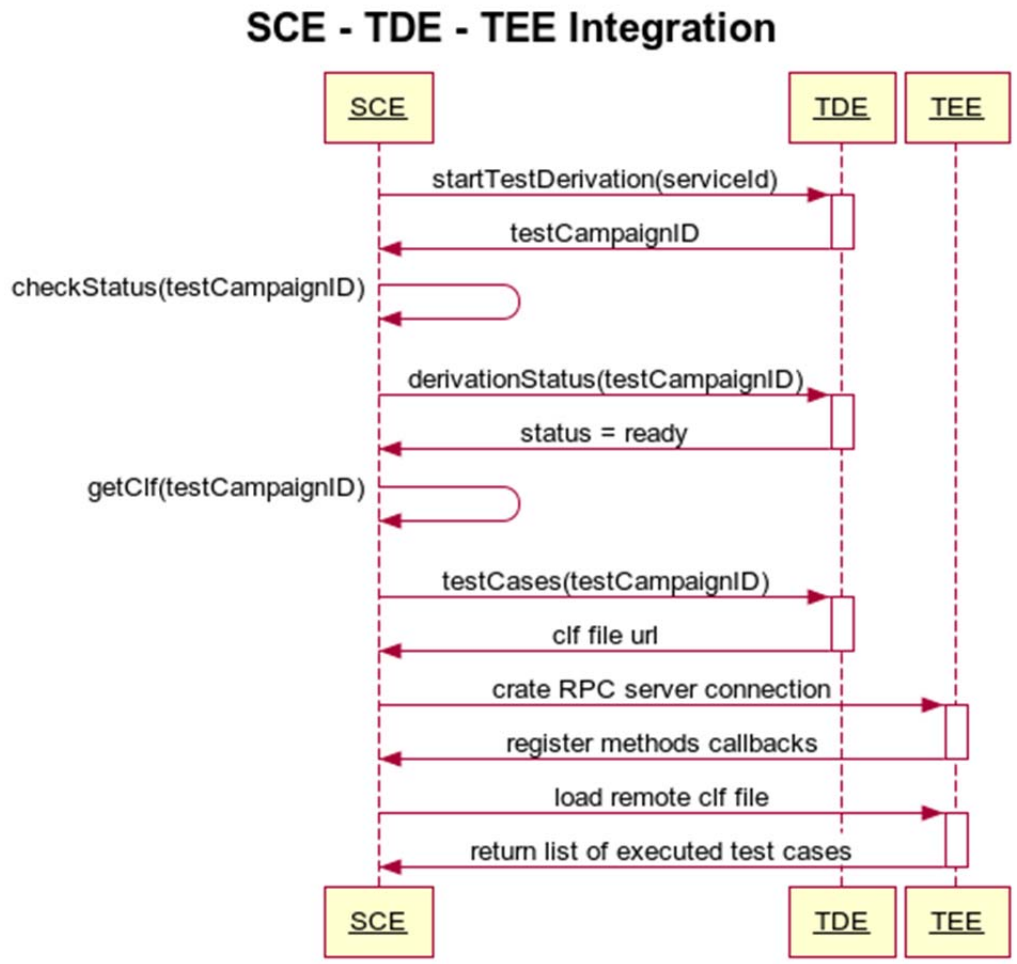

Figure 10: Interaction between SCE and Testing Components

Step 1: the SCE starts the test derivation providing the service ID to the TDE. This operation is automatically done for composite services or under user request for atomic services. The TDE gives back the test campaign ID.

Step 2: the SCE check the status of the request until the TDE responds that the test campaign is ready. Alternatively there is an Action Handler method that can be used.

Step 3: the TDE gives back the .clf file to the SCE, so it can display the list of test cases on the screen, allowing the user to select which of them he wants to be executed.

Step 4: once the list of test cases is ready, the SCE opens a connection with the TEE in order to execute test cases. This step is mandatory any time a test campaign needs to be executed.

Step 5: the .clf file is loaded into the TEE and it returns tests results one by one.

Step 6: this results are finally collected by the SCE to produce a global one to be shown to the user and, in the specific case of a composite service, to update the semantic annotations on the Knowledge Management.

3.3.4 Interaction between KM and TDE

Step 1: the TDE requests the description of the composite service from the Knowledge Management component.

Step 2: the Knowledge Management component returns the semantic description parameters of the composite service as a JSON string with the following structure:

```
{
  "compositeServiceID": "composite123123",
  "serviceDescription": "textual service description",
  "serviceCategory": "serviceCategory",
  "serviceLatitude": "51.243683",
  "serviceLongitude": "-0.591858",
  "serviceLogicalLocation": "locationFromIndoorOntology",
  "serviceRelatedEntity": "relatedEntity",
  "behaviourURL": "BehaviourURL",
  "composingServiceIDs": ["atomicserviceID1",
  " atomicserviceID2"],
  "usedAtomicOperationEndpoints": ["atomicserviceID1endpoint",
  " atomicserviceID1endpoint"],
  "compositeServiceTasks": [{
    "taskID": "StartTask",
    "operationName": "task1",
    "operationURL": "http://UseThisURLToStartTheTask",
    "inputParameters": ["inputParameter1",
    "inputParameter2"]
  }]
}
```

Step 3: afterwards, it connects to service specific knowledge bases (e.g. Ontology) to resolve the described parameters.

4. Graphical User Interface

As it was explained in section 3.2.1, the SCE performs the majority of the operations within the whole environment, acting as the core of the system, but without exposing any interface to interact with other components as it is the one that consumes them. This means that all operations are centralized on the SCE, and it takes care of calling the appropriate method of each component to perform any action, from searching services to executing a composite service.

Apart from that, the SCE offers a GUI to consolidate the usage of all involved components, allowing human interaction when it is needed or just showing results on the screen. As most of the operations are performed on the background, this GUI allows the user to monitor the progress of the operations that he is performing.

Once the user connects to the system, the first screen he will see is an Activiti console with the list of already available composite services or just business workflows that have been designed before.

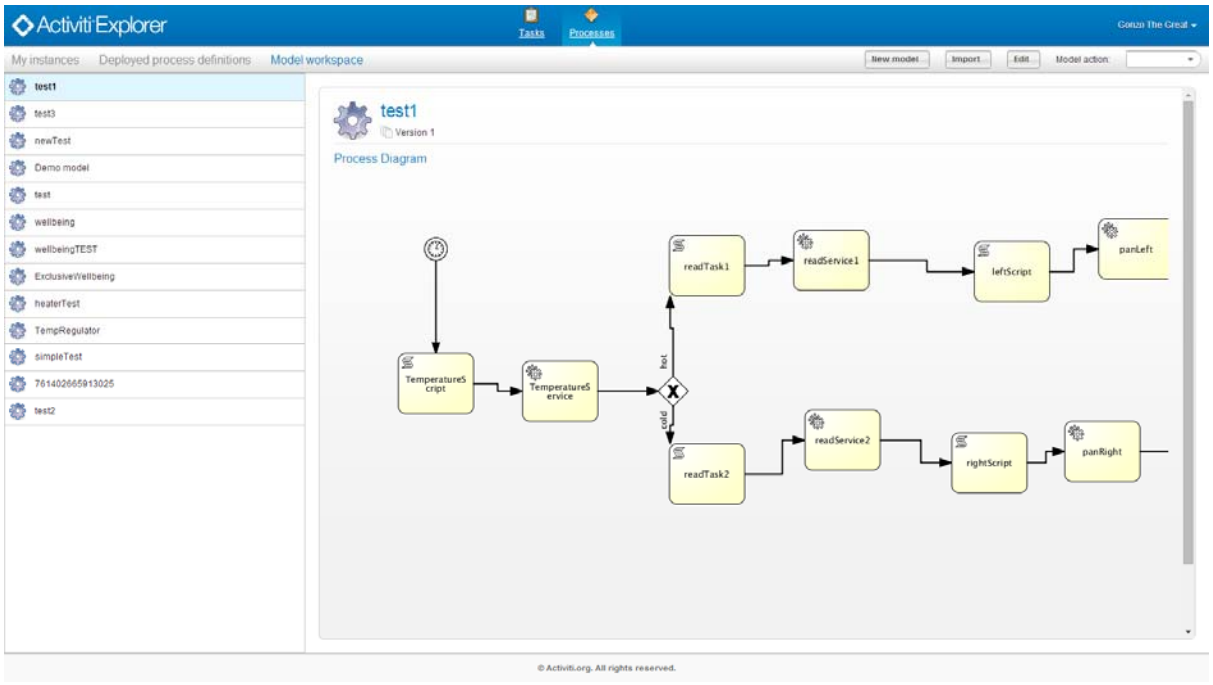

Figure 11: Activiti screen with the list of available business workflows and CSs

The user can select one of these composite services or business workflows to edit it, or create a new one from the scratch. In both cases, the Editor will be opened to proceed with the service composition.

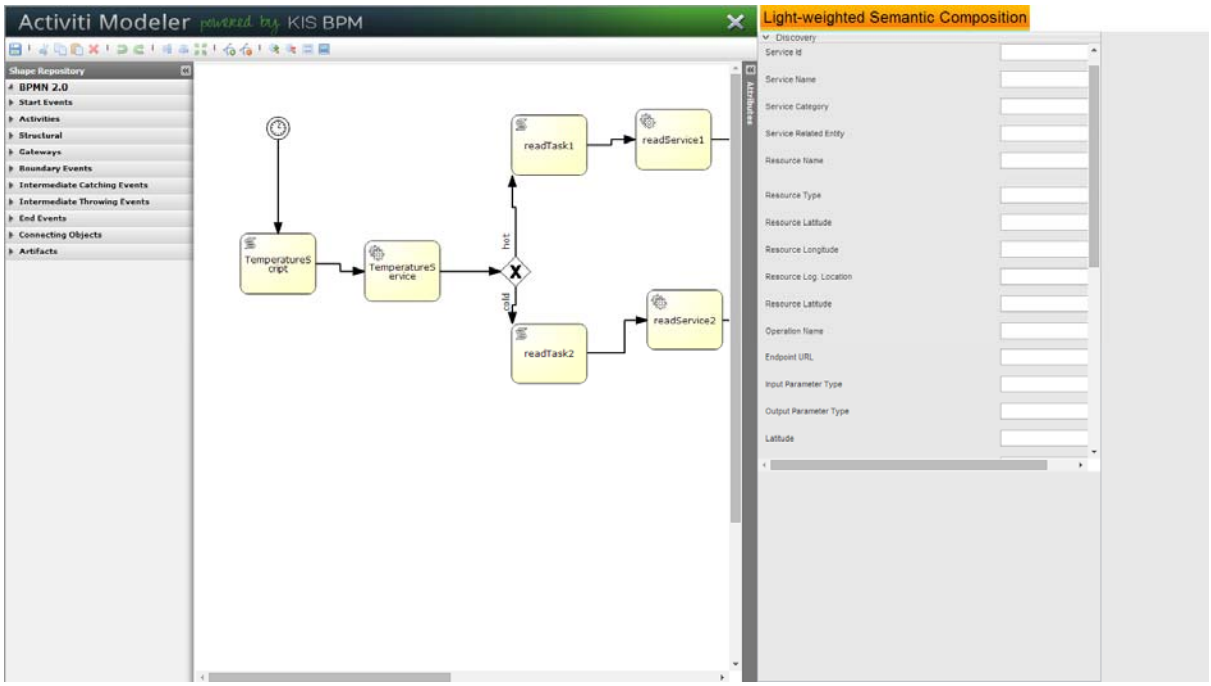

Figure 12: SCE Editor main screen

As it can be shown in Figure 11, the main screen is divided into 3 panels that provide different functionalities.

- *Left panel:* it contains the list of all elements, supported by standard BPMN2.0, which can be added to the workflow design. These include:
    - Start and end events: that are mandatory, otherwise the workflow won't be executable as it needs starting and end points.
    - Boundary and intermediate events: optional elements to be added, just in case the workflow needs some external action to succeed before continuing the execution.
    - Gateways: these are also optional, but recommended when different actions need to be executed in parallel or to establish a decision point.
    - Tasks: the most common ones are script, human and service tasks. Service tasks are the ones where atomic services will be bound, script tasks are used to execute procedures in background and human tasks are only needed where there is the need for some human interaction (i.e. management approval before sending a confirmation).
- *Central panel:* it contains the graphical representation of the business workflow. In order to easiest its design, all the elements of the left panel can be drag and drop to this central panel and they are automatically represented. There is also the possibility to add some attributes to each element, like ID or Name to easily identify them within the generated .bpm file.
- *Right panel:* this panel is called Light-weighted Semantic Composition, as it is the one who offers the functionalities for creating a service composition.

This right panel needs to be described more in details as all the functionalities provided by the IoT.est project are included there, although more details will be provided with D6.2 companion document [IOTESTD6.2]. This document is not included in the plan for WP6, as D6.2 was initially a software

deliverable, but a companion document will be provided with the software release in order to provide more details about the final integrated IoT.est framework.

Coming back to the right panel, the first thing that the user will see are the fields to perform a search. This search is mandatory to find services that can match with user requirements for a specific task, as it is shown in Figure 12.

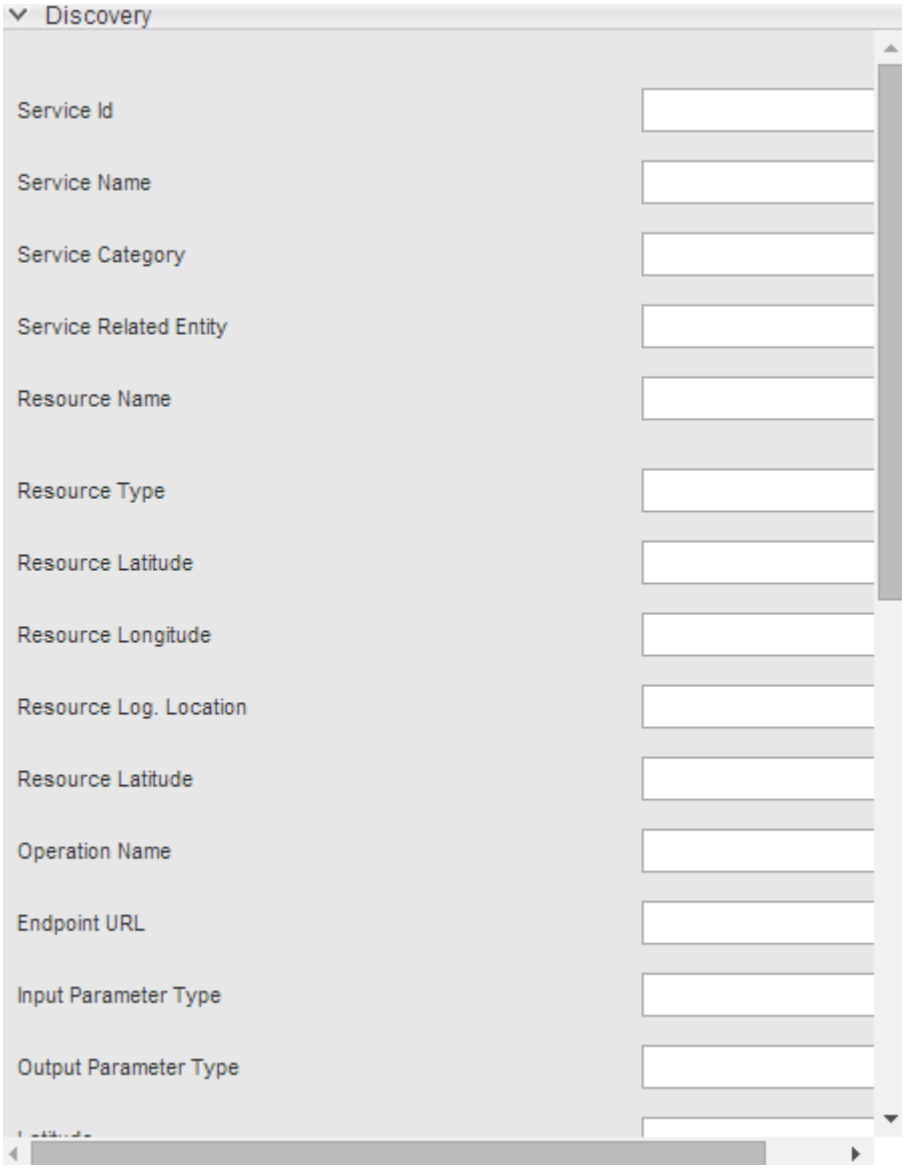

Figure 13: Search fields

None of these fields is mandatory, but if the user doesn't fill any of them the list of services that will be present includes all available services without having any relation with the needed functionality to be provided.

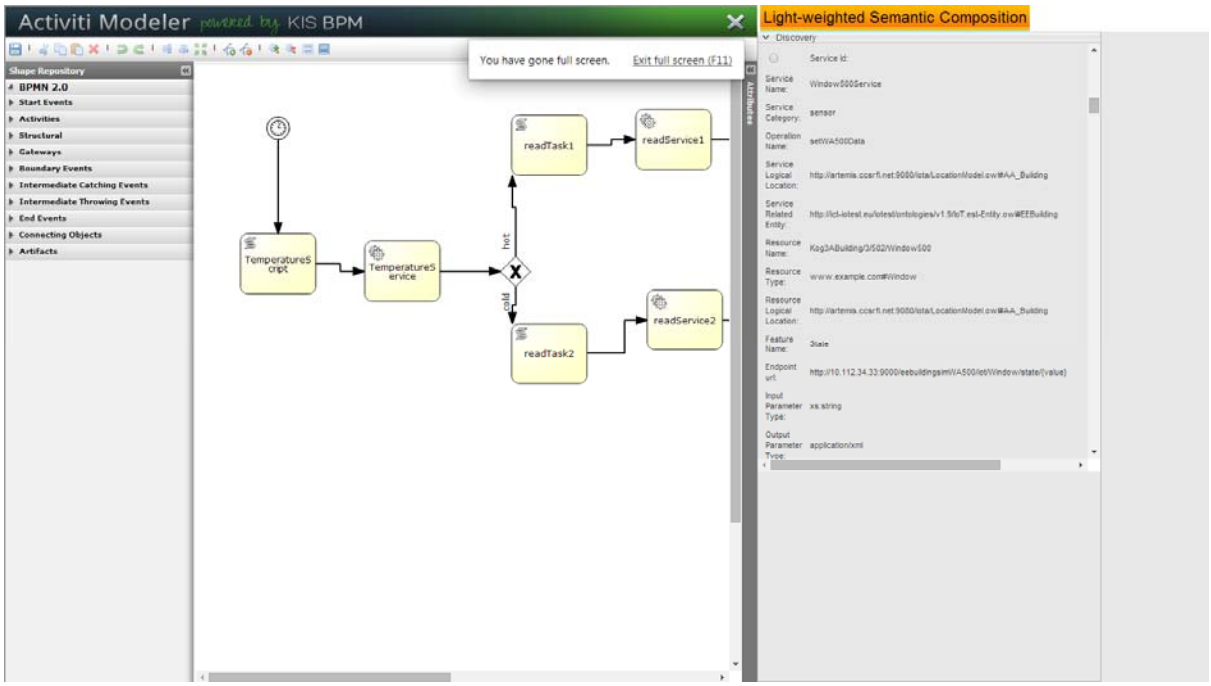

Figure 14: List of available services after perming a search

The list of services is presented as it is shown in Figure 13, indicating all the available information about it, as ID, Name, Operation, logical location, etc. The user can select any of the services listed based on their own criteria.

When a service is selected it can be tested and, if results are OK, bound to a specific service task. Once all service tasks have a bound service, the composition is finished and the user can deploy it. Deployment operation is performed on the background, so the user is not aware of it, but it includes deployment on the RTE and on the Sandbox, test derivation and execution and update of its annotations on the Knowledge Management. Once all these operations have been performed, the user will see results on the screen and can decide whether to execute it immediately or not.

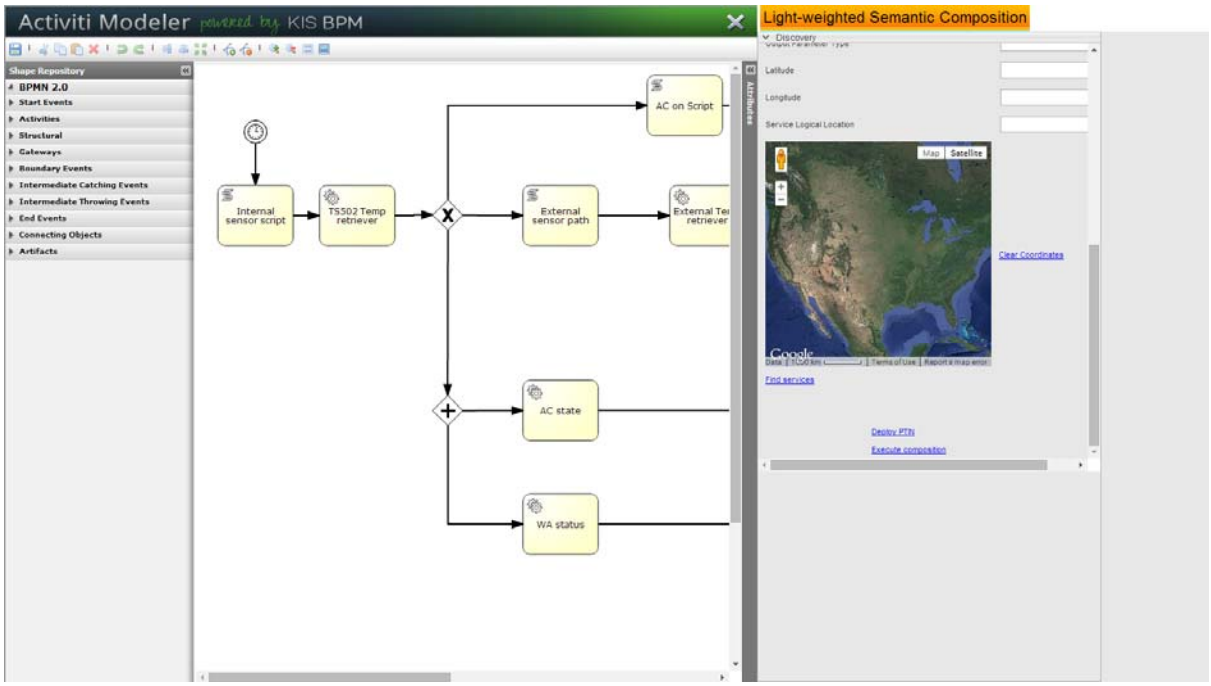

Figure 15: Different actions to be performed

5. Test Derivation Specification

The test derivation concept envisions a technology independent test creation for extensively described IoT-services. The realised implementations are based on the IoT.est service concept. It utilises RESTful interfaces to encapsulate IoT services for enhanced re-usability in composition. 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 [Hadley2006]. Input parameters as well as service responses are extensively semantically described. The implemented AS can be deployed in any Runtime Environment (RTE) and is registered in the Knowledge Management (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. It only uses AS interfaces to acquire sensor information and to control actuators. The interfaces are also described by WADL and a semantic description, which is used to enable re-usability for composition and testing.

5.1 Test Derivation in the Service Life Cycle

The test derivation is based on the knowledge driven IoT.est service life cycle [IOTESTD23], which enables a structured utilisation of semantic descriptions for knowledge representation, annotation and the process of test derivation. This ensures a high level of re-usability and testing during the service lifetime. Figure 15 shows the annotation of information in the service life cycle. Main life cycle stages, which collect data for the test derivation, are:

- **Service Composition and Creation:** During this phase developers are annotating relevant information for interfaces and parameters, which are used inside the service. Defining boundaries, more precise than on a technical data type level, enables better reusability in compositions and enables the test generation of functional conformance tests.
- **Finalisation of Functional Description of Services:** To enable the usage and testability of the service, in addition to the internal definitions, the external service interface and the behaviour of the service has to be described. By using RESTful services this leads to a complete description of all service methods including their parameters that can be called at the HTTP-based interface. Furthermore the coherence between these methods (e.g. registering a sensor before it can be used) has to be described.

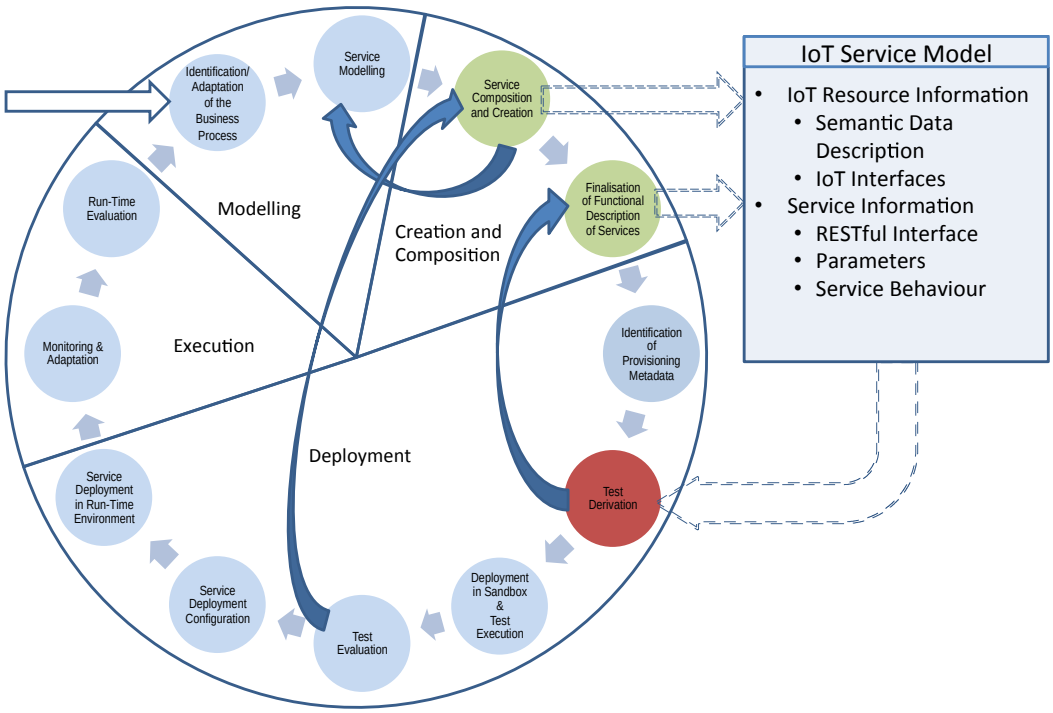

Figure 16 : Test Derivation in the Service Life Cycle

5.2 Service Interface Descriptions

In this Section the different types of service descriptions are described. These descriptions are used in Section 5.4 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 [Fielding2000], 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.

5.2.1 Service 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 Listing 1). 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.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02"
 xmlns:camschema="http://example.org/ValueRanges">
  <doc xmlns:jersey="http://jersey.java.net/"
  jersey:generatedBy="Jersey: 1.17.1 02/28/2013 12:47 PM" />
  <grammars>
    <include href="http://10.112.34.41:8080/TestDataStore/service/CameraService/xsd0.xsd"/>
    <include href="xsd1.xsd"/>
  </grammars>
  <resources base="http://10.112.34.42:8042/CameraService/iot">
    <resource path="/Camera">
      <resource path="/position/{id}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id"
          style="template" type="xs:string" />
        <method id="getPosition" name="GET">
          <response>
            <ns2:representation xmlns:ns2="http://wadl.dev.java.net/2009/02"
              xmlns="" element="PositionResponse" mediaType="application/xml" />
          </response>
        </method>
      </resource>
      <resource path="/zoom/{id}/{value}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id"
          style="template" type="xs:string" />
        <param name="value" style="template" type="xs:double" />
        <method id="setZoom" name="POST" />
      </resource>
      <resource path="/pan/{id}/{value}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id"
          style="template" type="xs:string" />
        <param name="value" style="template" type="xs:double" />
        <method id="setPan" name="POST" />
      </resource>
      <resource path="/tilt/{id}/{value}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id"
          style="template" type="xs:string" />
        <param name="value" style="template" type="xs:double" />
        <method id="setTilt" name="POST" />
      </resource>
    </resource>
  </resources>
</application>
```

Listing 1 - WADL Annotation with Enhanced Datatypes

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

5.2.1.1 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 Listing 2. 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.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="valve">
  <xs:simpleType>
   <xs:restriction base="xs:double">
     <xs:minInclusive value="-25.0"/>
     <xs:maxInclusive value="15.5"/>
   </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Listing 2 - Simple Parameter Restriction

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 17). This methodology results in 4 disjoint classes: iP1, vP1, vP2, iP2.

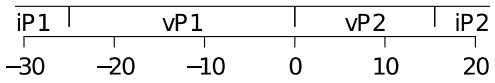

Figure 17 : Equivalence Class Partitioning Example

Further Examples of defining restricted parameters are shown in Listing 3.

Listing 3 - Parameter Restriction Examples

5.2.1.2 Semantic Parameter Description

The test data generation uses semantic annotations that can be linked to upper level ontologies like the Suggested Upper Merged Ontology (SUMO)[1] [Niles2001] 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 Listing 4. The *owlType* definition is declared for each semantic parameter and linked to a class definition within the ontology by the *requestLink* tag (Listing4:3:6). The *restriction* tag describes the predicate on which the interdependency is defined (Listing 4:7). Listing 5 shows the generated SPARQL Protocol and RDF Query Language (SPARQL) code that is used to find the matching entities in the ontology.

```
<owlTypeDefinition>
  <owlType name="Countryname" type="base">
    <requestLink uri="http://www.onto.org/SUMO.owl#Nation" />
  </owlType>
  <owlType name="Cityname" type="restricted">
                    <requestLink uri="http://www.onto.org/SUMO.owl#City" />
    <restriction uri="http://www.onto.org/SUMO.owl#geographicSubregion"
      value="{Countryname}" />
  </owlType>
</owlTypeDefinition>
```

Listing 4 - Semantic Parameter Interdependency

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

Listing 5 - SPARQL - Query

5.2.1.3 Geospatial Parameter Definition

Since IoT-based services 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 polygon1, 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)2) 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 17.

[1] Suggested Upper Merged Ontology (SUMO): http://www.ontologyportal.org/

- 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 17 shows the created equivalence partitions of the bounding box and the polygon. The precision improvement of the polygon is shown in Figure 18 since after comparison it shows that the bounding box describes a false positive valid equivalence class:

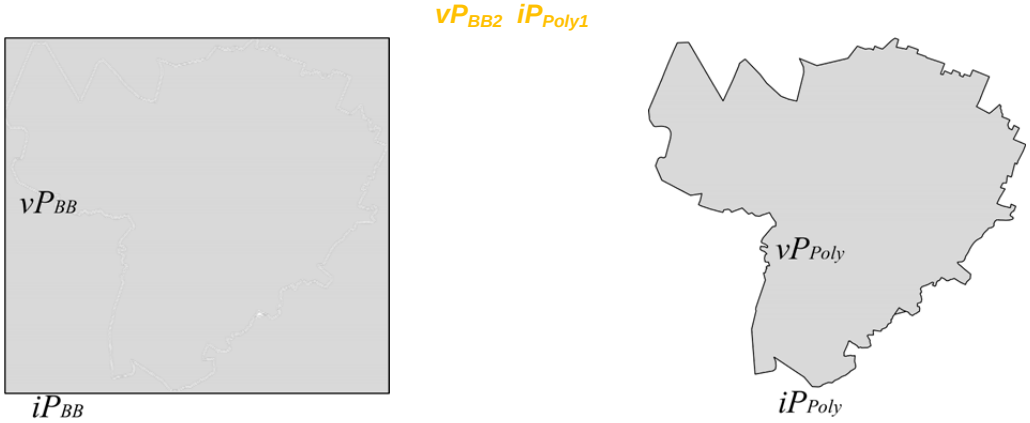

Figure 18 : Equivalence Classes defined by Bounding Box

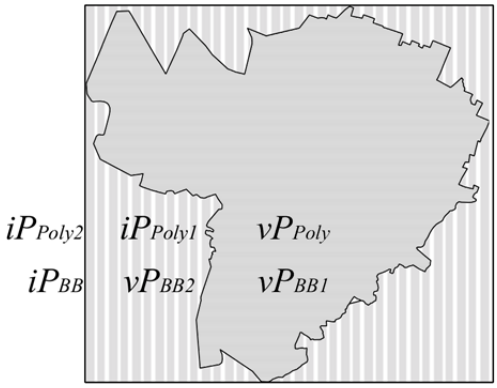

Figure 19 : Comparing Equivalence Classes

Figure 19 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.

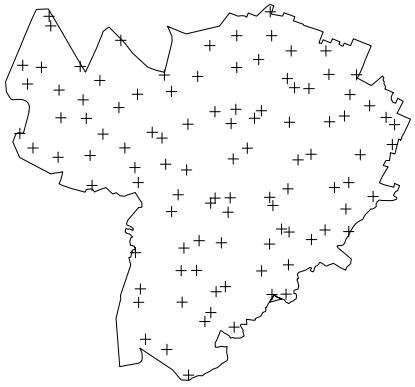

Figure 20 : Random Generated Test Data in the City Area

5.2.2 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. Listing 6 illustrates the main structure of a sequence description document.

```
<sequencespecification xmlns:ws1="application.wadl">
  <vars>
    <var name="cameraPan" type="double" />
  </vars>
  <paramsets>
    <paramset id="cameraPan">
      <param name="id">10.11.127.6</param>
      <param name="value"></param>
    </paramset>
    ...
  </paramsets>
  <results>
    <result name="testPosition" mediatype="application/xml">...</result>
  </results>
  <sequence mode="multiple" subSequenceType="All">
    <subsequence mode="single" subSequenceType="MutualExclusive">
      <subsequence mode="single">
        <wsuri path="ws1:/Camera/pan/{id}/{value}" paramset="cameraPan" />
        <wsmethod name="ws1:setPan" returnCode="2xx" />
        <setvar var="cameraPan">{value}</setvar>
      </subsequence>
      <subsequence mode="single">...</subsequence>
  </sequence>
</sequencespecification>
```

Listing 6 - 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*, 6:16) and a method (*wsmethod*, 6:18), 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.

5.3 Evaluation of an Example Service

To demonstrate the algorithms and the process of the test framework we use a camera control example service for illustrating the test derivation. 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 20). Between S3 and S4 the values can be set with the *setTilt* and *setPan* method and evaluated with the *getPosition* method at the transition back to S3.

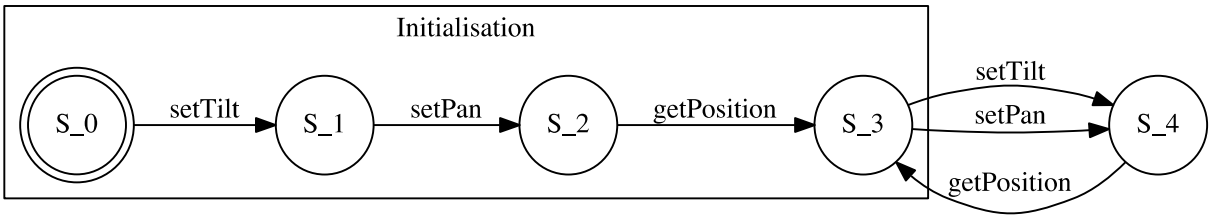

Figure 21 : Camera Example Service

5.4 Test Case Derivation Process

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 21 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.

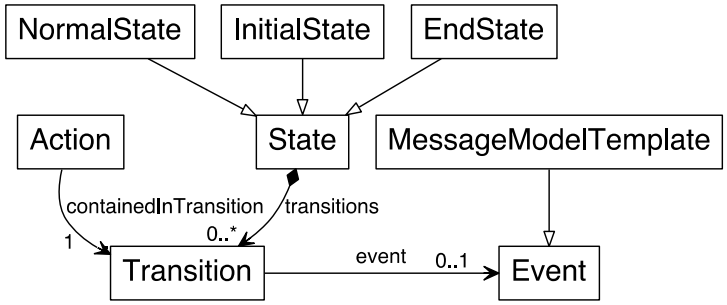

Figure 22 : Simplified EMF Model

The stages of the test derivation process and its information flow are shown in Figure 22. 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 details.

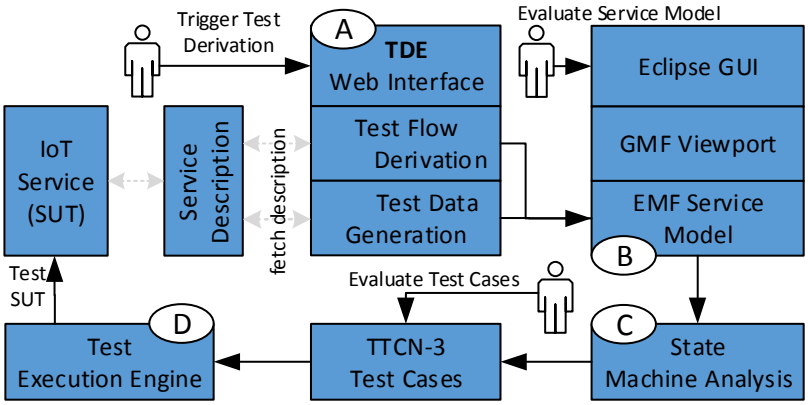

Figure 23 : Information Flow

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 [Juristo2012]. 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 5.2.1).

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 [Lin2011] 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 an 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 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 1/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 (Table 1/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 (Table 1/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.

After creation the EMF model can be evaluated and altered in an Eclipse GMF - Model Editor (see Figure 23).

Table 1 : Document Transformation to Model Object NormalState

Input:	Action:
<sequence mode=”single”> ... </sequence>	Create NormalState for each sequence.
<sequence mode=”single”></sequence> <sequence mode=”single”></sequence> or <sequence mode=”multiple”> <subSequence>...</subSequence> <subSequence>...</subSequence> </sequence>	If exists: create the next normal state.
<!-- WADL document --> <resources base=”http://10.1.1.42:8080/”> <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>	Create MMT event for a request to a IoT service from WADL.
<!-- Sequence description --> <sequence mode=”single”>	Create MMT event for a request to an IoT service

<wsuri path="ws1:/Camera/pan/{id}/{value}" paramset="cameraPan"/> <wsmethod name="ws1:setPan" returnCode="2xx"/> </sequence>	from Sequence.
<paramset id="cameraPan"> <param name="id">10.11.127.6</param> <param name="value">12</param> </paramset>	Test data 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"/>	Add HTTP status codes to MMT.
<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.

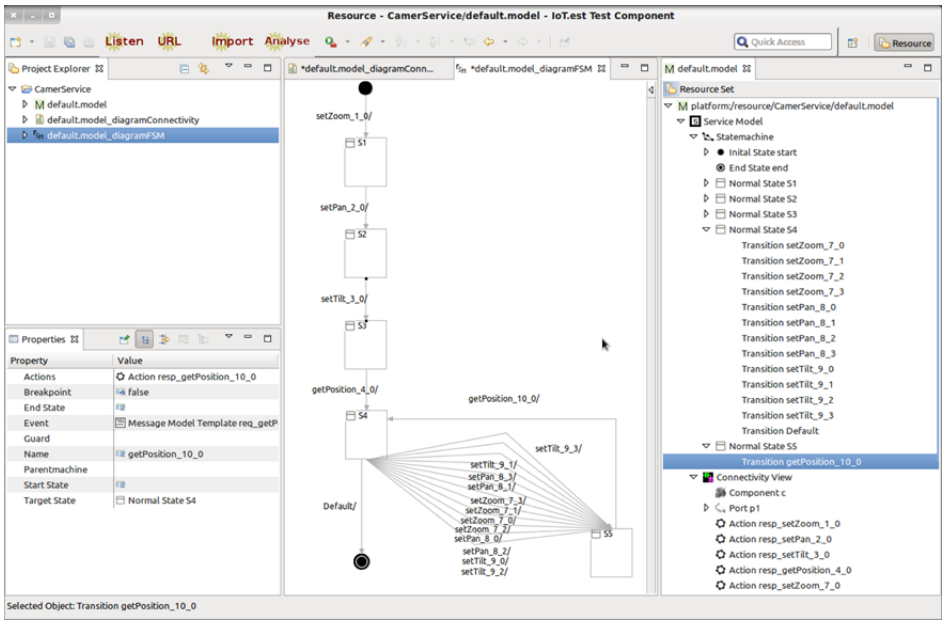

Figure 24 : EMF Model Evaluation in Eclipse

C. Creating Test Cases From the Service Model

The EMF model is used to analyse resulting state machines from the previous step. It can be chosen between transition coverage or transition and state-coverage based on the W method [Gargantini2005]. 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 transition directs to an 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 (*W set*) of input sequences that can be utilised to distinguish every pair of existing states in the model. The 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 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 State (e.g. $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_3$, see Figure 21 : Camera Example Service). 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 Velocity[2] 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. In the following the transformation step is outlined with some detail. Table 2 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 2 - TTCN-3 Translation of Model Object Initial State

Action:	TTCN-3 Output:
Add timeout timer	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 a received input by the SUT and an Action that describes the output reaction of the SUT to this input.

Table 3 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.

[2] The Apache Velocity Project: http://velocity.apache.org/

Table 3 - 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://10.1.1.42:8080/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;

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 *alt* element is used to form the possibilities of the SUT behaviour. At first, the failure case for delayed or unexpected responses is modelled. Then the followed approach assumes deterministic service behaviour with one possible valid reaction. This expected behaviour is included in the *alt* element including the jump to the next TTCN-3 function (state) created before. Table 4 shows the discussed transformation.

Table 4 - 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();} . ..}

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 5 reveals the resulting TTCN-3 output.

Table 5 - Translation of Model Object Normal State

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

At the final stage of the test case the model object *EndState* is reached. This completes the code creation by setting the verdict to pass. If all functions, requests and response messages have been transmitted during the test case execution this indicates that the SUT has the expected behaviour for this test case. Table 8 shows the resulting TTCN-3 code.

Table 6 - TTCN-3 Translation of Model Object EndState

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

D. Creating Test Cases from the Service Model

After compilation of the TTCN-3 test cases the whole test flow is executed by a web service interface triggered by the SCE. To visualise detailed test results it can evaluated using the TT-Workbench. This enables a visualised logging of test execution in a log report, which can be used to evaluate the detailed test results (see Figure 25).

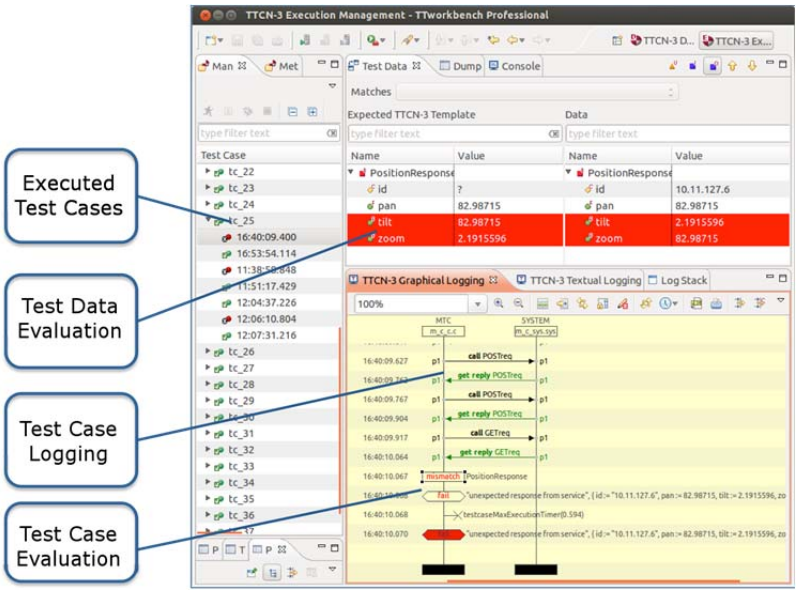

Figure 254 : Test Case Evaluation

5.5 Test Component Summary

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 IoT.est Framework lowers 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 trans- formation 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 in 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.

6. Summary and Outlook

The results on the work to define and implement the IoT.est SCE are presented in this document. All concepts introduced in the previous report [IOTESTD41] have been implemented and specified here. At the time this report has been completed further improvements on components are being developed; the documentation of those extensions includes the descriptions of the prototypes for each of the components specified in the architecture. While the functionality has been implemented and there is still work remaining for adding functionalities to the integrated IoT.est prototype. These developments are documented in D6.2 companion document [IOTESTD62].

This report describes the main functionalities of each component that is part of the Service Composition Environment as well as the way it interacts with other components. The decision to change the overall architecture was taken during the implementation phase, as it the work has shown that a centralized approach will benefit the whole process as the information is always flowing monitored by at least one component.

In general, WP4 has achieved its main objectives with the development of a Service Composition Environment that offers the possibility not only of composing goal-oriented services but to add testing functionalities what will result in a more trustable composition for the end user. Also results coming from WP3 have been incorporated to enrich with semantics the operation performed within the SCE and results from WP5 in order to give the user the possibility of interact with the framework along the whole IoT.est lifecycle, going for business modelling to service execution.

Different projects, initiatives and other related work were investigated in order to have the most complete vision as possible of the different options that already exist, and the IoT.est project was one step further consolidating this isolated approaches in only one framework providing all requested functionalities. But this framework hadn't been develop as an end point, as its modular approach allows further investigations and evolutions for each component. This is the ideal solution, as i.e. testing field offers a lot of possibilities to evolve the current solution modelling and emulating the correlation of sensors and actuators as part of the test concept. The semantic model can be also extended taking into account the complexity of describing IoT services with missing domain specific knowledge. And also the Service Composition Environment, providing more functionalities like the automation of the composition based on specific criteria or the support of different business modelling languages.

7. References

[Aydal2009] 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, Sept. 2009.

[Binkley2007] D. Binkley, “Source code analysis: A road map,” in 2007 Future of Software Engineering, FOSE ’07, (Washington, DC, USA), pp. 104– 119, IEEE Computer Society, 2007.

[Charfi2004] A. Charfi and M. Mezini, “Hybrid web service composition: business processes meet business rules”. In Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC’04), ACM, New York, NY, USA, pp 30-38, 2004.

[Deng2009] 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.

[Fielding2000] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, 2000.

[Fischer2012] 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), Sept. 2012.

[Gargantini2005] A. Gargantini, “4 conformance testing,” in Model-Based Testing of Reactive Systems, Springer, 2005.

[Hadley2006] M. J. Hadley, “Web application description language (wadl),” Sun Microsystems, https://wadl.dev.java.net, 2006.

[Huang2013] W.-l. Huang and J. Peleska, “Exhaustive model-based equivalence class testing,” in Testing Software and Systems (H. Yenign, C. Yilmaz, and A. Ulrich, eds.), vol. 8254 of Lecture Notes in Computer Science, pp. 49–64, Springer Berlin Heidelberg, 2013.

[IOTESTD23] D. Kuemper et al, “Framework for IoT Service Life Cycle Management”, IoT.est Deliverable D2.3, 2013.

[IOTESTD41] L. López et al. “Concepts of Goal-oriented IoT Service Composition and Testing (Design-Time)”, IoT.est Deliverable D4.1, 2013.

[IOTESTD51] E. Reetz et al. “Concepts of Automated IoT Service Provisioning and Adaptation (Run-Time)”, IoT.est Deliverable D5.1, 2013.

[IOTESTD52] P. Chainho et al., “Specification of Automated IoT Service Provisioning and Adaptation (Run-Time)”, IoT.est Deliverable D5.2, 2014.

[IOTESTD62] L, López et al., “Architecture of the Integrated System, Installation and User Manual”, IoT.est Deliverable D6.2, 2013.

[Juristo2012] 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, April 2012.

[Kuemper2013] D. Kuemper, E. Reetz R. Toenjes, “Test Derivation for Semantically Described IoT Services”, 22nd Future Network and Mobile Summit 2013, Lisbon, Portugal, 2013.

[Lin2011] M. Lin, Y. Chen, K. Yu, and G. Wu, “Lazy symbolic execution for test data generation,” Software, IET, vol. 5, April 2011.

[Meyer2013] 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, vol. 7908 of Lecture Notes in Computer Science, Springer, 2013.

[Niles2001] N. and A. Pease, “Towards a standard upper ontology,” in Proceed- ings of the International Conference on Formal Ontology in Information Systems - Volume 2001, FOIS ’01, (New York, NY, USA), ACM, 2001.

[Pretschner2005] A. Pretschner, “Model-based testing,” in 27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings, May 2005.

[Pu2006] Pu, K., Hristidis, V., Koudas, N., “Syntactic Rule Based Approach to Web Service Composition”. Proceedings of the 22nd International Conference on Data Engineering 2006 (ICDE’06), vol. 31, no. 31, 03-07 April 2006.

[Reetz2013] E. Reetz, D. Kuemper, K. Moessner, and R. Toenjes, “How to test IoT- based services before deploying them into real world,” in 19th European Wireless Conference (EW 2013), Guildford, United Kingdom, 2013.

[Rosenberg2005] F. Rosenberg and S. Dustdar, “Design and Implementation of a Service-Oriented Business Rules Broker”, Proceedings of the 2005 Seventh IEEE International Conference on E-Commerce Technology Workshops, 2005.

[Takanen2008] A. Takanen, Fuzzing for software security testing and quality assurance. Information security and privacy series, Artech House, 2008.

[Toenjes2012] R. Toenjes, 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.

[Tracey1998] 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, 1998.

[Walkinshaw2013] 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.

[Ziqin2012] Z. Yin, G. Zhang, T. Wang, S. Li, “Rule engine-based web services composition,” World Automation Congress (WAC), 2012 , vol. 1, no. 4, pp.24-28, June 2012.