Internet of Things - Architecture
IoT-A

Project Deliverable D2.5 - Adaptive, fault-tolerant orchestration of distributed IoT service interactions

Project acronym:	IoT-A
Project full title:	The Internet of Things - Architecture
Grant agreement no.:	257521

Doc. Ref.:	D2.5	
Responsible Beneficiary :	UNIS	
Editor(s):	Stefan Meissner (UNIS)	
List of contributors:	Tobias Jacobs (NEC), Markus Joos (SAP), Carsten Magerkurth (SAP), Stefan Meissner (UNIS), Sonja Meyer (SAP), Klaus Sperner (SAP), Matthias Thoma (SAP), Gerd Voelksen (SIEMENS)	
Reviewers:	Tobias Jacobs (NEC)	
Contractual Delivery Date:	31.08.2012	
Actual Delivery Date:	15.11.2012	
Status:	Final	
Version and date	Changes	Reviewers / Editors
v01 – 19.04.2012	Table of Contents draft	Stefan Meissner (UNIS)
v02 – 02.05.2012	Table of Contents revised after phone conference	Stefan Meissner (UNIS)
V03 – 03.08.2012	Gerd's contributions	Gerd Voelksen (SIEMENS)
V04 – 20.09.2012	First draft of fault tolerant business processes	Sonja Meyer (SAP)
V05 – 06.11.2012	Introduction included and Self-configuration merged	Tobias Jacobs (NEC), Stefan Meissner (UNIS)

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

	Dissemination Level	PU
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the Consortium (including the Commission Services)	
CO	Confidential, only for members of the Consortium (including the Commission Services)	

Executive Summary

IoT-A, the Internet of Things – Architecture, proposes the creation of an architectural reference model for the Internet of Things together with a definition of an initial set of key building blocks. One of the main outcomes of IoT-A is the efficient integration of Internet of Things (IoT) systems into the service layer of the Future Internet. Work package 2 deals with the integration of real-world IoT services with enterprise level applications and processes of the Future Internet. This deliverable contributes towards the work package 2 objective of adaptive orchestration of IoT resources and services with enterprise services.

The service layer of the Future Internet will support domain experts in defining applications that make use of IoT services. Work package 2 addresses **IoT Business Process Management** by providing business process modelling and execution techniques tailored to the needs and specifics of the Internet of Things as well as **Service Orchestration & Service Composition** methods to organise IoT services in order to support IoT specific applications.

This deliverable reports about the work carried out in Task 2.2 “Orchestration and Management of Distributed Services” and Task 2.4 that has been renamed to “Complex Event Processing in IoT Architectures” from previous title „Global state detection and Complex Event Processing”.

This document describes how IoT-aware service orchestration and service composition techniques presented in D2.3 can be supported with self-management capabilities which allow to build systems that are able to configure, to optimise, to repair and to protect themselves without or at least with minimal human user interaction. These capabilities contribute towards non-functional requirements like dependability, efficiency, and robustness that have been identified by WP6 as essential for IoT systems. The document is structured according the self-management aspects. Self-configuration deals with service behaviour adaptation upon service execution self-optimisation describes adaptation to changes in the execution environment during service runtime, self-healing is about repairing service orchestrations when services become unavailable or do not operate in a normal way, and self-protection describes techniques that can prevent services from being used in an unwanted manner. This report also presents an approach for applying fault tolerance in IoT-aware Business Processes.

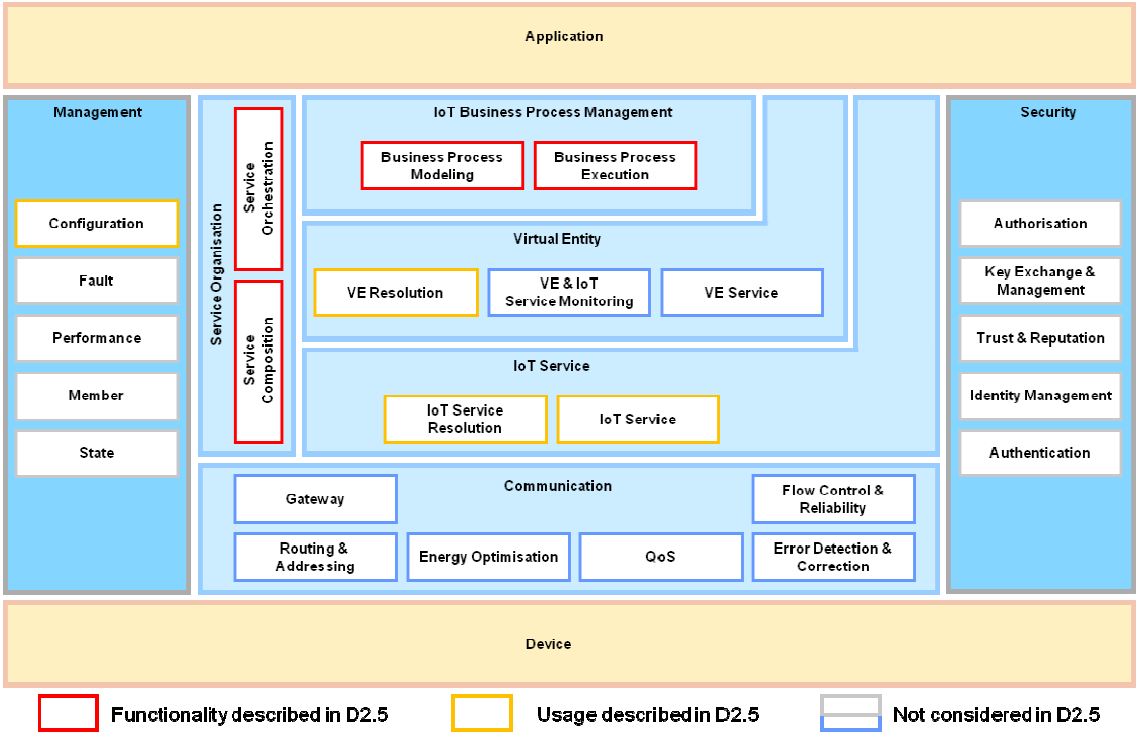

Table of Acronyms

Acronym	Definition
BPM	Business Process Management
BPML	Business Process Modeling Language
BPMN	Business Process Modeling Notation (up to Version 1.2) Business Process Model and Notation (from Version 2.0)
CI	Central Instance
D	Deliverable
eEPC	Event-driven Process Chain
ERP	Enterprise Resource Planning
GUI	Graphical User Interface
IAPMC	IoT-aware Process Modeling Concept
IOPE	Input Output Precondition Effect
IoT	Internet of Things
IoT-A	Internet of Things – Architecture
IR	Internal Report
JDBC	Java Data Base Connectivity
OMG	Object Management Group
OSGi	Open Services Gateway initiative
QoI	Quality of Information
QoS	Quality of Service
REST	Representational State Transfer
RWIP	Real World Integration Platform
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
UML	Unified Modeling Language
USDL	Universal Service Description Language
WP	Work Package
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Service Description Language
YAWL	Yet Another Workflow Language

Table of Content

Executive Summary ..... - 1 -
Table of Acronyms ..... - 2 -
Table of Content ..... - 3 -
Index of Figures and Tables ..... - 5 -
1. Introduction ..... - 6 -
1.1 Purpose of Deliverable ..... - 6 -
1.2 Introduction into adaptive and fault tolerant systems ..... - 6 -
1.2.1 Self-configuration ..... - 7 -
1.2.2 Self-optimization ..... - 8 -
1.2.3 Self-healing ..... - 8 -
1.2.4 Self-protection ..... - 8 -
1.3 Structure of Document ..... - 9 -
2. Self-Configuration in IoT Service Orchestration ..... - 10 -
2.1 State of the art in self-configuration in IoT and SOA systems ..... - 10 -
2.1.1 Self configuration in IoT systems ..... - 10 -
2.1.2 Self configuration in SOA systems ..... - 10 -
2.2 Updates to Service Description ..... - 11 -
2.2.1 Updates regarding Service Invocation ..... - 11 -
2.2.2 Updates regarding quality parameters ..... - 12 -
2.3 Resolution Phase ..... - 14 -
2.3.1 Resolution of Quality Parameters ..... - 15 -
2.3.1.1 Availability ..... - 15 -
2.3.1.2 Accuracy ..... - 16 -
2.3.1.3 Correctness ..... - 16 -
2.3.1.4 Precision ..... - 16 -
2.3.1.5 Summary ..... - 17 -
2.4 Self-Configuration during Service Composition ..... - 17 -
2.4.1 Self-Configuration of Service Compositions ..... - 20 -
2.5 Self-Configuration of IoT Services ..... - 21 -
2.5.1 Underlying RWIP Middleware ..... - 21 -
2.5.2 Rationale for an abstraction layer ..... - 22 -
2.5.3 Design of the abstraction layer ..... - 23 -
2.5.4 Gateway approach ..... - 24 -
2.5.5 Adding new devices ..... - 25 -
2.5.6 Extinction of devices ..... - 25 -
2.5.7 IoT Service strategies for self-configuration ..... - 26 -
2.5.7.1 Self-configuration of device integration ..... - 26 -
2.5.7.2 Self-configuration of device extinction ..... - 28 -
2.5.7.3 Self-configuration of information fusion ..... - 28 -
2.5.7.4 Self-configuration of service policies ..... - 28 -
2.5.8 Conclusions ..... - 29 -
2.6 Self-Configuration of Services for Global State Detection ..... - 29 -
2.6.1 Introduction to State Detection ..... - 30 -
2.6.2 IoT Service Self-Configuration ..... - 33 -
2.6.3 State Detection Service Self-Configuration ..... - 37 -
2.7 Conclusion ..... - 39 -
3. Self-Optimization in IoT Service Orchestration ..... - 40 -
3.1 Self-Optimization for Massive Service Orchestration ..... - 40 -
3.1.1 Introduction to Massive Service Orchestration ..... - 40 -
3.1.2 Relation to self-protection ..... - 41 -
3.1.3 Optimization models ..... - 41 -

3.1.3.1 Fixed Service areas vs. request-dependent Service areas - 41 -
3.1.3.2 Area coverage vs. point coverage vs. line coverage - 42 -
3.1.3.3 Single coverage vs. multi-coverage - 42 -
3.1.3.4 One-time coverage vs. continuous coverage and related mobility models - 43 -
3.1.3.5 Single-tenancy vs. Multi-tenancy - 43 -
3.1.4 Application scenarios - 43 -
3.1.5 Overview of existing algorithmic work and applicability to Massive Service Orchestration - 44 -
3.1.5.1 General theoretical background: Set Cover - 44 -
3.1.5.2 Existing results - 45 -
3.1.6 Open research challenges in Massive Service Orchestration - 46 -
3.2 Self-Optimization for Global State Detection - 46 -
3.3 Conclusion - 48 -
4. Self-Healing in IoT Service Orchestration - 49 -
4.1 State of the art in self-healing in IoT and SOA systems - 49 -
4.2 Replacing failed services in IoT Service Orchestration - 50 -
4.3 Repairing failed service compositions - 51 -
4.4 Self-Healing for Global State Detection - 52 -
4.4.1 Drop-out Repair - 53 -
4.4.2 Anomaly Detection and Repair - 54 -
4.5 Conclusion - 55 -
5. Self-Protection in IoT Service Orchestration (NEC) - 56 -
5.1 Introduction to Self-protection - 56 -
5.2 Self-protection in the context of IoT - 56 -
5.3 Failure models for IoT Service Orchestration - 57 -
5.4 Self-protection and the replication approach - 58 -
5.5 Fault-tolerant sensing - 59 -
5.6 Fault-tolerant actuation - 60 -
5.7 Summary - 61 -
6. Fault Tolerance of IoT-aware Business Processes - 63 -
6.1 Introduction - 63 -
6.2 Background - 63 -
6.2.1 Faults and Fault Tolerance - 64 -
6.3 Problem analysis - 65 -
6.4 Ratios for Fault Tolerance - 67 -
6.5 Conclusion - 70 -
7. Conclusions and Outlook - 71 -
8. References - 72 -
9. Appendix - 76 -
9.1 Introduction to Global State Detection using Complex Event Processing - 76 -
9.2 Information Architecture - 76 -
Event Description Language - 76 -
Rule Description Language - 77 -
9.3 CEP Application Domains - 78 -
Glossary - 81 -

Index of Figures and Tables

Figure 1 Abstracted meta model for self-organizing SOA [Liu 2008] - 11 -
Figure 2 Service Model extended by ServiceEndpoint - 12 -
Figure 4 ServiceArea - 13 -
Figure 5 IoT-A Service Model extended by Quality Parameters - 13 -
Figure 6 Self Configuration of IoT Service - 15 -
Figure 7 Input Output Matching with Quality Parameters - 17 -
Figure 8 Input Output Matching with Accuracy - 18 -
Figure 9 Input Output Matching with Availability - 19 -
Figure 10 Self-Configuration of Service Compositions - 20 -
Figure 11: Real World Integration Platform Site Manager - 22 -
Figure 12: IoT Service Associations from WP7 UC1 - 23 -
Figure 13: IoT Service Level Statements - 26 -
Figure 14: USDL Linked Data Vocabulary for Security - 27 -
Figure 15: Data Flow in a CEP-based State Detection Environment - 32 -
Figure 16: Storage and Discovery of the Event Channel Service Description - 34 -
Figure 17: Storage and Discovery of the Event Channel Service Description - 37 -
Figure 18: Example for the Publish-Subscribe Protocol for Event Distribution - 38 -
Figure 19: In Massive Service Orchestration. a set of Services needs to be selected such that the Service areas cover the Service request area. - 40 -
Figure 20 SENSEI Architecture [Strohbach 2010] - 49 -
Figure 21 Self-Healing of Service Orchestration - 51 -
Figure 22 Self-Healing of Service Compositions - 52 -
Figure 23: Process diagram of quality-based price process - 65 -
Figure 24: Fault event leads to process termination - 66 -
Figure 25: Fault events leads to defined fault handling process - 66 -
Figure 26: Rule for fault handling on top of business process - 66 -
Figure 27: Process diagram of quality-based price process - 67 -
Figure 28: Ecore Model of Reliability Metric - 68 -
Figure 29: IoT Process Components with Description Models - 69 -
Figure 30: Resource Confidence Value - 69 -
Figure 31: Rule Viewer for Generated DRL Code - 78 -
Figure 32: IoT-A Retail Use Case - Orchid Environment Monitoring - 80 -
Figure 33: CEP System Reference Architecture Components - 81 -

Table 1 REST methods used in for HTTP protocol methods - 12 -
Table 2: IoT Process Components effectible by faults - 65 -

1. Introduction

1.1 Purpose of Deliverable

The objective of this deliverable is to develop the concepts and approaches for autonomous computing in the context of IoT Service orchestration. It is a direct continuation of D2.3 [Meissner 2012], where it has already been outlined what the four fundamental self-* properties of autonomous systems mean for IoT-A in general and IoT Service Orchestration in particular. This deliverable is devoted exclusively to these properties, which is reflected also in the structure of the document.

In order to obtain a complete view, each of the four self-* properties (self-configuration, self-optimization, self-healing, and self-protection) is addressed from different points of view: How to express requirements, how to quantify the properties, and how to address these requirements on the communication level, on the architecture level, and on the algorithmic level.

1.2 Introduction into adaptive and fault tolerant systems

This deliverable describes approaches for orchestrating IoT Services within the IoT-A framework in such a way that the resulting composite services or processes satisfy essential non-functional requirements like dependability, security, robustness, and efficiency. These requirements not only have to be met at creation time, but their satisfaction needs to be ensured during the whole lifetime of the process, which means that it needs to be adaptive to unforeseen events like failure of underlying services, malicious attacks, or changes in the quality assessment of particular services.

In the domain of IoT, mobile and unreliable services are rather the rule than the exception: GPS devices do not work in tunnels, Battery-driven sensors can run out of power, network connections might become unavailable, etc. Furthermore, the large number of devices and services in typical IoT scenarios turns creation and maintenance of composite services into a task that cannot be solved manually by humans. In other words, composed IoT services will have to be self-managed systems to a large extent.

In general, a system is considered to be *self-managed* or *autonomous* if it establishes a set of properties or performs certain actions by itself, i.e. without interaction from outside the system. This definition applies not only to computational systems, but to all kinds of natural and artificial system like ecosystems, political systems, or even the human body.

In many cases the purpose of self-management of a system is purely to ensure its own existence, and often this property is the result of a long evolutionary process. However, and in particular in the case of computer systems, systems can also be *designed* to be autonomous. Here the purpose in addition is to save maintenance costs – as everything the system can do by itself does not need to be done by humans. Furthermore, many management functionalities can be done much more fast by the system itself than by external maintainers; think for example of the automatic identification and marking of bad hard disk sectors, as compared to this being done manually by even the most skilled system administrator.

System design always requires a careful trade-off between self-management and controllability Any management task a system performs by itself introduces the risk that events unforeseen at design time lead to failure or even undesired behaviour of the system. In cases like autonomously driving automobiles this can have severe consequences. Another example are systems that react to perceived threads by restricting access, and thus may be unusable during long periods of time.

Despite these challenges, in [Kephart and Chess 2003], autonomic computing is identified as the only possible way to overcome the ever-increasing complexity of computing systems, and we have already pointed out above that this holds even more for the Internet of Things, where the number and heterogeneity of devices and subsystems will be taken to an unprecedented level. The authors break the property of self-management down into four main sub-properties: Self-configuration, self-optimization, self-healing, and self-protection. In the remainder of this subsection we will explain what each of these self-* properties means in the context of IoT Service Orchestration, and in the following subsections of this chapter approaches to orchestrate Services accordingly will be described. An overview of the main challenges and approaches to meet them has already been described in D2.3 [Meissner 2012].

Coming from the business process perspective orchestrated or un-orchestrated services are executed in a predetermined process flow. By IoT-aware business processes as defined in [Meyer 2012], we understand business processes that specify IoT services among further components in the process model. Therein IoT services, devices, resources and entities are a particular type of potential process model components. Resolution and execution parameters can be defined as requirements on process component level. These requirements are specified for each process element individually of a dedicated process. The problem appears that some sensitive business processes or just parts of it shall not be fault-tolerant and shall be finished accordingly if a fault occurs while others shall be fault-tolerant. To provide a basis for decisions for process resolution and execution, reliability requirements can be defined on level of the individual process elements that are considered in the allocation process during the resolution. This includes not only requirements specifications for IoT services, but also for devices and resources. For example it could be specified, that a sensor in a dynamically triggered quality-based price determination process may be used only if it meets at least a typical precision of 98%.

1.2.1 Self-configuration

A self-configuring system can be described as having the property that it does not require manual configuration before it is ready to be used. In other words, a self-configuring system automatically adapts to its environment and to the needs of the user. Although the latter kind of self-configuration needs to be based on some user information, the specification of requirements should be as simple and intuitive as possible for the user.

In the context of IoT Service Orchestration, there are two kinds of components that offer the potential to self-configuration. The first kind of components are the services themselves.

There need to be mechanisms for services to register their own capabilities at the service resolution infrastructure. In traditional service oriented computing, the most important parameters are input, output, precondition and postcondition of the service, as well as non-functional properties like cost, security, and others. In the Internet of Things, additional properties like geographical service area, energy constraints, and communication constraints have to be added.

The second and more challenging kind of self-configuration takes place on the level of service orchestration. Here the user specifies a high-level service request and it is up to the orchestration engine to discover the set of services which can potentially serve the request, select the most appropriate one, and trigger and monitor its execution. In cases where it is not a single service which satisfies the user requests, multiple services have to be combined in an or semi-automatic or fully automatic manner. In cases where previously selected services become unavailable or failure occur (see self-healing), the self-configuration capabilities will be needed again for selecting and alternative.

1.2.2 Self-optimization

In general, self-optimization describes the process where a system adapts to its environment in a way that it can serve its purpose as good as possible. While the process of self-configuration ends as soon as the system has made itself able to serve the user request, self-optimization goes one step further by choosing among several feasible configurations the best one.

In the context of IoT Service Composition, there are multiple parameters that offer the potential to optimization. Among these are the typical QoS parameters like dependability, cost-efficiency, as well as more IoT-specific parameters like energy consumption. Some of these goals are contradicting each other, which means that methods for finding the best trade-offs have to be defined and implemented. Like self-configuration, self-optimization is to be performed not only during the initial service orchestration step, but the service quality needs to be monitored during execution. In cases where better services become available, the quality of previously selected services decreases or is experienced to perform worse than expected, the optimization methodology needs to be reiterated.

1.2.3 Self-healing

If an overall system is able to detect the situation that individual parts of it do not work as intended and automatically takes measures to either repair or replace these parts, the system is said to be self-healing. Self-healing systems must have three basic capabilities. The first capability is to be able to detect the situation that something in the overall system is erroneous. In such an event, the second capability is used to identify the part that has caused the error, and the third capability is to repair, update, or replace this erroneous part.

In IoT Service Orchestration, self-healing takes place during the execution of processes involving multiple services. Detecting and isolating errors is being performed by an execution monitoring component. If the erroneous service cannot be repaired by the orchestration engine (which can be typically assumed to be the case), the self-configuration and self-optimization routines are reiterated in order to find an alternative service composition.

1.2.4 Self-protection

According to [Kephard and Chess, 2003], self-protection relates to the protection of the overall system from failure. Overall failure can be either an effect of a malicious attack or of a cascade of failures experienced by a whole series of system components. Self-protection not only includes countermeasures against single failures or attacks to influence the behaviour of the whole system, but also to anticipate such situations and to try to avoid them.

A formal distinction between self-protection and self-healing could be expressed as follows: Assume that a system consists of multiple components and serves multiple purposes. The system is self-repairing if in case of faulty components it can take actions after which it again serves all its purposes. In contrast, the system is self-protecting if, in case of faulty components where self-healing fails, it can take actions after which it can still serve a subset of its purposes.

In IoT Service Orchestration, a cascading failure occurs for example when one service returns a wrong result, and this error is propagated through the execution of all subsequent services. Even worse, a single service might produce an infinite stream of random data, and if this data is fed into subsequent services without checking, all services in the composition become unavailable for future executions.

1.3 Structure of Document

At first we motivate a cloud-based data-sharing approach to IoT-service orchestration. The rest of the structure will be according to the self*-properties identified in the introduction taken from D2.3 section 5.4 IoT-aware aspects of Service Orchestration. The document will be completed by an approach for fault-tolerance of IoT-aware Business Processes.

2. Self-Configuration in IoT Service Orchestration

Services as defined for Service Oriented Architectures "[]... may autonomously vary in their implementation, deployment, operation, and management independently of their consumers."[Liu 2008] That means the implementation of the service is left to the developer of the Resource, but to be open to several potential service consumers the service needs to be as flexible as possible. The service then allows the service consumers to specify which particular configuration is desired so that the service behaviour will be adjusted accordingly. That adjustment of service behaviour to fulfil a service request is meant in the following by self-configuration. It is presented a brief overview about state of the art of self-configuration in IoT systems as well as in SOA systems followed by the explanation on how self-configuration techniques can be applied in IoT-A during the IoT Service Orchestration, Service Composition as well as for Global State Detection. It is further described how self-configuration is applied by a proof-of-concept demonstrator that has been contributed to IoT-A by a project partner.

2.1 State of the art in self-configuration in IoT and SOA systems

2.1.1 Self configuration in IoT systems

IoT devices are likely to be widely deployed in the environment so that it needs effort to maintain those devices on the site. There are two ways of dealing with widely deployed devices; either they have to be in the configuration as they were deployed until the end of their lifetime or until an engineer comes to reconfigure them, or the devices can be configured remotely over a communication link when it is required. There are several cases in which reconfiguration are required during a device's lifetime, such as applying bug-fixes or a changing of application use cases. Remote reconfiguration of devices is supported by some operating systems designed for embedded devices. Systems like Contiki, SOS, and Impala are suitable for wireless sensor networks while supporting dynamic module loading to configure applications on-the-fly [Toeyry 2011].

Several communication protocols have been proposed that support software updates over-the-air to Wireless Sensor Network-devices. In [Rossi 2008] a protocol has been developed that allows distribution of binary memory dumps over WSN-devices. Successive work addresses scalability issues by assuming noisy conditions in Wireless Sensor Networks of higher density [Rossi 2010].

2.1.2 Self configuration in SOA systems

According to [Binder 2011] Self-Configuration "refers to the ability of services to automatically configure themselves to adapt to different deployment environments".

In their view the service composition shall support partial matches to processing queries. They also propose that desired QoS parameters specified in the service request are broken down into requirements for individual services that take part in service compositions. Liu, Thanheiser, and Schmeck propose a SOA reference architecture in [Liu 2008] that equips each SOA element (each service) with an Observer/Controller (O/C) unit. The O/C units observe the SOA element itself as well as the environment of the SOA element. Based on the observations and predefined policies the behaviour of the SOA element can be controlled to in order to react to changes in the external and internal environment of the SOA element. The figure below illustrates the separation between the controlling management pane and the actual SOA Layer implementing the service logic.

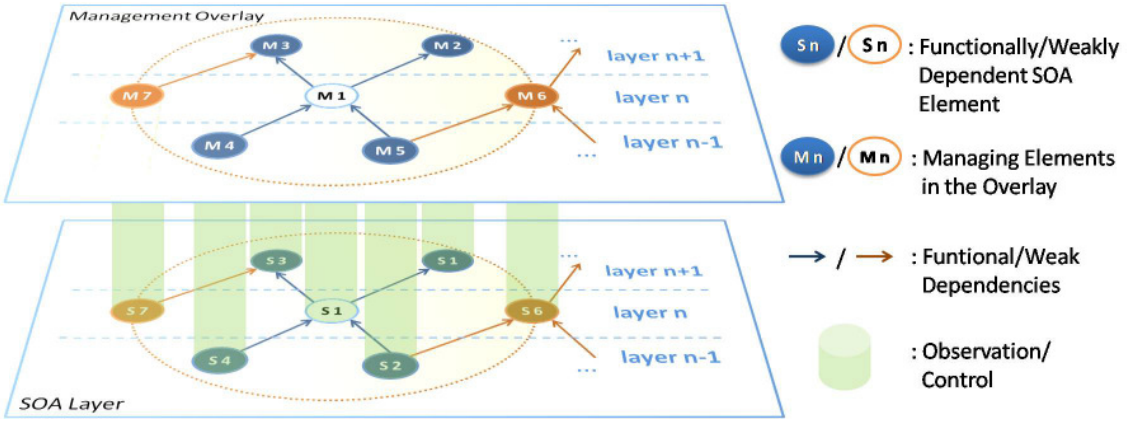

Figure 1 Abstracted meta model for self-organizing SOA [Liu 2008]

Figure 1 also illustrates that services interact with other services. Liu, Thanheiser, and Schmeck explain that each service in layer n can consume services from layer n-1 and deliver services to layer n+1. For each service in the SOA layers an O/C unit in the management overlay exists. Each service can "determine its immediate neighbourhood in the system" [Liu 2008] This immediate neighbourhood is then the scope the corresponding O/C unit is able to observe an to control. This design reduces the complexity of self-management and thus contributes to scalability of service orchestrations with self-configuring services.

2.2 Updates to Service Description

The service description model originally published in D2.1 [Martin 2012] did not include ways to express technical details on service invocation. It was also not specified how quality parameters can be assigned to describe service behaviour (**Quality of Service**) and the **Quality of Information** delivered by IoT services. Until this stage of the project there is no distinction made between Quality of Information and Quality of Actuation. The current assumption in this report is that Quality of Information parameters, like accuracy, precision and confidence can be applied in the same way as Quality of Actuation parameters. Thus the term Quality of Information mentioned in this work addresses sensing services as well as actuating services. Further research is needed in the future to assess whether this assumption is valid.

2.2.1 Updates regarding Service Invocation

Following extensions have been made to the Service Model originally published in D2.1 with respect to service invocation. In the revised version of this model depicted in **Fehler! Verweisquelle konnte nicht gefunden werden.** the *Service* is extended by a *ServiceEndpoint*. The *ServiceEndpoint* specifies how the service can be invoked and thus it provides technical details about the service endpoint such as network host (host name or IP address), network path as well as network port and network protocol. This information is needed to invoke a service, for instance, running on host 'test.iot-a.eu' under the URL 'test_service' accessible on port 8080 via HTTP protocol. A service user is then able to construct the service invocation call like this: 'http://test.iot-a.eu:8080/test_service/'. Within the *ServiceEndpoint* it can be given a URL to a technology specific service description, such as WSDL [WSDL 2007] or WADL [WADL 2009] where the service is further described.

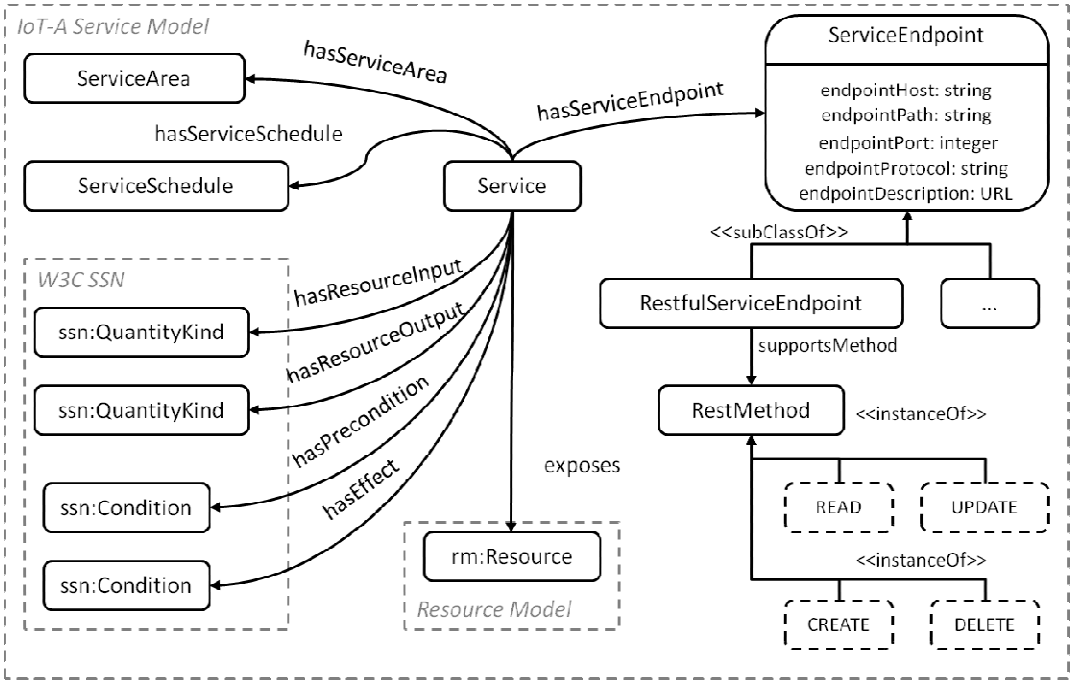

Figure 2 Service Model extended by ServiceEndpoint

The Service Model is open to any style of service, e.g., remote procedure call, but at this stage of the project only *RestfulServiceEndpoint* has been considered as subclass of *ServiceEndpoint*, specifying the standard [Fielding 2005] *RestMethods CREATE, READ, UPDATE, and DELETE.* In conjunction with a service invoked via HTTP this gives users the indication which HTTP-method to be used for which *RestMethod*:

RESTMethod	HTTP method
CREATE	POST
READ	GET
UPDATE	PUT
DELETE	DELETE

Table 1 REST methods used in for HTTP protocol methods

2.2.2 Updates regarding quality parameters

According to the IoT Service Description in D2.1 [Martin 2012] the functionality of IoT services are described mainly by the Input-Output-Precondition-Effect (IOPE)-set:

- Service (I)nputs
- Service (O)utputs
- (P)recondition that must be met before the service can be invoked
- (E)ffect that is the post-condition the service environment is in after invoking the service

Each of the service characteristics can be further defined by quality parameters that are typical for IoT services. In the work of [Wang et al. 2012] following parameters have been identified for specifying **Quality of Service**:

- AvailabilityQoS
- NetworkQoS
  - Delay
  - Jitter
  - PacketlossRate
  - Throughput

- RobustnessQoS

The Quality of Service parameters addressing availability of services have been considered in D2.1 already. As much as the *ServiceSchedule* specifies the temporal dimension of the service availability the Quality of Service parameter *ServiceArea* sets the spatial scope of a service. This feature is specific to IoT services since they have an impact on a part of the real world environment (in case of actuation) or gather information about a part of the real world (in case of sensing). The concept of *ServiceArea* was introduced in D2.1 already, but is now further refined as a union of geometric areas {*CircularArea ∪ PolygonalArea ∪ RectangularArea*}. The model is open to be extended with three dimensional spaces, like sphere and cubes if needed.

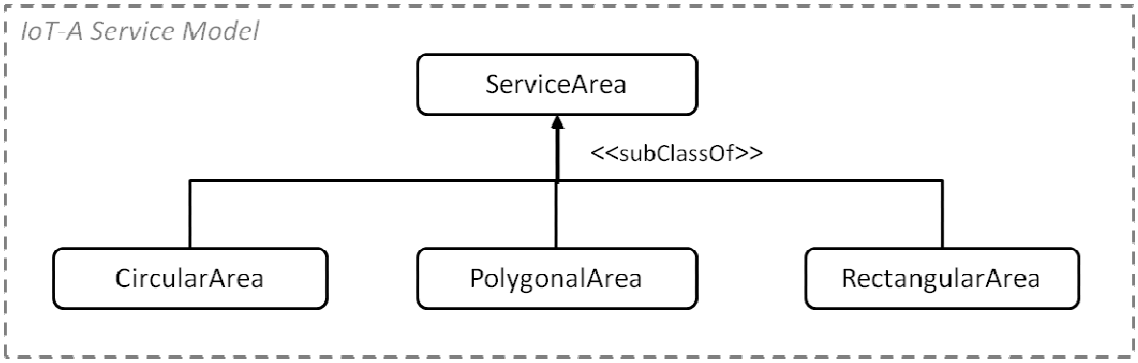

Figure 3 ServiceArea

The "Ontology for Knowledge Representation in the Internet of Things" [Wang 2012] addresses the Quality of Service and Quality of Information parameters that have been listed in D2.3:

- Performance (execution time, latency, roundtrip time, …)
- Dependability (availability, accuracy, precision …)
- Security and Trust (security, reputation)
- Cost and Payment (price, penalty)

As intermediate step this work specifies the Quality of Service parameters as properties of *Service* and the Quality of Information-parameters *Correctness, Precision, and Provenance* as properties of the IOPE parameters as illustrated in Figure 4.

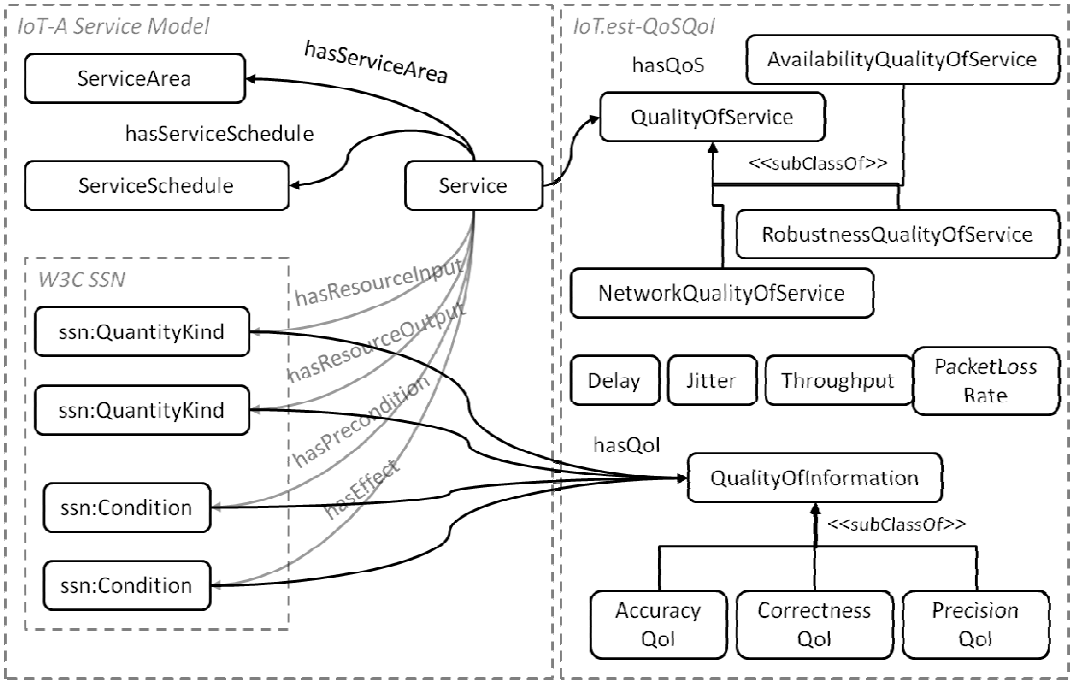

Figure 4 IoT-A Service Model extended by Quality Parameters

For future work it is aimed at aligning the Service Model with Linked USDL [SAP 2012] that is undergoing a standardisation process. Linked USDL have defined modules for pricing and security. Examples on their usage can be found in [Martin 2012].

2.3 Resolution Phase

This section describes how IoT-A's functional group 'Service Organisation' [Bauer 2012] handles service requests issued by IoT Service users in order to find suitable services that are able to satisfy the request. IoT-A's functional component 'Service Orchestration' [Bauer 2012] analyses the request and discovers IoT Services that are possible candidates to match the request.

Since service providers and service consumers (users) are loosely coupled, the service providers aim at offering services to as many consumers as possible. This means that services can fit several use cases and can therefore accept several service requests that do not match exactly but fulfil the requirements the service consumer has asked for. To illustrate this let imagine a user request that asks for a temperature value with the accuracy of 3%. On the other hand there is an IoT service available that provides temperature with an accuracy of 1%. If the 'IoT Service Resolution' component would consider services only that match the effect exactly the service providing an accuracy of 1% would not be offered as possible candidate. The service could advertise itself, by giving a range of values it supports, e.g. accuracy >1%, or with a discrete set of supported values, like, accuracy = {1%; 2%; 3%}. If specified like this the 'IoT Service Resolution' is able to find the service as candidate that is suitable to the request. Thus services can offer a range of parameter settings to potential users and are therefore able to configure themselves in order to support the actual service request. The approach followed in this report allows sending the configuration parameters together with the actual service invocation call as input parameters. Let us imagine a service output provides a quality of information accuracy = {1%; 2%; 3%}. This would lead to a configuration parameter 'accuracy=x' whereas x can be set to one of the allowed values taken from the output description, namely 1, 2 or 3%. This happens in step 5 and 6 of the following procedure that covers a service request initiated by a service user:

1. the user calls the Service Orchestration an provides a service specification that contains the requirements of the service the user wants to invoke
2. The Service Orchestration forwards the Service Specification to the IoT Service Resolution by sending a discoverService request
3. In case there are services available that match the service request a number of Service Descriptions belonging to the matching services is returned
4. The Service Orchestration selects the best matching service description (the service that is closest to the requirements user the user specified in the request)
5. The parameters to configure the service according to user's needs are extracted from the Service Description
6. The configuration parameters are sent to the user together with information about the service endpoint
7. The user invokes the service with usage of configuration parameters received before
8. The service configures itself according to the configuration parameters
9. The service is executed
10. The service result is sent as response to the user

The entire procedure is depicted in Figure 5:

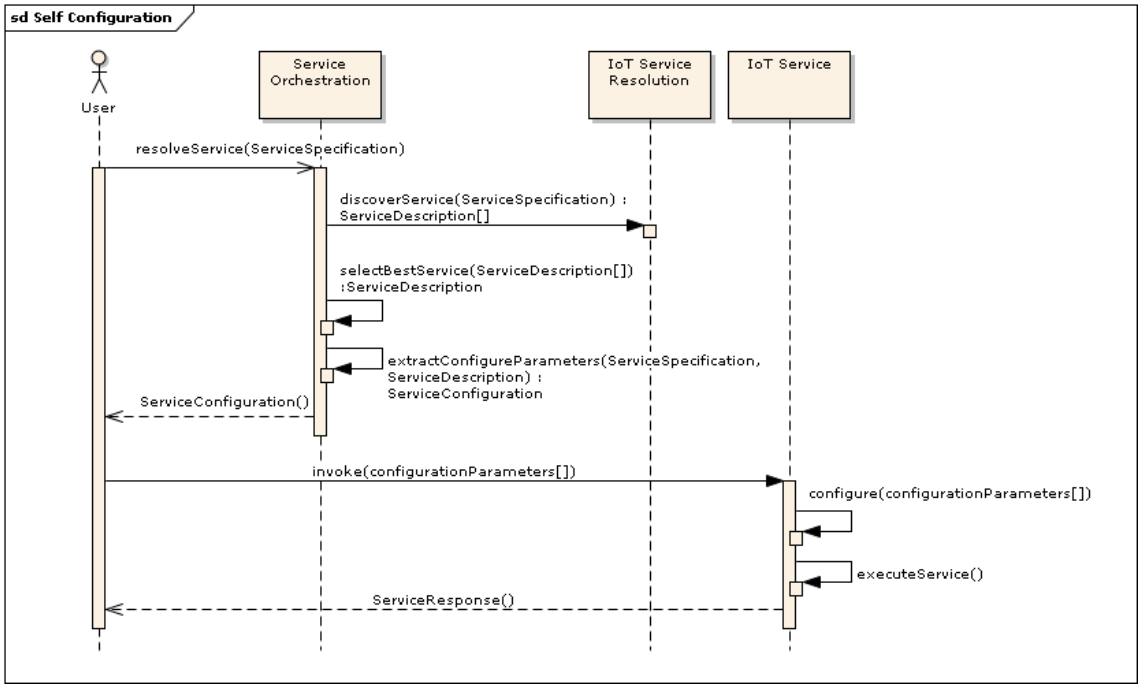

Figure 5 Self Configuration of IoT Service

2.3.1 Resolution of Quality Parameters

In section 5.1.1 of D2.3 [Meissner 2012] it is described how the services are resolved during service orchestration according to functional criteria that are specific to the use case services cover and service users are interested in. Service users are should be able to express requirements on quality parameters introduced in the subsection before additionally to the functional parameters, such as kind of information, e,g, temperature and so on. This subsection shows some examples on how quality parameters can be specified in ServiceSpecifications. The example specifications are written in SPARQL [SPARQL 2008], a semantic query language that can be used to query for services since the services are described in a semantic formats, namely RDF[RDF 2004].

2.3.1.1 Availability

For availability the IoT Service Description has defined the concept 'ServiceSchedule' that enables to specify dates when the IoT Service is supposed to run normally. This feature allows to better plan usage of Resources that rely on, e.g., solar energy and are thus only able to run at sufficient daylight conditions.

The following SPARQL query asks for services that start on 16:00 o'clock on 11th November 2012 and are available for more than 45 minutes:

```
PREFIX sm: <http://purl.oclc.org/net/unis/ServiceModel.owl>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX time:<http://www.w3.org/2006/time#>
SELECT ?service
WHERE {
      ?service sm:hasServiceSchedule ?schedule.
      ?schedule a time:Interval;
      ?schedule time:hasBeginning ?start;
                time:hasDurationDescription ?duraDesc.
      ?start a time:Instant;
```

```
                time:inXSDDateTime ?time.
        ?duraDesc a time:DurationDescription;
                          time:minutes ?duration.
FILTER ( ?time = "2012-11-11T16:00:00Z"^^xsd:dateTime &&
                ?duration > "45"^^xsd:decimal)
}
```

The results returned upon this request will also include services that have been up and running before the start time given in the request. But the results will definitely not include services that shut down before 16:45 on 11th November 2012.

2.3.1.2 Accuracy

The following example shows a service specification for a service that provides a temperature as output with the accuracy of at least 1%.

```
PREFIX sm: <http://purl.oclc.org/net/unis/ServiceModel.owl>
PREFIX qu-rec20: <http://purl.oclc.org/NET/ssnx/qu/qu-rec20>
PREFIX iotest: <http://ict-iotest.eu/iotest/ontologies/v1.0/IoT.est-
QoSQoI.owl>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?service
WHERE {
      ?service sm:hasResourceOutput ?output.
      ?output     a qu-rec20:Temperature;
                  sm:hasQoI ?qoi.
      ?qoi        a iotest:AccuracyQoI;
                  sm:qoiValue ?accuracy;
FILTER ( ?accuracy < "0.2"^^xsd:decimal)
}
```

2.3.1.3 Correctness

To specify the parameter correctness instead of accuracy following lines have to adjusted compared to the accuracy example above:

```
      ?qoi        a iotest:CorrectnessQoI;
                  sm:qoiValue ?correctness;
FILTER ( ?correctness = "0.99"^^xsd:decimal)
```

With this request a temperature is required with a correctness of at 99%.

2.3.1.4 Precision

To specify the parameter Precision instead of accuracy following lines have to adjusted compared to the accuracy example above:

```
      ?qoi        a iotest:PrecisionQoI;
                  sm:qoiValue ?precision;
FILTER ( ?precision > "96"^^xsd:decimal)
```

With this request a temperature is required with a correctness of 96%.

2.3.1.5 Summary

The list of presented parameters is not being considered as complete. With a query language that is not specific to a particular service specification the language is as extensible as the service specification model too. Security related parameters have not been given attention to in this document; security related aspects will be described in the upcoming deliverable D2.7.

The examples presented before focused on IoT Services only assuming the IOPE values are of types modelling measurements and observations [CSIRO 2011] as proposed in [De 2011]. These data types rather describe what data the sensing resource exposed by the IoT Service is able to produce and thus they are limited to physical properties, e.g. energy and temperature. These services are not necessarily associated to VEs, but if so the output of the IoT Service is associated to an attribute of the VE (*VE Service*). Since VEs are domain specific their attributes are domain specific too. From the Quality of Information point of view there is no reason to treat *VE Services* differently to IoT Services during the resolution phase.

2.4 Self-Configuration during Service Composition

In this subsection it is described how quality parameters are applied to service compositions consisting of more than one service. Basically the approach is analogue to the one described in section 5.5 of D2.3 [Meissner 2012].

Input-Output-Matching with Quality Parameters

The input-output-parameter-matching is extended by the quality parameters additionally. Also the quality parameters must match in order to decompose the composite service into atomic services that satisfy the requirements of the composite service as illustrated in Figure 6.

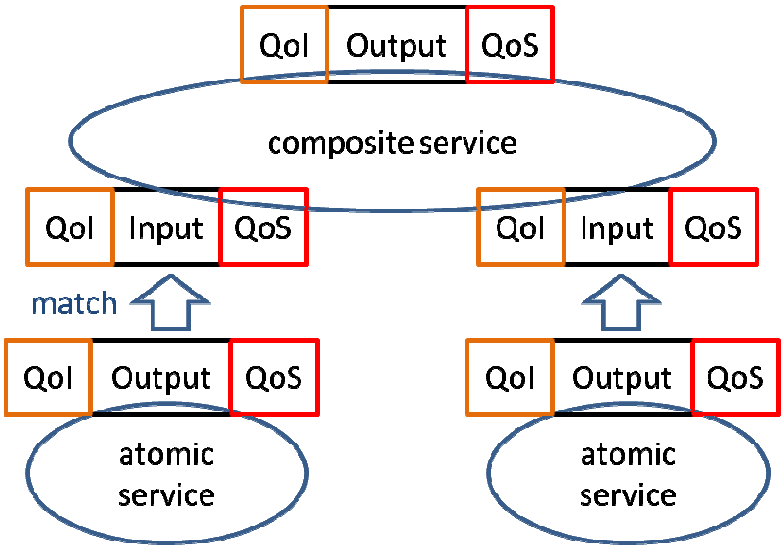

Figure 6 Input Output Matching with Quality Parameters

As explained in 2.3.1 an exact match of quality is not always needed; it is rather recommended to express requirements on quality parameters as inequality if applicable, e.g., for continuous number values or as range of discrete values, like particular elements in a set.

Example: Accuracy

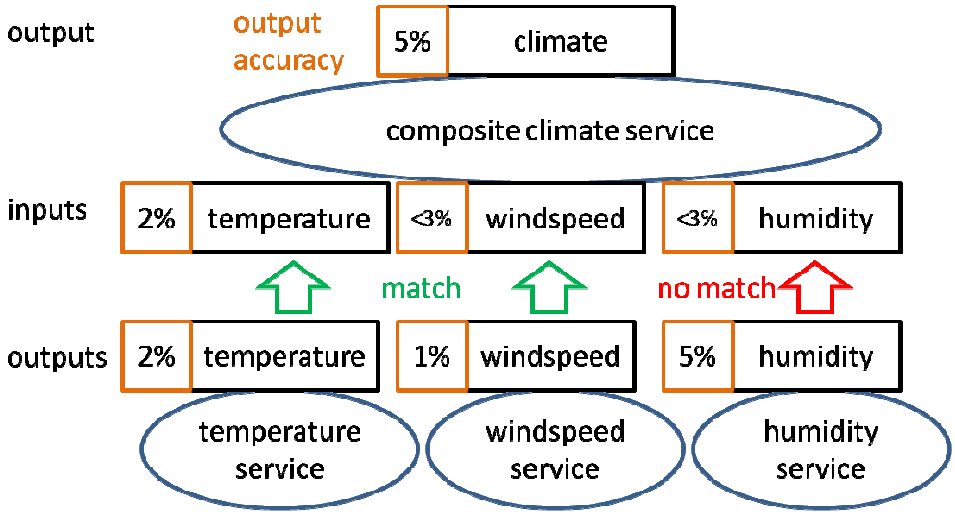

Figure 7 Input Output Matching with Accuracy

Figure 7 shows an example for a service composition annotated with the QoI parameter accuracy. In this example all the atomic services have been resolved by the physical properties (temperature, wind speed, humidity), but their adjacent accuracy quality does not match for all of them. For temperature there is an exact match (2% = 2%); for the wind speed service the accuracy provided by the atomic service is better than required by the input specification of the composite service (1% < 3%). So this is a valid match, but for humidity service the accuracy of the candidate atomic service is higher than the accuracy required by the composite service. The matching condition is not fulfilled ($5\% \not< 3\%$) and thus this humidity service is not a suitable candidate for the service composition.

Please note:

How the overall accuracy of the composite service is calculated based on the accuracy of the atomic services is a matter of implementation and therefore left to the provider of the composite service. For the service organization Functional Group the Service Description of composite services are seen as black boxes just describing themselves by its external interfaces together with their quality parameters.

Example: Availability - ServiceSchedule

The example depicted in Figure 8 illustrates the decomposition of the same composite climate service by the QoS parameter 'Availability' that is modelled as 'ServiceSchedule' by choosing weekdays to express the times of availability. The composite service is required to be available on Saturday (shown in the figure as 'Sat'). The atomic temperature service is also available Saturdays and thus matches exactly to the requirement of the composite service (Sat = Sat). The atomic service candidate for wind speed is available from Monday to Sunday and should therefore be considered as suitable candidate ($Sat \in \{Mon; \ldots; Sun\}$). The humidity service matches the requirement of the composite service by physical property, but it is only available on Fridays and cannot be considered as valid match.

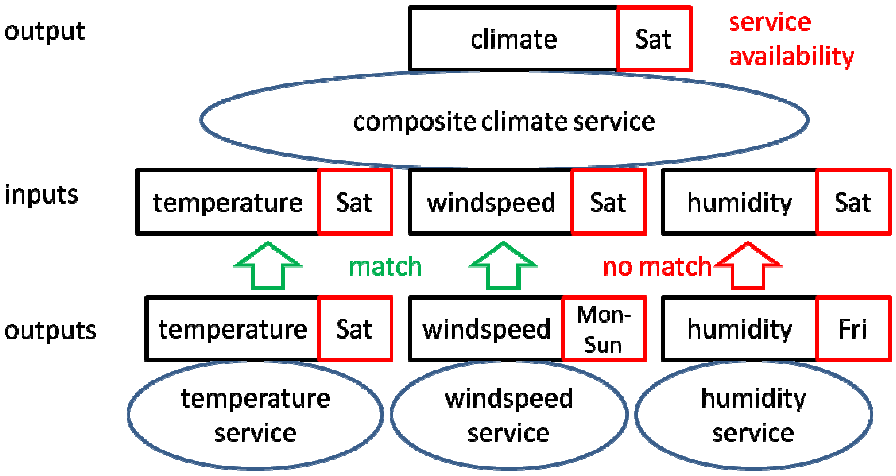

Figure 8 Input Output Matching with Availability

2.4.1 Self-Configuration of Service Compositions

If composite services need to be configured their orchestration is similar to the procedure depicted in Figure 5. The composite service is decomposed and the required inputs have to be satisfied first until the composition will be orchestrated. Only after to all of the required inputs have been resolved with suitable services the composition is seen as valid and the configuration process is initiated by the Service Orchestration as shown in Figure 9. The composite service will be configured after the atomic services have been configured.

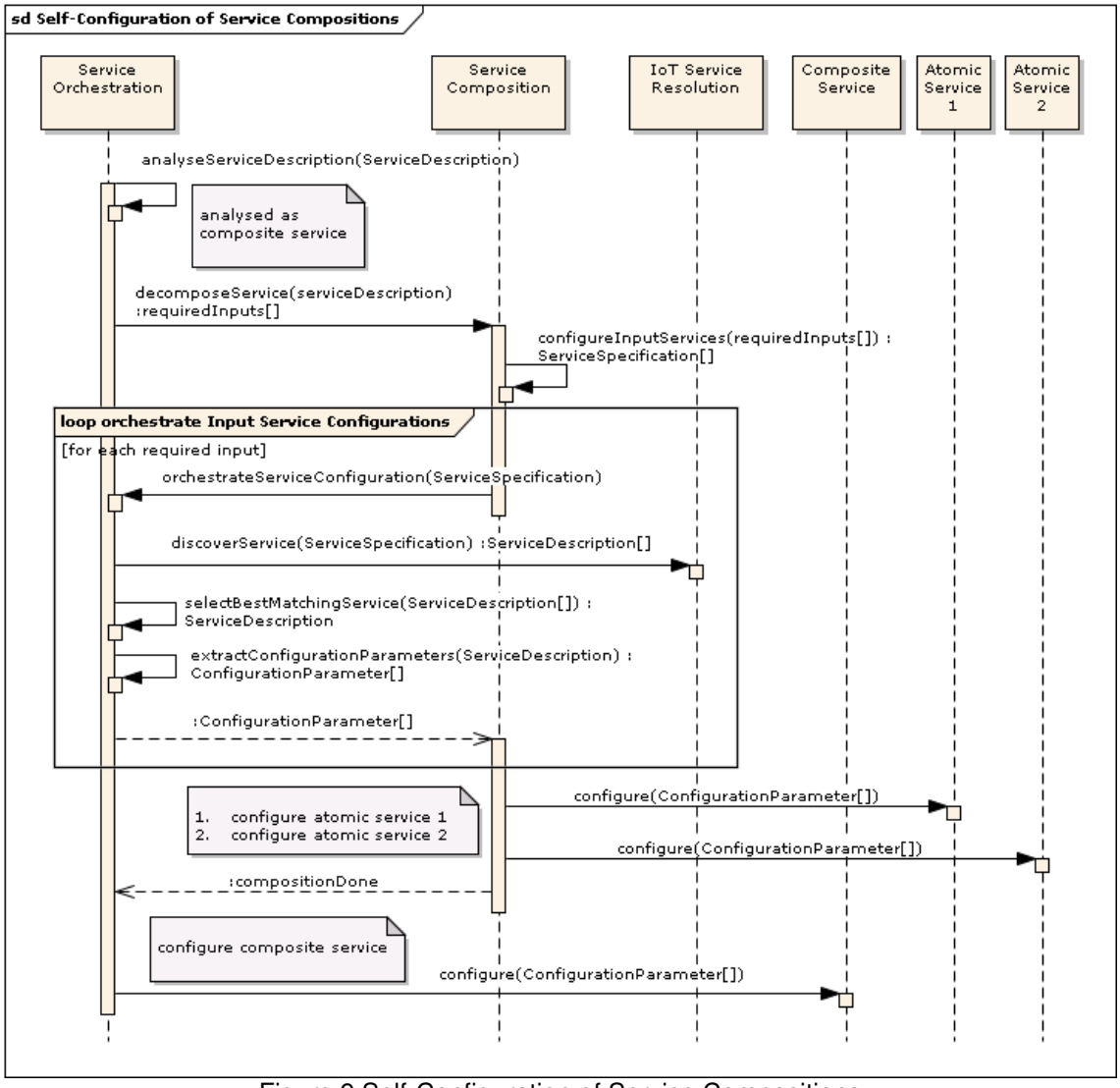

Figure 9 Self-Configuration of Service Compositions

In the example above the configuration of services is independent of the actual invocation which is different to the procedure shown in Figure 5 in which the configuration is executed on the IoT service after receiving the invocate-message. It is future work to investigate which method is more practical with respect to self-configuration.

2.5 Self-Configuration of IoT Services

In this section we discuss the introduction of an abstraction and management layer for IoT devices and IoT services. We propose to use a middleware for device integration (aka Real-World Integration Platform (RWIP)) as a Wireless Sensor Network Gateway and incorporate the new abstraction and management layer into this framework. The purpose of this layer is to automate the necessary steps for incorporating the services of a new device into the IoT stack. Furthermore, the necessary actions upon the extinction of a device are provided by this layer. A central component introduced in this section is used for automatically aligning and processing the sensor information from the devices associated with IoT services, so that the behaviour of the respective IoT services automatically adapts to the concrete devices associated with the services.

The foundation for the discussed abstraction and management layer is the Real World Integration Platform (RWIP) that we have already briefly introduced in D2.3. We will now first discuss the main concepts of the RWIP platform before the necessary augmentations to the platform are presented that allow for an effective self-configuration of IoT services.

2.5.1 Underlying RWIP Middleware

The Real World Integration Platform is a scalable and generic approach of a middleware platform for providing integration of a variety of different heterogeneous appliances. RWIP was brought into the IoT-A project as background IP and has been used successfully in both the IoT Week 2011 and IoT Week 2012 demonstrators. The goal of the RWIP is to connect SAP business systems to as many devices, systems and users as possible. The RWIP is a scalable distributed application for various domains written in Java using OSGi. SAP understands real-world integration as an alliance of monitoring and controlling devices that are functionally interacting together. The RWIP promises low footprint, runs on various platforms to allow a heterogeneous environment and provides connectivity to most of SAP's business systems.

The RWIP architecture consists of three main components: The Site Manager, the Central Instance and several Nodes. The system provides two different environments, a runtime and a configuration environment. This means that there is a Central Instance (CI) running in the background, whereas the Site Manager represents the user interface for controlling and managing Nodes. A setup that contains one CI, one or more Sites and a Site Manager is called an Organization. An Organization can contain several Sites. Sites can contain one or more Nodes which connect one or more Devices to the system.

A Node is an abstract entity that contains Devices. A Device is an abstract entity representing an Object respectively an Agent which may be attached to a device respectively a thing. However the term Object and Agent will be used as a synonym in this section. The RWIP definition of the Device does not fully match the definition given in the domain model of the IoT-A project. In RWIP, a Device could also relate to components that are only modelled in software. This, however, is not a real problem, as the typical definition of a Device in IoT-A is also what we typically understand is the device in RWIP. For the remainder of this section we refer to the more common term of an agent being used in RWIP.

Due to the separation of runtime and configuration environment an agent also consists of two components, a configuration part and a runtime part.

The Central Instance (CI) is a GUI-less application that stores the configuration data for all connected nodes in an embedded database, namely an Apache Derby database. The CI stores an Objects configuration, the infrastructure and relationship between Objects in a node, the different types of Objects, the message types for inter-agent communication, and the Objects code. Further the CI can contain additional libraries for instance Java Data Base Connectivity (JDBC) connectors.

The Site Manager is a GUI that allows configuration and management of the CI. Also, the Site Manager enables access to every Node in an Organization. With the Site Manager a Nodes configuration can be maintained. Furthermore, it provides control over the different Nodes, even during runtime. Since Nodes are implemented as OSGi bundles, the possible actions on a Node are typically the same as on an OSGi bundle; most importantly a Node can be started and stopped. The Site Manager also allows creating and managing the infrastructures hierarchy.

RWIP provides inter-agent communication via asynchronous events. Furthermore, agents can also define custom message types that allow for synchronous communication.

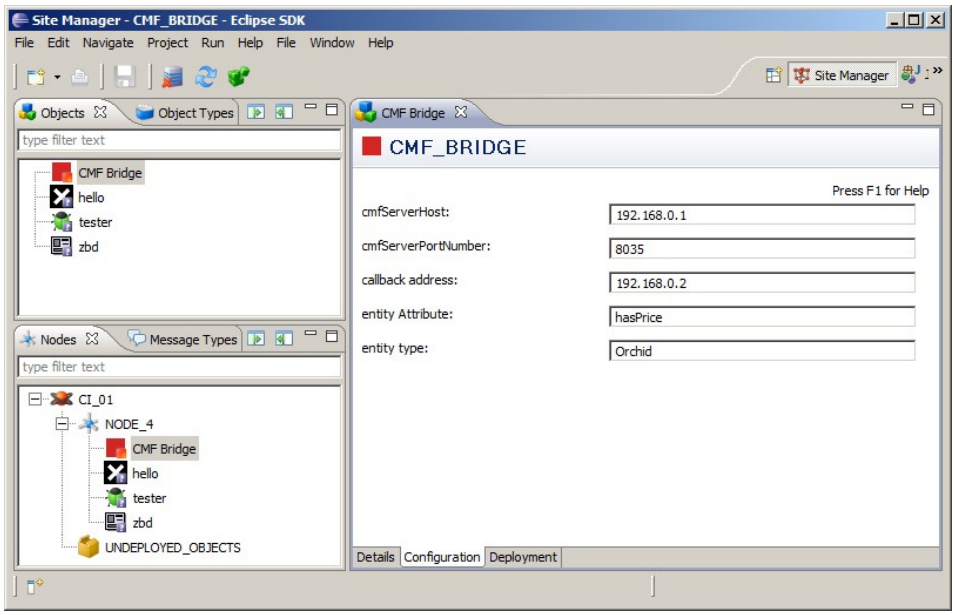

Figure 10: Real World Integration Platform Site Manager

Figure 10 shows a screenshot of the Site Manager with the node configuration that was used in the IoT Week 2011 demonstration. In the left part of the screenshot there is, among other agents, a so-called “zbd” device which relates to an electronic shelf label that displays pricing information in supermarkets using a proprietary radial protocol. With the Site Manager this device can be configured, started, stopped, etc. Important to note is the 1:1 relationship between the zbd device and the corresponding agent. From the perspective of the higher levels of the IoT-A functional view, the zbd agent is a prototypical IoT service in the sense of exposing the zbd resource hosted on the physical device. It can be bound, for instance, during process execution and effectively allows access to the respective resource via its IoT service interface.

2.5.2 Rationale for an abstraction layer

While the Real World Integration Platform works sufficiently well as a service platform for IoT services, there are two drawbacks when it comes to self-configuration of IoT services:

1.) The configuration of a node is performed in a static way, i.e. it is necessary for the user to graphically connect all individual physical devices with respective agents and take care of their states (i.e. if they are turned on or off) and the messages they exchange with other agents.
2.) There is a 1:1 relationship between agents and devices. This is appropriate for large IoT devices such as a fridge, a turbine, or a computer, as these devices usually come in quantities for which it is still feasible to individually administer each and every device and the abstraction of an “agent” seems appropriate due to the complex nature of each device. Thinking about other types of IoT devices such as sensors a wireless sensor

network, it would be impractical to model or the individuals sensors in a graphical administration tool.

In order to overcome these deficiencies we introduce an abstraction layer that effectively decouples the agent and hence the IoT service interface from the associated devices. For the consumer of the IoT services, the interfaces for invoking the services remain unchanged, however, the association of devices to services becomes dynamic and therefore allows for abstracting away both the quantity of devices associated with a single service and the state of the physical devices. In essence, this means that the service consumer can, for instance, query a service that provides the temperature of a given entity. It is then not transparent to the consumer, whether the service directly interacts with a certain resource hosted on some device, or if the information was still stored within the agent's memory. More importantly, it is also not transparent, how many physical devices are associated with an IoT service in any given moment in the case of WSNs. The IoT service then effectively acts as a gateway to the WSN.

2.5.3 Design of the abstraction layer

In order to realise transparency for the service consumer, the external interface of the IoT service does not change with the introduction of the abstraction layer. Likewise, for the design time components of the IoT service, the interfaces do not change, as long as no further configuration steps have to be introduced that require additional user interface components. These would have to be reflected in the configuration UIs of the respective agents/ services and would be accessed in the Site Manager just like any other agent configuration. In order to allow different associations between the OSGi runtime and the physical devices only the agent runtime OSGi bundle is augmented based on the Composite design pattern. An agent that previously represented a single IoT device would then be able to be associated with multiple devices of the same type, without introducing changes to the design time part of the agent or the IoT service interface to the outside. In order for this mechanism to work, the OSGI runtime is equipped with a novel component called “Multiplexer” that takes its name from the field of telecommunications and allows for combining information from different physical devices using different configurable strategies.

At the time of writing this deliverable, the implementation of the Multiplexer component and thereby the abstraction layer as such is not yet complete, so that currently only scenarios such as the Year 2 review are implemented in a running system that demonstrate how a mote device from WP5 is associated to a single agent in the domain of retail.

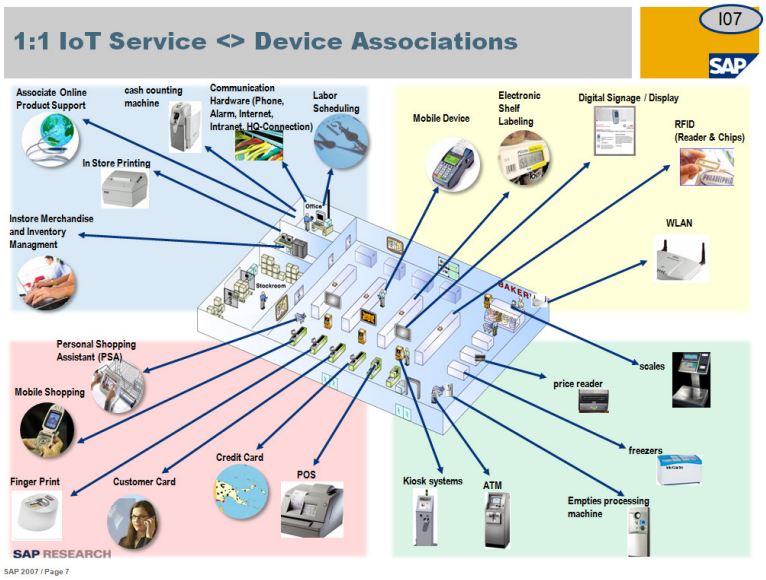

Figure 11: IoT Service Associations from WP7 UC1

As shown in Figure 11, the opportunities for providing IoT-A standardized device integration in the domain of retail are still manifold, although there are only individual devices associated with IoT services. The domain of retail includes a lot of physical devices, so that the design we have used so far definitely adds value. However, what we aim for conceptually is the automatic integration of multiple devices such as sensor nodes to a single agent whose individual sensor values are combined using different strategies within the Multiplexer component. This would allow for scenarios with distributed temperature sensors that e.g. are used for quality-based dynamic pricing schemes.

2.5.4 Gateway approach

One common way of integrating Wireless Sensor Networks is to treat such a network as one individual entity that is connected through a gateway. Such an approach completely separates the individual nodes from any party that is interested in obtaining the information from the WSN. For a service consumer, only the gateway as a single point of entry to the WSN is known. Such an approach naturally also works for a service platform such as the RWIP in which the WSN would be modelled as an individual agent. Any issues of changing the configuration of the WSN in terms of adding or removing devices or for dealing with the information gathered from the individual nodes (such as dealing with outliers or staleness of information) would, however, also be outside the scope of the service platform. While this is certainly appropriate for some scenarios, it might be necessary to have more control about the individual devices for which an IoT service interface is to be provided by the service platform and for which more control over the actual behaviour of the IoT service is required.

Because of that, a pure gateway approach is not always sufficient. Another reason why we sometimes need more control of the behaviour of the IoT service with respect to the configuration of the devices associated with the service are security constraints. Regarding for instance the WP7 use cases from the retail domain, we realise that certain devices change ownership when moving from one process step in the logistics domain to another one in the retail domain. There, a wireless sensor node is used during the transportation of goods from a distribution centre to a retail store in order to measure the temperature of the goods while they are on the road. During goods receipt at the supermarket, the temperature information would either have to be transferred to the temperature history in the ERP system at the store and the sensor would have to be dis-associated with the physical entity for which it had measured the temperature. Or, alternatively, if it had been part of a crate or if it had been fixed to that crate, the sensor itself might have to change ownership and effectively become part of the infrastructure of the retail store. In order for that scenario to become a reality, the dynamic association of new devices with an IoT service and the extinction of devices would have to include security features and would have to be under the control of the respective IoT service. In that example, the IoT service in the retail store would effectively need mechanisms to integrate and disintegrate devices, probably even to prioritise different devices depending on their origins, their quality, or other parameters, while at the same time the service interface and the functionality of the service, namely to provide a temperature value for a physical entity to which the IoT service is associated, would remain unchanged. For the service consumer, the changed physical configuration should be transparent and abstracted away already at the level of the service platform.

2.5.5 Adding new devices

The process of adding new devices highly depends on the nature of the device itself. While there are certain embedded platforms that allow for fully dynamic device orchestrations with self-advertising features and also certain communication paradigms such as the web of things exist, in which standard Internet protocols are used for all communication allowing, in principle, for auto configuration of communication partners, the majority of devices cannot be integrated in the fully automatic way. In the RWIP, the actual process of device integration is left to the agent developer. If we disregard the concrete steps necessary to make a device available for the RWIP runtime component, we still need to make sure that the individual devices are accessible via the appropriate interfaces to the devices and the respective information coming from the devices is accessible from any IoT service consumer.

As mentioned above, we utilise a Composite interface that adds to the existing, agent specific interfaces the methods of add(), remove(), and getChild(). By doing so, we can transparently replace the old interface that was not capable of integrating 1:n relationships between agents and devices with a new interface that allows for adding an arbitrary amount of suitable devices to the agent. The Multiplexer component can then call the getChild() methods in order to gain access to the individual devices associated with the agent and apply appropriate strategies for auto configuration, for instance prioritising the information given by individual sensor nodes depending on the credibility / trust of the node's origin or the variance of sensor values provided by the specific node.

2.5.6 Extinction of devices

The extinction of devices is performed in a corresponding way to the integration of devices. The composite interface's remove() method is called and the device is disassociated from the IoT service. Upon the removal of the last device, the IoT service is not automatically removed or disabled, but the Multiplexer component might default to a fallback strategy, for instance caching the final sensor value and return this value to service consumers for a certain amount of time, until finally the service is removed or returns null values /error codes or throws exceptions.

As pointed out in the previous section, the interaction of the OSGi agent runtime that represents the device or the devices and the actual devices themselves is not standardised and to be realised by the agent developer. Very often, it involves proprietary radio protocols that the IoT service abstracts away. For the extinction of devices, this implies that there is no standard way of unintentionally removing devices, e.g. when the devices are physically removed, run out of power, or simply break. The process of intentionally removing devices is straightforward, as this boils down to a call to the interface's remove() method from the Multiplexer component that in turn might be triggered by a respective action from the RWIP Site Manager. This is a standard process that the RWIP provides with respect to agent management. In the case of unintentionally removing devices, the proprietary interface functionality usually provides a time out mechanism for pings sent to or from the respective devices. If the device does not reply within a given timeframe or does not send out information by itself within a given timeframe, the device can be considered as unavailable and might be removed from the IoT service as a consequence.

Even though the interaction between agent runtime and device is proprietary and different for each device, the mechanisms for automatically determining, if a device needs to be removed from the IoT service are implemented in a device agnostic way, utilising the USDL based service level profiles that are discussed in more detail in D2.1. The service level statements valid for the IoT service naturally need to be fulfilled by the underlying devices as well. Figure 12 shows two basic service level statements that guarantee a certain response time and certain probability of responsiveness.

```
:ResponsivenessProbability
            a sla:Variable;
```

```
            rdfs:label "ResponsivenessProbability";
            sla:hasDefault [ a gr:QuantitativeValue;
                             gr:hasValue "95";
                             gr:unitOfMeasurement "percent" ]

:ResponseTimeThreshold
            a sla:Variable;
            rdfs:label "ResponseTimeThreshold";
            sla:hasDefault [ a gr:QuantitativeValue;
                             gr:hasValue "500";
                             gr:unitOfMeasurement "milli seconds" ]
```

Figure 12: IoT Service Level Statements

In the case of the traditional 1:1 association between IoT service and device, the service level statements must be fulfilled by that device. If the device failures to deliver, e.g. by not responding within the guaranteed response time, the removal of the device will be enforced and the IoT service will become unavailable. In the augmented 1:n associations facilitated by the abstraction layer, effective self-configuration measures can be taken that enhance the robustness of the IoT service, because it is no longer required for each and every device to fulfil the service level statements, as long as any of the devices reacts with in the require that timeframe. Naturally, this fact implies that adding new devices to an IoT service helps reaching service level policies and this is actually the essence of redundancy, i.e. one of the reasons why WSNs consist of multiple nodes that individually fail to provide the required is quality of service parameters, but as a federation are capable of providing accurate and timely information to the respective consumer.

2.5.7 IoT Service strategies for self-configuration

With the basic mechanisms of adding or removing devices to a single IoT service in place, the actual self-configuration strategies are discussed in this section. These strategies operate on the addition of devices, the extinction of devices, and the provision of device generated information to the consumer of the IoT service which mostly relates to effective means of sensor or information fusion, so that the information given by different devices converge to a meaningful result of the IoT service.

2.5.7.1 Self-configuration of device integration

If devices are not directly accessible from the service platform, because they are accessed only through a gateway, there is a 1:1 relationship between the agent runtime and the gateway and therefore self-configuration strategies are not applicable. If additional devices or nodes are to be integrated, then this will take place behind the gateway and is out of scope from a service platform perspective. For other cases, when it comes to the integration of additional devices and the association with an IoT service, there are three issues to take into consideration:

1.) The identification of the device
2.) Security concerns
3.) Service level policies

The first and fundamental issue to regard is whether the device can be identified and classified as function narrowly equivalent to the existing devices associated with the IoT service. As we are concerned with the IoT service level and not the entity service level, this basically means that the device must provide both the same kind of information as the existing devices as well as a spatial and temporal correlation to the existing devices associated with the service. The methods of obtaining the respective sensor information might be different between the devices, e.g. we might add a more accurate device to an IoT service or a device that has higher battery power, but still it is required that the information provided by the devices is equivalent on a device level. This point is important, as it is a much stronger constraint than merely being associated with the same entity on an entity service level. For the latter it would be sufficient to have e.g. two sensors measuring the location of an entity, one involving a satellite for GPS data and the other the nearest GSM cells to the respective entity (with the entity being, for instance, a car). On an entity service level, both would be equivalent in terms of providing the location information of the entity, but on an IoT service level, they would be fundamentally different and could not be subsumed under the same IoT service. Practically, this means that the identification of devices as being appropriate for adding to a given IoT service is not trivial and probably involves human intervention, since many IoT devices serving as sensors are not necessarily aware of their own locations and thus require a decision process performed by a human or some kind of surveillance mechanism outside of the device.

This security related issues concerning the integration of devices to an IoT service are obvious. If the device is not provided by the owner of the IoT service, then we can naturally not be sure about the real purpose and reliability of the device. Even simple sensor devices might provide inaccurate information on purpose or because of insufficient quality not matching the service level statements provided for the device.

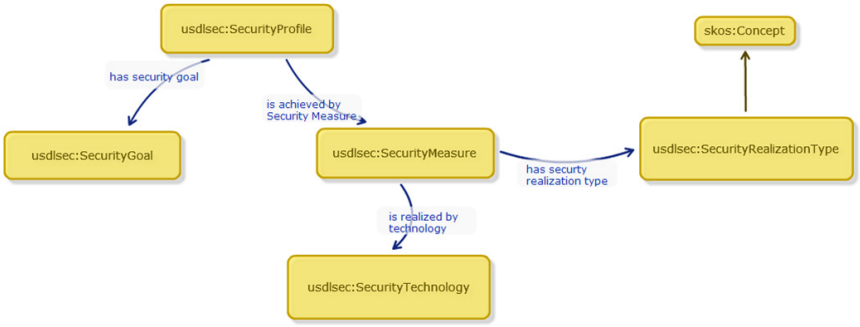

Figure 13: USDL Linked Data Vocabulary for Security

To some degree, there is an inherent risk in integrating unknown hardware that cannot be completely mitigated. For security critical applications it is therefore advised to not utilise device integration for unknown devices. The current approach for at least transparently providing a standardised way of expressing security features for the device is using self-descriptions modelled with the USDL Linked Data Vocabulary for Security (USDL-SEC). USDL-SEC aims at describing the main security properties and features of a service or device, described with USDL. In the same way as IoT service providers can use this specification to describe the security features of their services, and thus to support users in finding adequate alternatives to fulfil their needs, devices can also be described and these descriptions can be used for determining, if the device should be associated with an IoT service or not. The USDL-SEC description is shown in Figure 13.

Provided that devices are properly identified and security risks are either negligible or have been in some way mitigated, the final issue to look into is whether it makes sense from a service policy perspective to add the device to the IoT service. If the existing devices are already capable of fulfilling the requirements quality level of information, then it might make more sense to not add to the device, or to add the device and at the same time re-adjust the respective service policies. This is discussed below in more detail.

2.5.7.2 Self-configuration of device extinction

The automatic extinction of the device association to an IoT service is straightforward, when the device fails to respond within a given timeframe, which is configured using a respective USDL service statement, or does not send the required information by itself within a given interval. In that case it can be assumed that the device either does not exist any more, is broken, out of battery power, or for some other reason not available any more.

There is, however, also another case in which it might make sense to automatically remove the device from an IoT service, namely when the device is responding correctly, but the information provided by the device is not in sync with the results provided by other devices for the same IoT service. This might be due to malfunctioning or malicious behaviour, or simply due to accuracy or quality issues with the respective device. In any case, if the specific device does not contribute to the accuracy or robustness of the IoT service, it might make more sense to remove it, then to use the information from the other devices in order to compensate for this device's deficiencies. The removal of the device might then affect the service policies of the IoT service, i.e. the service might then be capable of providing "better" information and therefore, as a consequence of the self-configuration of the device extinction, is self-configuration of the service policies might be performed.

2.5.7.3 Self-configuration of information fusion

This aspect of self-configuration is still under investigation and experimentation. We believe that certain heuristics patterns should be available from the abstraction layer that can be used to configure how the IoT service should behave in the light of multiple devices providing sensor input. For instance, it should be configurable how to deal with

a.) The caching of information
b.) Assigning different weights to the output of certain devices

Depending on the application domain, it will be more or less appropriate to cache the information provided by the devices in order to save energy, or because there are no hard real-time requirements. The magnitude of caching will be determined by the service profile, but also by the interplay and number of different devices. In general, the more devices are available, the less need for caching might be required.

The more interesting point in information fusion relates to how to weigh and prioritise the different devices. A natural default might be to treat each device in the same way, especially if they come from the same vendor and where associated with the IoT service at a similar point in time. For scenarios such as the IoT-A work package 7 retail use case, in which a sensor device might change ownership from one party to the other during goods receipt at a supermarket, it might indeed make sense to treat the novel device with caution for a certain amount of time. The runtime performance and accuracy of the information provided by the device might be different than those of the devices already assigned to the IoT service. After the device has been in use for some time, it might then make sense to give a higher priority to the device, as it has proven to provide adequate performance (or not). The possibilities here are diverse; it just needs to be investigated, in how far complex patterns of prioritization actually provide more benefits compared to very simple ones.

2.5.7.4 Self-configuration of service policies

So far in the discussion, we have focused almost exclusively on the implications of breaking the 1:1 association of devices to IoT services, so that multiple devices can be represented by an individual service and therefore provide capabilities of dealing with the loss of individual devices or differing qualities of services of these individual devices, with the IoT service effectively becoming an additional layer of abstraction that provides a higher level of robustness and flexibility.

The IoT-A approach of IoT service modelling however also allows for self-configuration when a single device is exposed through an IoT service. Depending on the concrete sensor modality, the range and variance of meaningful values, and the nature of the application and deployment domain, the Multiplexer component might be able to detect outliers or improbable sensor values and react accordingly. While it might make sense to apply smoothing algorithms for the service results, the configuration of the device integration might also be adapted. Increasing outliers might result in increasing the frequency of querying the device, if applicable, so that a sufficient amount of plausible values per timeframe can be received, if the service policies guarantee a certain frequency. Likewise, the service policies themselves might be altered as a consequence of plausible sensor results. This means that the IoT service policies would not be provided as static USDL files any more, but that they would be dynamically generated and adapted according to the real runtime performance of the devices. In the light of the physical realities in terms of battery power or other factors that effectively influence runtime performance, the dynamic generation of service policies is a major advancement for realising insight to IoT services.

2.5.8 Conclusions

In this section we have outlined and discussed the potential of self-configuration of IoT services and presented our initial contributions based on our underlying service platform RWIP. As IoT services are to some degree more primitive than higher-level entity services, the capabilities space for self-configuration comes down to augmenting the standard 1:1 relationship of device and service allowing for different means of adding or removing devices as well as automatically adapting the fusion of information and the service policies of the IoT service. By implementing an additional abstraction layer within the IoT service platform, the benefits of self-configuration such as integrating additional devices when needed (e.g. when previous devices start failing or perform malicious actions) becomes available to the higher levels of the IoT-A stack without additional effort. In that respect, self-configuration capabilities of IoT services are a very sensible addition to the conceptual space of IoT services. It must be noted, though, that certain challenges such as developing proprietary protocols for adding devices without any human intervention still need to be solved on a broad scale, in order for self-configuration of IoT services to become a widespread solution in the realm of the Internet of Things.

2.6 Self-Configuration of Services for Global State Detection

This chapter introduces the idea of global state detection (GSD) and its integration into an IoT system. Since IoT systems may be very large and behave in a dynamic, sometimes unpredictable way, some mechanism determining the states of these systems are highly desirable. A more elaborated explanation of the relation between the IoT and the global state detection – particularly focusing on the utilization of Complex Event Processing (CEP) – was already presented in Chapter 5 of Internal Report IR2.3, see also Chapter 9[1]. Further details are under development for the future IoT-A Deliverable D2.6 "Events Representation and Processing", due by May 2013 which is targeted on the Reference Architecture of a CEP-based systems for state detection in the IoT.

It needs to be mentioned at this place that the IoT-A Work Package 2 Task 4 where this work has been developed is now renamed from "Global State Detection and Complex Event Processing" to "Complex Event Processing in IoT Architectures". The reason is the very narrow domain of global state detection. As listed in IR2.3, see Appendix 9.3, there are several further applications that utilize CEP and that are of great importance for the IoT. Examples cover system monitoring, system control, or anomaly detection. On the other hand, Task T2.4 will go into more detail of a reference architecture for a CEP-based IoT Service that provides functions for example for system monitoring, system control, or anomaly detection.

[1] As IR2.3 is not open to the public, the relevant sections are re-printed in the Appendix of this document, see Chapter 9

In this document, we will focus on one or several CEP-based services for global (or any other kind of) state detection and the related composition of such services within an IoT system in concert with other IoT services. As this document investigates the applicability of self* techniques on service composition and especially service orchestration for the IoT, there will be different sections referring to the issue of self* techniques for orchestrating a global state detection service. The present chapter deals with self-configuration scenarios along with an introduction to the context of global state detection using CEP. Chapter 3.2.2 adds self-optimization aspects and Chapter 4.4 focuses on self-repair/self-healing. Background information on the self* approaches in the context of IoT has already been given in Chapter 5.5 of Deliverable D2.3, see [Meissner 2012].

2.6.1 Introduction to State Detection

Detecting the global state of an IoT system is surely of high importance for system operation. However, there are many other states which might be of special interest beside the global state. Here are the examples:

- Sub-system state detection:
  some examples of sub-systems are as follows:
  - Location-related sub-system:
    Events related to entities or devices from a certain geographical area will be considered. Example: weather data in a given region;
  - Entity-related sub-system:
    A certain entity or a subset of entities of the IoT system is considered. Example: having a transport system, the state(s) one particular vehicle or all vehicles of a certain type;
  - Data-related sub-system:
    Events providing a certain type of information are processed. Example: traffic flow data;
  - Owner-related sub-system:
    If several component owners contribute to one joint IoT system, the overall state of the components belonging to one owner is determined. Example: the position data of the taxis of one operator.
- Process state detection, assuming a process is somehow different from a sub-system.
  - Business process:
    State of a particular order
  - Production process:
    State of a particular work item under development or load of the production line.

The utilization of Complex Event Processing (CEP) is the most appropriate way to detect the states of systems. The basic idea is that entities, devices or services generate events which describe their particular state.

Events are data objects of a specific type which are produced automatically and periodically. An event contains at least an event name, a timestamp, one or more values describing the state of the event generator, a reference to that event generator and some further information as listed in Chapter 9.2.

Such events are handed over to an event channel where they are maintained and from where they are delivered to the event customers. The event channel needs to cover all relevant components of the IoT system which produce or consume events, namely devices, services, processes, etc. It needs to provide an interface for event publish-subscribe access regarding event consumers. Thus, an event channel is a distribution service providing a platform or infra-structure for events which may supply several different consumers. An event customer, especially a global state detection service, subscribes for the events it needs for its purposes.

While Event Stream Processing (ESP) considers synchronized events of the same type from one or several sources, CEP accepts events of any type in any order, generated by any sources. Thus, an ESP service is easier to build and run compared to a CEP service because ESP requires a time-series analysis technique for evaluating an overall state or trend. CEP is somewhat more complicated due to its less restricted input data.

Usually, the heart of a CEP service is a rule engine which matches incoming events with rule preconditions in order to detect a given pattern. If such a pattern is detected, the rule produces a related result which may be a database or log file entry, a command that will be executed, or an input for a GUI for system visualization in a control room. Finally, the output of a rule may be again an event which is further processed within the CEP service, or which may be delivered to another CEP service. Such an event may also be re-supplied into the event channel for any further usage.

It is the very fundamental principle of a CEP service to produce an abstraction of the set of events being under observation. Usually monitoring a large set of discrete numeric values $v_i(x_j, t_k)$ is easier and handier to process – as long as the values remain within a given range – if an arithmetical average value as for example $\overline{V}_{i,k} = \frac{1}{|X_j|} \sum_{j \in X_j} v_i(x_j, t_k)$ is provided, where X_j designates the set of locations where the values originate from. Such an average may refer to time or location of one particular event type and may only incorporate a subset of last recent events or a subset of locations. The original events may be deleted. Thus, an aggregation of several events into a complex one is achieved. A reasonable and intended loss of information will help to reduce the flood of event data and to gain lesser but more meaningful, thus complex events.

Since the rules of a CEP service may produce further events or any other kind of output which again may be used to modify the devices or entities of the IoT system, the CEP-based global state detection service will be a significant part of the control loop within which the IoT system runs.

Having in mind that CEP services may produce events which are in turn processes by other CEP services, CEP services need some kind of arrangement among each other. According to Bruns and Dunkel, see [Bruns 2010], a composition of CEP services is called an Event-driven Architecture (EDA). This approach is more data- particularly event-related compared to the Service-Oriented Architectures (SOA) approach. However, these options are not contradicting as the EDA incorporates services, too.

The objective of this section is to show how IoT services, devices and entities on the one hand and CEP services on the other hand can be composed within an IoT system.

The following assumptions are made:

1. IoT-A and particularly Work Package 2 focuses on service orchestration only – leaving aside the approach of service choreography. This holds for CEP services, too.
2. Devices are sensors or actors, according to the IoT-A Reference Model, Deliverable 1.3 (see [Bauer 2012], Chapter 3.2). For both types of devices, it is assumed that they can deliver state events, even though actors are usually not constructed in such a way.[2]
3. The event channel is an IoT management service that distributes events from their generators to the consumers.

[2] For tags as the third type of device, no such assumption is made. In the context of IoT-A, a tag is a "Label or other physical object used to identify the Physical Entity to which it is attached" (see http://www.iot-a.eu/public/terminology or D1.3, see [Bauer 2012], Chapter 3.2.2, pp 44). Thus, tags do not maintain any internal state data.

4. Each device that produces events for global state detection is wrapped by a service which is able to communicate using a certain protocol and which can configure and control the device. For service modelling see [Martin 2012]

Particularly the last assumption helps various services to address devices – especially if these devices are strongly resource restricted in terms of storage, computing power, or communication capability. Therefore, such devices might lack in communication and interface standards. Thus, the device wrapper service connects devices with the rest of the IoT world. It needs to be device specific in order to fully control the device using the devices proprietary interface. Furthermore, the device wrapper service has to be IoT-enabled in order to accept and translate standardized commands or queries, particularly those for the global state detection.

In this case, the device wrapper service needs to be differentiated from the resource as it is defined in D1.3, see [Bauer 2012], Chapter 3.2.2.2, p 45. The device wrapper service requires a well-defined and standardized interface for device control in order to become part of the self* operations. Particularly, it needs to be managed by the IoT Service Infrastructure according to Chapter C2.1.2 of D1.3. Therefore, it is more than just a resource. However, a device wrapper service might utilize and wrap the devices' resources, independently of the recourse being "on device" or "networked".

Figure 14 displays the components and the data flow in an environment for global state detection using CEP. The system control loop is depicted by the feed-back arrow which represents the execution of control commands while the event arrow shows the flow of events generated by a CEP service which are used by other CEP services. The components are assumed to be modelled as services according to IoT-A Deliverable D2.2, see [Martin 2012]. This helps managing the components by using the interface of the IoT Infrastructure for look-up, resolution, and discovery as defined in IoT-A Deliverables D1.3 and D4.3, see [Bauer 2012] and [De 2012], respectively.

In addition to earlier versions, Figure 14 also displays the event feed-back into the Event Channel while the other feed-back arrow designates any control actions in general. The event feed-back indicates that events produced by a CEP system may be consumed further by other CEP systems according to the idea of an Event Driven Architecture. Regarding the Internet of Things, a CEP-based system that detects, processes, and generates events is a particular IoT subsystem.

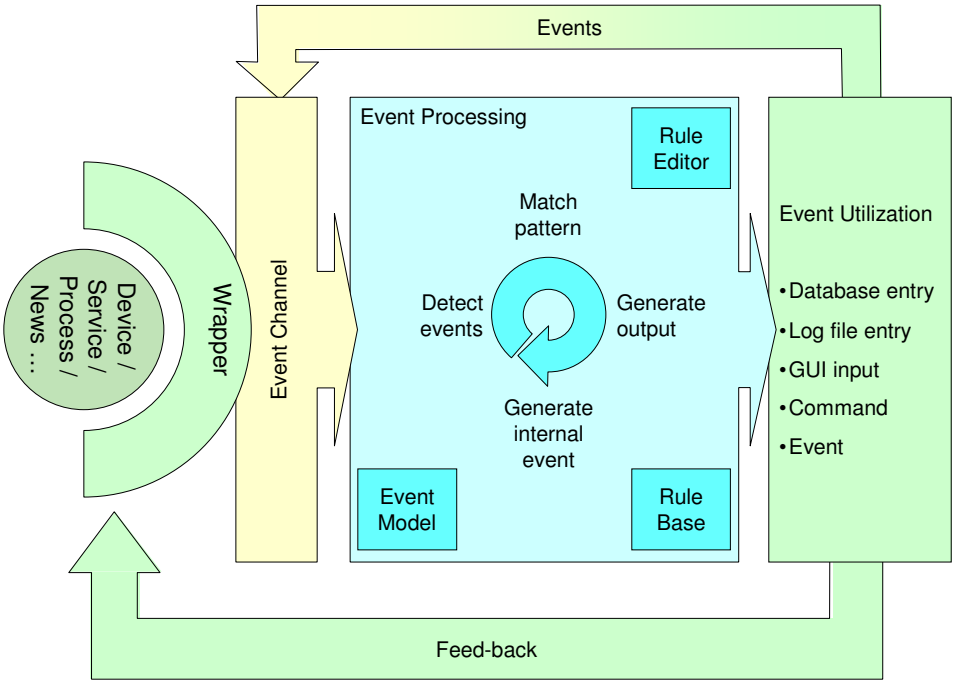

Figure 14: Data Flow in a CEP-based State Detection Environment

The following questions in terms fault-tolerant self*-based orchestration of these components need to be answered:

1. How to configure the device or the device wrapper service?
2. How to configure the (global) state detection / CEP service?
3. How to optimize the global state detection overall configuration?
4. How to repair the global state detection overall configuration in case of faults?

The first two questions will be handled in the following sections according to the self-configuration characteristics, Question 3 will be discussed in Chapter 3.2 in the focus of self-optimization, and the final question is subject of Chapter 4.4 regarding self-healing.

2.6.2 IoT Service Self-Configuration

There are two aspects of event source (= device or device wrapper service) configuration:

1) Configuration of the related event channel data:
   The event channel URL will be retrieved by and stored at the event source automatically.
2) Configuration of the event source policies:
   Data defining the event source behaviour, namely the frequency of event supply and the event specification in terms of event type, value range, value type, etc. These data need to be stored and utilized by the event source.

Configuration of the related event channel data

In order to retrieve a valid event channel URL, each IoT event source has to perform the proper operations provided by the resolution and look-up infrastructure which is described in [Bauer 2012], Chapter C2. We assume that the event channel URL is stored in some DHT component which can be addressed by a keyword as for example described in [de las Heras 2012], Chapter 4.4 and in [De 2012], Chapter 3.4 using a distributed hash table (= DHT). The DHT is one technological implementation option of the IoT Infrastructure for resolution, look-up, and discovery of services and associations. Its features are exploited for the self* functions described here. Having the infrastructure access operations in mind (see green bold typed tokens below), the event channel service publishes its description by executing a piece of code[3] like:

```
// code sequence executed by the event channel service
{
    ServiceDescription eventChannelSD = new ServiceDescription();
    eventChannelSD.setKeyword( "IoTEventChannel" );
    eventChannelSD.setURL( this.getURL() );
    this.serviceDescriptionID = insertService( eventChannelSD );
}
```

The service description object contains – among other data – its URL and a publicly known keyword by which it can be discovered. In the example above, the used keyword is just "IoTEventChannel" which is a string constant widely known in the IoT system. The method "insertService" stores the service description object of the event channel service including its URL into the IoT Infrastructure DHT[4].

[3] The code in this text is intended to give a taste of the functional behaviour and the data objects. It is not necessarily compile-save.

[4] At the point of time of writing this, there is a mismatch between Deliverable D1.3 (see [Bauer 2012]) and Internal Report D4.3 (see [De 2012]) concerning the names of some operators which obviously provide the same functions: what is named "insertServiceDescripiton", "updateServiceDescripiton", and "deleteServiceDescripiton" in D1.3 is just named

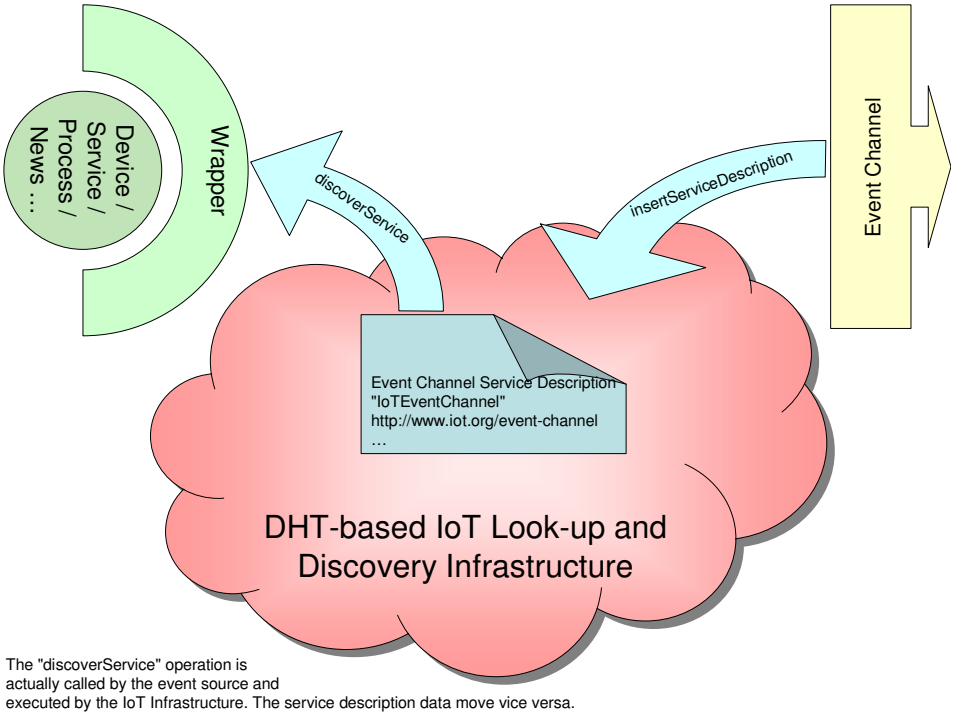

Figure 15: Storage and Discovery of the Event Channel Service Description

Now, the event source has to execute the method getEventChannelURL which may be implemented in the following way:

```
// declaration sequence elaborated by the event source

void notifyServiceResolution( SubscriptionID sID, ServiceURL serviceURL );
// the call-back function, implementaton see below.

void getEventChannelURL()
{
    ServiceDescription eventChannelDesc = discoverService( "IoTEventChannel" )[0];
    // or select any other event channel if the first one in the array is not ok
    if ( eventChannelDesc != null ) {
        this.setEventChannelSID( eventChannelSD.getServiceID() );
        this.setEventChannelURL( resolveService( this.getEventChannelSID() ) );
        this.setEventChannelSubscrID( subscribeServiceResolution(
                                        this.getEventChannelSID(),
                                        notifyServiceResolution ) );
        // passes the call-back function to the service resulution subscription
    }
}

void notifyServiceResolution( SubscriptionID sID, ServiceURL serviceURL )
{
    if ( this.getEventChannelSubscrID() == sID ) {
```

"insertService", "updateService", and "deleteService", respectively, in D.3. We will follow the naming convention of D4.3 which appears more credible (and shorter) though the insertion, update, and deletion operations refer to a service description object rather than to services.

```
            this.setEventChannelURL( serviceURL );
        } else {
            // check further subscription identifier and act accordingly
        }
    }
```

The method "notifyServiceResolution" is a call-back function which will be passed to the method "subscribeServiceResolution". It will be executed in whenever the URL of the related service designated by the service identifier has been modified. When it is executed, the new URL can be used to refresh the related data at the client site.

The methods "discoverService", "resolveService" and "subscribeServiceResolution" are provided by IoT-A Work Package 4. "discoverService" is used to retrieve an event channel service description object. For getting a more dense code, it is assumed, that a keyword is sufficient as a service description – otherwise, a service description object needs to be constructed which contains the keyword which might look like that:

```
ServiceSpecification eventChannelServSpec = new ServiceSpecification();
eventChannelServSpec.setKeyword( "IoTEventChannel" );
ServiceDescription eventChannelDesc = discoverService( eventChannelServSpec )[0];
```

Once the service description is found, the related service ID is extracted and subsequently used to resolve the event channel URL by applying the operation "resolveService". Finally, the service resolution is subscribed to get informed in case the URL is modified. Here, the "subscribeServiceResolution" method is called with the call-back function "notifyServiceResolution" as a parameter.

```
// simple code that provides a permanent up-to-date URL of the event channel service
{
    getEventChannelURL( notifyServiceResolution );
}
```

This is a simple demonstration of the operations provided by the lookup and resolution infra-structure. It helps event sources to find and refresh the URLs of the corresponding event channel where it delivers its events. In case, the event channel service will be cancelled completely, an extension of the above code sections will become necessary where a service discovery and a service discovery subscription will be utilized. However, the basic idea is about the same.

Configuration of the event source policies

The second aspect of the IoT Service self-configuration was the configuration of the event source regarding its behaviour policy. Just as in the case above, a DHT of the IoT Infrastructure eases the self-configuration process extremely. All an event source needs to know is the key-word of the data containing its behaviour instructions. In the following, we focus on configuring of the frequency of event deliveries. The other configuration data are to be handled in an equivalent way.

It is assumed, that there is an IoT system model as the result of the system design process which defines the default values and behaviour policies of the event sources in an appropriate event source model. The system model and particularly the event source model will maintain – among other data – the individual event delivery frequency of each of the event sources. These data need to be accessible using an appropriate interface. The following piece of code provides a specific data object which contains the individual behaviour configuration for each event source.

```
// code sequence executed at initialization time of the IoT system
forEach eventSourceModel in iotSystemModel {
    ServiceDescription sD = new ServiceDescription();
    sD.setKeyword( eventSourceModel.getID() + "_eventSourceConfiguration" );
    sD.setFrequency( eventSourceModel.getFrequency() );
    sD.setConfig( eventSourceModel.getConfig() );
    eventSourceModel.setServiceDescriptionID( insertService( sD ) );
}
```

Now the configuration data objects for the behaviour policies are stored in the DHT of the IoT resolution and look-up infrastructure according to a unique keyword which is constructed by appending the string constant "_eventSourceConfiguration" to the individual identifier of the event source model. The method "insertService" stores the service description objects into the DHT and returns the service description identifier.

In the next step, the event source needs to search its individual configuration data object from the DHT infrastructure by executing a code section like the one below:

```
// code sequence executed by the event source
{
    ServiceDescription mySD = discoverService( this.getID() +
                                               "_eventSourceConfiguration" )[0];
    if ( mySD != null ) {
        this.setFrequency( mySD.getFrequency() );
        this.setConfig( mySD.getConfig() );
    }
}
```

The construction methodology for the keyword and its individual identifier is all the event source needs to know to get in touch with the IoT system in order to extract its individual behaviour policy data by calling the "discoverService" method. Again, if the "discoverService" operation does not work directly with a keyword string parameter, the keyword needs to be inserted into a service specification object which will be used as the parameter value instead.

It needs to be mentioned here, that the self-configuration is processed just once, namely at the system start-up. A re-configuration of the system and especially of the event sources will require a system shut-down followed be a modification phase for the IoT system model data and a subsequent system restart, where the two code portions above are executed again. An approach of re-configuring the event sources at runtime is described later in Chapter 3.2 in terms of self-optimization techniques of the CEP subsystem for state detection.

2.6.3 State Detection Service Self-Configuration

The first aspect to be regarded here refers to the CEP-based service for state detection, designated as the event processing unit in Figure 14. This service needs to address the event channel, too. Therefore, a mechanism needs to be introduced that helps distributing the event channel URL. This mechanism of storing the service description into the IoT Resolution and Look-up Infrastructure – particularly into the DHT platform – is about the same as that for the event source, see Figure 16. In fact, even the event channel service description object may be the same in both cases.

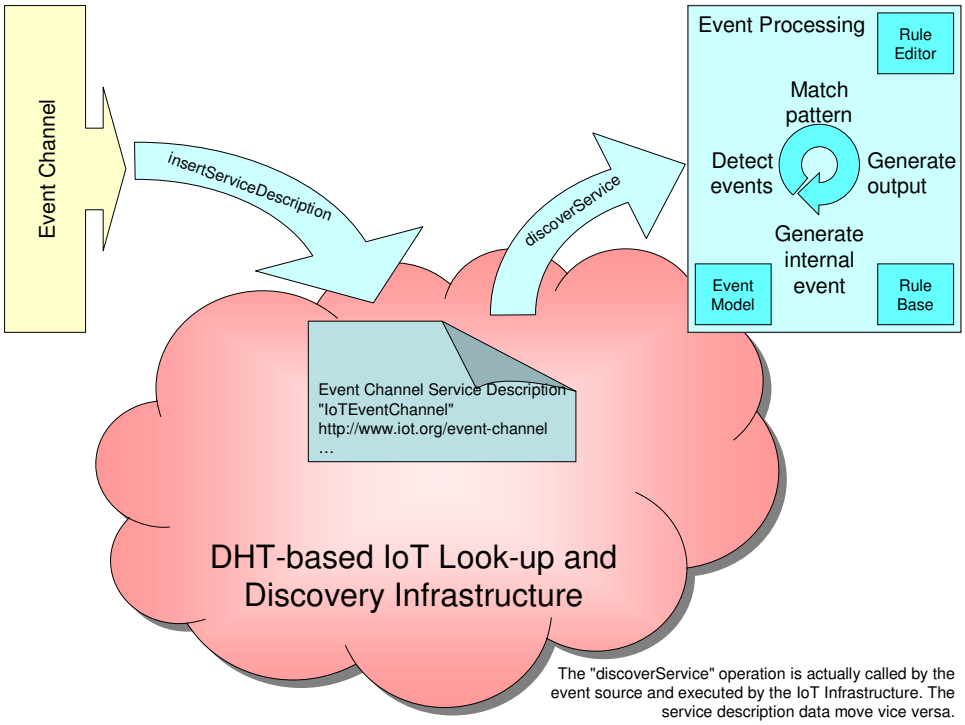

Figure 16: Storage and Discovery of the Event Channel Service Description

The other aspect refers to the subscription of the event types provided by the event sources via the event channel. Therefore, the Event Channel is assumed to provide a subscribe interface towards the event processing service as required in [Vidackovic 2010]. This feature can be utilized by both the event source and the event processing service site. As long as the event processing unit knows about the event characteristics needed by its rules, it can provide a much more precise subscription for events than just the event type.

In Chapter 9.2, some relevant event characteristics are listed like location and type of event source, event category (e.g. warning, error, alert), or event type (technical, system, business event). For example, a rule or a subset of rules from the rule base might require events from a particular location or from specific types of sources to determine the state of the related subsystem of interest. An event request could therefore specify the kind of desired event using an event descriptor object. On the other hand, event sources can provide their specific events accordingly. A match making component is able to detect fitting pairs of event offers from the event sources and event requests from the event processing service. Both components may utilize the same type of event descriptor.

As previously mentioned in Chapter 2.6.1 and in [Vidackovic 2010], the event channel is required to provide a publish-subscribe mechanism for event distribution. Thus, the events that match the desired event type are offered to the event consumers which subscribed for that event types. The event channel notifies the event consumer when a matching event was received. The event sources had to fetch the event from the event channel.

Here, the proposal is to extend the event description. Though the event type is highly important, other characteristics become essential when having the IoT in mind. This holds mainly for the event source location but also for the event source type, the event timestamp, or the event category. Therefore, the matching operation will become more than just the evaluation of the equation "event.getType() == desiredEventType". The new event matching will also regard range queries in time and space and might need ontologies for checking the event source types.

The event channel is the best place for the match making function as it services as the distribution platform for the events. Therefore, it is familiar with the event sources and the event consumers. Running the matchmaking component in the event channel makes this event channel a really powerful service in the IoT CEP system. It will have to maintain and match event requests and event offers in addition to event transport. The role of the event channel is depicted in Figure 17 which shows an example of exchanging messages for the event subscription.

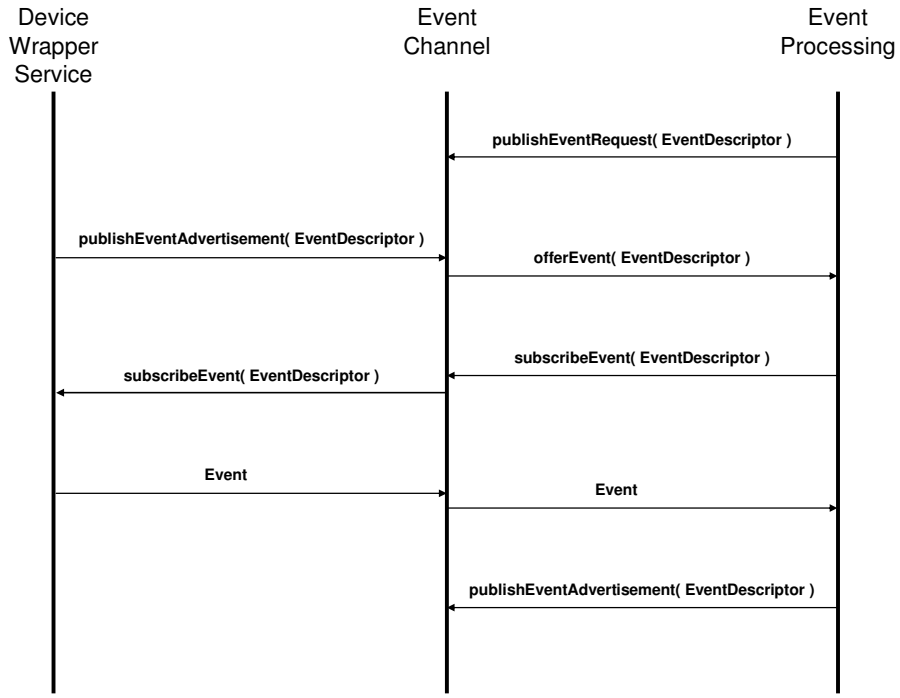

Figure 17: Example for the Publish-Subscribe Protocol for Event Distribution

The new idea is basically to collect event advertisements from the event sources and event requests from the event consumers. Event requests and event advertisements may arrive in any order. If two event descriptors match – of which one is from a source and the other is from a consumer – the related event consumer will receive an offer to which it may subscribe. In the following, events are directly transported to the consumer without and notification. An "unsubscribeEvent" operation will terminate the event delivery by the event channel.

The benefit is that events arrive faster because there is no message exchange for event consumer notification followed by an acceptance or rejection for each event. This process is moved into the match making operation which is executed once before events are delivered. Thus the new event channel concept will reduce the transport delays and also provide events more fitting to the consumers. Therefore, the event consumers – i.e. the event processors or state detection services – need lesser efforts for their internal event matching. In total, the event consumer are enabled to configure their event input more effective; the event processing proceeds more efficient due to the better fitting event that arrive from the event channel.

2.7 Conclusion

This section has given an overview about different aspects of self-configuration for IoT service orchestrations, service compositions and the services itself. Self-configuration deals with the automated inclusion of new IoT devices that are included into the system without human intervention in a plug-and-play manner. A demonstrator has been developed that showcases the extensibility of an IoT system by using self-configuration techniques. Introducing new devices and therefore resources allow extending the functionality of the system, but also the non-functional aspects such as quality of service and quality of information parameters have been considered in this section. It has been outlined how IoT services can describe themselves what configurations they support and how they can advertise such capabilities to potential service users in order to be offered by the IoT Service Resolution. It has been further explained how self-configuration is applied to service compositions that consist of more than one IoT service. The section has also given an overview on how self-configuration is applied for Global State Detection. For doing so a CEP subsystem for state detection makes fundamental use of the IoT service infrastructure with the focus on self-configuration.

3. Self-Optimization in IoT Service Orchestration

Self-optimization describes the process where a system adapts to its environment in a way that it can serve its purpose as good as possible. In IoT Service Orchestration, both Service Composition and Service Execution offer potential for self-optimization. This chapter has two main sections.

In the first section we identify the combinatorial optimization problems that need to be solved when massive amounts of distributed Services are orchestrated in order to work towards a larger task. The focus is on the minimization of resource (energy, bandwidth) usage by choosing the smallest possible set of Services such that the original requests can be satisfied. The Section also gives an overview of the existing approaches for solving these problems and points out gaps in the state of the art.

The second main section of this chapter is a continuation of Chapter 2.5.2. It describes a CEP-based IoT subsystem that is used for device re-configuration during runtime depending on the system states.

3.1 Self-Optimization for Massive Service Orchestration

3.1.1 Introduction to Massive Service Orchestration

In classic Service-oriented Computing, the overall service request is split into several tasks, and then one Service is selected for each of the tasks. The tasks are conceptually different, and so are the functional descriptions of the Services. If several Services of the same kind are used in one composition, then the reason for this is typically to improve the throughput, dependability, or some other non-functional property that could not be achieved by a single Service instance.

We claim that the IoT has a number of application scenarios where Service compositions deviate from this pattern. In problems of *Massive Service Orchestration*, the task is to build Service compositions from a large number of very simple atomic services. The atomic Services differ from each others neither in the kind of data they have as input and output, nor in the kind of sensing or actuation they perform. Their functional definition, however, is parameterized by one or more continuous parameters. These parameters describe what is has been defined in IoT-A as the *Service Area* of the Service – typically a geographic area, possibly enhanced by a temporal parameter if the Service is known not to be always available.

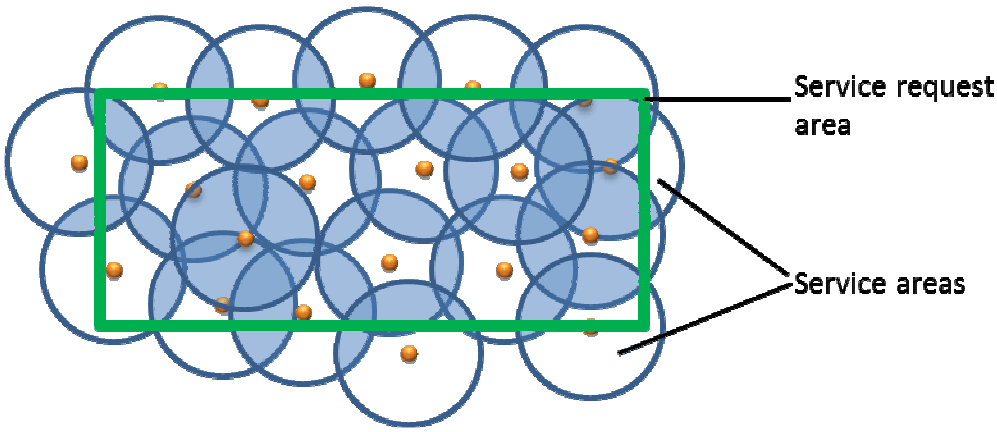

Figure 18: In Massive Service Orchestration. a set of Services needs to be selected such that the Service areas cover the Service request area.

A request in Massive Service Orchestration refers to a certain Service type and a certain Service request area. If the area is large and cannot be served by a single Service, multiple Services need to be selected such that the union of their Service areas contains the request area, as shown in the Figure above. The request area can have a temporal dimension, when the Service is required during a certain period of time. There are also cases where the request area is rather a set of points or lines than an object with a surface; a more detailed discussion of the different models can be found below.

In IoT application domains like Smart Cities, the request areas can be by orders of magnitude larger than the Service areas, which in turn leads to the necessity to orchestrate massive amounts of Services. Think for example of heat maps to be produced from temperature sensor readings in the whole city. Every single sensor delivers information about the state of the world only in its immediate surroundings, so, depending on the required level of details, a large number of sensors readings are required.

The naive way to react to a Service request of this kind would be to invoke all Services whose Service area overlaps with the request area. This might however cause unnecessary redundancy, especially when the density of available Services in the request area is high. In order not to waste communication bandwidth, energy and cost, one has to select a set of Services that is in minimum under the constraint that it can jointly serve the request, i.e., covers the request area.

3.1.2 Relation to self-protection

Making an optimal selection of Services turns out to be complex combinatorial problem and therefore should be automated. In other words, it is advisable that the self-optimization of an IoT Service middleware includes algorithms for setting up Massive Service Orchestrations with optimal or near-optimal resource usage.

Fault-tolerance, which is a way to achieve self-protection (see Section 5), plays a role in Massive Service Orchestration as an additional constraint the composition has to meet. For example, it can be required that every point in the request area is covered by two or more atomic Services, so that the overall Service can be delivered even when a number of atomic Services are faulty.

3.1.3 Optimization models

In what follows, we give a framework to categorize the variants of the optimal selection problem that appear in Massive Service Orchestration.

3.1.3.1 Fixed Service areas vs. request-dependent Service areas

In the above introduction to Massive Service Orchestration, each Service has been associated with a specific Service area. This Service area was fixed in the sense that the Service covers that area, regardless of how the request looks like. In the above figure, that Service area has been symbolized as a circle. However, this does not mean that circles are the only shape Service areas can have; they can in principle have any shape. The Service area of a fixed camera would for example be more accurately described as a triangle. From an algorithmic point of view, the shape of the Service area has an impact on how well optimal Service orchestrations can be computed. Most of the known results require that the shape of the Service areas is at least convex, i.e., any straight line segment that starts and ends in the Service area lies completely in it.

In the case of request-dependent Service areas, it is the request who determines the Service area of each Service. Although this is a very generic description, we believe that in almost all practical examples of this in the IoT domain the Service areas are circles, and the Service request simply specifies the radius of the circle. The circle radius represents the desired accuracy of the composed Service; the larger the radius the lower the accuracy requirement.

From a computational point of view, the request-dependence has an impact on the feasibility of preprocessing. Service requests can be handled faster when certain data about the Service areas (e.g. the intersection areas) is available. This kind of data can be pre-processed when the Service areas are fixed, but needs to be recomputed for each request defining its individual set of Service areas.

3.1.3.2 Area coverage vs. point coverage vs. line coverage

A typical use case of Massive Service Orchestration in the context of Smart Cities is the monitoring of city districts by sensors. Here the Service request area is a true two-dimensional area, so we speak of area coverage. In scenarios of point coverage, it is a number of point-shaped objects that have to be served. More specifically, the request is given as a set of points, and the goal is to select Services so that each of the points is inside the Service area of one or more selected Services.

Finally, line coverage problem instances are characterized by a line that needs to be covered by Service areas. Typical real-world examples are monitoring of roads or railway lines.

Apart from very special cases, point coverage problems are regarded as easier than area coverage problems. There exist techniques for converting area coverage problems into equivalent point coverage problems, see below.

Line coverage problems can either be treated as special cases of area coverage. Alternatively, each Service area can be reduced to the line segment it covers, which results in a one-dimensional area coverage problem to solve.

3.1.3.3 Single coverage vs. multi-coverage

As pointed out in Subsection 3.1.2, the Service request might specify that each point in the request area needs to be covered by two or more Services. In that case we speak of a multi-coverage problem.

The obvious reason for demanding multi-coverage is to obtain improved robustness against failure of individual Services. For example, when each point is covered by two Services, then the composite Service is guaranteed to work correctly as long as there are no two faulty Services whose areas overlap.

We remark that when the Service area is request-dependent, there are two independent ways for the user to increase the number of selected Services. One is to choose a smaller Service area, and the other is to increase the number of Services by which each point in the request area needs to be covered.

These two parameters are not completely independent. For example, whenever the request area is connected and needs more than one Service for its coverage, then a 2-coverage (not necessarily the optimal one) with circular Service areas of radius r can be obtained by computing a single coverage for radius r/3. This is because in the single coverage each point of the request area is covered by one radius r/3 circle and has a distance of at most r to some other circle center of the single coverage.

3.1.3.4 One-time coverage vs. continuous coverage and related mobility models

When the Service request asks for delivery of a snapshot of the current state of the world, or for a one-time actuation, then a coverage problem needs to be solved, then selected Services are triggered, and finally result is aggregated and returned to the user. In cases like this we speak of one-time coverage.

In contrast, we speak of a continuous coverage when the requested Service persists over a longer period of time, e.g. for monitoring an area and regularly reporting the current status to the user. If both the set of available Services and the request area are static, then any algorithm for computing a one-time coverage can as well be applied for continuous coverage. This picture changes radically when Services and/or the request area are mobile. Then the selection of Service constantly needs to be updated.

3.1.3.5 Single-tenancy vs. Multi-tenancy

In the context of Massive Service Orchestration, we speak of multi-tenancy when not only one service request has to be satisfied, but multiple Service request areas overlap. Of course the middleware can simply address each of these requests separately, but the opportunity (and challenge) is to save energy and communication costs by exploiting overlaps between the requests. For example, if two Service requests differ only in the request area and there is a large overlap between these areas, then the middleware can handle this like a single request with the union of the areas. The situation becomes more challenging when the requests differ e.g. in terms of the Service area radii, or if one request is continuous and the other one is a one-time request.

3.1.4 Application scenarios

We describe a number of Application scenarios and categorize them according to the framework introduced in the previous subsection.

a) Persons can subscribe to a Service where they receive a warning whenever a radiation hot-spot is in their immediate surroundings. The location of the person is derived from the radio cell.
In this scenario the Service areas of the radiation sensors are likely to be fixed. As the whole surrounding area of the person needs to be monitored, we have to solve an area coverage problem. Whether single coverage or multi-coverage is used depends on what the Service provider considers more appropriate; both is thinkable. The Service request is clearly a continuous one, because the radiation level is continuously checked. As the user is mobile, so is the request area. Finally, a Service of that kind is likely to be used by many users, i.e. the potential optimizations that come with multi-tenancy can be applied.

b) The average temperature of a city district is recorded every 10 minutes to create statistical records.
A service of that kind could be offered with several levels of accuracy, in which case the Service areas are request-dependent. The request area clearly is a real geographic area, not a set of points or a line. The choice between single or multi-coverage could also be left to the user. Although the Service request is continuous, neither the Service areas nor the request area are mobile, which means that algorithmically this case can be treated like a one-time request. Also here multi-tenancy is possible.

c) An operator of public displays sells a Service where it is guaranteed that the customer's content (information, advertisement) is scheduled so that for any point in the city at least once per hour the content is shown at some public display in distance of at most 1km. This condition translates into each public display having a fixed circular Service area of radius of 1km. In the described scenario the request area is always the whole city,

although it is also thinkable to offer the Service for user-definable city districts. Also multi-coverage could be additionally offered, which here means that the content is displayed on two distinct screens near each location in the city. The Service request is clearly continuous here. Algorithmically one could compute one Service cover and invoke it every hour, but also more flexible approaches are possible. There might be multiple customers using the Service, but overlaps cannot be used to save resources here, as the scenario is about an actuation Service.

d) The route of a bus line is monitored by fixed and mobile noise sensors (microphones which report only the noise level due to privacy reasons) in order to detect accidents and traffic jams, so the buses can be redirected if necessary.
Here the service area of each microphone is fixed. The request area is a line; note that here the line is not straight in general. Multi-coverage is advisable in this use case, as individual microphones might from time to time report noise because of a nearby event that is not related to traffic (birds, wind, etc.). The Service request is continuous, and as some of the Services are mobile this has to be taken into account by the middleware. When multiple bus lines are monitored like this, then one can benefit from the cost-saving potential of multi-tenancy.

3.1.5 Overview of existing algorithmic work and applicability to Massive Service Orchestration

3.1.5.1 General theoretical background: Set Cover

All Service coverage problem versions can be described as variants of the general Set Cover Problem. Instances of the latter problem are characterized by a ground set U and a number of subsets U_1, … , U_n of U. The objective is to select the least possible number of subsets so that their union equals U.

A well-known and straightforward algorithm for this problem is the so-called Greedy approach: always add the subset to the collection which contains the largest number of elements not yet covered. This algorithm can be proved to guarantee an approximation factor of log |U|, where |U| is the cardinality of U. This means that the Greedy algorithm always computes a collection of subset at most log |U| as large as the optimal solution to the respective problem instance. Although there are indeed problem instances where the Greedy algorithm performs exactly that bad, these instances are artificially constructed and on most real-world instances Greedy performs better than this lower bound.

One might be led to think that there are more sophisticated algorithms for the Set Cover Problem which have better performance guarantees. One solution would be to enumerate all possible covers and then choose the best one. This is however computationally infeasible, as the runtime required for this method would be exponential in the number of given subsets.

There is indeed strong theoretical evidence that indeed there is no efficient algorithm for Set Cover which can guarantee an approximation factor substantially better then log |U|; the existence of such an algorithm would imply that P=NP. This has been proven in the 90s as a consequence of a major breakthrough in complexity theory [Vazirani 2001].

As the algorithmic and complexity results on the general Set Cover Problem do not yield efficient algorithms for optimally solving the combinatorial problems that come with Massive Service Orchestration, the only hope to obtain near-optimal solutions is to exploit the specific geometric structure of those problems. In the subsequent section we give an overview of the existing work going into that direction. Most of the literature has studied geometric coverage problems in the context of sensor networks.

3.1.5.2 Existing results

An overview of the geometric covering problems that are studied in the sensor networks literature can be found in [Thai 2008]. Both area coverage and point coverage have been previously addressed.

As for the area coverage problem, a special case has been shown to be efficiently solvable to the optimum in [Sun 2007]. Here both the request area and the Service area are circles of the same size, and the request area contains all Service area centres. The authors give an algorithm solving this problem to optimality in time n log n, where n is the number of Services.

Unfortunately, less specific – and more realistic – classes of problem instances turn out not to be optimally solvable in polynomial time (unless P=NP). In [Ko 2011] it is shown that the area cover problem is NP-hard even when the Service areas are unit size discs and the Service request area is connected.

At least testing the feasibility of Service covers seems to be computationally tractable. In [Huang 03], an efficient method to test whether multi-coverage is achieved by given set of disks is presented.

Still considering the area coverage problem, Xu et al. have given algorithms with arbitrarily good approximation guarantees. Their method also applies to the multi-coverage problem. In addition, it takes into account the possibility to adjust the Service area of individual Services, where larger areas lead to increased cost [Xu 2008]. The drawback of their method is that its runtime is dependent on the geographical density of available the Service set. The more the Service areas overlap, the longer it takes to compute the solution.

The problem of covering points with geometric areas is also known as the discrete geometric covering problem. The process of transforming instances of the area coverage version into instances of the point coverage version is therefore also referred to as discretization. Probably the most intuitive approach to discretize the area coverage problem is to place points in the request area on a grid. The finer the grid is, the more accurately the discretized version represents the original problem. One such method has been given in [Xu 2008]. The disadvantage of the grid method is that it is difficult to give a guarantee that an optimal solution to the discrete version is a feasible solution to the original problem instance. Furthermore, the number of grid points is larger for finer-grained grids – so the problem size blows up with the accuracy of approximation.
An alternative discretization method has been given in [Ko 2011]. This method returns a number of points that is quadratic in the number of Services whose areas intersect the request area, and it can be guaranteed that any solution for the resulting discrete instance is also a cover of the request area.

The point coverage problem is computationally difficult as well. This follows from the complexity of the area coverage problem and the fact that area coverage can be reduced to point coverage [Ko 2011]. The good news however is that the problem can be approximated arbitrary well by a simple local search method when the Service areas belong to a class of geometric objects that includes discs [Mustafa 2010].

There is a slightly stronger version of the multi-coverage problem, where it is demanded that the selected set of Services can be partitioned into some number of completely independent single covers. Clearly, any solution that can be decomposed into k single covers is a k-cover, but the reverse implication is not true in general. However, for certain classes of Service area shapes it can be shown that any k-cover can be decomposed into k/y single covers, where y is some constant depending on the shape of the Service area. The decomposability of multi-covers for points has been investigated in [Gibson 2010].

A dynamic service coverage problem has been addressed in [Weinschrott 2011]. Here the problem is motivated by a participatory sensing application, where participants move along a road network and the goal is to use their sensors to detect objects that are as well mobile. The authors propose both a centralized and a distributed algorithm to select the participants whose sensors are switched on or off. Due to the restriction to roads, the work basically solves a coverage problem with one spatial dimension, but takes also time into account. The objective to detect mobile objects requires a different kind of reasoning as compared to the standard objective of just monitoring a region over a certain time period.

3.1.6 Open research challenges in Massive Service Orchestration

Massive Service Orchestration comes with a number of research challenges regarding algorithms for selecting the Services that become part of the orchestration. The basic covering problems have been addressed, though not completely solved, by the sensor networks community.

The IoT adds a number of new dimensions to these problems. Firstly, the mobility of Services and the mobility of requests have only been addressed for very special cases in the existing literature. Secondly, the co-existence of more than one request and the corresponding potential for optimization by Service re-usage has not been considered at all to the best of our knowledge. This is due to the nature of wireless sensor networks; they are typically deployed for one specific purpose.

In this section on Massive Service Orchestration the focus has been on the combinatorial problems that need to be solved for efficient Service Composition, including the maintenance of compositions during runtime. However, also the simultaneous communication between masses of Services during Execution is far from trivial. Here the main problem is scalability, as a single network node cannot communicate with arbitrary numbers of Services.

3.2 Self-Optimization for Global State Detection

A CEP-based IoT subsystem for (global) state detection may be used to control the IoT system particularly in case of a critical situation. As already pointed out in Appendix 9.3, CEP-based applications focus mainly on detecting irregularities of various kinds. Consequently, the CEP-based system may initiate appropriate countermeasures based on its output, probably in terms of commands.

The state of a system is a collection of the states of its single components – thus it is a complex data structure that needs to be handled. It is reasonable to subdivide the states in abstract groups. A simple example refers to aggregating a green, a yellow, and a red state which express a regular, a critical and an alarm situation, respectively. According to this example, it is obvious that only few events need to be regarded as long as the state is green and the situation is regular. Thus, the event frequency might be low. If there is a system transition to the yellow state referring to a critical situation, the user might prefer more detailed information at a higher rate. This implies that the event source needs to produce its events more often than originally determined by the default configuration.

It needs to be mentioned that the aggregation of events in order to detect any of the given coloured states is already a result of applying rules in the CEP service. The rule that produces the state information may delete its original input events or store them in a log file without bothering the user, as long as it detects a green state.

Let $\vec{x}(t_i), i \in N$ be a sequence of event vectors $\vec{x}(t_i) \in \vec{X}$ at given discrete points of time as for example $t_i = t_0 + i \cdot \Delta t$, where each vector component x_j is uniquely related to its event source a_j. Furthermore, x_j is the most up-to-date available event generated by a_j. Each event vector $\vec{x}(t_i)$ is evaluated by executing the state rule $\sigma : \vec{X} \rightarrow S$ that maps an event vector $\vec{x}(t_i)$ to a system state value $s_i = \sigma(\vec{x}(t_i)) \in S$, where the set S of states is finite with an order relation '<' such that $s < s'$ if state s is less critical than s'. According to the example above $S = \{"green", "yellos", "red"\}$ with $"green" < "yellow" < "red"$.

Furthermore, there is a partial function $\phi : \vec{X} \rightarrow \vec{X}$ for filtering events. ϕ maps an event vector onto itself, if the state is not "green". For other state, it is undefined which means that the events are deleted. Thus: $\phi(\vec{x}) = \begin{cases} undef & if \quad \sigma(\vec{x}) = "green" \\ \vec{x} & otherwise \end{cases}$. So, ϕ prohibits the CEP service from being flooded with events though no critical or alert state was currently detected.

Let $\vec{x}(t_{i+1})$ be the event vector succeeding $\vec{x}(t_i)$ such that $\sigma(\vec{x}(t_i)) < \sigma(\vec{x}(t_{i+1}))$. In this case, a higher frequency of events is desired which means that – from a functional point of view – the factor Δt that determines the duration between two event evaluations needs to be reduced. However, in practice the events are already delivered to the event processing service before they are analyzed. The loads of the network, the event channel, and event sources would not be reduced. Therefore, the reduction of Δt needs to be performed where the event comes into being.

It was already described how self-configuration of the event sources proceeds. With the self-optimization as a run-time version of self-configuration, the mechanisms are about the same as in Chapter 2.6.2. With the rules σ and ϕ from above, it is easy to trigger a re-configuration procedure spontaneously. The trigger function $\tau : S \times S \rightarrow \mathrm{A}$ will be used to produce an action from a command suite A initiated by a state transition. For two succeeding states $s = \sigma(\vec{x}(t_i))$ and $s' = \sigma(\vec{x}(t_{i+1}))$, the trigger function maps in the following way:

$$\tau(s, s') = \begin{cases} noOperation() & if \quad s = s' \\ increaseFrequency() & if \quad s < s' \\ decreaseFrequency() & if \quad s > s' \end{cases}.$$

The increase and decrease frequency procedures are about the same; they are defined for the IoT control site in the following way (just as in the section on event source self-configuration in Chapter 2.6.2):

```
// code sequence executed in case of system state transition
forEach eventSourceModel in iotSystemModel {
    if ( related eventSource is relevant for the state transition s -> s' ) {
        ServiceDescription sD = new ServiceDescription();
        sD.setKeyword( eventSourceModel.getID() + "_eventSourceConfiguration" );
        sD.setFrequency( adaptFrequency( getSystemState() ) );
        // here the new system state dependent frequency is determined
        sD.setConfig( adaptConfig( getSystemState() ) );
        // maybe any other important configuration alingnments
        updateService( sD ) );
    }
```

```
}
```

The adaptFrequency method could be implemented by indexing a sequence of frequency values using the systems states: $s \in S \quad \Rightarrow \quad f := frequencyCollection[s]$.

It is, of course, a difficult issue to detect the device or devices which ultimately cause the state transition. Thus, there will be no further details on the selection of related event sources that are relevant for the state transition $s \rightarrow s'$.

On the event source site, the availability of a new configuration including a new frequency has to be detected. Again, the subscription mechanism of the IoT Resolution and Lookup Infrastructure helps avoiding polling for that situation. Similarities with the code for event source self-configuration in Chapter 2.6.2 are obvious:

```
void notifyServiceLookup( SubscriptionID sID, ServiceDescription serviceDesc );
// the call-back function, implementaton see below.
…
{
    this.setServiceLookupSubscrID( subscribeServiceLookup(
                                        this.getServiceID(), notifyServiceLookup ) );
        // passes the call-back function to the service discovery subscription
    }
}
…
void notifyServiceLookup( SubscriptionID sID, ServiceDescription serviceDescripiton )
{
    if ( this.getServiceLookupSubscrID() == sID ) {
        this.setFrequency( serviceDescripiton.getFrequency() );
        this.setConfig( serviceDescripiton.getConfig() );
    } else {
        // check further subscription identifier and act accordingly
    }
}
```

The summary of the previous sections is that the self-optimization scenario for the CEP subsystem for state detection can be implemented easily by utilizing the IoT Infrastructure for service resolution which is under development in IoT-A WP4.

3.3 Conclusion

This section has focused on the self-management capability of IoT services with respect to optimization. Existing service orchestrations are able to be adapted to changes in the availability of IoT services. The original service request is used as the goal that needs to be optimized. It has been outlined how the service orchestration and service composition can be modified in order to provide a service that is optimal for the user with the currently available services. The IoT infrastructure for resolution, look-up, and discovery provides a substantial instrument for managing service organization functionality as well the CEP subsystem services particularly in the focus of self-optimization.

4. Self-Healing in IoT Service Orchestration

In this section it is described how services already resolved and bound to applications can be replaced by others due to connectivity loss caused by mobility or faulty resources. This self-healing functionality makes use of WP4's resolution frameworks that takes care for monitoring the availability of services and sending out notifications about resources appeared or disappeared in the IoT system.

4.1 State of the art in self-healing in IoT and SOA systems

The architecture developed during the ICT-FP7-project 'SENSEI' as depicted in Figure 19 proposed a solution that leaves the service orchestration process to the central functional component *Semantic Query Resolver*, but forwards service requests together with the services orchestrated upon request time to another functional component called *Execution Manager* [Strohbach 2010]. The *Execution Manager* monitors service orchestrations for longer lasting service requests, e.g. subscriptions to Resources. If the orchestration originally composed breaks the *Semantic Query Resolver* will be triggered again with the original service request to find a replacement for the broken service. This approach makes the service orchestration stateless since it is not concerned with monitoring service orchestrations for subscriptions over time and therefore contributes to better scalability. Delegating this functionality to another central component as proposed with the *Execution Manager* can still be seen as a risk of performance bottle neck and a single point of failure.

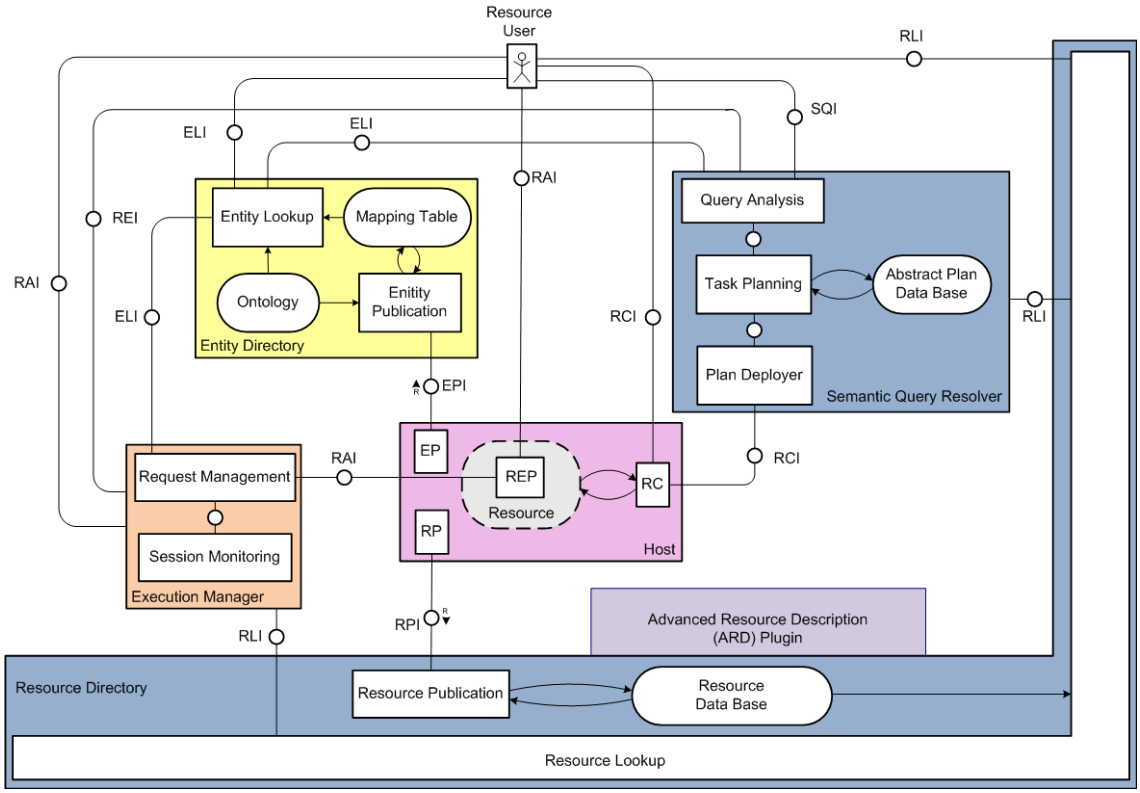

Figure 19 SENSEI Architecture [Strohbach 2010]

4.2 Replacing failed services in IoT Service Orchestration

The following sequence diagram shows how IoT services are replaced by the service orchestration with a minimum of required user interaction to repair the service orchestration. Self-healing is useful for service interactions that last over a period of time, like subscriptions to services. The service will then send notifications to the service user that has provided the service with a notification callback address. The service keeps sending notifications to this callback address as long as it is in a working state. As soon as the service fails no more notifications arrive at the service user. In case the notification period is long the user might not realise that a service has failed in the meanwhile though. The assumption is that a failing service will be detected by the IoT Service Resolution FC [Nettsträter et al. 2012] so that the Service Orchestration FC can get notified about the failed service. If an IoT service is associated to an VE the respective Association becomes invalid and requires repairing as well. In case of VE Services the VE Resolution gets notified about invalid associations. A prerequisite to get notifications about failed services and associations is a subscribe request to the respective resolution component developed in WP4:

```
subscribeServiceLookup
subscribeServiceDiscovery
subscribeAssociationLookup
subscribeAssociationDiscovery
```

An example for Self-Healing of service orchestrations is illustrated in the following sequence diagram:

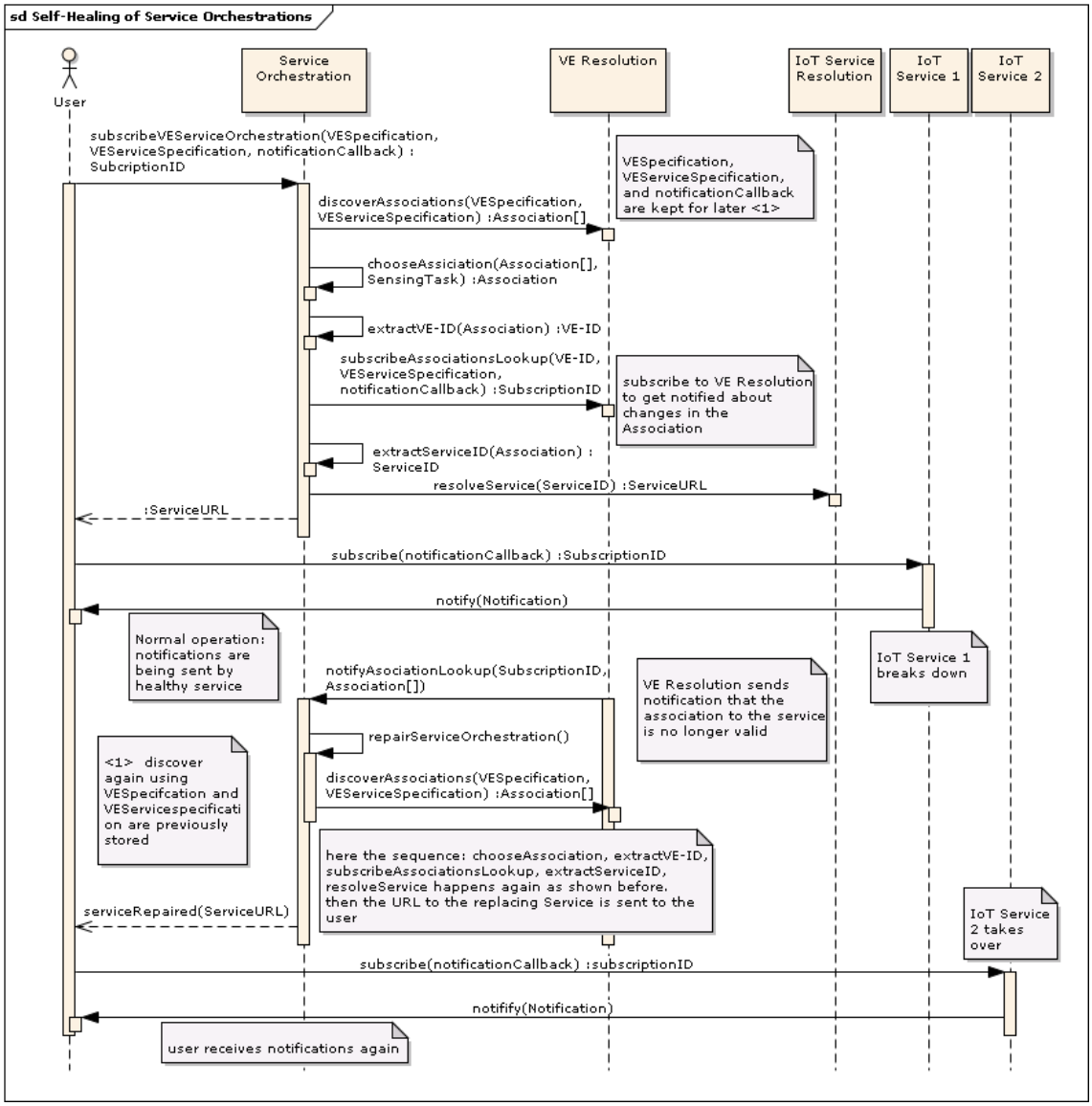

Figure 20 Self-Healing of Service Orchestration

4.3 Repairing failed service compositions

In this subsection an approach is presented that repairs service compositions without human user interaction. To achieve this, the Service Composition FC keeps track of the atomic services that are assigned to composite services. If one of the atomic services becomes unavailable a replacement service will be looked for. If a suitable service can be orchestrated the composite service will be repaired without user interaction. The detailed process involving functionalities of the IoT Service Resolution is illustrated with a sequence diagram in Figure 21.

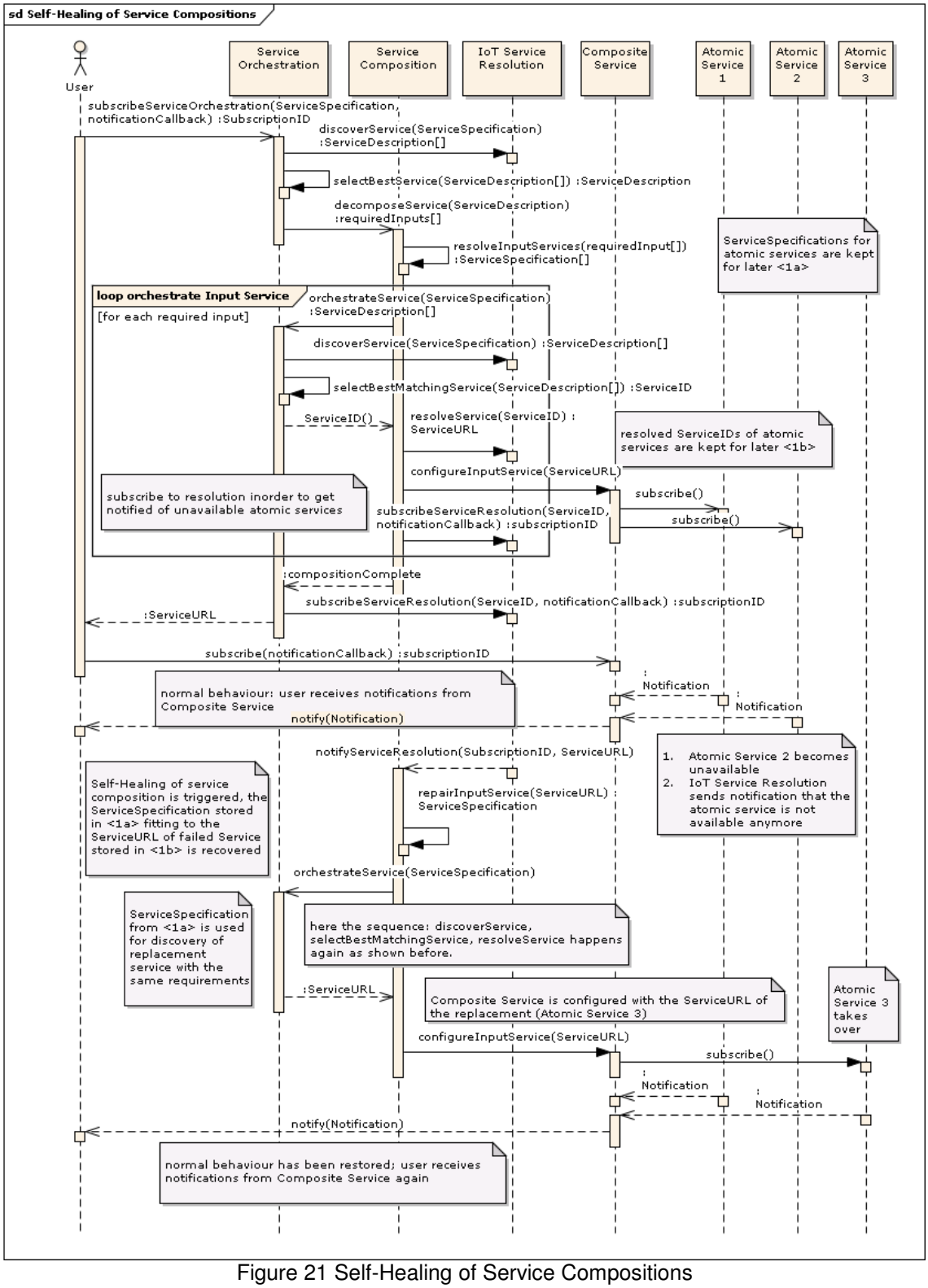

Figure 21 Self-Healing of Service Compositions

4.4 Self-Healing for Global State Detection

The CEP subsystem for IoT depends strongly on the availability of the events. Problems may occur, if events do not reach the rule-based CEP services which are responsible for the state detection. The loss of events may have several reasons:

- The event channel works faulty such that events get lost;
- The event channel itself breaks down;
- The devices and/or the device wrapper service break down.

In the following, we will focus on the second and third point. The first point is caused by internal faults which may be caused by little robustness or any other internal deficits.

The loss of events caused by break-downs of either the event channel or the event source can be treated equivalently as described in the following chapter.

Beside the break-down of the event source, a faulty working event source is an even more challenging situation. In this case, the repair proceeds in the same way as for other services, but the detection of that event source is tricky. This issue will be discussed in Chapter 4.4.2 under the topic of anomaly detection.

4.4.1 Drop-out Repair

The break-down or dropout of a service is usually detected when the utilization of the service is unsuccessful. In the context discussed here, services like the event channel and the event source deliver data; they are not really utilized. Therefore, the absence of events is not really an indicator of a dropout of the respective services: there might not just any events produced.

The useful indicator in case of the event source dropout is the event delivery frequency. As described in Chapter 2.6.2 and particularly the configuration of the event source policies, the event delivery frequency is a value taken from the system model date for device initialization. Thus, the frequency value is assumed to be a realistic and reasonable value. The basic idea is to take the frequency value for a device as the time-out value for the dropout detection. If no specific event arrives within a period of time, which is defined by the device-specific minimum frequency, a dropout of the related event source can be assumed.

If there are no events at all from any event source, there may be a complete break-down of all related devices. However, there is quite a high probability that the event channel break-down is the reason.

In any case, the solution is to start a "resolveService" operation call. This operation from the IoT service resolution interface provides the caller with the URL of the service addressed by the service identifier in the parameter:

```
String url = resolveService( serviceID );
```

The URL may be used to extract the ip-address of the service which is again used for a ping command. If the service does not reply correctly, the ping result indicates this and the caller is then sure about the dropout.

Alternatively, a service specification object that specifies the service of interest can be used to start a service subscription:

```
serviceDiscoverySubscriptionID = subscribeServiceDiscovery(
                                        ServiceSpecification, notificationCallback );
```

The notification call-back will be executed whenever new fitting services are available:

```
notifyServiceDiscovery( SubscriptionID, ServiceDescription[] );
```

By means of the array of service description object, the caller will be able to detect also services that are no longer available.

4.4.2 Anomaly Detection and Repair

Anomaly detection is wide field of research. In the following sections, an overview is given of what are the problems and which potential solutions are available. This Chapter is mostly base on the survey on anomaly detection by [Chandola 2009]

The basic problem of anomaly detection is the notion of what is normal and what is not. In the context of state detection for IoT systems, the state space is defined by the individual states of all devices and components of interest. According to Chapter 3.2, the state space is $\vec{X}$ and the subset of $N \subset \vec{X}$ which defines the normal states is given by inverting the state rule $\sigma : \vec{X} \rightarrow S$ in the following way: $N := \{\vec{x} \in \vec{X} \mid \sigma(\vec{x}) = "green"\}$, assumed the abstraction of states is given by $S = \{"green", "yellos", "red"\}$. In this case, a point anomaly can be defined because the values of $\vec{x}$ determine the state of the system exclusively.

Contextual anomaly refers to divergence of state vectors that are close – where close may refer to closeness in time, closeness in space, or logical closeness. To identify the device that behaves irregularly in terms of time, the difference $\vec{x}(t_{i+1}) - \vec{x}(t_i)$ helps to detect the component x_j with its associated device a_j, that diverges more than the others. These two types of anomaly – among others – are described in more detail in [Chandola 2009]. In the following, some excerpts of that paper are presented which are of importance for IoT.

The mayor problem of anomaly detection is caused on the evolution of normal behaviour. The system may change its state fundamentally but still remain in a normal state. Other problems refer to:

- Borders between normalcy and anomaly may be fuzzy due to noise data in the events
- The notion of anomaly is different for different application domains and sub-systems
- Selection of reference data for training or validation that describe system normalcy
- The difference between normal noise and anomaly of data

Usually, there are data to define normalcy which may be used as training data for learning systems or for comparison operations in rules. If there are no such data, the system and its components are assumed to work well in general and only very few components are faulty. Otherwise, no regular states can be detected automatically as the system behaves in a chaotic way.

The two main types of faults in the context auf IoT refers to "wear and tear" of mechanical components and to sensor anomaly.

In case of a defect caused by continuous usage, the analysis refers to time-dependent, probably streamed data coming from the devices. The problem here is that the anomaly detection needs to be executed permanently in real-time. Also, the transition from normal to anomaly may proceed very slowly such that it is difficult to decide whether the overall state is ok or not. In this case, reference data for normalcy are extremely necessary. With these date, the detection of an irregular event source and its related device is quite simple. Besides rule-based system, statistical models, neuronal networks, and spectral analysis techniques are useful.

In the case of detecting sensor anomaly, the main problem is to decide whether there is a faulty sensor or a highly outstanding situation it senses. Data that are available are mostly streaming data from distributed sources like audio, video, or others that arrive continuously but at discrete points of time. Usually, there is some noise added to the values due to environmental reasons.

The anomaly detection needs to proceed in real-time. A so-called lightweight anomaly detection may be located on the sensors just as the on-device resources in the IoT-A domain model. Since sensors may be mobile, the loss of events may be caused when the sensor enters a dead zone. In this case, a sensor dropout may be detected though it might be in good shape but without connectivity. The techniques recommended for sensor anomaly comprise distributed data mining, Bayesian networks, statistical modelling, spectral analysis techniques, and nearest neighbour-based techniques. The latter one refers to the already mentioned closeness of event sources in terms of time, space, or logic proximity. The nearest neighbour-based method can be expressed in a rule-based system where the events from close sensors are correlated with each other. In case of a dropout or a detected anomaly, the nearest neighbours are taken to calculate substitutional event values which may replace the original but faulty sensor values until proper countermeasures have been executed.

4.5 Conclusion

In this section it has been described in which situations in an IoT system a self-healing mechanism should be applied and how such a mechanism can be designed. The approach makes use of IoT-A's resolution infrastructure that keeps track of dynamic changes in the IoT service environment. Based on notifications from the resolution infrastructure the self-healing functionality can be triggered as explained in this section. For Global State Detection is has been shown that self-healing is also required for anomaly detection. For detecting misbehaviour the notifications of the supporting resolution framework are not enough since they only detect changes in availability or configuration of services, but not in their operation behaviour during service execution. CEP services as they have been described in this section are able to detect abnormal behaviour of services that are used as event streams. In this report it is proposed to incorporate the functionalities for monitoring service orchestrations composed by the Service Orchestration and Service Composition FC by the management component dedicated to the respective Resource as depicted in 2.1.2. This will reduce the risk of having a single point of failure that would lead to a loss of many service compositions in case of failure of a single component responsible for all Resources.

5. Self-Protection in IoT Service Orchestration (NEC)

5.1 Introduction to Self-protection

In this chapter we elaborate on concepts for orchestrating IoT services in such a way that the resulting composite services are robust against both the failure of individual services and malicious attacks.

In order to reason about the degree to which a system is self-protecting against attacks, one requires a precise definition of what an attack is. There exist a great variety of different attack types, and even the strongest model can never guarantee that it covers all possible attacks. For a simple example, think of an encryption system where it can be formally proved that without knowing the password the plain data can be retrieved only with extremely small probability, but the password itself is stored in an unsecured storage. Here the attack model in the formal proof does not take into account the attack of retrieving the password.

The same observation holds with respect to failure: Any notion of robustness against failure depends on the failure model. In this chapter, we do not make a distinction between the attack model and the failure model, but we also consider malicious attacks as a certain type of failure. For example, a client willingly behaving in an unexpected way in order to perform a malicious server attack will not be distinguished from a client behaving in an unexpected way because of say a software bug. Therefore, when speaking of failure in this chapter, this will always implicitly include malicious behaviour.

In the preceding chapter one particular approach for self-protection has already been described: By *self-healing* mechanisms, malfunctioning parts of the system are automatically repaired or replaced with healthy components. This approach is based on the assumption that every part of the system is in principle repairable or replaceable. Whenever this assumption does not hold, additional approaches for self-protection are required, so that even in the existence of faulty (and non-replaceable) parts the overall system performance remains at a reasonable level. Think for example of network protocols like TCP which include means to re-send lost or corrupted data packets. Here a faulty network connection is rightly considered to be neither automatically replaceable nor repairable, but the protocol keeps connectivity alive, and the faultiness just leads to a decrease in bandwidth.

This chapter's focus is on methods for keeping the overall system's performance at a reasonable level despite the (unrecoverable) failure of certain components. The desired behaviour is that there is a reasonable relation between the number and severity of faults on the one hand and the system performance on the other. This behaviour is also commonly denoted as *graceful degradation*, and the opposite of is a so-called cascade of failures, where a fault in one unimportant system part spreads throughout the whole system, causing an overall failure.

5.2 Self-protection in the context of IoT

There is not much that has to be said to motivate the importance of self-protection in the context of IoT in general and IoT Service orchestration in particular. Due to the distributed and mobile nature of the IoT, unreliability of communication is rather the rule than an exception. Furthermore, devices are typically resource-constrained and thus can often not implement the strongest security features, and, finally, the vast number of participants in IoT communication makes failure of some components even more probable. As IoT Services typically expose functionality of resources on real-world hardware devices, automatic repair or replacement is not an option in general.

In the following we give a number of examples explaining the concept of graceful degradation in the context of IoT.

a) A process for automatic traffic optimization uses a service which in real time returns the traffic flow through a certain street intersection, and this Service becomes suddenly unavailable. Instead, the process calculates an estimate of the traffic flow from the measurements from neighbouring street intersections. The accuracy of this alternative measurement is lower, but at least there still is real-time data the process can work with.

b) A street light is automatically switched on when vehicles or persons are detected to pass by. The success of this operation is monitored by a light sensor, so that in the case of failure nearby lamps can be switched on instead and so the person is not completely in the dark.

c) A history of previous sensor measurements is recorded. When a Service exposing the resource on a sensor fails, an estimation based on the history can be provided instead. Here again the reliability of the data is lower than before, but the alternative would be to have no data at all.

d) The heat map of a whole city district is computed from a large number of sensor measurements. A certain sensor might produce an extremely high or low heat value. By taking neighbouring sensors into account the overall system concludes that this sensor must be faulty and excludes it from the data provider set.

5.3 Failure models for IoT Service Orchestration

As this document is about Service orchestration, we do not discuss self-protection of individual Services, but rather lay the focus on the orchestration as a whole. Individual Services will be treated to a high degree as black boxes. This is compatible with the paradigm of Service-Oriented-Computing, where individual Services are fully characterized by their functional and non-functional properties, but *how* the Service is implemented is not relevant for anybody but the Service operator. In the IoT context, how the Service interacts with the real world through sensors and actuators is also part of the functional description. According to the black box approach, we assume that failures of individual Services is something that cannot be avoided from time to time, and it is up to the orchestration methodology to prepare for this situation.

Services communicate with each other via a network. A realistic and commonly accepted assumption in the context of Internet communication is that the network is *asynchronous.* In the context of Service failure models this means that there is no fixed upper bound on the transportation delay of messages even then the communicating Services are non-faulty[5]. What is assumed, however, is that all messages *eventually* arrive when sent and received by non-faulty participants. We remark that this does not exclude modelling of permanent failure of communication links: Whenever the communication link to a Service permanently fails, this can be treated like a failure of the Service itself. In this chapter we consider both synchronous with fixed upper bounds on message delays and asynchronous communication.

As for the Services, we do not assume any restriction on which messages are sent in case of failure. They can do anything from simply not reacting, flooding the network with random messages to performing malicious attacks. Faulty Services can even team up to coordinate their attacks. Traditionally, the only restriction is that the computational resources of failed services are limited, so they cannot forge signatures or perform other breaks of cryptography. This failure model is commonly known as *Byzantine failure* [Lamport 1982]. In addition, we have to assume some limitation of what failed Services controlling actuators can do; details will be given in Section 6.6.

The total number of failures that can occur is assumed to be bounded by some known constant δ. This assumption is in line with nearly all failure models in literature; see e.g. [Castro 2002].

[5] The notion of synchronous and asynchronous networks used in this chapter is common in the context of fault-tolerance, see e.g.[Ramasamy 2005]. An alternative would be to speak of realtime and non-realtime communication.

5.4 Self-protection and the replication approach

In Service Oriented Computing, self-protection against Byzantine failure is typically achieved by a combination of cryptography and replication. Cryptography is employed to protect messages from being faked by faulty Services, while replication is used here such that number of non-faulty replicas is guaranteed to be sufficient for deriving which of the returned messages is correct (e.g. by majority voting).

In general, these approaches cannot be directly adopted for IoT Service orchestration. In contrast to traditional Services, the nature of IoT Services is not to *compute*, but to interact with the real world. This conceptual difference has a number of consequences:

- Computing services can in principle be replicated arbitrarily often, and there are even techniques to achieve that each replica runs in a different environment and thus the correlation of failures is limited [Pu 1996]. IoT Services, in contrast, expose the capabilities of devices to interact with the real world. Direct replication of such Services would require replication of the (hardware) devices, which is typically not possible, especially in highly dynamic environments like when sensor readings from automobiles are made available by Services. To have more than one Service expose the resources on a devices is also not an option, as this would create a dependence between these Services, i.e., the failure of one makes the failure of the others more probable.
- When traditional Services are invoked, they can change their own state and return a response depending on the input received. IoT Services exposing actuators, in contrast, change the state of the real world. Performing the same actuation a number of times might not be desirable, see Section 6.6 for a more detailed discussion of this issue.

We describe a two-fold generalization of the replication approach, which we see as a first step to overcome these shortcomings. In the subsequent sections we then outline the requirements and approaches for achieving failure-tolerant sensing and actuation.

When a number of replicas of a traditional web Service are executed in parallel, each replica simultaneously serves two different purposes: Firstly, it executes the desired service, and, secondly, its result can be used to validate the correctness of other replicas. We split these two purposes and distinguish e*xecutor Services* from *monitoring Services*. Executor Services are invoked in order to achieve a certain effect on the real world and/or obtain certain information about it, whereas monitoring Services return results that can be used to check whether or not an executor Service behaves as desired.

The second generalization is the additional consideration of accuracy as a parameter. While traditional Web service replicas typically have the same characteristics regarding both functional and non-functional properties, ranking IoT Services according to their accuracy of real-world interaction is necessary for the definition of self-protecting IoT Service orchestrations. We remark that accuracy relates not only to measurements, but also to actuations: The more accurate an actuation is, the more exactly it serves its purpose.

With these generalizations of the replication concept, protocols for safe execution of sensing and actuation Services can be defined. Roughly speaking, each task is executed by the most accurate execution Service and monitored by a number of monitoring Services, which again are selected according to their accuracy.

In order to validate the feasibility of the above failure and replication model, we show how the examples of graceful degradation given in the introduction of this chapter can be described in terms of the generalized replication approach.

a) Using an aggregate of measurements from nearby street crossings can be interpreted as an alternative execution Service with a lower accuracy, which is used when the most accurate Service becomes unavailable.

b) The street light under consideration is the selected execution Service, while the light sensor is exposed by the monitoring Service here. The nearby street lamps can be modelled as an alternative execution Service with lower accuracy.
c) Like in a), estimates from history are an alternative execution Service.
d) In the heat map example, many individual Services exposing temperature sensors are orchestrated to provide the complete heat map. Here neighbouring temperature sensors of any individual temperature sensor take the roll of monitoring Services. As individual Services are not essential for the overall orchestration, there is no need for defining alternative execution Services here.

5.5 Fault-tolerant sensing

Under the assumption that there can be up to δ faulty Services, it is straightforward to see that at least 2δ+1 replicas of a sensing Service are necessary to guarantee fault-tolerance. Otherwise, if the number of replicas is smaller than that, δ faulty Services could team up to provide the same piece of wrong information. There is no way to find out whether these δ Services or the remaining ≤ δ Services are the non-faulty ones. As the information the Services provide comes from the real-world, there also is no way to use signatures or other ways to prove that the delivered information is correct.

This lower bound on the number of replicas holds for both the synchronous and asynchronous network model, as the described behaviour of δ faulty Services is possible in both cases.

In the synchronous case it is also straightforward that the bound is tight: With 2δ+1 replicas, simple majority voting can be used to determine what the true response is, while digital signatures or message authentication is employed to validate that the sender of the response is really the one it claims to be.

In general, the main problem with asynchronous networks is that there is no way to determine whether a Service not responding is faulty or if it just needs more time to respond. Simply considering such a Service as faulty is not feasible, because this would lead to the potentially wrong conclusion that at most δ-1 of the received responses can be faulty. Despite these problems, in the case of sensing Services it is still sufficient to have 2δ+1 replicas: As it is known that there are δ+1 Services which will provide the correct response after a finite period of time, one can simply wait until the reception of δ+1 responses that are compatible with each others.

Note that this method is also a reasonable approach in the synchronous case, especially when the upper bound on the response time is high.

As pointed out above, perfect replicability of sensing Services is an unrealistic assumption, so the used replicas potentially differ in terms of accuracy. Therefore, the notion of compatibility of responses is not limited to exact equality, but rather describes that the returned values are similar enough that they confirm each other. Inevitably, this also means that faulty Services which just slightly deviate from the truth will remain undetected. There also is no generic formula to define compatibility; the exact definition highly depends on the domain-specific characteristics of the sensors.

After having received δ+1 mutually compatible responses, the one with the highest accuracy is chosen. In terms of the preceding section, the most accurate non-faulty Service gets the role of the executor service, while the others take the role of the monitoring Services. Note that the selection rule implicitly assumes that the accuracy parameter of each Service is known a priori. If this is not the case, a faulty Service might claim to measure with high accuracy and then return an inaccurate value. But even if this happens, the faulty value must be at least compatible to the measurements reported by the non-faulty Services.

5.6 Fault-tolerant actuation

The byzantine model assumes that faulty Services can behave arbitrarily within the bound of their computational capabilities. As actuation Services rather interact with the real world than compute something, the notion of being able to do “anything” needs to be carefully taken into consideration.

We assume here that faulty actuation Services can arbitrarily influence the whole world. Just as computing Services are bound to their computational capabilities, actuation Services are assumed to be bound to the physical constraints of the devices they expose. There are of course actuators thinkable where a failure potentially has impact on large parts of the world (airplanes, power plants, etc.). However, such kind of Services have to be made fault-tolerant already on the level of the individual Service and so they play no particular role in the context of Service orchestration.

If we speak of Services that are able to set a certain set of properties of the real-world to a set of states, it is natural to assume that also faulty Services are able to do that – in an arbitrary way. Unfortunately, the existence of only one such faulty Service would make it impossible to do any fault-tolerant actuation on this particular part of the world, as any change for good could immediately be undone by the faulty Service. The only way to deal with such Services is to deactivate or repair them. As this rather belongs to the topic of self-healing, consideration of such Services remains out of the scope of this chapter.

Instead, we consider a more specialized failure model for actuation Services: We assume that they can return any response within their computational bounds, but, as for the physical actuation, they can only either do the requested actuation or not do it.

We define synchronous networks here so that not only non-faulty Services respond after a fixed time interval, but also that the real-world actuation is completed within this interval. Even faulty Services cannot delay their actuation further beyond this time bound, i.e., either they perform the actuation in time or they do not perform the actuation

Even in this restricted model it turns out that the availability of strategies for achieving fault-tolerant actuation depends on how actuation is assumed to take place. We categorize the different models of actuation as follows.

- *Idempotent Actuation* effectuates that after its completion a certain part of the world is in a certain state, regardless of what state it has been in before. This implies that applying the same actuation twice or more often in a row will have the same effect as applying it only once. Examples are actuations like “set room temperature to 21 degree” or “close the door”.

- *Additive Actuation* is a kind of actuation where applying the same actuation more than once will have an undesired the effect on the real world. Examples are actuations like “turn on some idle heater” or increase the room temperature by 1℃””. This model of actuation also includes pressing an on/off switch.

- *Reversible Actuation* can be undone by another kind of actuation. Note that this category is orthogonal to the others, i.e., both idempotent and additive actuation can either be reversible or not.

Idempotent Actuation is by far the simplest to handle. Under the assumption that at most δ Services are faulty, one just needs to trigger δ+1 replicas of the actuation Service. This works out both in the case of synchronous and asynchronous networks, and it is not even necessary to monitor the success of the actuation, as one knows that at least one of the δ+1 replicas is non-faulty.

In order to make Additive Actuation fault-tolerant, one inevitably requires monitoring Services that report whether or not the actuation has been successful. When the network is synchronous, one instance of the actuation Service is executed, and the execution is then examined by 2δ replicas of the monitoring Service. This number is necessary because if otherwise δ monitoring Services report an unsuccessful actuation one would not know whether these δ Services are faulty, or if the actuator and the remaining <δ monitoring Services are faulty.

To see that 2δ replicas of the monitoring Service are sufficient for fault-tolerance, note that δ monitoring Services are sufficient as witnesses for successful actuation. This is because otherwise both the actuator Service and these witnesses would have to be faulty, which is not possible by the assumption of at most δ faults. On the other hand, δ+1 witnesses of faulty actuation are sufficient. When the number of monitoring Services is 2δ, then both faulty and non-faulty actuation obtain the respective sufficient minimum number of non-faulty witnesses.

When the actuator Service has turned out to be faulty, another instance of it needs to be triggered and monitored. As there are at most δ faulty actuation Services, after δ+1 iterations of this process the actuation is guaranteed to be successful. Therefore, the total number of required Service replicas is bounded by 3δ+1. In practice one would of course not trigger monitoring Services again that have in former rounds already turned out to be faulty. However, this optimization does not change the total number of Services used for the whole actuation process.

We conclude the discussion of fault-tolerant additive Actuation for synchronous networks with two remarks. Firstly, after triggering the actuation it is important to wait until the maximum time it might require to complete has passed before executing the monitoring Services. Even if the actuator Service returns a message earlier that this time bound, it might be faulty and therefore delay the actuation. Then all monitoring Services could potentially report an unsuccessful actuation although the actuation is performed later. The second remark is that the approach described here is suitable for both reversible and irreversible actuation, as reversing is not used at all.

We finally turn our attention towards additive actuation for asynchronous networks. The fact that under this assumption there is no way to find out if a Service is faulty or just slow makes fault-tolerance a severe problem here. When an actuation Services has been triggered, it is not known when, if at all, the actuation will take place. If the actuation is not reversible, then the problem is unsolvable: triggering more than one actuation Service replica might set the world irreversibly into an undesired state, whereas only one actuation Service might never do the actuation. In the case of reversible actuation, the only solution would be to trigger δ+1 replicas of the actuator Service, continuously control the effect by monitoring Services, and revert any undesired effect by additional actuator Services – a highly nontrivial task given that also the monitoring Services and the reverting actuations have to deal with the asynchrony of the network. We are convinced that this is not feasible in practice and therefore come to the conclusion that, when synchronism of the network cannot be guaranteed, actuator Services should be designed to be idempotent.

5.7 Summary

In this chapter we have described a concept for achieving self-protection in IoT Service Orchestration. The concept is based on a generalization of the replication paradigm in traditional fault-tolerant Service Oriented Computing. The special characteristics of sensing and actuating Services imply special requirements on the number of alternative sensor Services and/or Services for controlling actuation. It turns out that sensing is easier and requires less replicas than traditional web Services (2δ+1 as opposed to 3δ+1). Actuation is even simpler in the case of idempotent operation (δ+1 replicas), but requires 3δ+1 Services for in the additive model. Even worse, in the latter model these services cannot all be executed in parallel, and the whole approach is only feasible in the case of a synchronous network with upper bounds on communication delays. As the latter is likely not to be achievable in the IoT, one of our main findings is that idempotent actuation is highly desirable at the Service level. Furthermore, there

seems to be no generic way to deal with arbitrary actuation of faulty Services at the Service orchestration level, so it is advisable to prevent this at the level of individual Services.

As a final remark, we point out that in practice there is no guarantee on the number of Services that can be faulty- the true value of δ is unknown. What has described in this chapter is a method to obtain a system that is robust against a certain number of failures, but from this method one cannot derive a system that is universally dependable The upper bound δ is a constant in the failure model, and the occurrence of more faults is not captured by the model anymore.

6. Fault Tolerance of IoT-aware Business Processes

This section examines how fault tolerance parameters can be expressed in IoT-aware Business Processes. Therefore we show that process metrics combined with defined and undefined fault handling is a suitable mean to express fault tolerance requirements to IT process elements such as services in order to predict /influence the fault tolerance handling during the deployment.

While the self-protecting of services of section 5 is based on a rather pessimistic model, where services can fail and then behave arbitrarily bad, this section assumes that the expected behaviour of services is known and can be expressed as a non-functional property.

Highly reliable business processes are the central nervous system of ERP applications. Through the integration of sensors and actuators via the envisioned Internet of Things conventional business processes must fulfil a certain degree of fault-tolerance. To meet the needs of the IoT and business processes, a possible initial step is to express these requirements measurable at business process level- concretely for the three central elements device, resource and service. Process metrics are a suitable instrument to formulate these requirements during the process modelling and to consider them during the resolution and execution phase of a business process. For the implementation these metrics can be expressed by using the extension mechanism of the extended and IoT-aware BPMN model 2.0 and therewith be demonstrated as part of the IoT4BPM Platform.

6.1 Introduction

Coming from the business process perspective, services are executed in a predetermined process flow. By IoT-aware business processes as defined in [Meyer 2012], we understand business processes that specify IoT services among further components in the process model. Therein IoT services are a particular type of potential process model components. With the help of process metrics and parameters resolution and execution requirements can be defined for the overall process models. This process metrics can be defined for each process element, such as for any IoT service. The problem appears that some sensible business processes shall not be fault-tolerant and shall be finished accordingly if a fault occurs while others shall be fault-tolerant. A specific metric can such be used for the predicted fault tolerance handling during the orchestration of IoT services.

6.2 Background

We focus on conservative modelling standards for complementing existing business processes by a straightforward IoT integration in order to maximize a potential industrial application and foster a widespread adoption similar to the adoption of core web technologies on the Internet. Thus purely academic approaches such as Petri nets or YAWL are excluded [Aalst 2005]. Business processes are represented by many methods and techniques. According to [Freund 2012] the most commonly used notation standards are BPMN [BPMN], eEPC [Ferscha 1994] and UML [UML] all providing graphical elements. Several meta-models have been developed to convert these graphical notations into executable ones such as Web Services Business Process Execution Language (BPEL) [BPEL 2007]. Due to the totally different concepts, conversions from BPMN 1.2 or EPC are still limited by many problems [Weidlich 2008]. The latest version of BPMN bridges this gap by providing a detailed xml serialization format serving as an interface between process modelling and execution by combining an end-user friendly notation with a detailed technical process specification. In terms of process element annotation, we focus on mapping different types of IoT conditions to a standard business process model, in comparison to [Schnabel 2012] who provided process facilitated process models in an end-user friendly way.

For the IoT-A research work we support BPMN 2.0 [BPMN], which was evaluated by [Meyer 2011] as the most IoT suitable state of the art approach, even though it still lacks features to completely cover the IoT domain. [D2,2] implements an IoT-aware Process Modeling Concept (IAPMC) enhancing BPMN 2.0 in order to close these gaps. The process model comprises a graphical and a-non graphical XML representation, which is an initial step for progressing with any further BPM lifecycle phase. Following [Freund 2012] who distinguishes between two modes of modelling a business process we aim to provide the second and detailed mode: the technical process model. In our case, the objective of the technical level is the implementation of the technically described details by using a process execution engine. Therefore, based on [Weske 2007] we consider the process model as a set of IoT and non-IoT process tasks having resolution and execution restrictions between these tasks. The central component and outcome of the process design is the IoT aware business process model serving as a clearly defined interface between the process design phase and resolution and execution phase.

To extend a process notation for the modelling of IoT processes, the main IAPMC components contain the following three concepts:

As a particular type of a process task the new subclass IoT Service is presented. The IoT Service is a fully automatic activity without any human interaction that directly starts after the process flow reaches the activity. This atomic activity can be further subdivided in device specific tasks such as actuation and sensing tasks. These tasks are designed to pass information to/from process internal devices, which cannot be accessed by the process execution engine. If this information is transferred, the process task is performed. Every IoT service may include a service description containing any number of parameters that describe the IoT service requirements for the service resolution and execution.

User studies have shown that an IoT Device is often presented as a separate lane / pool in the process model. This means an IoT device behaves like a fully-automated process participant who carries out a business role of an organization unit and is responsible for the assigned IoT and non IoT process tasks. Therefore we introduce in analogy to the recommendations for mobile phones of [Kozel 2010] a subclass to the BPMN Participant for the technical process participants IoT Device. However, an IoT Device in a business process doesn't only overtake the role of a participant, but in the same time the role of a performer which has further implications to the structure of the meta-model of the IAPMC.

Building on [Walewski 2011], for the IAPMC we consider as a Physical Entity any physical object that is relevant to a dedicated business process. Thereby, Physical Entities in a process can be relevant for one or more IoT Services of one or more IoT Devices. Business processes often go across departmental and operational borders. Physical entities can exist within these borders, but might also exist beyond it. We introduce the concept Physical Entity refining the approach of [Sperner 2011] as a passive process participant, having the peculiarity to be not responsible for process activities. This means that a Physical Entity behaves like a passive process participant, which corresponds to the collapsed participant view of BPMN 2.0. For all three core concepts IoT Service, IoT Device and Physical Entity requirements can be expressed such as defined by WP4.

6.2.1 Faults and Fault Tolerance

Definition faulty for business processes

A process component can be considered as faulty once its behaviour is no longer consistent with its specification. It's actually fault is defined as the deviation between its monitored behaviour and the wished behaviour of the process component. If the fault in the business process is not handled it can lead to unexpected results.

Definition fault tolerance for business processes

In analogy to the faulty definitions of section 5, a fault tolerant business process is a business process that enables the process execution engine to continue with its operation somehow even if some part of the business process execution fail. That means that the process execution proceeds with the current process or a modified resolved process although problems either with the hardware or software occur.

6.3 Problem analysis

For providing a problem analysis it has to be clarified, which components of an IoT-aware business process are likely to fail exactly. In an IoT-aware business process defined by D 2.2 single services, complete devices, its resources or even the whole process may fail. The problem appears that some sensible business processes shall not be fault-tolerant and be finished if a fault occurs while others shall be fault-tolerant.

We present a short example: Imagining a quality-based price process, where the price of a sensible good is adapted to its quality. The quality of the good is measured by sensors such as that the faulty measurement of the sensor would directly influence the price to which the good is sold and finally billed to the customer's credit card.

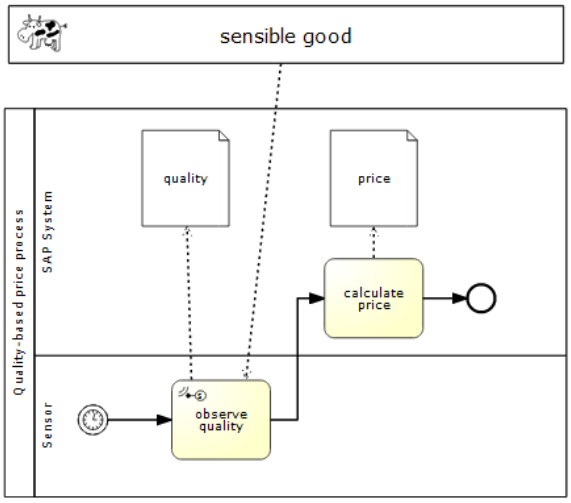

Figure 22: Process diagram of quality-based price process

The following table contains a short analysis of which IoT process components might be effected by faults.

IoT Process Component	Can be affected by faults	Example Process
IoT Device	Yes	Sensor
IoT Service	Yes	Observe qualiy
IoT Resource	Yes	Native software component "observe quality"
Entity	No	Sensible good

Table 2: IoT Process Components effectible by faults

Providing a fault-tolerant process design might be one option:

For a possible fault occurring when executing the sensing task one option might be that the process shall be ended completely. This case could be expressed like demonstrated in the following model.

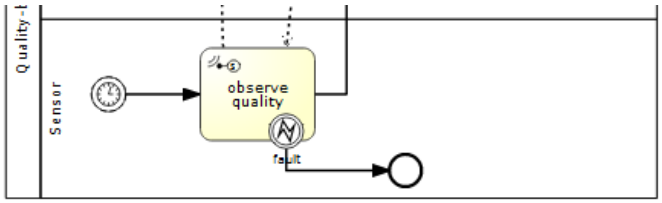

Figure 23: Fault event leads to process termination

Another option might be that the process shall proceed and start a dedicating fault handling sub process. This case could be expressed like demonstrated in the following model.

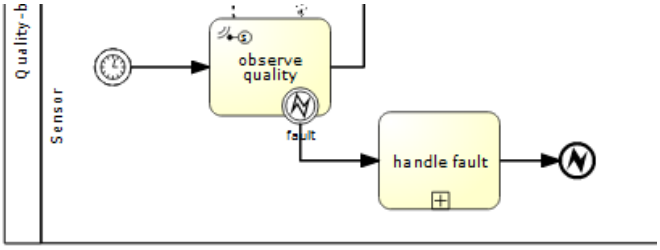

Figure 24: Fault events leads to defined fault handling process

The third option includes that even though a fault tolerant behaviour is wished no dedicated fault process flow is defined in the model. In this case, it has to be expressed that a certain process element or even the complete process shall support fault tolerance or not.

Wrapping up the a rule such as shown in Figure 25 could be applied to any occurring faults on top of business processes as foreseen by the rule definition:

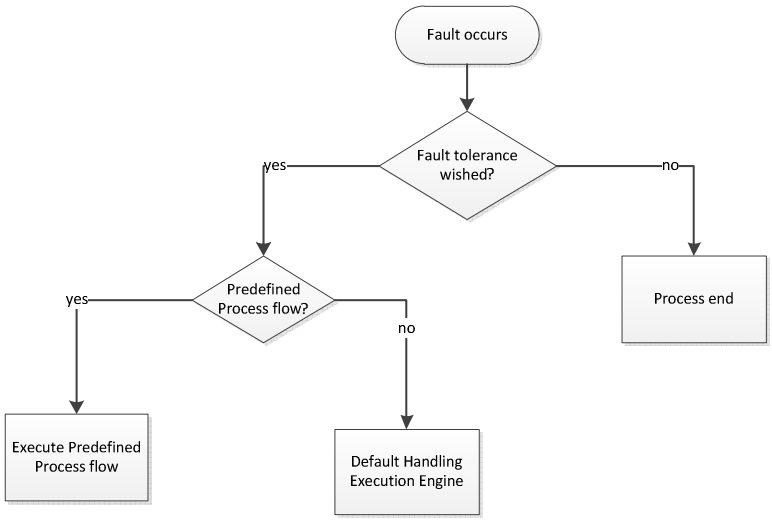

Figure 25: Rule for fault handling on top of business process

Building on the process example one could argue in different directions for the definition of quality requirements for fault avoidance. We exclude the consideration of faults occurring in the central business system such as faults occurring during the price calculation task, while we concentrate on the IoT specific process components.

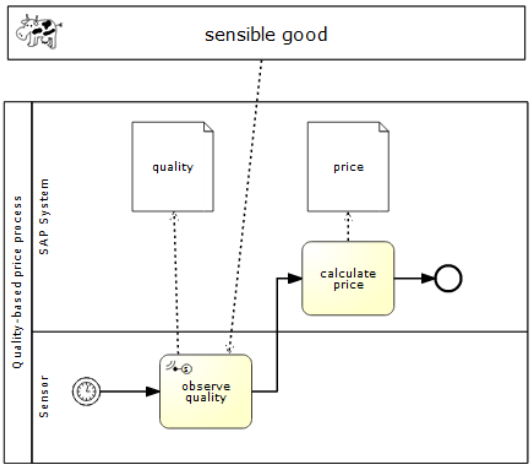

Figure 26: Process diagram of quality-based price process

Quality Requirements for fault avoidance:

- The IoT service accessing the on device resources of the sensor should feature a certain minimum of quality as defined in section 6.4 regarding the occurrence of faults. The fulfilment of such a requirement would ensure that the IoT service is suitable enough to take part in this sensible business process. This indicates that the service is highly reliable on the resources and the device it accesses and should be able to reflect its faulty properties.
- The sensor device properties should feature a certain minimum of quality regarding the occurrence of faults of its on device resource accessed by the service.
- The resource properties should feature a certain minimum of quality regarding the occurrence of faults in its provided information.
- The overall business process should feature a certain minimum of quality regarding the occurrence of faults, where all further process components rely on.

Transferring this use case to the effected IoT process components, quality requirements for fault avoidance could be defined for: **Service, Resource, Device and Process.**

6.4 Ratios for Fault Tolerance

As shown in the example of last section, BPM foresees three types of error events: start, interim and end. These standard elements can be defined for triggering and handling defined errors within a business process. Mostly an error event presents an exception end state of a process activity. It is drawn at the boundary of the task and can be understood as an XOR gateway. The default exit of the task is still the usual out coming branch, but in case an error occurs the exception exit is used and the task is finished immediately. The error end event finishes the process flow branch of type error. The error start event only occurs as part of a sub process.

Before directly determining the fault tolerance of a process component or even the process in general the following **criteria** are used to evaluate which components should be fault-tolerant:

- How critical is the process element? Maybe not every process element has to be fault-tolerant. That would resolve that some of the process elements should be fault tolerant while others don't. In this case a specification could take place for every process element.
- How likely is the component to fail? Maybe not every process element is likely to fail. If the process component is not likely to fail, it might not make sense to define the fault procedure. Or otherwise before the defining the process element it might be important to express a requirement, that the process element shall be not likely to fail to a certain degree.

These criteria could be taken into account already during the business process modelling phase before resolving the process the first time. This means that fault tolerance could be expressed for the effected IoT process components as part of the quality requirement. Based on [Sensei] and its unification of [IEEE MASS] we apply the following quality model for defining the requirements on process level:

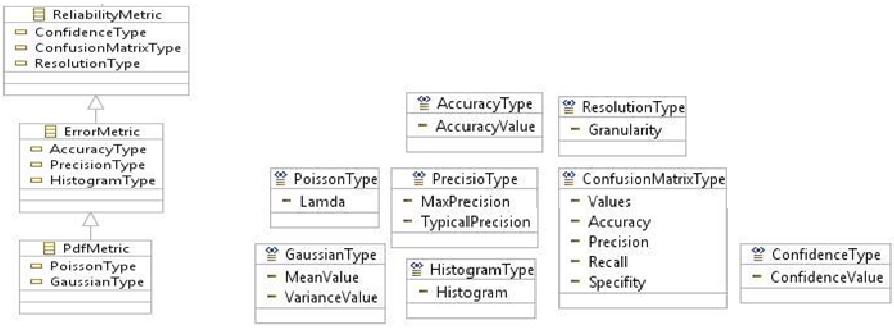

Figure 27: Ecore Model of Reliability Metric

Based on that model, a fault avoidance vector (reliability metric) per process component can be specified. This enables the process modeller to respond to faulty real-world process components by modifying the process accordingly, as well as to express the required values per process component.

For the usage of the fault avoidance vector during the modelling of business processes we distinguish between the following cases:

- First, we consider the case that a new real-world component is added to the process model. At modelling time no information about the device, service or resource, which will be bound to the component during the resolution phase, is available. Nevertheless, if the modeller wants to ensure that the device, service or resource, which will be bound to the components, fulfils certain fault avoidance, he can annotate the desired process component.
- Second, a certain fault avoidance vector for a process component, which is newly added to the process model, can be represented to the modeller. Prerequisite for this case is the direct integration of the resolution infrastructure into the process modelling environment. To provide the information to the modeller, the modelling environment reads the descriptions of the appropriate components from the repository. When saving the model to a file, the modelling environment writes the vectors of the selected components to the process model in order to provide a compensation component, if the selected component is not available during runtime.

In order to store the values of the vector in the process model, two types of extensions must be realized.

On the one hand, the standard XML schema representation of the component in the BPMN 2.0 process model needs to be extended to provide the ability to annotate the values. On the other hand, the symbolism of the existing BPMN 2.0 component would have to be expanded to represent these values in the process model as well. In our case the second extension will be made as part of the modelling environment as attributes are directly reflected in the process model. Based on [Meyer 2012] the fault avoidance vector will be integrated to the description model of the relevant components (green) as shown in the following figure.

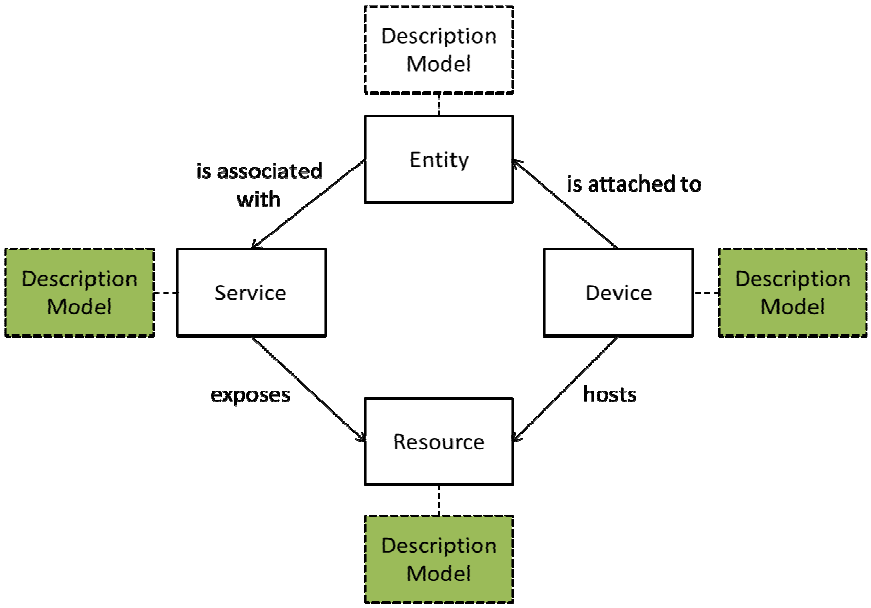

Figure 28: IoT Process Components with Description Models

The BPMN 2.0 meta-model is extensible by default, but the extensions are limited to some restrictions that need to be fulfilled in order to be BPMN 2.0 compliant. [BPMN]

The proposed symbolism is based on the standard definition of the modelling environment. The new components contain the fault avoidance parameters as part of the description category IoT attributes. Figure 29 shows a manually entered vector by the process modeller for the process component service.

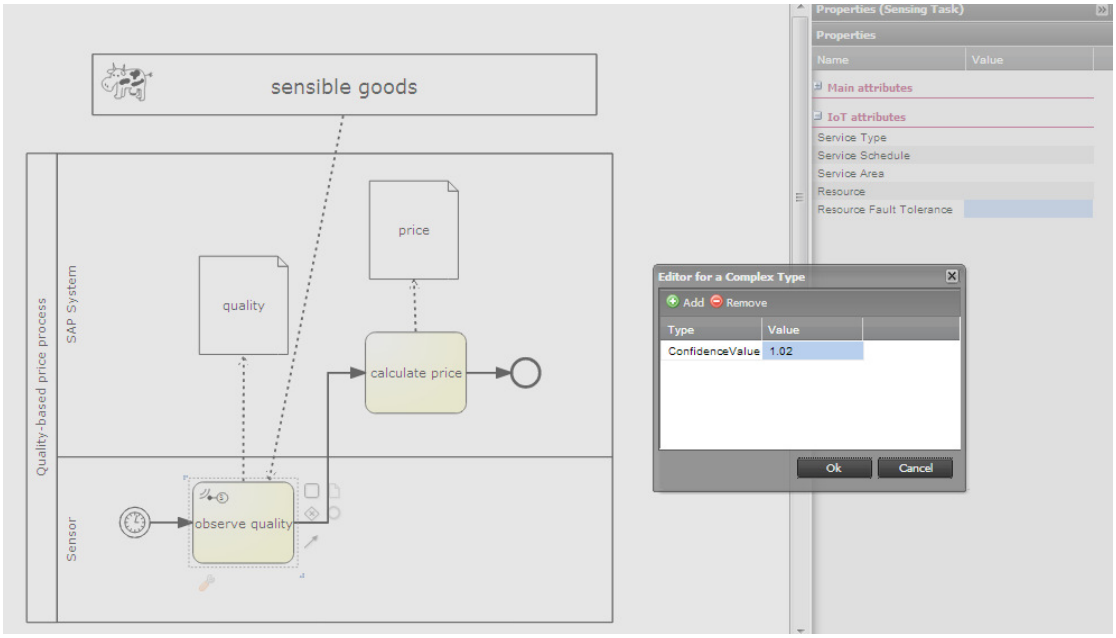

Figure 29: Resource Confidence Value

The following code extraction belongs to a sensing task of the XML BPMN process model. It depends on a resource in the background, which shall fulfil the confidence value of 1.02.

```
<sensingTask completionQuantity="1" id="sid-1F6DF202-B221-409B-B138-897E3981F241"
      implementation="webService" isForCompensation="false" name="observes Temperature"
      startQuantity="1">
      <extensionElements>
         <signavio:signavioMetaData metaKey="bgcolor" metaValue="#ffffcc" />
```

```
        </extensionElements>
        <incoming>sid-F5E38CE8-6BB7-41E5-8A58-DCA5AF4CC749</incoming>
        <outgoing>sid-17DC114D-1D75-490B-B0A7-6D6987F8A637</outgoing>
        <dataOutputAssociation id="sid-C7F4AC1C-F10B-4763-A81A-77C4CD9488EF">
           <sourceRef>sid-1F6DF202-B221-409B-B138-897E3981F241</sourceRef>
           <targetRef>sid-5A455268-DF61-4243-9EB1-A2CA646F73A3</targetRef>
        </dataOutputAssociation>
        <iotPerformer>
           <resourceParameterBinding
           parameterRef="rm:hasFaultTolerance:ConfidenceType:ConfidenceValue">
                    <expression>1.02 </expression>
           </resourceParameterBinding>
        </iotPerformer>
</sensingTask>
```

6.5 Conclusion

The fault-proneness of IoT components such as sensors and actuators, its services and resources is a significant obstacle to successfully include real-world components in business processes. In this section, we have shown how to improve this situation considerably with a new bottom-up modelling approach that introduces fault avoidance aspects to the business process level. In order to reach that goal, we applied the reliance metric of [sensei] and introduced these parameters as new quality aspects to the IoT components of a business process model.

Utilizing an appropriate abstraction mechanism we made the fault avoidance parameters available to the effected IoT-specific components of the business process description, thus extending BPMN 2.0 modelling concepts.

7. Conclusions and Outlook

The deliverable focuses on automatically orchestrating IoT services in a way that it satisfies certain predefined quality of service requirements. As this is a quite ambitious goal we identified certain sub-objectives we investigated, namely self-configuration, self-optimization, self-healing and self-protection. We concluded by showing how fault-tolerance can be integrated into business processes.

We first described a comprehensive approach on Self-Configuration in IoT Service Orchestration based on the service platform RWIP and self-configuration for Global State detection, thus addressing the automated inclusion of new IoT devices in an automatic plug-and-play way. We have shown how IoT services are able to describe themselves, what configurations they support and how they can advertise their capabilities to service consumers. We argued that due the often simple nature of IoT services (compared to high-level services) the capabilities space for self-configuration comes down to augmenting the standard 1:1 relationship of device and service allowing for different means of adding or removing devices as well as automatically adapting the fusion of information and the service policies of the IoT service. Nonetheless, certain challenges remain open, like developing proprietary protocols for adding devices without human intervention in order for self-configuration of IoT services to become a widespread solution in the Internet of Things.

We then went on with self-optimization where we used the original service request as the goal that needs to be optimized. We outlined how the service orchestration and service composition can be modified in order to provide a service that is optimal for the user with the currently available services.

The section on self-healing described in which situations in an IoT system a self-healing mechanism should be applied and how such a mechanism can be designed. Our approach makes use of IoT-A's resolution infrastructure. Furthermore, the usefulness of self-healing in Global State Detection for anomaly detection has been shown. For detecting misbehaviour the notifications of the supporting resolution framework are not enough since they only detect changes in availability or configuration of services, but not in their operation behaviour during service execution.

In the next chapter we introduced a concept for self-protection based on a generalization of the replication paradigm. It turned out that sensing is easier and requires less replicas than traditional web Services ($2\delta+1$ as opposed to $3\delta+1$). Actuation is even simpler in the case of idempotent operation ($\delta+1$ replicas), but requires $3\delta+1$ Services for in the additive model. Even worse, in the latter model these services cannot all be executed in parallel, and the whole approach is only feasible in the case of a synchronous network with upper bounds on communication delays. As the latter is likely not to be achievable in the IoT, one of our main findings is that idempotent actuation is highly desirable at the Service level

Afterwards we explained how Fault Tolerance can be integrated into IoT-aware Business Processes. Unfortunately, compared to traditional business processes, the fault-proneness of IoT components such as sensors and actuators is much higher. This is a major problem when designing Business processes for IoT. In this section we explained a new bottom-up modelling approach that introduces fault avoidance aspects to the business process level. Utilizing an appropriate abstraction mechanism we made the fault avoidance parameters available to the effected IoT-specific components of the business process description, thus extending BPMN 2.0 modelling concepts.

8. References

[Aalst 2005] W. van der Aalst and A. Ter Hofstede, —YAWL: yet another workflow language, Information Systems, vol. 30, no. 4, pp. 245–275, 2005.

[Bauer 2012] M. Bauer, M. Boussard, N. Bui, F. Carrez, P. Giacomin, St. Haller, E. Ho, Chr. Jardak, J. De Loof, C. Magerkurth, St. Meissner, A. Nettsträter, A. Olivereau, A. Serbanati, M. Thoma, J. W. Walewski: "Deliverable 1.3 – Updated Reference Model for IoT v1.5", IoT-A, 2012.

[Binder 2011] Binder, Walter, Daniele Bonetta, Cesare Pautasso, Achille Peternier, Diego Milano, Heiko Schuldt, Nenad Stojnic, Boi Faltings, and Immanuel Trummer.“Towards Self-Organizing Service-Oriented Architectures.” In 2011 IEEE World Congress on Services, 115–121. 2011. IEEE. doi:10.1109/SERVICES.2011.44. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6012702.

[BPEL 2007] Web Services Business Process Execution Language (WS-BPEL), Version 2.0, April 2007, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[BPMN] Business Process and Notation (BPMN), Version 2.0, 2012.

[Bruns 2010] Ralf Bruns and Jürgen Dunkel: "Event-Driven Architecture", Springer Verlag Berlin Heidelberg, 2010.

[Castro 2002] M. Castro and B. Liskov, Practical Byzantine Fault Tolerance and Proactive Recovery, ACM Transactions on Computer Systems, v. 20 n. 4, pp. 398-461, 2002.

[Chandola 2009] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” ACM Comput. Surv.,vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009.

[De 2011] De, Suparna, Payam Barnaghi, Martin Bauer, and Stefan Meissner. 2011. “Service Modelling for the Internet of Things.” Federated Conference on Computer Science and Information Systems. Szczecin.

[De 2012] S. De, G. Cassar, B. Christophe, S. Ben Fredj, M. Bauer, N. Santos, T. Jacobs, E. Zeybek, R. de las Heras, G. Martín, G. Völksen, A. Ziller: "Concepts and Solutions for Entity-based Discovery of IoT Resources and Managing their Dynamic Associations", IoT-A Project Deliverable D4.3, 2012.

[de las Heras 2012] Ricardo de las Heras, Martin Bauer, Nuno Santos, Nils Gruschka, Gerd Voelksen, Benoit Christophe, Sameh Ben Fredj, Suparna De, Gilbert Cassar, Ebru Zeybek, Alexis Olivereau, Irene Sainz Villalba: "Concepts and Solutions for Identification and Lookup of IoT Resources", IoT-A D4.1, Report, 2012.

[Drools 2012] JBoss Community, “Drools Fusion,” 2012. [Online]. Available: http://www.jboss.org/drools/drools-fusion.html

[Ferscha 1994] A. Ferscha, Qualitative and quantitative analysis of business workflows using generalized stochastic petri nets: Proceedings of Workflow Management-Challenges, Paradigms and Products CON'94: Citeseer, 1994.

[Fielding 2005] Fielding, Roy T. “Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)”, 2005. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[Freund 2012] J. Freund, B. Rücker, and T. Henninger, Praxishandbuch BPMN: Hanser, 2010.

[Gibson 2012] Matthew Richard Gibson. Clusters and covers: geometric set cover algorithms. Dissertation, University of Iowa, 2010.

[Huang 2003] Chi-Fu Huang and Yu-Chee Tseng. 2003. The coverage problem in a wireless sensor network. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications* (WSNA '03). ACM, New York, NY, USA, 115-121.

[CSIRO 2011] Lefort, Laurent (CSIRO, Australia and W3C Semantic Sensor Network Incubator Group). “Ontology for Quantity Kinds and Units: Units and Quantities Definitions.”, 2011, W3C Working Draft. http://www.w3.org/2005/Incubator/ssn/ssnx/qu/qu-rec20.

[Kephart 2003] Kephart, J O, and D M Chess. 2003. “The vision of autonomic computing.” Computer 36 (1): 41-50.

[Kifer 2012] M. Kifer and H. Boley, “RIF Overview,” 2010. [Online]. Available: http://www.w3.org/2005/rules/wiki/Overview

[Ko 2012] Ren-Song Ko. The Complexity of the Minimum Sensor Cover Problem with Unit-Disk Sensing Regions over a Connected Monitored Region. International Journal of Distributed Sensor Networks, 2012.

[Kozel 2010] T. Kozel, BPMN mobilisation: Proceedings of the European conference of systems, and European conference of circuits technology and devices, and European conference of communications, and European conference on Computer science: World Scientific and Engineering Academy and Society (WSEAS), 2010.

[Lamport 1982] Lamport, L.; Shostak, R.; Pease, M. (July 1982). "The Byzantine Generals Problem". ACM Transactions on Programming Languages and Systems 4 (3): 382–401. doi:10.1145/357172.357176

[Liu 2008] Liu, Lei, Stefan Thanheiser, and Hartmut Schmeck. A Reference Architecture for Self-organizing Service-Oriented Computing.” In Architecture of Computing Systems – ARCS 2008, ed. Uwe Brinkschulte, Theo Ungerer, Christian Hochberger, and Rainer Spallek, 4934:205–219. 2008. Springer Berlin / Heidelberg. http://dx.doi.org/10.1007/978-3-540-78153-0_16.

[Martin 2012] G. Martín, St. Meissner, D. Dobre, M. Thoma: "Resource Description Specification" IoT-A Deliverable D2.1, 2012

[Meissner 2012] S. Meissner, S. Debortoli, K. Sperner, C. Magerkurth, D. Dobre: "Project Deliverable D2.3 – Orchestration of Distributed IoT Service Interactions"

[Meyer 2011] S. Meyer, K. Sperner, C. Magerkurth, and J. Pasquier, —Towards modeling real-world aware business processes in Proceedings of Web of Things 2011, San Francisco, CA, USA, 2011.

[Meyer 2012] S. Meyer, K. Sperner, C. Magerkurth, S. Debortoli, M. Thoma, G. Romero: "Project Deliverable D2.2 – Concepts for Modelling IoT-Aware Processes"

[Mustafa 2010] Nabil H. Mustafa and Saurabh Ray. 2010. Improved Results on Geometric Hitting Set Problems. *Discrete Comput. Geom.* 44, 4 (December 2010), 883-895..

[Pu 1996] C. Pu, A. Black, C. Cowan, and J. Walpole. A Specialization Toolkit to Increase the Diversity of Operating Systems. In ICMAS Workshop on Immunity-Based Systems, 1996.

[Ramasamy 2005] HariGovind V. Ramasamy and Christian Cachin. 2005. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In Proceedings of the 9th international conference on Principles of Distributed Systems (OPODIS), 2005.

[RDF 2004] "Resource Description Framework (RDF): Concepts and Abstract Syntax." http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[Rossi 2008] Rossi, Michele, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, Albert F. Harris III, and Michele Zorzi. "SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks Using Fountain Codes." In 2008 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, 188–196. 2008. IEEE. doi:10.1109/SAHCN.2008.32. http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4557755&contentType=Conference+Publications.

[Rossi 2010] Rossi, Michele, Nicola Bui, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, and Michele Zorzi. "SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes." IEEE Transactions on Mobile Computing 9 (12): 1749–1765. 2010. doi:10.1109/TMC.2010.109. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5611465.

[SAP, Research. 2012] "Linked USDL." http://linked-usdl.org/.

[Schnabel 2011] F. Schnabel et al., —Empowering Business Users to Model and Execute Business Processes: Business Process Management Workshops, in Lecture Notes in Business Information Processing, M. Muehlen and J. Su, Eds.: Springer Berlin Heidelberg, 2011, pp. 433–448.

[SPARQL 2008] "SPARQL Query Language for RDF." http://www.w3.org/TR/rdf-sparql-query/.

[Sperner 2011] K. Sperner, S. Meyer, C. Magerkurth —Introducing Entity-based Concepts to Business Process Modeling, 3rd International Workshop and Practitioner Day on BPMN, Lucerne, Switzerland

[Sperner 2012] K. Sperner, S. Meyer, G. Völksen, M. Thoma: "Internal Report IR2.3 - Initial Design and Implementation Report", IoT-A IR2.3, Report, 2012.

[Strohbach 2010] Strohbach, Martin, Stefan Meissner, Claudia Villalonga, Fernando López, Frederic Montagut, Vincent Huang, Harald Kornmayer, and Jochen Bauknecht. "D2.6 Sensor Information Services with Open Service Interfaces.". 2010

[Sun 2007] Min-Te Sun, Chih-Wei Yi, Chuan-Kai Yang, and Ten-Hwang Lai. 2007. An Optimal Algorithm for the Minimum Disc Cover Problem. *Algorithmica* 50, 1 (December 2007), 58-71.

[Thai 2008] My T. Thai, Feng Wang, David Hongwei Du, and Xiaohua Jia. 2008. Coverage problems in wireless sensor networks; designs and analysis. *Int. J. Sen. Netw.* 3, 3 (May 2008), 191-200.

[Toeyry 2011] Timo Toeyry "Self-management in Internet of Things". 2011 https://wiki.aalto.fi/download/attachments/59704179/toyry-self-management.pdf

[UML] Unified Modeling Language (UML) Specification, 2010.

[Vidackovic 2010] K. Vidackovic, T. Renner, and S. Rex, "Market Overview Real-Time Monitoring Software," Frauenhofer-Institut für Arbeitswirtschaft und Organisation IAO, Report, 2010.

[Vazirani 2001] Vijay Vazirani. Approximation Algorithms. Springer, 2001.

[WADL 2009] "Web Application Description Language." http://www.w3.org/Submission/wadl/.

[Walewski 2011] J.W. Walewski et al., Project Deliverable D1.2 – Initial Architectural Reference Model for IoT, June 2011, —http://www.iot-a.eu/public/public-documents/documents-1/1/1/d1.2/at_download/file□.

[Wang 2012] Wang, Wei, Suparna De, Ralf Toenjes, Eike Reetz, and Klaus Moessner. "A Comprehensive Ontology for Knowledge Representation in the Internet of Things." In 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, 1793–1798. 2012. IEEE. doi:10.1109/TrustCom.2012.20. http://www.computer.org/csdl/proceedings/trustcom/2012/4745/00/4745b793-abs.html.

[Weidlich 2008] M. Weidlich, G. Decker, A. Großkopf, and M. Weske, —BPEL to BPMN: the myth of a straight-forward mapping,□ On the Move to Meaningful Internet Systems: OTM 2008, pp. 265–282, 2008.

[Weinschrott 2011] Weinschrott, H.; Weisser, J.; Durr, F.; Rothermel, K.. Participatory sensing algorithms for mobile object discovery in urban areas. Pervasive Computing and Communications (PerCom), pp.128-135, March 2011.

[Weske 2007] M. Weske, Business process management: concepts, languages, architectures: Springer-Verlag New York Inc, 2007.

[WSDL 2007] "Web Services Description Language (WSDL) Version 2.0 Part 0: Primer." http://www.w3.org/TR/wsdl20-primer/.

[Xu 2008] Xiaochun Xu; Sahni, S.; Rao, N.; , "Minimum-cost sensor coverage of planar regions," Information Fusion, 2008 11th International Conference on, vol., no., pp.1-8, June 30 2008-July 3 2008.

9. Appendix

9.1 Introduction to Global State Detection using Complex Event Processing

Complex Event Processing (CEP) is a technique to analyze events from various sources. In IoT, and according to the IoT Domain Model (of which the latest version was published in D1.3 (see [Bauer 2012]), sources may be sensor devices or services. This, of course, includes distributed mobile event sources.

Other event sources may be business or production processes or applications. As a reaction on the input events, the CEP system produces data which again may be input to a GUI, a trigger for an application or a log file or database entry. However, the CEP output can also be an event for further processing which enables a modular networked structure of the CEP system. Such structures are called Event-Driven Architectures (EDA, see also [Bruns 2010]). On the other hand, an event may also be processed internally as if it had been received from an external event source.

The events appear as single events, event streams, or as floods of events. Accordingly, there is a subdivision between Simple Event Processing (of single events), Event Stream Processing (ESP, of one or more synchronized event streams each of the same event type), and CEP, respectively. The latter one covers the others as it processes events of various types which correlate in some temporal, spatial, or causal relation. However, the ESP-CEP distinction is not really sharp.

According to the FhG IAO study (see [Vidackovic 2010]), a further structure definition determines the Event Channel which is the entity where events are pushed while the Event Consumer – the CEP system – selects the relevant events from this channel. In [Bruns 2010]0, this Event Channel is called a Mediator. This component enables a loose coupling of event sources and event consumers. It also arranges for the event distribution if sources and consumers are distributed.

9.2 Information Architecture

The main information units are the events and the rules. These two information formats depend strongly on each other, because the rule has to access, evaluate, and interpret the events.

Event Description Language

An event has to deliver at least the following data

- Timestamp (MM-DD-hh-mm-ss-YYYY)
- Event category (information, error, warning, alert)
- Event type (Technical event, system event, business event)
- Event name or event ID
- Event values, each providing

- Numerical value
- Dimension description

- Event source (sensor, service, etc.)
  - Event source type
  - Event source ID
  - Event source location

An evaluation of event description languages will follow in IoT-A D2.6

Rule Description Language

Rules are modelled in various formats. Most of the CEP Systems provide a proprietary format, sometimes along with an editor. In the following, two of them are presented in order to give a taste of what they look like: RIF and the Drools DRL (Drools Rule Language).

RIF / XML

The following example RIF/XML rule (see [Kifer 2012]) shows the main parts which comprise a left hand side (lhs) and a right hand side (rhs) along with declarations of variables which have an access scope for the whole rule (declare) and a declaration of variables for the right hand side only (declareactionvar). The left hand side evaluates to a Boolean value while the right hand side provides a statements for execution.

```
<rule>
  <declare classname="Real" varname="X" />
  <declare classname="Real" varname="Y" />
  <lhs>
    <op name="<">
      <const type="Real" value="0.0" />
      <op name="+">
        <var name="X" />
        <var name="Y" />
        <const type="e" value="2,71" />
      </op>
    </op>
  </lhs>
  <declareactionvar classname="Real" varname="z" />
  <declareactionvar classname="Real" varname="pi" />
  <rhs>
    <execute>
      <op name="assign">
        <var name="X" />
        <op name="+">
          <var name="Y" />
          <const type="Integer" value="1" />
        </op>
      </op>
      <op name="assign">
```

```
        <var name="Z" />
        <var name="pi" />
      </op>
    </execute>
  </rhs>
</rule>
```

Drools DRL

Drools DRL Code (see [Vidackovic 2010] and [Drools 2012]) is an open source CEP system. Its even processing language is a declarative language applying production rules on events. For the development of event processing rules, an Eclipse-based IDE is provided. Figure 30 shows the rule viewer and the gives a taste of a rule for a loan application.

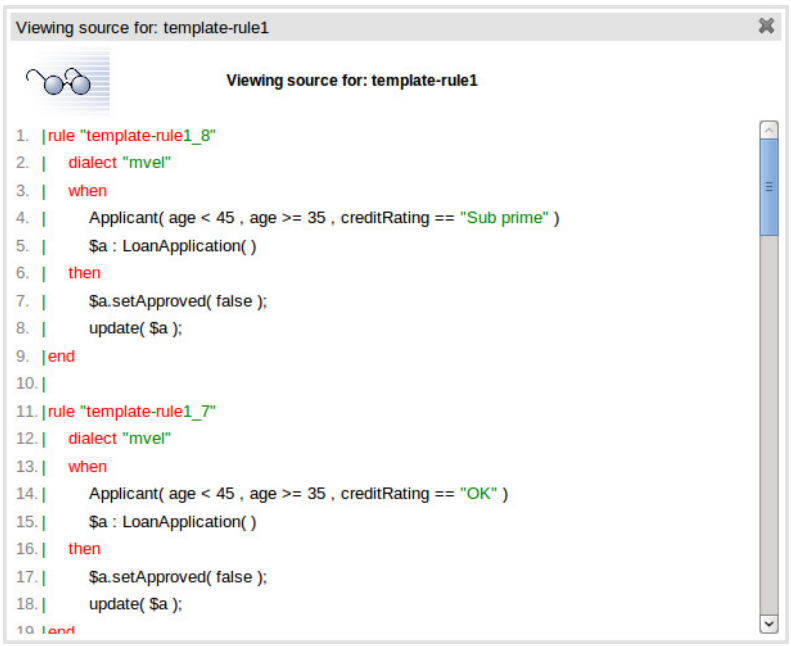

Figure 30: Rule Viewer for Generated DRL Code

9.3 CEP Application Domains

CEP is the appropriate technology for calculating the state of a system. However, there is a more precise subdivision of this objective necessary: A highly complex system might not be sufficiently characterized by one state description model. Here are some special aspects of state detection that appear reasonable for complex systems.

- State detection
  - Global system state detection
  - Process state detection
  - Sub-system state detection: sub-systems may be characterized as follows:
    - Location-based sub-system: events related to a certain geographical area will be considered like the weather in a given region

    - Item-based sub-system: a certain item of the system is considered: on vehicle from a logistic system.
    - Data-based sub-system: events providing a certain type of information are processes like traffic flow data.

There are, furthermore, lots of application domains which might utilize CEP technologies; here are some examples which all also apply to the context of IoT systems. The main categories refer to anomaly detection, system monitoring, and control loops.

- Anomaly detection (see also [Chandola 2009])
    - Intrusion or fraud detection
    - Drop-out detection
    - Surveillance and motion detection
    - Health supervision
- System monitoring
    - Production line monitoring
    - Business processing monitoring
    - Network monitoring
    - Work piece state detection
    - Weather and weather forecast
- Control loops
    - Stock trading
    - Business processes
    - System control
        - Production plant control
        - Robot control
        - Traffic management
    - Network management
    - Power and energy management (smart grits)

The list above is surely not complete and also includes overlaps. For example, monitoring applications are usually parts of control loops. Other overlaps refer to the business processes and the related production lines.

However, the idea of Global State Detection (GSD) is just one of several applications for IoT systems.

Most CEP publication deal with application from the commercial or business domain. IoT, however, has some further applications which might need CEP technology. While health supervision has already been mentioned above under the topic "Anomaly Detection", the IoT-A Retail Use Case needs to be mentioned where the price of goods depends directly on the conditions under which the goods a stored. In Figure 31, the goods are orchids, which are priced according to their quality which again depends strongly on the environmental conditions comprising temperature, humidity, and illumination at various places. These values are sensed and the corresponding events are processes in order to determine a price which guarantees complete sale, profit, and user satisfaction. Additionally, the sensors are monitored, too. Beside prices, the system also detects spaces in the shelves which might indicate a necessary order from the nursery.

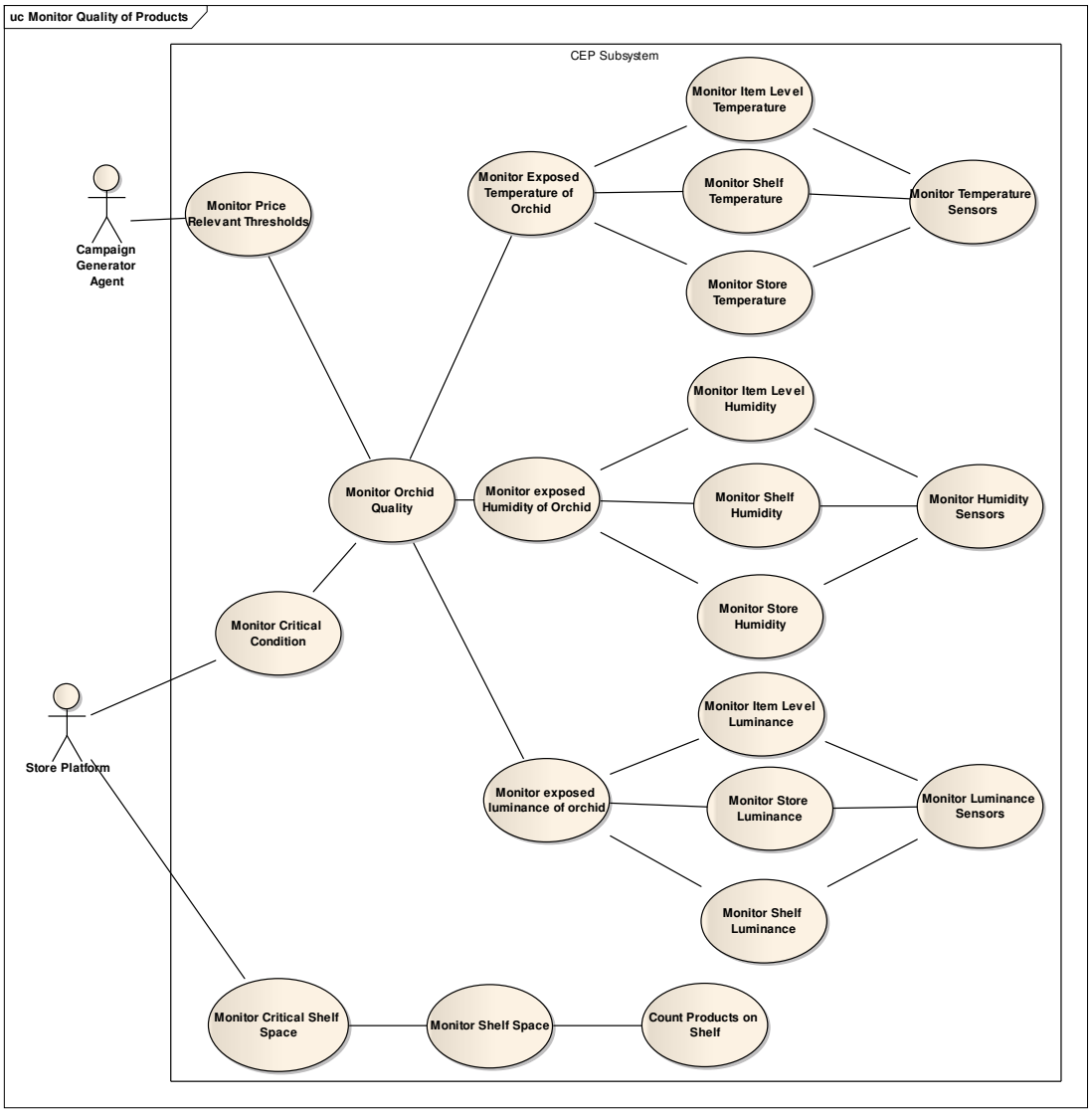

Figure 31: IoT-A Retail Use Case - Orchid Environment Monitoring

Whether or not a control loop is in operation depends on the way the output of the CEP system is handled. If an anomaly detection result or a system monitoring output is input to a human or a module who/which again affects the system by some modifications and thus influences the event sources, the control loop is closed and at work as indicated in Figure 32. Only if the CEP results are stored in a log file or a data base for a post-mortem analysis, there is no real closed control loop. Though post-mortem analysis is a highly popular CEP application domain, the most important applications are related to online management of running systems in order to avoid a system crash and, hence, a post-mortem analysis.

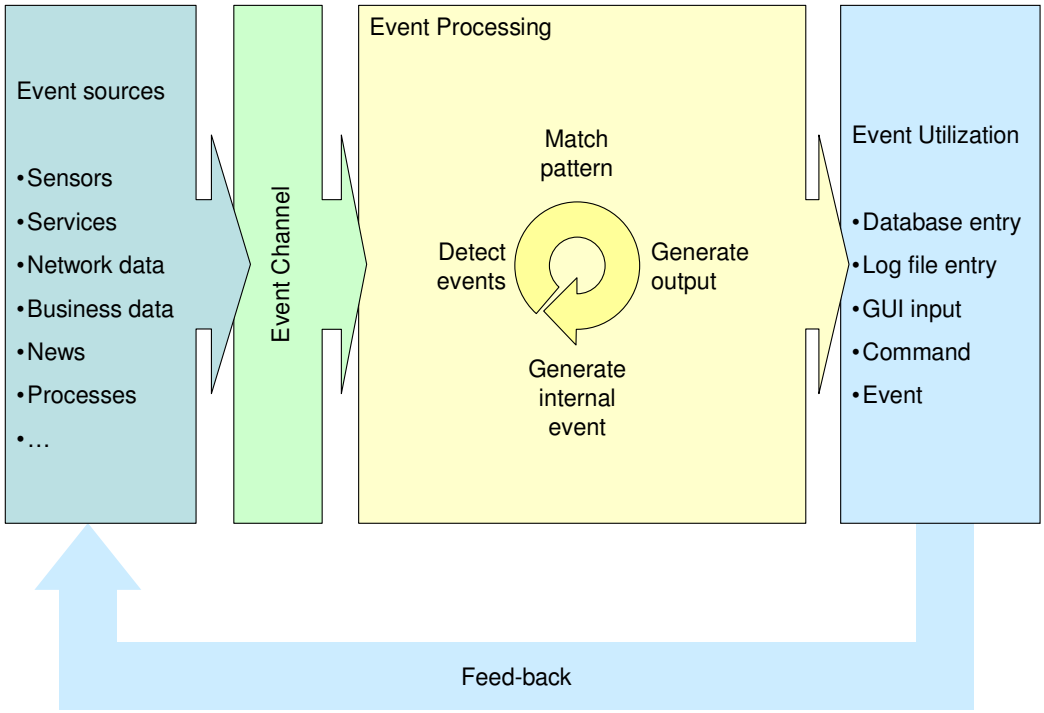

Figure 32: CEP System Reference Architecture Components

Glossary

The following table describes the definitions of the main concepts relevant for the work in WP2. It consists of the official terms of the project, as they are referenced at http://www.iot-a.eu/public/terminology as well as the WP2 specific terms that are relevant for WP2 deliverables, but not necessarily for the work done in other work packages. These WP2-specific terms such as Business Process Management, Business Process Execution, Service Composition, Service Orchestration, Service Choreography, Complex Event Processing, etc. mostly relate to the Future Internet technologies and higher level enterprise systems that WP2 is concerned with.

Term	Definition	Source
Active Digital Entity	Any type of active code or software program, usually acting according to a *Business Logic*.	Internal
Actuators	"An *actuator* is a mechanical device for moving or controlling a mechanism or *system*. It takes energy, usually transported by air, electric current, or liquid, and converts that into some kind of motion."	[Sclater2007]
Address	An address is used for locating and accessing – "talking to" – a *Device*, a *Resource*, or a *Service*. In some cases, the ID and the Address can be the same, but conceptually they are different.	Internal
Application Software	"Software that provides an application *service* to the *user*. It is specific to an application in the multimedia and/or hypermedia domain and is composed of programs and data".	[ETSI- ETR173]
Architectural	The IoT-A architectural reference model follows the definition of the IoT	Internal

Reference Model	reference model and combines it with the related IoT reference architecture. Furthermore, it describes the methodology with which the reference model and the reference architecture are derived, including the use of internal and external stakeholder requirements.	
Architecture	"The fundamental organization of a *system* embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution".	[IEEE-1471-2000]
Architecture Vision	"A high-level, aspirational *view* of the target *architecture*."	[TOGAF9]
Aspiration	"*Stakeholder* Aspirations are statements that express the expectations and desires of the various *stakeholder*s for the *service*s that the final [*system*] implementation will provide."	[E-FRAME]
Augmented Entity	The composition of a *Physical Entity* together with its *Virtual Entity*.	Internal
Association	An association establishes the relation between a *service* and *resource* on the one hand and a *Physical Entity* on the other hand.	Internal
AutoID and Mobility Technologies	"Automatic Identification and Mobility (AIM) technologies are a diverse family of technologies that share the common purpose of identifying, tracking, recording, storing and communicating essential business, personal, or product data. In most cases, AIM technologies serve as the front end of enterprise software *system*s, providing fast and accurate collection and entry of data. AIM technologies include a wide range of solutions, each with different data capacities, form factors, capabilities, and "best practice" uses. AIM technologies also include mobile computing devices that facilitate the collection, manipulation, or communication of data from data carriers as well as through operator entry of data via voice, touch screens or key pads.	[AIMglobal]
Business Logic	Goal or behaviour of a system involving Things serving a particular business purpose. Business Logic can define the behaviour of a single Thing, a group of Things, or a complete business process.	Internal
Business Process Execution	Business Process Execution (BPE) consists of the manual, semi-manual and automated execution of business processes over actors. An important precondition for the automated execution of business processes is the technical specification of the business process model by technical analysts. The relevant industry standards for describing an executable business process model are BPEL and BPMN. Common IT-Systems supporting the execution of a huge number of automated processes use a process execution engine as a central controlling component.	Internal
Business Process Management	Business Process Management (BPM) is a structured approach that employs methods, policies, metrics, management practices and software tools to manage and continuously optimize the activities and processes of an organization. Process management uses an iterative process revision cycle known as the BPM lifecycle, which enables the continuous improvement of business processes. Depending on the methodology the following lifecycle phases are distinguished: Definition, Modelling, Simulation, Deployment, Execution, Monitoring, Analysis, Optimization and Discovery. In the IoT-A project we mostly focus on parts of the lifecycle phases Modelling, Deployment and Execution.	Internal
Business Process Modeling	Business Process Modelling is the activity of representing processes of an enterprise by abstracting the business process in a graphical or non-graphical model for progressing with one of the other BPM lifecycle phases in any order. Modelling a business process is typically performed by business analysts. The industry relevant standards are UML, EPC and BPMN. WP2 focuses on extending business process modelling approaches by IoT-specific characteristics.	Internal

Complex Event Processing	Complex event processing (CEP) describes a method of processing a multitude of events triggered by various information sources, and consecutively derive more meaningful events from a set of low level events. In IoT-A CEP is considered as an approach to bridge real-world events to the execution of business processes. We use the term event processing in the narrowest sense, i.e. analysis of events and subsequent decision making in terms of taking actions in real-time, are not considered as event processing, but are rather considered part of the business logic.	Internal
Constrained Network	A constrained network is a network of devices with restricted capabilities regarding storage, computing power, and / or transfer rate.	Internal
Controller	Anything that has the capability to affect a *Physical Entity*, like changing its state or moving it.	Internal
Credentials	A credential is a record that contains the authentication information (credentials) required to connect to a resource. Most credentials contain an user name and password.	Internal
Device	Technical physical component (hardware) with communication capabilities to other IT systems. A *device* can be either attached to or embedded inside a *Physical Entity*, or monitor a *Physical Entity* in its vicinity.	Internal
Digital Entity	Any computational or data element of an IT-based system.	Internal
Discovery	Discovery is a *service* to find unknown *resources/entities/services* based on a rough specification of the desired result. It may be utilized by a human or another *service*. Credentials for authorization are considered when executing the discovery.	Internal
Domain Model	“A domain model describes objects belonging to a particular area of interest. The domain model also defines attributes of those objects, such as name and identifier. The domain model defines relationships between objects such as “instruments produce data sets”. Besides describing a domain, domain models also help to facilitate correlative use and exchange of data between domains”.	[CCSDS 312.0-G-0]
Energy-harvesting Technologies	“*Energy-harvesting* (also known as power harvesting or energy scavenging) is the process by which energy is derived from external sources (e.g., solar power, thermal energy, wind energy, salinity gradients, and kinetic energy), captured, and stored. Frequently, this term is applied when speaking about small, wireless autonomous *device*s, like those used in wearable electronics and wireless *sensor network*s.	[Wikipedia EH]
Future Internet	Future Internet is a summarizing term for worldwide research activities dedicated to the further development of the original Internet. In the IoT-A project we focus mostly on the Internet of Things and the Internet of Services as major constituents of the Future Internet that are consolidated within WP2.	Internal
Gateway	A Gateway is a forwarding element, enabling various local networks to be connected.	Internal
Global Storage	*Storage* that contains global information about many *entities of interest*. Access to the *global storage* is available over the *Internet*.	Internal
Human	A human that either physically interacts with Physical Entities or records information about them, or both.	Internal
Identity	Properties of an entity that makes it definable and recognizable.	Internal

Identifier (ID)	Artificially generated or natural feature used to disambiguate things from each other. There can be several Ids for the same *Physical Entity*. The set of Ids is an attribute of an *Physical Entity*.	Internal
Information Model	"An information model is a representation of concepts, relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse. The advantage of using an information model is that it can provide sharable, stable, and organized structure of information requirements for the domain context. The information model is an abstract representation of entities which can be real objects such as devices in a network or logical such as the entities used in a billing system. Typically, the information model provides formalism to the description of a specific domain without constraining how that description is mapped to an actual implementation. Thus, different mappings can be derived from the same information model. Such mappings are called data models."	[AutoI]
Infrastructure Services	Specific services that are essential for any IoT implementation to work properly. Such services provide support for essential features of the IoT.	Internal
Interface	"Named set of operations that characterize the behaviour of an entity."	[OGS]
Internet	"The *Internet* is a global *system* of interconnected computer networks that use the standard *Internet* protocol suite (TCP/IP) to serve billions of *user*s worldwide. It is a network of networks that consists of millions of private, public, academic, business, and government networks of local to global scope that are linked by a broad array of electronic and optical networking technologies. The *Internet* carries a vast array of information *resource*s and *service*s, most notably the inter-linked hypertext documents of the World Wide Web (WWW) and the infrastructure to support electronic mail. Most traditional communications media, such as telephone and television *service*s, are reshaped or redefined using the technologies of the *Internet*, giving rise to *service*s such as Voice over *Internet* Protocol (VoIP) and IPTV. Newspaper publishing has been reshaped into Web sites, blogging, and web feeds. The *Internet* has enabled or accelerated the creation of new forms of *human* interactions through instant messaging, *Internet* forums, and social networking sites. The *Internet* has no centralized governance in either technological implementation or policies for access and usage; each constituent network sets its own standards. Only the overreaching definitions of the two principal name spaces in the *Internet*, the *Internet*-protocol address space and the domain-name *system*, are directed by a maintainer organization, the *Internet* Corporation for Assigned Names and Numbers (ICANN). The technical underpinning and standardization of the core protocols (IPv4 and IPv6) is an activity of the *Internet* Engineering Task Force (IETF), a non-profit organization of loosely affiliated international participants that anyone may associate with by contributing technical expertise."	[Wikipedia IN]
Internet of Things (IoT)	The global network connecting any smart object.	Internal
Interoperability	"The ability to share information and services. The ability of two or more systems or components to exchange and use information. The ability of systems to provide and receive services from other systems and to use the services so interchanged to enable them to operate effectively together."	[TOGAF 9]
IoT Service	Software component enabling interaction with resources through a well-defined interface. Can be orchestrated together with non-IoT services (e.g., enterprise services). Interaction with the service is done via the network.	Internal
Local Storage	Special type of *resource* that contains information about one or only a few *entities* in the vicinity of a *device*.	Internal
Location Technologies	All technologies whose primary purpose is to establish and communicate	Internal

	the location of a *device* e.g. GPS, RTLS, etc.	
Look-up	In contrast to *discovery*, *look-up* is a *service* that *addresses* exiting known*resources* using a key or *identifier*.	Internal
M2M (also referred to as machine to machine)	"The automatic communications between *device*s without *human*intervention. It often refers to a *system* of remote *sensor*s that is continuously transmitting data to a central *system*. Agricultural weather sensing *system*s, automatic meter reading and *RFID* tags are examples."	[COMPDICT-M2M]
Microcontroller	"A *microcontroller* is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM. *Microcontroller*s are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications. *Microcontroller*s are used in automatically controlled products and *device*s, such as automobile engine control *system*s, implantable medical *device*s, remote controls, office machines, appliances, power tools, and toys. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output *device*s, *microcontroller*s make it economical to digitally control even more *device*s and processes. Mixed signal *microcontroller*s are common, integrating analog components needed to control non-digital electronic *system*s".	[Wikipedia MC]
Network resource	*Resource* hosted somewhere in the network, e.g., in the cloud.	Internal
Next-Generation Networks (NGN)	"Packet-based network able to provide telecommunication *service*s and able to make use of multiple broadband, QoS-enabled transport technologies and in which *service*-related functions are independent from underlying transport-related technologies"	[ETSI TR 102 477]
Observer	Anything that has the capability to monitor a *Physical Entity*, like its state or location.	Internal
On-device Resource	*Resource* hosted inside a *Device* and enabling access to the *Device* and thus to the related *Physical Entity*.	Internal
Operator	The operator owns administration rights on the services it provides and/or on the entities it owns, is able to negotiate partnerships with equivalent counterparts and define policies specifying how a service can be accessed by users.	Internal
Passive Digital Entities	A digital representation of something stored in an IT-based system.	Internal
Physical Entity	Any physical object that is relevant from a user or application perspective.	Internal
Perspective (also referred to as architectural perspective)	"Architectural perspective is a collection of activities, checklists, tactics and guidelines to guide the process of ensuring that a *system* exhibits a particular set of closely related quality properties that require consideration across a number of the *system*'s architectural *view*s."	[ROZANSKI2005]
Privacy	Information Privacy is the interest an individual has in controlling, or at least significantly influencing, the handling of data about themselves.	Internal
Process Execution Engine	The process execution engine (PEE) is the central component with the complete process overview, which handles the specified technical process model. The PEE decides which service calls take place under which conditions and controls the process by orchestrating services and resources via interfaces. Automated processes are impacted by activities of human operators, services, and their descriptions, as well as	Internal

	resources and their descriptions. The PEE supports the process controlling with key indicators for evaluating and monitoring the process.	
Reference Architecture	A reference architecture is an architectural design pattern that indicates how an abstract set of mechanisms and relationships realises a predetermined set of requirements. It captures the essence of the architecture of a collection of systems. The main purpose of a reference architecture is to provide guidance for the development of architectures. One or more reference architectures may be derived from a common reference model, to address different purposes/usages to which the Reference Model may be targeted.	Internal
Reference Model	"A reference model is an abstract framework for understanding significant relationships among the entities of some environment. It enables the development of specific reference or concrete architectures using consistent standards or specifications supporting that environment. A reference model consists of a minimal set of unifying concepts, axioms and relationships within a particular problem domain, and is independent of specific standards, technologies, implementations, or other concrete details. A reference model may be used as a basis for education and explaining standards to non-specialists."	[OASIS-RM]
Resolution	Service by which a given *ID* is associated with a set of *Addresses* of information and interaction *Services*. Information services allow querying, changing and adding information about the thing in question, while interaction services enable direct interaction with the thing by accessing the *Resources* of the associated *Devices*. Based on a priori knowledge.	Internal
Resource	Computational element that gives access to information about or actuation capabilities on a *Physical Entity*.	Internal
Requirement	"A quantitative statement of business need that must be met by a particular *architecture* or work package."	[TOGAF9]
RFID	"The use of electromagnetic or inductive coupling in the radio frequency portion of the spectrum to communicate to or from a tag through a variety of modulation and encoding schemes to uniquely read the *identity* of an RF Tag."	[ISO/IEC 19762]
Self-* properties	Self-* properties subsumes the desired capabilities of information systems to be self-configuring, self-organizing, self-managing, and self-repairing.	Internal
Sensor	A sensor is a special *Device* that perceives certain characteristics of the real world and transfers them into a digital representation.	Internal
Security	"The correct term is 'information security' and typically information security comprises three component parts: 1.) Confidentiality. Assurance that information is shared only among authorised persons or organisations. Breaches of confidentiality can occur when data is not handled in a manner appropriate to safeguard the confidentiality of the information concerned. Such disclosure can take place by word of mouth, by printing, copying, e-mailing or creating documents and other data etc.;2.) Integrity. Assurance that the information is authentic and complete. Ensuring that information can be relied upon to be sufficiently accurate for its purpose. The term 'integrity' is used frequently when considering information security as it represents one of the primary indicators of information security (or lack of it). The integrity of data is not only whether the data is 'correct', but whether it can be trusted and relied upon; 3.) Availability. Assurance that the systems responsible for delivering, storing and processing information are accessible when needed, by those who need them.	[ISO27001]
Service	"Services are the mechanism by which needs and capabilities are brought together"	[OASIS-RM]

Service Choreography	Service Choreography is a form of Service Composition in which the participating services interact without being coordinated by a central component. The messaging schema between the elementary services is defined from a global point of view outside the involved services.	Internal
Service Composition	Service Composition denotes the composition of loosely coupled elementary services to form a higher-order composite service to provide additional functionality by re-using existing functionality. IoT-A focusses on hierarchical service composition that follows a black box approach keeping the composition implementation opaque to service consumers.	Internal
Service Composition Model	A Service Composition Model is a formal graphical or textual representation of a Service Composition.	Internal
Service Orchestration	Service Orchestration is a form of Service Composition in which the coordination of the involved services is performed by a central component. This component is called the orchestrator and defines the messaging schema, needed to fulfil the composite service.	Internal
Stakeholder (also referred to as system stakeholder)	"An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a *system*."	[IEEE-1471-2000]
Storage	Special type of *Resource* that stores information coming from *resources* and provides information about *Entities*. They may also include *services* to process the information stored by the *resource*. As *Storages* are *Resources*, they can be deployed either on-device or in the network.	Internal
System	"A collection of components organized to accomplish a specific function or set of functions."	[IEEE-1471-2000]
Tag	Label or other physical object used to identify the *Physical Entity* to which it is attached.	Internal
Thing	Generally speaking, any *physical object*. In the term '*Internet of Things*' however, it denotes the same concept as a *Physical Entity*.	Internal
Unconstrained Network	An unconstrained network is a network of devices with no restriction on capabilities such as storage, computing power, and / or transfer rate.	Internal
User	A *Human* or any *Active Digital Entity* that is interested in interacting with a particular physical object.	Internal
View	"The representation of a related set of concerns. A *view* is what is seen from a *viewpoint*. An architecture *view* may be represented by a model to demonstrate to stakeholders their areas of interest in the architecture. A*view* does not have to be visual or graphical in nature".	[TOGAF 9]
Viewpoint	"A definition of the perspective from which a *view* is taken. It is a specification of the conventions for constructing and using a *view* (often by means of an appropriate schema or template). A *view* is what you see; a *viewpoint* is where you are looking from - the vantage point or perspective that determines what you see".	[TOGAF 9]
Virtual Entity	Computational or data element representing a *Physical Entity*. Virtual Entities can be either Active or Passive *Digital Entities*.	Internal
Wireless communication technologies	"Wireless communication is the transfer of information over a distance without the use of enhanced electrical conductors or "wires". The distances involved may be short (a few meters as in television remote control) or long (thousands or millions of kilometres for radio communications). When the context is clear, the term is often shortened to "wireless". Wireless communication is generally considered to be a branch of telecommunications."	[Wikipedia WI]

Wireline communication technologies	"A term associated with a network or terminal that uses metallic wire conductors (and/or optical fibres) for telecommunications."	[setzer-messtechnik2010]
Wireless Sensors and Actuators Network	"Wireless sensor and actuator networks (WSANs) are networks of nodes that sense and, potentially, control their environment. They communicate the information through wireless links enabling interaction between people or computers and the surrounding environment."	[OECD2009]

References for Glossary

[AIMglobal] Association for Automatic Identification and Mobility, online at: http://www.aimglobal.org/

[AutoI] Information Model, Deliverable D3.1, Autonomic Internet (AutoI) Project. Online at: http://ist-autoi.eu/autoi/d/AutoI_Deliverable_D3.1_-_Information_Model.pdf

[CCSDS 312.0-G-0] Information architecture reference model. Online at: http://cwe.ccsds.org/sea/docs/SEA-IA/Draft%20Documents/IA%20Reference%20Model/ccsds_rasim_20060308.pdf

[COMPDICT-M2M] Computer Dictionary Definition, online at:http://www.yourdictionary.com/computer/m2-m

[E-FRAME] E-FRAME project, available online at: http://www.frame-online.net/top-menu/the-architecture-2/faqs/stakeholder-aspiration.html

[EPCglobal] EPC Global glossary (GS1), online at: http://www.epcglobalinc.org/home/GS1_EPCglobal_Glossary_V35_KS_June_09_2009.pdf

[ETSI-ETR173] ETSI Technical report ETR 173, Terminal Equipment (TE); Functional model for multimedia applications. Available online: http://www.etsi.org/deliver/etsi_etr/100_199/173/01_60/etr_173e01p.pdf

[ETSI TR 102 477] ETSI Corporate telecommunication Networks (CN); Mobility for enterprise communication, online at:http://www.etsi.org/deliver/etsi_tr/102400_102499/102477/01.01.01_60/tr_102477v010101p.pdf

[IEEE-1471-2000] IEEE 1471-2000, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems"

[ITU-IOT] the Internet of Things summary at ITU, online at: http://www.itu.int/osg/spu/publications/internetofthings/InternetofThings_summary.pdf

[ISO/IEC 2382-1] Information technology -- Vocabulary -- Part 1: Fundamental terms, online at:http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=7229

[ISO 27001] ISO 27001: An Introduction To Information, Network and Internet Security

[OGS] Open GeoSpatial portal, the OpenGIS abstract specification Topic 12: the OpenGIS Service architecture. Online at: http://portal.opengeospatial.org/files/?artifact_id=1221

[OASIS-RM] Reference Model for Service Oriented Architecture 1.0 http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[OECD2009] "Smart Sensor Networks: Technologies and Applications for Green Growth", December 2009, online at: http://www.oecd.org/dataoecd/39/62/44379113.pdf

[Sclater2007] Sclater, N., Mechanisms and Mechanical Devices Sourcebook, 4th Edition (2007), 25, McGraw-Hill

[setzer-messtechnik] setzer-messtechnik glossary, July 2010, online at: http://www.setzer-messtechnik.at/grundlagen/rf-glossary.php?lang=en

[TOGAF9] Open Group, TOGAF 9, 2009

[Wikipedia EH] Energy harvesting page on Wikipedia, online at:http://en.wikipedia.org/wiki/Energy_harvesting

[Wikipedia IN] Internet page on Wikipedia, online at: http://en.wikipedia.org/wiki/Internet

[Wikipedia MC] Microcontroller page at Wikipedia, online at: http://en.wikipedia.org/wiki/Microcontroller

[ROZANSKI2005] Software Architecture with Viewpoints and Perspectives, online at: http://www.viewpoints-and-perspectives.info/doc/spa191-viewpoints-and-perspectives.pdf

[Wikipedia WI] Wireless page on Wikipedia, online at: http://en.wikipedia.org/wiki/Wireless