Executive Summary

The information generated from the Internet of Things (IoT) potentially enables better understanding of the physical world for humans and supports creation of ambient intelligence for a wide range of applications in different domains such as smart cities, healthcare and intelligent transportation. Integrating service oriented computing mechanisms and semantic technologies to create a semantic service layer on the top of IoT is a promising approach to facilitate seamless access and management of the information from the large, distributed and heterogeneous sources. Architecture of the IoT has been already developed in several projects; however, a reference architecture for the IoT services is still not available. This architecture is needed to support the service oriented computing for IoT services and many existing business services and applications.

This report represents the efforts of the IoT.est project towards developing a reference architecture for service creation and testing in an Internet of Things environment. The architecture design extends the existing architectures on Internet of Things and takes the unique characteristics of the Internet of Things based services into consideration. The knowledge management component in the architecture is developed based on a semantic description framework, which provides a foundation for knowledge based service discovery, composition and testing. The architecture contains an IoT service composition environment with re-usable components for a number of common functionalities such as data integration and management, sensor data mining, and pattern extraction. Employing those re-usable components facilitates service creation and testing. Another notable feature of the architecture is that it provides mechanisms to test IoT services in the service providers’ infrastructure before deployment. Furthermore, the service runtime environment contains service monitoring mechanisms for context-aware service adaptation which is able to respond to changes in the environment. The developed architecture serves as a reference framework for the IoT.est project and other Internet of Things enabled services and applications.
# Contents

List of Abbreviations ............................................................................................................. 5

1. Introduction .......................................................................................................................... 6
   1.1 Purpose of the Document .............................................................................................. 6
   1.2 Relation to Other Tasks ................................................................................................ 6
   1.3 Structure of the Document ............................................................................................ 7

2. State-of-the-art Architectures in IoT .................................................................................... 8
   2.1 IoT-A Architecture ......................................................................................................... 8
   2.2 SOA4ALL Architecture .................................................................................................. 10
   2.2.1 Distributed Service Bus ............................................................................................ 11
   2.2.2 SOA4All Studio .......................................................................................................... 11
   2.2.3 Platform Services ....................................................................................................... 12
   2.2.4 Business Services and Processes ............................................................................. 12
   2.2.5 SOA4ALL Summary .................................................................................................. 12
   2.3 SENSEI .......................................................................................................................... 12
   2.4 Test Architectures ........................................................................................................... 14
      2.4.1 Modelling of test environments with UML testing profile .................................... 14
      2.4.2 Test automation with TTCN-3................................................................................ 15
      2.4.3 Software-tests with JUnit ....................................................................................... 15
      2.4.4 JMeter ..................................................................................................................... 15

3. The IoT.est Architecture .................................................................................................... 17
   3.1 Overview of the Architecture ....................................................................................... 17
   3.2 Functional View of Architecture .................................................................................. 18
      3.2.1 Knowledge Management ....................................................................................... 19
         3.2.1.1 Service Registration .......................................................................................... 20
         3.2.1.2 Service Lookup ............................................................................................... 20
         3.2.1.3 Service Update ................................................................................................. 20
         3.2.1.4 Service Removal .............................................................................................. 21
         3.2.1.5 Service Discovery ............................................................................................ 21
         3.2.1.6 IoT.est Description Ontology .......................................................................... 21
         3.2.1.7 Distributed Service Description Repository .................................................. 21
         3.2.1.8 Domain Knowledge Base ............................................................................... 21
         3.2.1.9 Lifecycle Management ..................................................................................... 21
         3.2.1.10 Role Management ......................................................................................... 22
      3.2.2 Service Composition Environment ......................................................................... 22
         3.2.2.1 Designer GUI .................................................................................................... 22
         3.2.2.2 Governance ...................................................................................................... 23
         3.2.2.3 Artefact Manager .............................................................................................. 23
         3.2.2.4 Design Time Composer .................................................................................. 23
         3.2.2.5 Workflow Manager ........................................................................................ 23
3.2.2.6 Data Binding Manager ................................................................. 24
3.2.2.7 Knowledge Management Client ..................................................... 24
3.2.3 Test Component ........................................................................ 24
3.2.3.1 Test Design Engine ................................................................. 25
3.2.3.2 Test Execution Engine .............................................................. 26
3.2.4 Service Runtime ........................................................................ 27
3.2.4.1 Message BUS ........................................................................ 27
3.2.4.2 Management ......................................................................... 28
3.2.4.3 Test Emulation Components .................................................... 28
3.2.4.4 Semantic Query ..................................................................... 28
3.2.5 IoT.est Services ......................................................................... 28
3.3 Interfaces ....................................................................................... 28
3.3.1 Knowledge management interfaces .......................................... 28
3.3.2 Runtime interface ..................................................................... 29
3.3.3 Test control interface ................................................................. 29
3.3.4 Network emulation interface ..................................................... 29
3.3.5 Resource Emulation Interface ..................................................... 29
3.3.6 Test Component Interface ........................................................ 29
3.3.7 Service interface ..................................................................... 29
3.4 Dynamic View of Architecture ....................................................... 29
3.4.1 Service Composition Environment ............................................ 30
3.4.1.1 Design Phase ....................................................................... 30
3.4.1.2 Testing Phase ....................................................................... 32
3.4.1.3 Deployment Phase ............................................................. 36
3.4.2 Runtime .................................................................................... 37
3.4.2.1 Deployment Preparation ....................................................... 37
3.4.2.2 Configuration ..................................................................... 38
3.4.2.3 Service Usage ..................................................................... 39
3.4.2.4 Service Monitoring .............................................................. 40
3.4.2.5 Service Adaptation .............................................................. 41
4. Requirements Traceability Matrix ....................................................... 42
   4.1 Service Registry and search/query engine ..................................... 42
   4.2 Service composition .................................................................. 43
   4.3 Test ......................................................................................... 45
   4.4 Service Runtime ...................................................................... 47
5 Conclusion and Outlook .................................................................. 48
References .......................................................................................... 49
## List of Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Architectural Reference Model</td>
</tr>
<tr>
<td>AS</td>
<td>Atomic Service</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Model and Notation</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create Remove Update Delete</td>
</tr>
<tr>
<td>CS</td>
<td>Composed Service</td>
</tr>
<tr>
<td>DSB</td>
<td>Distributed Service Bus</td>
</tr>
<tr>
<td>EFSM</td>
<td>Extended Finite State Machine</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>FG</td>
<td>Functionality Group</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphic User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transfer Protocol Secure</td>
</tr>
<tr>
<td>IMAP</td>
<td>Internet Message Access Protocol</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IoT-A</td>
<td>Internet of Things Architecture</td>
</tr>
<tr>
<td>IoT.est</td>
<td>Internet of Things Environment for Service Creation and Testing</td>
</tr>
<tr>
<td>IOPE</td>
<td>Input Output Precondition Effect</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Message Service</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>POP3</td>
<td>Post Office Protocol</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAI</td>
<td>Resource Access Interfaces</td>
</tr>
<tr>
<td>REP</td>
<td>Resource End Points</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RDFS</td>
<td>Resource Description Framework Schema</td>
</tr>
<tr>
<td>RM</td>
<td>Reference Model</td>
</tr>
<tr>
<td>RPI</td>
<td>Resource Publication Interface</td>
</tr>
<tr>
<td>SAWSDL</td>
<td>Semantic Annotation for Web Service Definition Language</td>
</tr>
<tr>
<td>SCE</td>
<td>Service Composition Environment</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol and RDF Query Language</td>
</tr>
<tr>
<td>SQR</td>
<td>Semantic Query Resolver</td>
</tr>
<tr>
<td>SUT</td>
<td>System under Test</td>
</tr>
<tr>
<td>TCEE</td>
<td>Test Case Execution Engine</td>
</tr>
<tr>
<td>TDE</td>
<td>Test Design Engine</td>
</tr>
<tr>
<td>TEE</td>
<td>Test Execution Engine</td>
</tr>
<tr>
<td>TTCN3</td>
<td>Testing and Test Control Notation Version 3</td>
</tr>
<tr>
<td>U2TP</td>
<td>UML 2.0 Testing Profile</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VE</td>
<td>Virtual Entity</td>
</tr>
<tr>
<td>WS</td>
<td>Web Service</td>
</tr>
<tr>
<td>WSAN</td>
<td>Wireless Sensor and Actuator Network</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Definition Language</td>
</tr>
<tr>
<td>WSMO</td>
<td>Web Service Modelling Ontology</td>
</tr>
</tbody>
</table>
1. Introduction
Adoption of the IoT based business services and applications is hindered by the architectural gap between heterogeneity of the underlying Internet of Things (IoT) layer and the high-level diverse business processes on the top. Adding a semantic service layer on the IoT based on the service oriented paradigm enables interoperability between IoT and the high level business world. The architecture of the IoT has been proposed such as in the IoT-A project [IoT-A], however, a reference architecture for the IoT service layer is still missing. This reference architecture is needed in order to support better integration of the Internet of Things into the service layer. This will facilitate the service oriented computing for IoT services and many existing business services and applications. One of the most important tasks of the IoT.est project is to address the limitations by proposing an architecture with open interfaces and re-usable components. Considering the highly dynamic physical world environment in which IoT resources and services operate, service testing is included into the service design and deployment time of the service lifecycle. Compared to many well-designed business services and applications, IoT services that run on constrained devices and environments are less reliable, therefore, more efficient monitoring and adaptation mechanisms are also needed. The architecture design will also analyse how different disciplines of IoT, service engineering and testing can be integrated.

1.1 Purpose of the Document
The purpose of this document is to present the analysis and design of a global reference architecture for IoT service creation, testing and provision. The important components, interfaces, functionalities, interactions among components and data flow are identified. The architecture design extends the existing architecture on IoT [IoT-A] by adding a semantic service layer for IoT based service creation and provision. It provides an abstraction to address the heterogeneity of underlying IoT and supports real world use cases. The architecture also includes reusable service components with common open interfaces to ease composition with high level business services. The reference architecture will be continually refined and updated as the IoT.est project progresses and knowledge is shared between the project and the IoT community.

1.2 Relation to Other Tasks
The IoT.est reference architecture provides an abstract guidance to develop compliant architectures for various IoT based services that potentially have domain or application-specific requirements. It also serves as a basis for the architectural design in the tasks related to service creation, runtime management and testing in the IoT.est project. The reference architecture represents a high-level view of the individual activities and their interaction. The related tasks, in which the architecture components will be developed in detail, are listed as follows:

- A3.1 semantic description framework for IoT services: it is the foundation of the knowledge representation component in the architecture. The framework defines a semantic model for describing and annotating IoT services, resources and other relevant concepts.
- A3.3 directory and discovery services, and query interfaces: the directory stores semantic descriptions of IoT resources, services, and their tests; the discovery and query interface provide means for other architectural components such as service composition environment and service runtime environment and test component to interact with the directory.
- A3.4 development of reusable test components and test interfaces for IoT services: reusable test component is an important part of the test component in the architecture that aims to develop testing programs for various IoT services.
- A4.1 knowledge based composition of IoT services: it is one of the core component in the architecture which aims to develop methods for IoT aware service composition based on the semantic descriptions for services.
- A4.4 test integration into IoT services composition: Service testing is novel concept introduced in the project that aims to ensure correct functionalities of IoT services. The test and service composition components in the architecture will be seamlessly integrated.
- A5.1 semantics for service resource requirements and network capabilities: it is part of the IoT.est semantic description framework focusing on resources (e.g., platform, system and deployment) and networking aspects.
- A5.3 automated large scale service deployment: service runtime deployment is another core component in the IoT.est architecture that aims to support automated deployment process by testing IoT services for their suitability in the designated network, application and environment.
- A5.4 performance monitoring and service adaptation: due to the dynamic environment and unreliability of the IoT, deployed services need to be monitored and service adaptation and compensation may need to be performed. It is part of the service runtime environment.

### 1.3 Structure of the Document

The rest of the report is organised as follows:

- **Section 2** reviews the related work on IoT architecture, service computing and software testing, in particular.
- **Section 3** elaborates the design of the reference architecture for IoT service creation, testing, composition and deployment. An abstract architecture that gives a high level overview of the reference architecture is presented first, followed by the IoT.est project specific architecture and detailed component descriptions. The interfaces of the architecture components and the information flow are also explained.
- **Section 4** provides a requirement traceability matrix which aims to explain how the architecture related requirements that have been identified in Activity 2.1 and Deliverable D2.1 are fulfilled by the current architecture design.
- **Section 5** concludes the report and points out the future work.
2. State-of-the-art Architectures in IoT

The section provides a review of the existing work on IoT architecture design, service computing and test architecture. The discussion is focussed on some of the most representative research works and projects.

2.1 IoT-A Architecture

The goal of the IoT-A project is to develop an Architectural Reference Model (ARM) for the IoT, which involves identification, use and specification of standardised components, understanding the underlying business models and deriving protocols and interfaces. The ARM is envisaged as a combination of a Reference Model (RM) and a Reference Architecture (RA). The Reference Model seeks to promote a common understanding of the IoT domain with a set of models identifying the main concepts of the IoT, their interactions and constraints. It serves as a basis of the RA. The RM, detailed in [Walewski, 2011], includes an IoT Domain Model as a top-level description, an IoT Information Model explaining how IoT information is going to be modelled, and an IoT Communication Model in order to understand specifics about communication between many heterogeneous IoT devices and the Internet as a whole. The domain model identifies the concepts of entities, resources and services as important actors in IoT scenarios. The related entity, resource and service model are presented in detail in [De et al, 2012] and a review will be presented in IoT.est Deliverable D3.1 [Wei et al, 2012]. The IoT-A project identifies entities to be the ‘thing’ in the Internet of Things and thus an entity is the main focus of interactions by humans and/or software agents. An entity could be a human, animal, car, store or logistic chain item, electronic appliance or a closed or open environment.

The virtual representation of the physical entity is termed as a Virtual Entity (VE) and is made possible by a device which either attaches to an entity or is part of the environment of an entity so it can monitor it. The actual software component that provides information on the entity or enables controlling of the device, is a ‘resource’. As implementations of resources can be highly dependent on the underlying hardware of the device, a ‘service’ provides a well-defined and standardised interface, offering all necessary functionalities for interacting with entities and related processes. The services expose the functionality of a device by accessing its hosted resources. The IoT.est information model will be an extension of the entity-resource-service models developed in the IoT-A project. The interpretation of entities, resources and services will be followed in a way similar to that defined in IoT-A but the service related model will be enhanced and extended to focus on a service oriented paradigm for the IoT. Moreover, the IoT.est project will introduce modelling aspects that will allow testing of services to be part of both the design and deployment phases of the service lifecycle.

The Reference Architecture aims at guidelines, best practices, views and perspectives that can be used for building fully interoperable concrete IoT architectures and systems. Once the IoT-A Reference Architecture is defined, it can be used by multiple organisations to implement compliant IoT architectures in specific application domains. The RA is an amalgamation of views and perspectives. The IoT-A RA interprets a view as a functional view which is defined in terms of the key functional groups that such an architecture needs to provide. The Functional View, is composed of 7 functionality groups (FGs), as shown in Figure 1.
1. **Applications**: This functionality group describes the functionalities provided by applications that are built on top of an implementation of the IoT-A architecture. The business process modelling component provides an environment for the modelling of IoT-aware business processes which can be executed in the process execution component.

2. **Process execution and service orchestration**: Orchestration and access of IoT Services to external entities and services is organised by this functionality group. The group provides a set of functionalities and APIs to expose and compose IoT Services so that they become available to external entities and can be composed by them. The process execution functional component executes IoT-aware process models by utilising IoT services that are orchestrated by the service composition and orchestration component.

3. **Virtual entity (VE) and information**: This group contains functionality to associate VEs to relevant services as well as a means to search for such services. When queried about a service of a particular virtual entity, this functionality group will return addresses of the service related to this particular virtual entity. The VE resolution component consists of functionalities to retrieve the list of services that are related to a VE. The corresponding associations between a VE and services that can be relevant to the VE are derived and maintained by the VE and IoT service monitoring component. The VE history storage stores and retrieves information recorded about a VE.

4. **IoT services and resources**: This group provides the functionalities required by services for processing information and for notifying application software and services about events related to resources and corresponding virtual entities. The IoT service component defines service descriptions. The IoT service resolution component stores and retrieves information about a service and can be used to insert and update service descriptions, retrieve stored descriptions and provide the address of a given service. The resource history storage component provides storage capabilities for
measurement data generated by resources and additional functionalities related to the processing of the stored information.

5. **Device connectivity and communication:** This functional block provides the set of methods and primitives for device connectivity and communication (referring to the possibility for a device to be part of a network, and the possibility for this device to be source or destination of messages, respectively). The communication unification component provides access to IoT devices and is meant to be agnostic to the device technology. The device interoperability component enforces interoperability between devices communicating either directly or through a gateway. The communication reliability component provides uniform interfaces for retrieving data from different devices. The device traceability component provides methods such as access logs, hand-over etc. for improving traceability for devices that can be subject to different availability. The establishment of communication is handled by the communication trigger component which is based on policies, events or schedules.

6. **Management:** This group of functionalities is tasked with efficient management of computational resources. When an application makes use of the different components of the architecture, the QoS manager is tasked with ensuring the consistency of the expressed QoS requirements. The device manager component is responsible for setting a default device configuration and for firmware upgrades. The production rule system component can be used to express and enforce conditions that can trigger pre-defined actions. This can be used to verify the integrity of VEs, services and the platform.

7. **Security:** Security functions have to be consistently applied by the different groups of functionalities. Specifically, access-control policies shall consistently be applied in order to prevent unauthorised applications from obtaining access to sensitive resources. Thus, this functionality group consists of components that provide functionality to decide upon access grants (authorisation component), distribute symmetric keys for M2M communication (key exchange component), provide certificates to bind a VE to IP addresses (certification authority component), user identity or chosen pseudonym assertion (authentication authority component) and maintain device or service reputation (trust authority component).

Since the IoT.est project extends the IoT-A service model, it is worth noting the concept and categorisation of services as identified in IoT-A. An IoT Service, as defined in IoT-A, provides a uniform abstraction and access to Resource functionalities. The interaction with a Physical Entity can be accomplished via one or more services associated with the corresponding Virtual Entity (VE). This association becomes important in the process of look-up and discovery. According to the level of abstraction, IoT Services have been categorised into three groups:

- Resource-level Services expose a device’s functionalities by accessing the software component of the device, i.e. the Resource. These services refer to a single Resource.
- Virtual-Entity level Services are associated to a single VE that can give access to the VE’s attributes either by reading the attribute values or to update the attributes.
- Integrated Services are formed as a result of the composition of the above two kinds of services.

### 2.2 SOA4ALL Architecture

SOA4All [SOA4ALL] establishes a service delivery platform, which domain independent services can be used to solve service specific problems. This platform can be also expanded to expose functionalities such as mobile services or sensors networks to fully enable the ‘Everything as a Service’ paradigm.

SOA4All architecture can be divided into four different parts:

- SOA4All Studio
- Distributed Service Bus
- SOA4All Platform Services
- Business Services
Each of these parts will be further explained in the following subsections. Figure 2 below shows the overall SOA4All platform.

**2.2.1 Distributed Service Bus**
The Distributed Service Bus (DSB) is allocated in the middle of the SOA4All architecture diagram, enabling web-style communication and collaboration among semantic spaces, to increase the scalability of distributed and autonomous services interaction, and service bus technologies, such as distributed service registration. Semantic space is a software developed within SOA4ALL aiming to decouple distributed Semantic Web applications in terms of time, space and reference.

**2.2.2 SOA4All Studio**
The SOA4All Studio is a web-based user front-end consisting of three different platforms:

1. **Provisioning Platform**, that has two main components:
   a. A set of tools to semantically annotate services.
   b. A process editor that allows users to create, share and annotate executable process models based on light-weight process modeling language.
   Annotations are based on WSMO-Lite [WSMO-Lite, 2010], a minimal extension to SA-WSDL [SAWSDL, 2007], while MicroWSMO [MicroWSMO, 2009] is used to annotate services that are not described in WSDL, such as RESTful services [Fielding, 2000] or web APIs. SOA4All provides a minimal service model in RDFS able to capture the semantics for both, services and APIs, allowing them to be treated homogeneously.
2. **Consumption Platform**, the gateway for service consumers, allowing users to formalize formal specifications, known as goals, of different objectives. These objectives are transformed into service templates, which define further constraints on the services and their executions, referring to internal process activities. At runtime, these templates are resolved to specific services.
3. **Analysis Platform**, which collects and processes monitoring events from the service bus, extracting and producing meaningful information displayed to users. These users can choose

![Figure 2. SOA4ALL architecture](image-url)
particular services to be monitored, what other kind of data can be also collected for all other services empowered through the bus.

2.2.3 Platform Services
Platform services provide different functionalities, such as service discovery, ranking and selection, composition and invocation. The service bus offers these consumable services as if they were business services because, although they have different purposes, they operate with service semantic descriptions, service templates and processes. The SOA4All platform services are shown at the bottom of figure 2.

2.2.4 Business Services and Processes
SOA4All enhances business services, so-called semantic web services, in terms of automation, composition and invocation. Their descriptions are published and used for reasoning with functionalities, interfaces, non-functional properties and context data, as they are considered the main enablers of the automation processes related to this kind of services.

Moving to the corners of figure 2, we enter the domain of semantic services descriptions and processes: (1) represents the semantic services descriptions, either in form of annotated RESTful services (3) or WSDL endpoints (4); thus (1) represents the so-called Semantic Web services. In the right top corner of Figure 1, (2) represents processes and mash-ups.

2.2.5 SOA4ALL Summary
SOA4ALL provides the experience in the field of semantic web description, composition of services and also in terms of deployment of services. Platform services could be used in activity 3.2 as an inspiration to find out the minimal composition of service delivery platform and might be used as re-usable components in IoT.est. Based on the experience of SOA4All and other projects we learnt that heavy semantic descriptions are not the right approach due to the difficulty to be learned and filled by an end user. Lightweight service descriptions instead are more adequate. In addition to this, framework components from SOA4ALL can be used as a base for some of the IoT.est components.

The main challenges that IoT.est addresses while in SOA4ALL were not priorities are: the assistance and integration of testing and the fulfilment of the specific needs of the IoT; mainly in terms of the dynamic appearance/disappearance of services, monitoring and adaptation at runtime.

2.3 SENSEI
The SENSEI project [Sensei-Project] aimed at enabling a future (real world aware) Internet through Wireless Sensor and Actuator Networks (WSANs) that provide an infrastructure to enable augmentation of the physical world and interaction with it. The architecture weaves heterogeneous WS&AN and processing resources into a homogeneous real world resource fabric, enabling an open market space for real world-awareness and interactions. The SENSEI architecture, detailed in [SENSEI, 2010] is depicted in Figure 3.
The concept of the SENSEI Resource is central to the SENSEI architecture as it provides a unifying abstraction for simple devices such as sensors, actuators, processors or software components (e.g. fusion algorithms. Sensor-based resources typically provide information in the form of observations or measurements, which include the value, but can also provide some meta-information regarding what type of information is provided, e.g., location or temperature. However SENSEI offers the option to applications to use the SENSEI information model which is centered on the entities of interest (EoI). EoIs are representations of people, places and things. For these entities of interest, relevant aspects like the activity of a person or the current location of a car are modelled as context attributes. Applications can base their requests on entities of interest and context attributes. The SENSEI information model is shown in Figure 4.
A resource is unique within a SENSEI system and is described by an associated Resource description. The SENSEI architecture differentiates between Resources and their interaction endpoints, which are termed Resource End Points (REP). The REP is the software process which represents the physical resource in the SENSEI Resource Layer and implements one or more Resource Access Interfaces (RAIs) to the resource. Thus, REPs can be considered as equivalent to web resources and are uniquely identified by a URI. Also, the resource and its REP may be hosted in different devices. The resource descriptions are hosted in a Resource Directory, which exposes the Resource Publication Interface (RPI). The resource descriptions consist of a number of keywords or free-text tags describing the resource along with the REP locator. The Advanced Resource Description Plugin to the resource directory defines an ontology that describes the resource semantically by its location, resource type (Sensor, Processor, Actuator), and its operations. The semantic operation description defines semantically the operations offered by the resource. For each operation it specifies the inputs that a resource takes in order to provide each output, the pre-conditions and the post-conditions derived from invoking an operation and the temporal availability of the operation. The Entity Directory maintains the associations between the properties of the EoI and the resources supplying information or interaction capabilities related to these EoIs in the Mapping Table. Both the resource and entity directory support simple search functions, e.g. those based on keywords and also lookup functionality based on IDs. For more advanced interactions, the Semantic Query Resolver (SQR) includes advanced resource tasking and service composition functionalities. The SQR analyses the received queries and finds the appropriate resources. If the query requires a number of resources to interact, the SQR constructs a plan involving a sequence of resources. The Execution Manager supports sessions, event notifications and resource adaptation. For this, the Execution Manager can setup and monitor resources. With the functionalities provided by the SQR and Execution Manager, SENSEI users can ask the system for complex queries which involve interpretation, splitting into smaller tasks and execution of the desired operations.

The SENSEI project concentrated on WSANs and how these could enable interactions with the physical world. However, the IoT.est project will consider all types of smart and connected objects that can allow interactions with the physical world. Thus, IoT.est needs to define a much broader concept of services that can offer a standardised interface offering all functionalities to interact with entities in the physical world. Moreover, IoT.est will also include test functionalities in the service descriptions so that testing can be part of both the design and deployment phases of a service lifecycle. The SENSEI architecture can be used as a reference for a resource-based architectural design. A similar abstraction of a ‘resource’ will be adopted in IoT.est. Concepts of semantic organisation and access as identified in SENSEI will also be applied in IoT.est for the service layer design.

2.4 Test Architectures
This section provides a review on some of the existing architectures for tests in the area of software engineering and highlights the relevance to IoT.est.

2.4.1 Modelling of test environments with UML testing profile
The UML 2.0 Testing Profile (U2TP) [OMG] is a graphical notation for testing applications and systems. U2TP provides concepts for designing and developing black-box-tests. In accordance with UML, U2TP is only a language and therefore it only provides a notation but no guidance on how to use it. U2TP extends UML 2.0 with test specific concepts like Test Architecture, Test data, Test behaviour and Test time. It reuses the UML 2.0 syntax and is based on the UML 2.0 meta-model. The test architecture contains test components and/or related classes from which test configurations and test structures may be specified. It provides a basis for defining the types that will realise the behaviour of a test case. It creates the interfaces and connections between the test components and the System Under Test (SUT). Test data refers to the specification of types and values that are received from or sent to the SUT. These data are used during the test execution as a stimulus for the input and for comparing the output of a SUT. Test behaviour addresses the dynamic aspects of a test procedure. It defines test cases which are a complete technical specification of how the SUT should be tested. It also contains the general conditions which are necessary to review and validate a test execution. These conditions are: necessary pre-conditions before a test run, quantity of input values, expected output values and necessary post-conditions after a test-run. The test time is defined to
specify time-quantified test procedures, such as time limits and restrictions and also time analysis of the duration of test steps during test execution. To realise and utilise U2TP as a test framework, the abstract U2TP notation has to be transformed into a test specific programming language like TTCN-3 [ETSI] or JUnit [JUnit].

2.4.2 Test automation with TTCN-3

Testing and Test Control Notation Version 3 (TTCN-3) [ETSI] is an international description language for automated testing of communication based systems. The standards were released by the European Telecommunications Standards Institute (ETSI). The core language is described in text format and there are also graphical representation formats that are defined for easier management. These include the graphical format (GFT) and the tabular format (TFT). The main advantages of TTCN-3 are its extensibility with external data definitions, integration of external functionalities and the adjustment to the systems under test (SUT) over so called adaptors. TTCN-3 is used to specify the tests and the order in which they will be executed. To use TTCN-3 a TTCN-3-tool, or a TTCN-3 compiler (or interpreter), with corresponding runtime environment is needed. Due to the abstract description language the test-cases are described independently from the runtime environment. This leads to simplicity and reusability in describing and running the tests. For the tests SUT is regarded as a black-box, where only the interfaces to the system are considered. Stimuli to the system are given and the reactions in the interfaces are evaluated. Test-cases contain the functions to be validated and can be merged to a test-suite [Grabowski'03]. A drawback in TTCN-3 is a lack of simplicity in the generation of tests in a service creation environment. It requires users to learn a new and complex language in order to create test-cases. Extensibility and validation of complex services derived from single components is also a challenging task in TTCN-3 [Hierons'06]. In overall TTCN-3 does not consider (automated-) test-friendly development and composition of services.

2.4.3 Software-tests with JUnit

JUnit [JUnit] is an open source framework to enable the execution of unit tests on Java-classes. JUnit has become a standard for automated testing in java development. JUnit enables definition of test-cases, executing them independent of each other, and collecting the defined tests in test-suites. The test-suites allow to group tests and to create test reports. JUnit test-cases are similar to test codes written in Java programming language and are used to test objects and their behaviour in unit-tests or module-tests [Tan'04]. JUnit has emerged from the agile test-driven software development and therefore follows the test-first approach. However it does not provide an acceptance-test or system-test where the customer validates and evaluates the software. The main advantage of JUnit is that tests can be developed before (or when) programming starts using the same programming language. The programmer does not have to learn a new language to describe the tests. JUnit allows to get instant feedback to a modification of a class and to determine if the changes affect other parts of the system [Do'06]. However JUnit has limitations for complex systems as extensive dependencies evolve, e.g. in software architectures like Java EE or JSLEE. Developed classes depend on other components and services, which depend on data-objects or protocol-messages. This leads to dependencies which appear only at run-time and under different environment-parameters. In theory classes should be tested without testing their dependencies. To reach this objective, dummy-objects, so called mock-objects, are introduced. Mock-objects emulate and simulate the behaviour of the depending components.

2.4.4 JMeter

JMeter [Apache'10] is an open source software from the Apache Software Foundation for load testing, functional behaviour testing and to measure performance. It offers a graphical user interface to define tests on a variety of static and dynamic resources such as files, servlets, Perl scripts, Java objects, databases, FTP servers and Web Services. The server types supported include Web (HTTP and HTTPS), SOAP, JDBC databases, LDAP, JMS and POP3 and IMAP-enabled mail. The HTTP methods supported include GET, PUT, POST, DELETE and OPTIONS. Hence, alongwith the SOAP method support, both RESTful and standard Web Services can be tested. The test is defined in terms of samplers which invoke the server interfaces. Test results can be visualised in real-time via a number of listeners that measure and cache performance metrics. JMeter has a multithreading
framework that allows concurrent sampling by many threads and simultaneous sampling of different functions by separate thread groups.

In overall, current test architectures do not consider the (automated-) test-friendly development of services for the IoT. For a simple and fast development of services, a service developer needs more effective and rapid development test methods. Test cases should be generated automatically by the service creation environment using the description of the IoT services. The test framework should also be able to verify domain parameters and variables while testing the services. The IoT.est Architecture will extend and utilise the current solutions and will focus on IoT services to analyse and propose solutions employing automated tests in IoT service creation and deployment. It will integrate the test functionalities into the IoT.est SCE and run-time platform to construct and provide reliable and adaptable IoT services.
3. The IoT.est Architecture
This chapter explains the IoT.est reference architecture which represents a semantic service and test extension of the IoT architecture. The reference architecture is designed based on the core functionalities derived from the analysis on use cases and scenarios [IoT.est-D2.1, 2012] and is presented in Section 3.1. The abstract architecture provides guidance to develop compliant IoT based service systems that potentially have domain or application-specific requirements. Section 3.2 gives the functional view of the architecture which is specific to the IoT.est project. The components inside each functional block in the architecture are elaborated. Section 3.3 explains the interfaces defined for the architecture and Section 3.4 shows the interaction among the functional blocks.

3.1 Overview of the Architecture
Rapid development of the IoT creates potentially new business opportunities for many application domains, especially those need to interact with the physical and their ambient environments. It also has a significant impact on the ways that future business values are created and delivered. The IoT.est project aims to build a dynamic service creation and composition environment which integrates the business world with the physical world and to contribute to the creation of new business values.

Designing the architecture for the IoT.est project needs to take the heterogeneity of the current IoT infrastructures and the gap between low-level IoT services and high-level business services (e.g., the legacy services) into consideration. For this reason, the abstract architecture is derived from the use cases and requirement analysis as presented in the IoT.est project Deliverable 2.1 [IoT.est-D2.1, 2012].
Figure 5 shows high-level abstract architecture for IoT service creation and testing. The purposes of the six major components in the architecture are summarised as follows:

**Knowledge management**: this component is responsible for registration, storage, search and query of the (IoT based) service descriptions as well as some management tasks. IoT.est services will be described and annotated according to the semantic description ontology and knowledge base which can be external to the project (e.g., reusing of the existing ontologies and knowledge base). The semantic service descriptions are stored in distributed service registries. The component also contains a search and query component that is able to locate appropriate services in response to service lookup and discovery queries. It is anticipated that an extraordinarily large number of services will be exposed in the future and these capability constrained services are usually unreliable and often operated in highly dynamic environments, more effective and efficient service discovery and lookup techniques are needed. The component also contains management functionalities related to the service lifecycle and roles.

**Service composition environment**: this component allows end users to compose IoT based services in a semi-automated way based on the business goals. A service design graphical user interface will be implemented to facilitate the service creation and composition as well as handling of interactions among different components inside the service composition environment.

**Service runtime environment**: Service runtime environment enables provisioning of IoT enabled business processes. It is related to the deployment and execution phases of the service life-cycle. The runtime environment monitors context and network parameter (e.g. QoS) and initiates automated service adaptation in order to fulfill service level agreements.

**Test Component**: The derivation and execution of tests of semantically described IoT.est services is managed by the Test Component. The test derivation is triggered by the Service Composition Environment. It fetches the service description from the Registry and search/query engine where it also stores information about its test results. It handles the testing of the IoT.est service in a controllable instance of the Service Runtime (Sandbox Instance) and emulates the external Resources.

**IoT.est services**: This component represents the collection of the IoT services and reusable service components developed in the project. Since IoT.est’s target is to allow IoT specific services to be described, annotated and bounded in a uniform manner, the term service is generic and not linked to any fixed options regarding the protocol, input/ output format or any other specific SLA details.

**External Resources**: The external resources are those not designed and developed within the IoT.est project. The resources can be services which can be discovered and used with IoT based services for service composition.

### 3.2 Functional View of Architecture

Based on the high-level reference architecture defined in Section 3.1, this section provides details on each functional component that needs to be implemented in the IoT.est project. Figure 6 shows the overview of the IoT.est architecture: the major functions that need to be provided in each component, their interfaces and interactions are identified. The following sub-sections present design details and identify fundamental requirements for each individual functional component. The interfaces and the interactions among functional components are elaborated in Section 3.4.
3.2.1 Knowledge Management

Knowledge management is the most fundamental component in the IoT.est architecture and provides the basic functionalities (e.g., service registration, search and query, and service lifecycle and role management) used by other components such as the Service Composition Environment, Service Runtime, Test Components and IoT.est Services. The specific architecture for this component is illustrated in Figure 7.

![Figure 6. Overview of the IoT.est architecture](image-url)
There are a number of functional modules to be implemented, namely, Service Registration, Service Discovery, Service Update, Service Removal, Service Lookup, IoT.est Description Ontology, Distributed Service Description Repository, Service Lifecycle Management and Role Management.

### 3.2.1.1 Service Registration
The Service Registration module receives a service registration request and stores the service description in a service description repository. Due to the potentially large number of IoT services and diverse service types, the module might need to make decisions on how to select an appropriate service description repository and where to store the service description in a distributed environment. A service registration client (in the service composition environment for example) might be able to proactively register the service descriptions. The registration interface is explained in Section 3.3.1.

### 3.2.1.2 Service Lookup
The Service Lookup module takes as input one or more service specific parameters and returns the information of the exact service that a user looks for. It serves as the basis for the other two operations: Service Update and Removal. The major difference between service lookup and service discovery is that the former aims to locate the exact service which matches the user requests (or none if no match found), while the latter aims to retrieve one or more matched services, usually in a ranked list.

### 3.2.1.3 Service Update
The Service Update module is responsible for the update of service descriptions stored in the service description repository. Service descriptions usually include dynamic information related to the non-functional properties (e.g., QoS) and test results of the services. Once the up-to-date information is obtained, the service descriptions in the repository must be updated accordingly. The service update

---

**Figure 7. Architecture of the knowledge management component**
process might be realised in an automated way, for example, an event detected by the service monitoring is able to send an update request to the module.

3.2.1.4 Service Removal
The Service Removal module deletes the description entry for a service if needed. In case a known service becomes (permanently) unavailable, authorised users should be able to remove its entry from the service registry. A service removal operation can also be performed by the service registry if the description has been in the registry longer than the valid period.

3.2.1.5 Service Discovery
The service discovery aims to retrieve services (including IoT based services and ordinary Web services) according to the criteria specified by users. This component consists of two major subcomponents: service search and service recommendation.

Service Search
The search is performed based on the service descriptions stored in the registry. This procedure can be generally realised using two strategies: logic based search (e.g., using the SPARQL language [SPARQL]) and text based search. It is expected that limited amount of textual information will be generated in the domain of IoT where devices such as sensors can automatically register and connect to the networks; therefore, the search procedure is primarily based on the logic based approach. How to enable users to generate expressive and effective logical queries is one of the key factors for design the service discovery component.

Service Recommendation
The service search functionality enables users to retrieve a list of services according to users' needs. However, in most cases, the system should be able to find the most appropriate service instances in the context of the applications, for example, service composition. This requires the service discovery component to implement an effective service recommendation function based on many factors such as the users' needs and tasks, application context, and possibly the trustworthiness of the underlying IoT resources.

3.2.1.6 IoT.est Description Ontology
The IoT.est description ontology is designed as the semantic description framework for the project. It consists of several modules which are responsible for describing different aspects of the project domain, for example, resources, test and services. More information on the description ontology is available in IoT.est Deliverable 3.1.

3.2.1.7 Distributed Service Description Repository
Design of the IoT service description repository should be distributed due to the distributed nature of the IoT. Service discovery mechanisms used in the service repository should hide the complexity of distributed query processing from the users or applications; and the interface of the repository is the only entry point to retrieve service descriptions. The main challenge here is to develop efficient distributed search mechanisms to meet the users and applications' requirements.

3.2.1.8 Domain Knowledge Base
The IoT.est services and resources will be linked to the concepts in the domain knowledge base according to the description ontology. Besides the knowledge base developed in the IoT.est project, other existing sources of the domain knowledge are considered, for example, the Geonames ontology [Geonames] which provides geographical information about locations and the linked open data [Linked-Data] on the semantic Web which provide general knowledge in many domains.

3.2.1.9 Lifecycle Management
Lifecycle Management provides means to the other modules to update all the information related to the lifecycle of the services. This module allows the update of the status of the service as well as provides further information that goes with the status. For example, a service that has passed the validation tests, consequently can be certified; the change of the lifecycle status of the service includes other information that proves that the service has passed all the tests.
3.2.1.10 Role Management
Role Management provides the typical operations that allow to create, update, delete or modify roles and to associate and describe the functionality that the user role is allowed to access. Depending on the user role, the user will be able to access certain functionalities. For example, user A who is an integration tester will be able to certify the tests, however user B who is a software developer will not be allowed.

3.2.2 Service Composition Environment
The service composition environment allows the end user to drive the whole lifecycle of a composed service. Concretely this module will focus on the composition of IoT services in an (semi-) automated and efficient way based on the business process goal in an application domain. The following diagram shows the architecture of the service composition environment and the main interaction with the other modules in the architecture.

The service composition environment (See Figure 8) consists of the following components:

3.2.2.1 Designer GUI
The Designer is the Graphic User Interface and handles the interaction between the Service Composition Environment with the final user. The access to the designer will be a role based access adapting the content and allowed functionalities to the role. Due to the specific issues of IoT environment encoded in the ontology model it’s considered that specific consistency checks are covered. Therefore, the Designer GUI will offer a meaningful way to design, test and afterwards execute “doable” services, providing feedback to the designer when a certain condition is not met (usually the ones from the SLA). The Designer GUI is composed of two modules:

- Editor, that using a graphical representation, provides the means for specifying business processes in a business process model. It covers most standard operations to build workflows and to edit data flows. It will use standard languages and notation to describe the resultant workflow.
- Plugins: extends the usual functionalities of editor in order to provide means to:
  o describe IoT services according to the service description ontology
  o allow end user to use ontologies to describe their objectives. That is, assist the user in providing information about IOPE (Input, Output, Preconditions and effects) and functional capabilities of the goal that the user want to achieve
  o display the result of the current reusable services that can be used to fulfill the user’s goal
  o select predefined templates
  o extended palette for IoT specific functions such as dynamic invocation of services
  o browse domain knowledge catalogue
  o trigger the derivation of test cases
  o trigger the execution of test cases
  o trigger the deployment of the composed service
  o manage the artifacts for deployment

3.2.2.2 Governance
The Governance component manages the access to system functions taking into account the end user role and will handle the lifecycle of the IoT.est services for the software development phases. For example, a business analyst looking for creating a service to fulfill a business need will be looking for a simple set of steps that will rapidly achieve his objective. On the other hand, for the developer, the Governance component would be responsible to set the GUI view showing the components that need maintenance, completion or creation if not available.

3.2.2.3 Artefact Manager
The Artifact Manager handles the additional information that needs to be provided for the execution of the IoT.est composed services. Examples of artifacts are those related to services, such as rules and process definitions, service descriptions, configuration files, databases schemas, etc.

While the Knowledge Management Client and the Register and Search/Query Engine will search and reason for the best service available to achieve a certain goal, the Artifact Manager provides services for querying, uploading, exploring, and managing Artifacts.

Managed artifacts are checked for sanity against targeted environment inside the designer in order to keep Knowledge Management content on synch against whole life cycle phases: design, test generation, test execution and analysis.

3.2.2.4 Design Time Composer
The design-time composer service provides semi-automatic assistance in resolving unbound activities within a process specification. As such, within IoT.est, the module is mainly supporting the activities of the GUI component and others that are part of the consumption platform of the SCE. In other words, the design-time composer supports the entire life-cycle of service composition, from supporting the process specification through elaboration of process and template expansions to the discovery and binding of service endpoints as activities within the processes. Side issues that are tackled by the composer service are data mediation and resolution of compatibility problems at the level of service inputs and outputs via the so-called service connectors and semantic link operators and the validation of the workflow. The Design time Composer is responsible for the validation of the workflow at the beginning of the deployment.

3.2.2.5 Workflow Manager
This component handles the translation or the extension and adaptation from the specific representation language in which the business process is described with a language that is executable. There are business processes notations that can be executed (example: BPMN 2.0), but others need to be translated to a specific execution language (ex.: BPEL). In the first case, this module will adapt and extend the produced description in order to be executed successfully. In the second case, this module will handle the translation from the specification language to the execution language. In any case this module will request the design time composer assistance in the validation of the current workflow before starting the transformation process.
The Workflow Manager shares with the Data Binding Manager the capability to check against Knowledge Management and known artifacts and the environment limits the capabilities of generated workflows. This step will bring a reality check in the Test generation phase for a better tailored and realistic test generation.

### 3.2.2.6 Data Binding Manager

This module is in charge of completing the data flows (data mapping) and the technical details of invocations to the endpoints using the semantic description (grounding) and artifacts manager in order to provide all the necessary information to the Service Runtime Deployment. It will handle the data flows of the tasks described and is responsible for the data binding with the endpoints and to adapt the workflow to the specific technology of the endpoint (for example REST or WS).

Additionally, this component is responsible for the verification that the service configuration has all its dependencies satisfied through the interaction with the Artifact Manager.

### 3.2.2.7 Knowledge Management Client

The Knowledge Management Client is the component in charge of dealing with the communication with the Knowledge Management module and providing an abstraction layer to work with semantic search using reasoning support in matching services. This module will translate the high level of abstraction requests from the other components to the specific invocations that needs to be executed against the Knowledge Management module. At the same time, it will handle the communication with the Lifecycle Management, Role Management and Knowledge Base modules.

### 3.2.3 Test Component

The Test Component is responsible for the derivation and execution of tests of semantically described IoT.est Services. Figure 9 shows the interaction with the main IoT.est architecture components together with the identified internal functional test components. The test derivation is triggered by the Service Composition Environment and the resulting tests are stored into the service Registry (in the knowledge management component). The Test Design Engine derives the test in an (semi-) automated manner from the semantic description of the IoT.est Services and the knowledge from the service developer can be utilised to modify the created tests. Based on the tests that are selected by the service developer, the Test Execution Engine handles the testing of the SUT in a controllable Service Runtime (Sandbox Instance). Test results are linked to the semantic description of the SUT.
3.2.3.1 Test Design Engine

The Test Design Engine (TDE) is responsible to generate test cases for new or changed services and preparing their execution. The TDE is trigged via the Test Component Interface by informing about a changed or new developed Atomic Service (AS) or Composed Service (CS) by sending the location of a semantic description. In the first phase the Codec Discovery and Creation will identify which protocols are utilised by the service interfaces and, if required, create a new codec in order to abstract the protocol flow. After the generation of a SUT model defining the internal behaviour of the service, based on an extended finite state machine single test cases for functional and non-functional tests are created. Information for the generation of test cases will be derived out of the semantic description of the service that is stored in the Registrar and Search/Query Engine. The order of test case execution is defined as a test flow. The test cases are described with a standardised test language. During creation the test cases will be enriched with generated test data based on the Input, Output, Precondition and Effect (IOPE) of the semantic service description. The Test Case and flow creation produce executable code from the test cases and assure a native test case execution by sharing the test case executable code inside the Test Component with the Test Execution Engine (TEE). A test case and flow description is published to the Registrar and Search/Query Engine. In case of a CS it is mandatory to rerun the test derivation process on every change of related atomic or composed services that could be utilised during the run-time binding facilitated by the query engine. The test process has to be manually initiated by the developer due to allocation of sandbox resources and human test interaction in order to assure that any changes or new realisations of services which might be selected at run-time are detected and result in a proper re-testing procedure.

Codec Discovery and Creation

In order to test services numerous codecs for an abstracted communication with the SUT’s interfaces have to be maintained whereby there’s a high reusability of the prepared codecs. The Codec Discovery and Creation component enables the TDE to obtain a suitable codec library that can be used for test case and flow execution. It fetches the protocol relevant semantic service description of a service and analyses the required codec capabilities. Existent codecs can be used out of a Codec Pool whereby unavailable codecs have to be (semi-) automatically created. Depending on the thoroughness of the semantic description a varying amount of developer interaction is needed.

Test Case and Flow Creation

Utilising the Codec Pool and a SUT model derived by the TDE the Test Case and Flow Creation generates particular test cases to ensure test coverage of service states defined in the semantic service description. It covers functional tests (e.g. data types, value ranges and quality of result information and correct results) as well as non-functional tests (e.g. QoS and robustness). Due to the infinite number of test data based on the protocol and interface information, besides a semantic description of relevant system parameters/protocol values, a Test Data Generation is used to improve test data based on its pertinence.

A variety of test cases will be assembled to test flows in order to provide best possible coverage of service’s Extended Finite State Machine (EFSM) based on statement-, branch-, path- and condition coverage. The initial description of the generated test cases flow is written to the Registrar and Search/Query Engine so a test developer is able to manually adapt test flows and cases. Executable test cases and flows are stored in the Test Component so the Test Execution Engine is able to access them.

Test Data Generator

The Test Data Generation utilises the semantic service description, information regarding different i.e. physical value types as well as information to derive the SUT’s EFSM. It uses algorithms for test data reduction and optimisation regarding statement-, branch-, path- and condition coverage. The Test Data Generation also utilises a Test Oracle to use developer input for proper identification of the functional test results.

Test Case Compiler
Utilising a standardised testing language (e.g. TTCN3) suitable for service testing the Test Case Compiler creates machine executable or interpretable code. It is invoked by the Test Case and Flow Creation during the test case creation. Compiled test cases will be used by the Test Execution Engine for test case execution.

3.2.3.2 Test Execution Engine
The Test Execution Engine (TEE) is the central component to coordinate the test flow and the related test case execution. The execution is triggered from the Service Composition Environment via the Test Component Interface. While the Test Case Execution Engine takes care of the test execution it access emulation components. These emulation components enable to execute the SUT under controlled and realistic conditions. This includes also the emulation of the Registrar and Search/Query Engine (in case of composite services) in order to control the runtime service selection. The SUT will be isolated from all resources which are located outside of the sandbox. This prevents dependencies and enables a larger range of test situations.

Test Case Execution Engine
The Test Case Execution Engine (TCEE) is responsible to run Test Cases. Therefore, the TCEE sets up and controls the emulation components, interacts with the SUT and triggers the test logging process. Codecs, developed during the test derivation phase, are discovered within the Codec Pool and ensures that the TCEE can encrypt the data stream from the SUT service interface.

Codec Pool
Data streams from and to the SUT need to be coded and decoded during the test execution. Based on the semantic description of the SUT required codecs are discovered or created during the test derivation. While the test cases are executed the provided codecs abstracts the data stream and enable a generic protocol and value validation of the SUT responses. The Codec Pool enables the utilisation of the created codecs.

Service Interface Emulation
Composite Services typically consists of several IoT.est Services which are not controllable from the service developer during the test execution (different service provider infrastructure). Therefore, all external involved IoT.est Services need to be emulated (interface emulation). The Service Interface Emulation component is responsible to create the consequential service interface testing stubs.

Service Search and Query Engine Emulation
Due to runtime (re-) selection of services from the Composite Service, the test environment needs to make sure that only controllable services and service interface stubs are selected during the test execution. The Service Search and Query Engine Emulation will mediate every service search request to an appropriate emulated service interface.

External Resource Emulation
The functional and non-functional behaviour of IoT.est services are dependent on the interaction with external resources (Web or IoT). External resources usually are not controllable and, in case of IoT resources, can affect the real world. In addition, in order to test the majority of possible service states it is required to inject specific input values to the SUT (thresholds, boundaries, etc.). IoT.est services can therefore be stimulated with emulated IoT Resources inputs. To cope with the variety of different heterogeneous and often proprietary protocols used for IoT Resources the resource emulation interface is designed as a generic interface which already abstracts the communication layer and enables injection at a payload level.

Network Emulation
Network characteristics can have major influences on the behaviour of IoT services. The network emulation component assures that the robustness of the SUT can be tested against hash network conditions.

Runtime Emulation
Resources of the service runtime have a critical impact on the non-functional behaviour of services. The Sandbox resources (Load, RAM, etc.) need to be adjusted to reflect the productive service execution environment. The runtime emulation controls the resource allocation and enables to verify if some conditions or combinations of conditions can result in a SLA violation.

Test Logging
The logging component takes care of the test results and stores the test history in a database. The test results are linked to the semantic description of the SUT and can be accessed by authorised users.

3.2.4 Service Runtime
Service runtime will allow the provision of IoT enabled business processes. It gives support to the deployment and execution phases of the service life-cycle. Service runtime allows monitoring context and network parameter (e.g. QoS) modifications, which can trigger the need of service adaptation in order to guarantee the fulfilment of the SLA. The Service runtime architecture is presented in Figure 10.

![Service Runtime Architecture](image)

3.2.4.1 Message BUS
The Message BUS is an infrastructure that enables a high level of interoperability between IoT.est services in a scalable and flexible way, facilitating the SOA realisation. It allows spanning the IoT.est Service Composition over heterogeneous systems making the bridge between IoT.est service consumers and IoT.est service providers. The main task of a message BUS is therefore to provide connectivity between multiple IoT.est services probably owned by different stakeholders. It executes an intelligent routing of messages between IoT.est services and, when needed, performs protocol transformation in order to ensure their compatibility. Moreover, it is responsible for adapting the data formats, guaranteeing a comprehensive data exchange between IoT.est service consumers and providers. Additionally, the message BUS ensures a secure communication by restricting the communication to authorised IoT.est services.
3.2.4.2 Management

IoT.est service runtime encompasses a set of management functionalities which are employed to enforce a consistent and harmonious governance of IoT.est services life-cycle. The IoT.est management is the central point for monitoring within the distributed system assuring the SLA accomplishment. The runtime monitoring mechanisms enable service adaptation to environment changes and modification of network parameters, which can lead to the reselection of involved components.

3.2.4.3 Test Emulation Components

The Network and Runtime Emulation Components are controlled by the Test Execution Engine. They enable to test the influence of the network and runtime behaviour to the quality of services as well as their robustness. The Network Emulation Component will encapsulate the SUT and ensures that each communication with the service interfaces can be adapted (bandwidth, delay, packet loss, etc.) according to the current executed test case. The Runtime Emulation Component will be responsible to adjust the available resources of the runtime environment.

3.2.4.4 Semantic Query

The Semantic Query component represents an intermediary between the service runtime and the service search and query engine. It forwards service search and query requests to the service search engine for processing. Details of the semantic search and query engine can be found in Section 3.2.1.

3.2.5 IoT.est Services

Current Internet of Things services platforms take in account two possible approaches: one based on extended heavy weight Web Services (SOAP/WSDL) and one using the capabilities exposed by REST architectural style. Both development approaches have been considered from both functional and non-functional criteria point of view. The majority of considered aspects stress specific subjects like resource-constrained devices, reliability of data or naming and addressing strategies. From this point of view, the REST approach is considered more appropriate due to low footprint, simplicity of the messaging style, openness to common Web technologies and easiness for developers. As a consequence most of the current standardisation is made around REST interfacing model.

At the same time from integration point of view, the IoT.est addresses both models of services together with semantic annotation capabilities. Still, it is necessary to make the difference between the level of exposure IoT where REST is the best fit, against the business process capability, when WS* allows more complex models. Those aspects should be reflected both on Service Design and Runtime where an alignment and correct abstraction of potential mix of WS* and REST services might happen in real world scenarios.

Based on the large stream of current approaches and standardisation efforts IoT.est will consider the interfacing with the IoT world based on REST. Those services receive the considered level of semantic description allowing both business process generation and testing capability as described. Where business services will need to be addressed as part of a process and IoT services are involved (including test), a WS* interface will be considered as accessible from design time and run time. Still, due to specific focus of IoT.est project, it is expected to provide best available connectivity in the IoT by REST.

3.3 Interfaces

This section describes the interfaces of the different components in the IoT.est architecture as well as their interactions among those components.

3.3.1 Knowledge management interfaces

The Knowledge Management interfaces cover the following:

- Lifecycle Management interface: handles request for updating the lifecycle status of a service, as well as adding further information that complements the status (i.e., details about facts that explain the reason because a service has been test certified).
- Role Management interface: handles request for adding/deleting or modifying the functionalities assigned to a role, as well as Create Remove Update Delete (CRUD) requests over the role itself.
- Domain Knowledge Interface: provides means to other modules to access background information. One of the sources of the background information can be a product catalogue that can group sets of services that are related with a product and describes the product.
- The Service Registry and Search interfaces include three interfaces for other components to use: service registration interface, service search interface and service query interface. These interfaces are used by the Service Composition Environment, Service Runtime and IoT.est Services components. The registration interface accepts a service registration (for new services that have not registered with the registry) request and stores the service descriptions in the distributed service description repository. The service search interface is an intermediary between service discovery requests and the service discovery component which performs service search, recommendation and ranking based on the semantic service descriptions. The service query interface accepts requests on service lookup/update/remove and forwards them to the respective processing components. The semantic descriptions of the services in the registry will be updated according to the requests.

This interface can be accessed by any of the components of the system, however within the project SCE and the Service Runtime will be the main users of these interfaces.

3.3.2 Runtime interface
This interface is used by the Service Composition Environment to automatically deploy the service package in the run-time environment. Different run-time environments should support this interface including pre-production run-time environments and production run-time environments.

3.3.3 Test control interface
The Test Control Interface enables the execution of the service runtime as a test instance (Sandbox). The Test Case Execution Engine utilises this interface to set up and control the service runtime. It triggers the test deployment process, handles bindings and resource allocations specified for the current test case.

3.3.4 Network emulation interface
The Network Emulation component controls the SUT network connection via the Network Emulation Interface. This interface receives messages to change the network connection behaviour (bandwidth, delay, packet loss, etc.).

3.3.5 Resource Emulation Interface
The Resource Emulation Interface is located within the SUT. Messages from the Resource Emulation component are utilised to inject data transmissions from emulated resources into the SUT.

3.3.6 Test Component Interface
The Test Component interface enables an external initiation of test generation and test execution as well as the adaptation of test cases and test flows. It enables control of the TDE and TEE and reports the current status of the derivation/execution process but does not provide the exchange of semantic descriptions or detailed test results.

3.3.7 Service interface
Due to the fact that the service interface is not specific to any standards, the Service Interface will be responsible to translate to grounding specific data as they are produced in common format accepted in Knowledge Management component and checked for validity in Design Studio.

3.4 Dynamic View of Architecture
This section presents a dynamic view of the IoT.est components with activity/sequence diagrams that show interaction between the different functional blocks of the architecture.
3.4.1 Service Composition Environment

A dynamic view of the SCE is provided showing how processes operate with one another and in what order using UML sequence diagrams. This is only a subset of the possible representations that are the most representative.

Before any other interaction takes place, the user needs to pass the authentication and authorisation process as shown in Figure 11 below:

![Figure 11. Login sequence diagram](image)

The authentication process validates that the user is able to enter into the system and the authorisation process determines which functionalities will be available for the user.

3.4.1.1 Design Phase

After successful login, authorised users will be able to design a new service based on the composition of the IoT.est services available.

In order to do so, the end user can access the designer interface. In addition to the Editor, the end user will find plugins that extend the functionalities of the editor. One of those plugins will allow the selection of predefined templates to facilitate the design process. For each task defined within a workflow, the user can define semantically the goals that they would like to achieve. To facilitate the process, the graphical user interface through a semantic search wizard (plugin) will provide the means to select different ontologies. Then using those ontologies, the end user can describe semantically inputs, outputs, functional capabilities and non-functional capabilities that describe the user goal for a given task. The Design Time Composer will drive the semantic search performed by the Registrar and Search/Query Engine and will provide to the GUI the set of feasible IoT.est services that in collaboration or individually fulfil the user goal. This process can be executed for each of the tasks within the workflow. The user can also add all the necessary artefacts that will be needed for successful deployment. Additionally users will be able, through the GUI, to annotate semantically the composed service that is being created. Once this process is finished, the user can mark the service as ready to be tested.

The following sequence diagram (see Figure 12) illustrates all the interactions involved in the usual design scenario.
Figure 12. Design phase sequence diagram
3.4.1.2 Testing Phase

The testing phase can be divided into five sub-phases. The Service Composition Environment starts the first testing phase by preparing the SUT (registration, life cycle management, etc.). Afterwards, the test derivation phase handles the (semi-) automated test creation. Then, the test flows and the test cases can be modified and selected for the test execution by the Service Composition Environment. Subsequently, when the test results are verified, the end user can certify the SUT, if the use has the rights to perform this action. The flows of the five testing phases are explained below:

1) Service test preparation
After successful login, authorised users will be able to initiate the test phase of IoT services registered in the system, with special focus on new or modified composed services. When the composed service is ready for testing, authorised users will handle the testing phase. First step is to update the lifecycle of the composed service. Figure 13 shows the interaction flow of this testing phase.

![Figure 13. Service Composition Environment Test Interaction](image)

2) Test derivation
The detailed test derivation interaction is shown in Figure 14. The derivation process is triggered by the Service Composition Environment and controlled by the Test Design Engine. The semantic description of the service is provided by the Registrar and Search/Query Engine. After the generation of Codecs and a SUT the test case creation is supported by a Test Data Generation to find a moderate number of test cases for sufficient test coverage. The definition of the test cases is used to compile machine executable/interpretable test cases. An initial test flow describes a succession of the test cases and publishes the descriptions to the Registrar and Search/Query Engine that can be adapted by the user. Compiled test cases are stored in the test component.
3) Test adaptation
As shown in Figure 15, the Service Composition Environment will be able to access, modify and select test cases and test flows for the test execution. Knowledge from the service developer can be utilised to improve the created tests (e.g. include knowledge about expected return values, boundary changes, change test case order, etc.).
4) Test execution

Figure 16 (Detailed Test Execution Interaction) shows the interaction flow of the test execution. Authorised users will be able to access the GUI of the SCE and start the test execution process. The process internal to the SCE is identical to the one described at the deployment preparation phase (3.4.1.3) and will be detailed in that section. The main difference resides in specific particularities of each environment. At a high level of abstraction the SCE at the beginning of the test executions checks that the workflow is valid, translates the workflow to the execution language (including binding and data mapping), and allows the user to include the necessary artifacts and metadata to achieve the deployment. Then, the Test Execution Engine requests the test cases and flow information from the Search and Query Registry Interface. Afterwards, the test cases are executed as specified in the test flow. For each test cases execution, the required codecs are selected and the emulation components are initialised. The Test Case Execution Engine will then run the test case by interacting with the SUT and simultaneously control the emulation components. Test results are logged and linked to the semantic description of the SUT.
5) Test certification

After the execution is completed the GUI can show a resume of the results of the tests. At any time the end user can initiate again the process. If the end user is satisfied with the result of the execution of the tests, then testing phase can be certified, if the user role has the rights to perform this action.

Figure 17 shows the flow of the test certification.
### 3.4.1.3 Deployment Phase

**Deployment preparation**

After successful login, authorised users will be able to initiate the deployment preparation phase of certified IoT.est services registered into the system as described in Figure 18(a). In order to do so, end users will need to update the artefacts and recommendations about service deployment metadata. These Recommendations are used in the next steps to semi-automate the definition of Deployment Metadata according to service design metadata and business policies previously defined.

After the deployment and operation metadata is set the deployment package is generated and adjusted by the user according to selected runtime environment. Consequently the Workflow Manager will perform the validation of the current workflow and will start the translation from the representation language to the execution language. Once this process is finished, the data binding process will be initiated. During the data binding process, the data mapping and the actual grounding from the semantic description to the specific interfaces will be executed. The objective is to generate a complete workflow that can be executed in the supported available workflow engines, apart from further configuration that will be completed at deployment phase.

If the service package is successfully generated the workflow is ready for Quality Assurance test to be performed in the service runtime sandbox. Consequently the lifecycle status is updated according to IoT.est service lifecycle model.

---

**Figure 18(a). Deployment phase sequence diagram**

**Deployment Authorisation**

After a workflow is marked as ready for deployment, authorised users will be able to execute the deployment that is described in the figure 18(b). Authorized end users can modify the deployment metadata and review the current information. After the end user validation of the deployment the necessary artefacts including the workflow will be notified to the Service Runtime. The specific aspects in the runtime will be addressed in the next section (3.4.2 Runtime).
3.4.2 Runtime
Some of the most relevant IoT.est Runtime use case behaviour is shortly described in the following sections by using UML Message Sequence Charts.

3.4.2.1 Deployment Preparation
Before the service is deployed in the service runtime, the service package with all artefacts needed by the Service Runtime must be generated including the service binding.

The main message flows are (see Figure 18):

Steps 1 to 4: The user with Deployment Manager permissions asks the SCE to list services that, according to IoT.est service lifecycle model, are ready to deploy status. The SCE returns this list from the Registry after role-based authorisation is performed.

Steps 5 to 8: The user selects one service from the list that should trigger the Registry to perform recommendations about service deployment metadata. These Recommendations are used in the next steps to semi-automate the definition of Deployment Metadata according to service design metadata and business policies previously defined.

Steps 9 to 12: The user obtains the service deployment and operation metadata that is saved in the Registry. The user should use as much as possible recommendations previously provided by the Registry.

Step 13: After the deployment and operation metadata is set the deployment package is generated and adjusted by the user according to selected runtime environment.

Steps 14 to 15: As soon as the service package is successfully generated the service is ready for Quality Assurance tests to be performed in the service runtime sandbox. The service status is set in the Registry according to IoT.est service lifecycle model.
3.4.2.2 Configuration

As soon as the Service is deployed in the runtime, and before it is activated, the service configuration must be performed.

The Main Message flows are (see Figure 19):

Steps 1 to 4: The user with Deployment Manager permissions asks the Runtime Governance to list services that, according to IoT.est service lifecycle model, are ready to be configured. The Runtime Governance returns this list from the Registry after role-based authorisation is performed.

Steps 5 to 6: the service configuration information is validated and eventually completed by the User according to runtime environment characteristics and deployment Metadata retrieved from the Registry.

Step 7 to 9: the service is configured by running all configuration processes defined in the service metadata. Internal service configuration uses Service Management Interface primitives (e.g. monitoring configuration by subscribing to certain usage conditions according to Key Performance Indicators (KPIs) derived from Operation Metadata) while other provisioning processes are run in external support systems.

Steps 10 to 11: as soon as the service is successfully configured the service status is set in the Registry according to IoT.est service lifecycle model.
3.4.2.3 Service Usage

The capabilities provided by IoT.est services that are active in the IoT.est Runtime can be invoked by service consumers. In case usage conditions are reached (defined in Operation Metadata) usage events are published with usage context data e.g. consumer identity, service capability Id and usage Id and amount like sensed data.

The Main Message flows are (see Figure 20):

Steps 1 to 3: The consumer, with appropriate permissions, invokes composite service capabilities through the Runtime Message Bus. The Runtime Message BUS routes the invocation to the real Functional Composite Service endpoint eventually performing any transformation and adaptation needed in terms of data and protocols.

Steps 4 to 5: the invocation received by the composite service triggers the execution of a certain workflow among its service components according to its business logic. Atomic Service components interact with encapsulated IoT resources as needed.

Step 6 to 7: In case usage conditions are reached (defined in Operation Metadata) usage events are published to Runtime Management services with usage context data e.g. consumer identity, service capability Id and usage Id and amount like sensed data.
3.4.2.4 Service Monitoring

When the service is contracted, it is necessary to start monitoring the service KPIs (Key Performance Indicators). If the Service SLA is violated, it may be necessary to trigger some adaptation actions to correct the deviations.

The Main Message flows for the Service Monitoring use case are (see Figure 21):

**Step 1**: In case Monitoring usage conditions are reached (defined in Operation Metadata) usage events are notified to Runtime Monitoring Management services with usage context data e.g. consumer identity, service capability Id and usage Id and amount like sensed data.

**Steps 2 to 3**: The monitoring service asks the runtime governance for Operation Policies applicable to the usage context that are retrieved from the Registry.

**Steps 4 to 5**: The monitoring service asks the SLA service to evaluate the SLA policies. In case the Service SLA is violated it may be necessary to trigger alarms and some adaptation actions to correct the deviations (e.g., reserve more resources for the service).
3.4.2.5 Service Adaptation

The Service Adaptation provides the necessary functionality to resolve the problem and get the service back to a normal operational state as efficiently as possible. This should involve automated adaptation actions including reconfiguration, re-assignments and/or temporary work-around. These actions will be initiated by the Adaptation service according to adaptation requests from Monitoring service. Adaptation service uses adaptation policies according to contracted SLAs.

The Main Message flows for the Service Adaptation use case are (see Figure 22):

**Steps 1:** In case the Service SLA is violated the Monitor Service asks the adaptation to resolve the problem and gets the service back to a normal operational state as efficiently as possible.

**Steps 2 to 3:** The Adaptation service asks the runtime governance for SLA and Adaptation Policies according to contracted SLAs which are retrieved from the Registry as part of the Operation Metadata.

**Step 4:** The Adaptation service performs adaptation actions according to evaluated policies e.g. allocate more network resources.

![Figure 22. Message flow in service monitoring](image)

![Figure 23. Message flow for the example of allocate more network resources in service adaptation](image)
4. Requirements Traceability Matrix

This chapter reviews the requirements defined in the IoT.est Deliverable D2.1 [IoT.est-D2.1, 2012] and provides mappings between those requirements and the components in the IoT.est architecture. This is to ensure that the requirements can be potentially fulfilled by the architecture design. Wherever applicable, new requirements are identified and documented.

4.1 Service Registry and search/query engine

Table 1. Service Registry and search/query engine traceability matrix

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Description</th>
<th>Architecture Component</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.3.0.4</td>
<td>Directory and discovery services, and query interface Based on the service description framework</td>
<td>Service discovery and query interfaces</td>
<td></td>
</tr>
<tr>
<td>R.3.1.1</td>
<td>IoT service description: Identify distinct characteristics of the IoT services with use case and requirement analysis; also based on existing semantic description frameworks on Web services</td>
<td>IoT.est description ontology and distributed service description repository</td>
<td>The service description is stored in the distributed repository</td>
</tr>
<tr>
<td>R.3.1.2</td>
<td>IoT resource description: IoT resource description model is existent and can be used for generating emulation data for testing</td>
<td>IoT.est description ontology</td>
<td></td>
</tr>
<tr>
<td>R.3.1.3</td>
<td>Descriptions for the characteristics of the physical entities that of interests</td>
<td>IoT.est description ontology</td>
<td></td>
</tr>
<tr>
<td>R.3.1.4</td>
<td>Identify the physical entities of interests on the linked data and their relationships if applicable</td>
<td>IoT.est description ontology and distributed service description repository</td>
<td>The semantic description is stored in the distributed repository</td>
</tr>
<tr>
<td>R.3.1.5</td>
<td>QoS and QoI are important information for service composition and adaptation</td>
<td>IoT.est description ontology</td>
<td>The information is included in the semantic description for IoT services</td>
</tr>
<tr>
<td>R.3.1.7</td>
<td>Provide descriptions for the test components in the semantic description framework</td>
<td>IoT.est description ontology and distributed service description repository</td>
<td></td>
</tr>
<tr>
<td>R.3.1.8</td>
<td>Identify data for testcases that can be reused for testsuites after testcase derivation</td>
<td>IoT.est description ontology and distributed service description repository</td>
<td></td>
</tr>
<tr>
<td>R.3.1.11</td>
<td>Service Status is part of Service Description</td>
<td>IoT.est description ontology</td>
<td></td>
</tr>
</tbody>
</table>
A number of new requirements have also been identified for the search and query engine.

Table 2. Service Registry and search/query engine new requirements

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Description</th>
<th>Architecture Component</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR.3.3.1</td>
<td>Distributed repositories should be implemented to store the service descriptions</td>
<td>Distributed service description repository</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.2</td>
<td>The component should allow the registration of new services</td>
<td>Service registration</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.3</td>
<td>The component should support service discovery based on different criteria</td>
<td>Service discovery</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.4</td>
<td>Service suggestion and recommendation should be supported</td>
<td>Service recommendation</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.5</td>
<td>A ranking mechanism should be designed for service discovery</td>
<td>Service ranking</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.6</td>
<td>The component needs to support service description update</td>
<td>Service update</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.7</td>
<td>The component needs to support service description lookup</td>
<td>Service lookup</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.8</td>
<td>The component needs to support service description removal</td>
<td>Service removal</td>
<td>New requirement not presented in D2.1</td>
</tr>
<tr>
<td>NR.3.3.9</td>
<td>The component needs to support the registration of tests results and feed it in ranking, discovery and recommending actions</td>
<td>Service update</td>
<td>New requirement not presented in D2.1</td>
</tr>
</tbody>
</table>

4.2 Service composition

In the following table there is the relationship of the requirements from D2.1 that have been covered in the current architecture:

Table 3 Service composition traceability matrix

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Description</th>
<th>Architecture Component</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.4.0.1</td>
<td>IoT.est SCE should ensure rapid time to market in IoT service development.</td>
<td>All</td>
<td></td>
</tr>
</tbody>
</table>
| R.4.0.2        | IoT.est SCE should address the problem of providing IoT services | Design Time Composer | Design Time Composer and templates will be
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Provided in</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.4.0.3</td>
<td>The IoT.est SCE should be easily set up.</td>
<td>GUI</td>
<td>The GUI will be provided as a web based interface, so no need for installation.</td>
</tr>
<tr>
<td>R.4.0.5</td>
<td>The IoT.est SCE should be compatible with the software for other software lifecycle phases</td>
<td>All</td>
<td>The Service environment through the GUI and the Governance model will integrate the different lifecycle phases.</td>
</tr>
<tr>
<td>R.4.0.6</td>
<td>The GUI of the SCE should be able to provide both low level accesses to functionality and also high abstraction level through GUI.</td>
<td>GUI and Governance</td>
<td>Different roles will have different functionalities.</td>
</tr>
<tr>
<td>R.4.0.7</td>
<td>Role based access could be provided in order to adapt the user functionalities and the options depending of the user role.</td>
<td>GUI / Governance</td>
<td></td>
</tr>
<tr>
<td>R.4.0.10</td>
<td>SCE is run-time environment agnostic</td>
<td>All</td>
<td>The architecture does not depend on any specific technology.</td>
</tr>
<tr>
<td>R.4.0.11</td>
<td>SCE is used to assist on the definition of Business Requirements to be fulfilled by Technical Services.</td>
<td>Governance / GUI</td>
<td>Different roles access different functionality.</td>
</tr>
<tr>
<td>R.4.1.1</td>
<td>IoT.est SCE should assist the process of including the user goals</td>
<td>GUI</td>
<td></td>
</tr>
<tr>
<td>R.4.1.2</td>
<td>IoT.est should be able to deal with RESTful and also with web services.</td>
<td>DataBinding Manager</td>
<td>The architecture is not limited to an specific technology</td>
</tr>
<tr>
<td>R.4.1.3</td>
<td>IoT.est SCE will allow the identification of service components that matches the user goals using a knowledge-based approach.</td>
<td>GUI/DesignTimeComposer</td>
<td></td>
</tr>
<tr>
<td>R.4.1.4</td>
<td>IoT.est SCE will allow the composition of services and the definition of the process flows using knowledge based approach.</td>
<td>GUI / DesignTimeComposer</td>
<td></td>
</tr>
<tr>
<td>R.4.1.5</td>
<td>IoT.est SCE will provide Design Time Composer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
inference mechanism to identify and select re-usable services components.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Description</th>
<th>Architecture Component</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.4.1.6</td>
<td>IoT.est SCE will provide an extension to the current SCEs to design and develop services tailored to IoT.</td>
<td>GUI, Governance</td>
<td>Specific templates and tools will be provided.</td>
</tr>
<tr>
<td>R.4.4.1</td>
<td>Integration of testing in the service creation environment</td>
<td>GUI, Governance</td>
<td>All test related components</td>
</tr>
<tr>
<td>R.4.4.2</td>
<td>The information obtained in the tests should be taken into account in order to highlight faults identified in the tests</td>
<td>All test related components</td>
<td></td>
</tr>
<tr>
<td>R.4.4.2b</td>
<td>The IoT.est SCE will support testing procedures and will have means to communicate with and to utilise service test components.</td>
<td>All test related components</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3 Test

Table 4. Test requirement traceability matrix

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Description</th>
<th>Architecture Component</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.3.4.1</td>
<td>Test component for the emulation of external input from IoT resources the System under Test (SUT).</td>
<td>External Resource Emulation</td>
<td></td>
</tr>
<tr>
<td>R.3.4.2</td>
<td>A Standardised test language which can be distinctly interpreted and executed by a testing framework will be used by IoT.est</td>
<td>All test related components</td>
<td></td>
</tr>
<tr>
<td>R.3.4.3</td>
<td>Support for SOA based (e.g. WSDL/XML/RESTfull ) access to the SUT to trigger, adopt and update the test components</td>
<td>Test Control Interface, Resource Emulation Component, Network Emulation Component</td>
<td></td>
</tr>
<tr>
<td>R.3.4.4</td>
<td>Codec derivation from protocols that are used by the SUT</td>
<td>Codec Creation/Derivation</td>
<td></td>
</tr>
<tr>
<td>R4.2.1</td>
<td>The IoT Service Description should include an interface description of the SUT</td>
<td>Designer GUI, IoT.est Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R4.2.2</td>
<td>The IoT Service Description should include also a semantic description</td>
<td>Designer GUI, IoT.est Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R4.2.3</td>
<td>The IoT Service Description should include Deployment, Connectivity, Security information</td>
<td>Designer GUI, IoT.est Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R4.2.4</td>
<td>The IoT Service Description includes information on how to provide sensor data for emulation of</td>
<td>Designer GUI, IoT.est Ontology</td>
<td></td>
</tr>
<tr>
<td>Ressources</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.4.2.5</td>
<td>The IoT Service Description may include valid usage scenarios</td>
<td>Designer GUI, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.4.2.6</td>
<td>The IoT Service Description should include error behavior description</td>
<td>Designer GUI, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.4.2.7</td>
<td>The IoT Service Description could provide operations that should be called before and after a test was executed</td>
<td>Designer GUI, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.4.2.8</td>
<td>The IoT Service Description could provide a reset option</td>
<td>Designer GUI, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.4.3.1</td>
<td>Clear semantic description of SUT Capabilities can be interpreted.</td>
<td>IoTest Description Ontology</td>
<td></td>
</tr>
<tr>
<td>R.4.3.2</td>
<td>Standardized interface description of SUT.</td>
<td>IoTest Service</td>
<td></td>
</tr>
<tr>
<td>R.4.3.3</td>
<td>A Model for the creation of abstract test suites is existent and test suites can be derived</td>
<td>Test Design Engine</td>
<td></td>
</tr>
<tr>
<td>R.4.3.4</td>
<td>Test suite can be compiled and deployed to the test environment</td>
<td>Test Components</td>
<td></td>
</tr>
<tr>
<td>R.4.3.6</td>
<td>Interpretation of SUT and Knowledge base enables IoTest to generate test data</td>
<td>Test Data Generator</td>
<td></td>
</tr>
<tr>
<td>R.4.3.7</td>
<td>A Common Testing Standard (like TTCN-3) is supported</td>
<td>Test Components</td>
<td></td>
</tr>
<tr>
<td>R.4.3.7</td>
<td>The IoT Service Description should support the definition of common data/values for parameters</td>
<td>Designer GUI, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.5.2.1</td>
<td>Interface to monitoring tools/services</td>
<td>Service Monitoring, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.5.2.2</td>
<td>Explicit definition of real world interaction in the business process description</td>
<td>Designer GUI, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.5.2.3</td>
<td>Explicit definition of network requirements of the SUT in the business process description</td>
<td>Designer GUI, IoTest Ontology Description</td>
<td></td>
</tr>
<tr>
<td>R.5.2.4</td>
<td>Explicit definitions of resource requirements of the SUT</td>
<td>Designer GUI</td>
<td></td>
</tr>
<tr>
<td>R.5.2.5</td>
<td>Metadata for Regression-Testing (e.g. Version, machine interpretable interoperability information) This metadata need to be write-able by the test environment to mark conflicts</td>
<td>Designer GUI, IoTest Ontology Description, Service Update</td>
<td></td>
</tr>
<tr>
<td>R.5.2.6</td>
<td>Standard Test cases description which represents a Finite State Machine of the SUT</td>
<td>Test Case and Flow Creation</td>
<td></td>
</tr>
<tr>
<td>R.5.2.7</td>
<td>Test interface for each SUT</td>
<td>IoTest Service</td>
<td></td>
</tr>
<tr>
<td>R.5.2.8</td>
<td>The sandbox environment is able to reproduce elastic conditions in order to create necessary</td>
<td>Sandbox Instance, Emulation</td>
<td></td>
</tr>
</tbody>
</table>
testing environments. For example, the sandbox is able to emulate the network bandwidth decrease and to reproduce CPU overload.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Description</th>
<th>Architecture Component</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.5.0.1</td>
<td>IoT Service operation and business support have no impact on IoT Service features delivery</td>
<td>All</td>
<td></td>
</tr>
<tr>
<td>R.5.0.2</td>
<td>Deployment and Operation metadata semantic model must be independent from run-time technology</td>
<td>All</td>
<td></td>
</tr>
<tr>
<td>R.5.0.3</td>
<td>Trustworthy deployment and execution of services</td>
<td>Message BUS, Governance</td>
<td></td>
</tr>
<tr>
<td>R.5.3.1</td>
<td>Deployment metadata is derived from design metadata and business policies</td>
<td>Service Registry</td>
<td></td>
</tr>
<tr>
<td>R.5.3.4</td>
<td>Trustworthy Service deployment is an automatic process taking into account its dependencies availability</td>
<td>Governance</td>
<td></td>
</tr>
<tr>
<td>R.5.3.5</td>
<td>Business and Operation decisions are automatically Applied.</td>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>R.5.4.1</td>
<td>To adapt services to new conditions when network and environment variables change</td>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>R.5.4.2</td>
<td>In order to react to changes in the environment and the performance it is necessary to define correctly the monitoring methods and mechanisms.</td>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>R.5.4.3</td>
<td>The information retrieved in the monitoring should help in the identification of other services that can compensate or replace the service or functionality affected.</td>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>R.5.4.4</td>
<td>Users should be able to express the compensation requirement and the acceptable threshold for the quality loss if other resources are utilised when the service becomes unavailable.</td>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>R.5.4.5</td>
<td>Services automatically adapt to fulfil business and operation decisions</td>
<td>Management</td>
<td></td>
</tr>
</tbody>
</table>

4.4 Service Runtime

Table 5. Service runtime traceability matrix
5 Conclusion and Outlook

Development of the IoT based business services and applications is hindered by the gap between the heterogeneous Things layer and the high-level business layer. One of the objectives of the IoT.est project is to address this issue by proposing an architecture with open and standardised interfaces and re-usable components. Furthermore, the architecture needs to address the problems of scalability, heterogeneity and interoperability of the IoT. The concept of IoT services that are able to expose capabilities of their corresponding resources defines the paradigm of service-oriented computing in IoT.

The IoT services often operate in dynamic environments, and in many cases, the resources underlying these IoT services are mobile, unreliable, and capability-constrained. All these factors make the IoT services different from most existing legacy services on the Web, i.e., they are not as reliable and stable as those well-engineered and maintained business services. The IoT services can be combined with other applications and services to compose complex, context-aware business services. In a service composition process involving the IoT services, monitoring, adaptation and compensation are also important design considerations to ensure continuous service access and reliable response to consumers' requirements.

This activity of the IoT.est project has developed such a reference architecture based on service oriented computing and semantic technologies. The requirement traceability analysis shows that the requirements identified previously in the project have been covered by the architecture. The architecture work so far has developed the important components, interfaces, functionalities and interactions among them. It will be used as a reference framework not only for other activities and tasks in the IoT.est project, but also for other projects which aim to develop service oriented IoT systems and applications. The reference architecture will be continually updated as the IoT.est project progresses and knowledge is shared between the project and the IoT community.
References


[Linked-Data] Linked Data - Connect Distributed Data across the Web, http://linkeddata.org/


[SENSEI-Project] SENSEI (Integrating the Physical with the Digital World of the Network of the Future), www.sensei-project.eu/

[SOA4ALL] Service Oriented Architectures for All (SOA4All) project, http://www.soa4all.eu/.


