

Master's Thesis

# **A Context Sensor Development Framework for SOA-based Collaborative Working Environments**

carried out at the

Information Systems Institute  
Distributed Systems Group  
Vienna University of Technology

under the guidance of

**Univ.Prof. Mag. Dr. Schahram Dustdar**

and

**Mag. Christoph Dorn**

as the contributing advisor responsible

by

**Florian Schöllhammer**

Matr.Nr. 0325948

Vienna, May 2009

---

## ACKNOWLEDGEMENTS

First of all, I want to express my sincere gratitude to Mag. Christoph Dorn for his constant support, helpful advice and patience in all stages of development of this thesis.

I am also indebted to my dear sister, Sonja Lengauer, who offered me continuous support in all kinds of English related questions and was kind enough to proofread this thesis.

Finally, I wish to express my deepest feelings of appreciation to my beloved parents. Without their mental and financial support I would not have been able to write this thesis. It is to them that I would like to dedicate this thesis.

## ABSTRACT

*In collaborative working environments (CWE), context describes people, artefacts, activities and resources. One major challenge in this domain is to sense context changes. However, the integration and deployment of context sensors often poses a serious problem, since in pervasive SOA-based collaboration environment the sensor developer is not able to change 3rd party services by adding sensor modalities. To mitigate this shortcoming, our approach focuses on analysing the message exchange between service provider and service consumer. In this thesis we design and implement a toolkit that simplifies the development and deployment of SOA-based context sensors, filtering and forwarding SOAP messages as well as managing context data.*

# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background Information</b>	<b>4</b>
2.1	Context and Context-Aware Systems . . . . .	5
2.2	Collaboration and Collaborative Working Environments . . . . .	7
2.3	Service Oriented Architecture (SOA) and Service Composition . . . . .	9
2.4	Context Information in Collaborative Working Environments . . . . .	11
<b>3</b>	<b>Review of the State of Art</b>	<b>13</b>
3.1	A short History of Context-Aware Systems . . . . .	14
3.2	Recent Context-Aware Systems . . . . .	14
<b>4</b>	<b>Problem Statement</b>	<b>17</b>
4.1	The Sensing-Problem in the Software World . . . . .	18
4.2	The Room Reservation Use Case . . . . .	19
4.2.1	Description . . . . .	20
4.2.2	Analysis . . . . .	20
4.3	Development of a possible Solution . . . . .	21
4.3.1	Sensor Integration . . . . .	21
4.3.2	Sensor Composition . . . . .	22
4.4	A Context Sensor Development Framework . . . . .	23
4.4.1	The Scope . . . . .	23
4.4.2	The Solution . . . . .	24
<b>5</b>	<b>Concept of CSDF</b>	<b>27</b>
5.1	The Context Sensor Development Framework . . . . .	28
5.1.1	Service Interceptor . . . . .	28
5.1.2	Controller . . . . .	30
5.1.3	Session Service . . . . .	34
5.1.4	Sensor . . . . .	35
5.1.5	Generator . . . . .	41
5.2	Sensor Model . . . . .	43
5.2.1	Input/Output Specification . . . . .	44
5.2.2	Control Specification . . . . .	45
5.2.3	Service Specification . . . . .	45
5.2.4	Sensor Specification . . . . .	46

5.2.5	Resources and Type System . . . . .	46
5.3	Sensor Filtering . . . . .	48
5.3.1	What is a Filter? . . . . .	49
5.3.2	Definition of Filter . . . . .	49
5.3.3	Filters and Session Management . . . . .	50
5.3.4	Filter Techniques . . . . .	51
5.4	Sensor Composition . . . . .	53
5.4.1	Composition in General . . . . .	53
5.4.2	Compatibility of Sensors . . . . .	55
5.4.3	Loop Detection . . . . .	57
5.4.4	Types of Links . . . . .	58
5.4.5	Active and Passive Sensors . . . . .	60
5.4.6	Composition at Runtime . . . . .	61
5.5	Development Circle . . . . .	64
5.5.1	Create the SensorModel . . . . .	64
5.5.2	Code Generation . . . . .	66
5.5.3	Write Business Logic . . . . .	67
5.5.4	Code Test . . . . .	67
5.5.5	Deployment . . . . .	68
5.5.6	Configuration . . . . .	68
5.5.7	Integration Test . . . . .	69
5.5.8	Activation . . . . .	70
5.6	Service Interaction . . . . .	70
5.6.1	Initialization of Controller . . . . .	70
5.6.2	Registration of a Sensor . . . . .	71
5.6.3	Service Interaction and Sensor Invocation . . . . .	73
<b>6</b>	<b>Sensor Model</b>	<b>77</b>
6.1	Introduction . . . . .	78
6.1.1	Ecore . . . . .	78
6.1.2	SensorModel ID . . . . .	78
6.2	Static Definitions . . . . .	79
6.2.1	SensorModel . . . . .	80
6.2.2	InputOutputSpecification . . . . .	80
6.2.3	PortAbstract . . . . .	83
6.2.4	PortExtract -> PortAbstract . . . . .	84
6.2.5	PortUpdate -> PortAbstract . . . . .	84
6.2.6	IOInput -> IODefinition . . . . .	85

---

6.2.7	IOOutput -> IODefinition . . . . .	85
6.2.8	IOSet -> IODefinition . . . . .	85
6.2.9	IODefinition . . . . .	86
6.2.10	DataSpecification . . . . .	87
6.2.11	Assertion . . . . .	89
6.2.12	AssertionExpression -> Assertion . . . . .	90
6.2.13	AssertionXPath -> AssertionExpression . . . . .	90
6.2.14	AssertionRegex -> AssertionExpression . . . . .	91
6.2.15	AssertionWSOperation -> Assertion . . . . .	91
6.2.16	NamespaceDefinition . . . . .	92
6.2.17	QoSAttribute . . . . .	92
6.2.18	IOReference . . . . .	93
6.2.19	ControlSpecification . . . . .	95
6.2.20	Standard . . . . .	96
6.2.21	StandardStatus -> Standard . . . . .	96
6.2.22	StandardUserDefined -> Standard . . . . .	97
6.2.23	ControlParameter . . . . .	97
6.2.24	ControlAccess . . . . .	99
6.2.25	ControlAccessDefault -> ControllAccess . . . . .	100
6.2.26	ControlAccessUser -> ControllAccess . . . . .	100
6.2.27	ControlStandardAccess . . . . .	101
6.2.28	ServiceSpecification . . . . .	101
6.2.29	ServiceDescription . . . . .	102
6.2.30	ServiceWS . . . . .	102
6.2.31	ServiceSensor . . . . .	103
6.2.32	SensorSpecification . . . . .	103
6.2.33	Resource . . . . .	105
6.2.34	ResourceWithNamespace -> Resource . . . . .	105
6.2.35	ResourceSchema -> ResourceWithNamespace . . . . .	106
6.2.36	ResourceSchemaXsd -> ResourceSchema . . . . .	106
6.2.37	ResourceSensor -> ResourceWithNamespace . . . . .	106
6.2.38	ResourceWSDL -> ResourceWithNamespace . . . . .	107
6.3	Dynamic Definitions . . . . .	108
6.3.1	DataSet . . . . .	108
6.3.2	DataValue . . . . .	109
6.3.3	ParameterValue . . . . .	109
6.3.4	PortReference . . . . .	110
6.3.5	SensorInfo . . . . .	110

---

6.3.6	SensorInfoPort . . . . .	111
6.3.7	SensorInfoIO . . . . .	112
<b>7</b>	<b>CSDF Web Services</b>	<b>113</b>
7.1	Sensor Services . . . . .	114
7.2	SensorIO . . . . .	114
7.2.1	GetIOSpecification . . . . .	115
7.2.2	GetPort . . . . .	115
7.2.3	ListAllForwards . . . . .	116
7.2.4	GetPortForwards . . . . .	116
7.2.5	Type: tForward . . . . .	117
7.3	SensorControl . . . . .	118
7.3.1	ListAllStandards . . . . .	118
7.3.2	GetStandard . . . . .	119
7.3.3	ListAccessForKey . . . . .	119
7.3.4	GetParameterValue . . . . .	120
7.3.5	SetParameterValue . . . . .	121
7.3.6	ListResources . . . . .	123
7.3.7	GetResourceByNamespace . . . . .	123
7.3.8	GetNamespaceByPrefix . . . . .	124
7.4	SensorService . . . . .	125
7.4.1	ListAllServices . . . . .	125
7.4.2	GetSelf . . . . .	125
7.5	SensorCore . . . . .	126
7.5.1	Invoke . . . . .	126
7.5.2	UnregistrationNotification . . . . .	128
7.5.3	IsAlive . . . . .	129
7.6	SensorManagement . . . . .	130
7.6.1	Initialize . . . . .	130
7.6.2	Activate . . . . .	131
7.6.3	IsActive . . . . .	132
7.6.4	Passivate . . . . .	133
7.6.5	Type: tServiceType . . . . .	134
7.6.6	Type: tForward . . . . .	134
7.7	Controller . . . . .	135
7.7.1	Register . . . . .	135
7.7.2	Unregister . . . . .	137
7.7.3	SetActiveStatus . . . . .	138

7.7.4	ListAllServices . . . . .	139
7.7.5	ListAllServicesDetails . . . . .	139
7.7.6	GetServiceByCore . . . . .	139
7.7.7	GetServiceByRequirements . . . . .	140
7.7.8	ListAllActiveServices . . . . .	141
7.7.9	Initialize . . . . .	141
7.7.10	Shutdown . . . . .	142
7.7.11	GetCompatibleInputPorts . . . . .	143
7.7.12	GetCompatibleOutputPorts . . . . .	144
7.8	Session Service . . . . .	144
7.8.1	Get . . . . .	145
7.8.2	Set . . . . .	146
7.8.3	Delete . . . . .	147
7.8.4	SessionCreate . . . . .	148
7.8.5	SessionDestroy . . . . .	149
<b>8</b>	<b>How To</b>	<b>151</b>
8.1	Introduction . . . . .	152
8.2	The SensorModel . . . . .	153
8.2.1	Creating a new SensorModel . . . . .	153
8.2.2	Edit the SensorModel . . . . .	154
8.2.3	Adding the Input/Output Specification . . . . .	154
8.2.4	Adding the Control Specification . . . . .	158
8.2.5	Adding the Service Specification . . . . .	160
8.2.6	Adding the Sensor Specification . . . . .	161
8.2.7	The finished SensorModel . . . . .	163
8.3	Code Generation . . . . .	164
8.3.1	Setup . . . . .	165
8.3.2	Code Generation . . . . .	165
8.4	The generated Code . . . . .	166
8.4.1	Eclipse Project . . . . .	166
8.4.2	Generated Packages . . . . .	167
8.4.3	Generated Files . . . . .	169
8.4.4	Commands . . . . .	170
8.4.5	The altered SensorModel . . . . .	171
8.4.6	Extension Points . . . . .	173
8.5	Code Extension . . . . .	174
8.5.1	Writing the Extraction Port Code . . . . .	174



8.5.2	Writing the Update Port Code . . . . .	175
8.6	Code Test . . . . .	177
8.6.1	Session-data Files . . . . .	177
8.6.2	Test of the Extraction Port . . . . .	178
8.6.3	Test of the Update Port . . . . .	179
8.7	Deployment . . . . .	181
8.7.1	Deploying the Sensor . . . . .	181
8.7.2	Empty Initialization . . . . .	182
8.7.3	Overview of ConfigAssistant . . . . .	183
8.7.4	Initialization via the ConfigAssistant . . . . .	188
8.8	Integration Test . . . . .	189
8.8.1	SOAP Request and Response . . . . .	189
8.8.2	Start the integration test . . . . .	191
8.8.3	Final Activation . . . . .	193
<b>9</b>	<b>Evaluation</b>	<b>194</b>
9.1	Use Case: Mailinglist . . . . .	195
9.1.1	Description . . . . .	195
9.1.2	Sensor Design . . . . .	195
9.1.3	Sensor Logic . . . . .	198
9.2	Use Case: Room Reservation . . . . .	198
9.2.1	Description . . . . .	198
9.2.2	Sensor Design . . . . .	199
9.2.3	Sensor Logic . . . . .	201
9.3	Use Case: Load Document . . . . .	202
9.3.1	Description . . . . .	202
9.3.2	Sensor Design . . . . .	202
9.3.3	Sensor Logic . . . . .	204
<b>10</b>	<b>Conclusion and Outlook</b>	<b>206</b>
10.1	Conclusion . . . . .	207
10.2	Summary of Contributions . . . . .	208
10.3	Outlook . . . . .	209
<b>A</b>	<b>Installation Guide</b>	<b>212</b>
A.1	System Requirements . . . . .	214
A.2	Prerequisites . . . . .	214
A.2.1	Java JDK . . . . .	214
A.2.2	Ant . . . . .	214

---

A.2.3	Logging Service . . . . .	215
A.3	Installation . . . . .	215
A.3.1	Tomcat Apache . . . . .	215
A.3.2	Eclipse . . . . .	216
A.3.3	Eclipse Add-ons . . . . .	216
A.3.4	Axis2 . . . . .	218
A.4	Configuration . . . . .	219
A.4.1	Eclipse Settings . . . . .	219
A.4.2	Session Service . . . . .	220
A.4.3	Controller . . . . .	220
<b>B</b>	<b>Source Code Listing</b>	<b>222</b>
B.1	Ecore Meta-Model . . . . .	223

# LIST OF FIGURES

1	Sentient Object Model . . . . .	16
2	Use Case: Room Reservation . . . . .	20
3	CSDF Concept . . . . .	29
4	Is-Alive Concept . . . . .	31
5	Sensor Interfaces . . . . .	36
6	Schematic Overview of the Sensor . . . . .	37
7	Integration of Resources . . . . .	47
8	Sensor Composition . . . . .	54
9	Sensor Composition . . . . .	54
10	Examples of loops . . . . .	57
11	Forward Example 1 . . . . .	59
12	Forward Example 2 . . . . .	59
13	Example of active and passive Sensors . . . . .	61
14	Invocation of active and passive Sensors . . . . .	64
15	CSDF Development Circle . . . . .	65
16	A SensorModel created via the EMF Model Editor . . . . .	66
17	Initialization using the ConfigAssistant . . . . .	69
18	Controller Initialization Sequence . . . . .	71
19	Sensor Initialization Sequence . . . . .	72
20	Service Interaction and Sensor Invocation (Part 1/2) . . . . .	74
21	Service Interaction and Sensor Invocation (Part 2/2) . . . . .	75
22	MessageSensor - Overview . . . . .	152
23	SensorModel Wizard (Page 1) . . . . .	154
24	SensorModel Wizard (Page 2) . . . . .	155
25	SensorModel Wizard (Page 3) . . . . .	155
26	SensorModel Editor . . . . .	156
27	Input and Output Specification of SensorModel . . . . .	158
28	Control Specification of SensorModel . . . . .	160
29	Service Specification of SensorModel . . . . .	161
30	Sensor Specification of SensorModel . . . . .	163
31	Libraries in the Build Path of Project . . . . .	167
32	The ConfigAssistant . . . . .	184
33	ConfigAssistant - Port Details . . . . .	185
34	ConfigAssistant - Enter Forward . . . . .	185
35	ConfigAssistant - Compatible Ports . . . . .	186

36	Use Case: Mailinglist . . . . .	196
37	Use Case: Room Reservation . . . . .	199
38	Use Case: Load Document . . . . .	203

---

# 1 INTRODUCTION

## INTRODUCTION

Context, though naturally comprehended by humans, is something computers usually fail to perceive. Making computers aware of the context they are operated in can substantially enhance their capabilities and efficiency. The device to gather aspects of the environment - the actual context - is called sensor. The physical world resorts to hardware-sensors in order to measure environmental attributes such as temperature, velocity, location, humidity, etc. Software-sensors, on the other hand, are programs that calculate and collect data like transfer rates, service invocations, message exchange, etc.

A Collaborative Working Environment (CWE) is a computer system that supports experts in a very heterogeneous environment (e.g. different organisations, physical separation, multiple teams, various projects, time shift) in their work. Especially in the domain of CWE, context awareness plays a crucial role to efficiently assist the users in the specific tasks to be accomplished.

In Service-of-Architecture (SOA)-based systems, however, becoming aware of context changes proves to be a challenging task. Frequently, the sensor developer is not authorised to extend existing software with sensory logic. Particularly when dealing with third party services, sensor integration may not be feasible.

One way to overcome this obstacle is to extract context solely from service requests and responses. Rather than directly attaching a sensor to the service provider, the message transfer between service provider and service consumer is observed and analysed. To support sensor developers in creating such context sensors, a development framework is required.

Yet another problem in the context sensing domain is the composition of sensors. Context attributes typically cannot be described by atomic, measurable aspects of the environment, but are a combination of such. An important requirement for a context development framework is hence the capability to compose and reuse sensors in a flexible and dynamic way.

In this thesis, we design and implement the Context Sensor Development Framework (CSDF) - a framework which allows the development and deployment of SOA-based context sensors, based on the datamining-approach described above. Sensors in CSDF are defined by a programming language-independent model, the so-called SensorModel. Following the specification of the SensorModel on part of the developers, CSDF automatically generates the language-specific sensor implementation. The only task that remains to be done is to implement the actual business logic of

the sensor.

The framework supports extensive mechanisms for sensor composition. Sensors can be both producers and consumers of context information. So-called Forwards allow for the combination of sensors in flexible ways, enabling the developer to build up complex context extracting networks. In addition, CSDF integrates a sensor registry, so new sensors can be added dynamically. The registry also provides a mechanism to identify sensors compatible with a given specification, by means of which the developer can seamlessly integrate a newly developed sensor into existing sensor compositions.

This thesis is structured as follows: Chapter 2 and 3 are intended to give background information and study the history of context-aware applications. Chapter 4 will see an analysis of the problems of state-of-the-art systems as well as a solution developed to solve these challenges. Chapter 5 is then devoted to a thorough analysis of CSDF and its components. The following two chapters, chapter 6 and 7, will discuss the SensorModel and Web services of CSDF in detail. A guide explaining how to develop sensors will be provided by means of a concrete example in chapter 8. In chapter 9, we shall evaluate the possibilities of CSDF by means of real world scenarios, whereas the final chapter, chapter 10, will summarise the results and contributions made in this thesis.

---

## 2 BACKGROUND INFORMATION

This chapter starts with an introduction of why context awareness is beneficial for computer applications, before moving on to the definition of context and explaining the major concepts and components in context-aware systems. The second part explains the term collaborative working environment (CWE) and give examples of frameworks. Part three offers a description of the concepts of SOA (service oriented architecture) and the final part is devoted to analysis of context information in the setting of CWE.



## 2.1 CONTEXT AND CONTEXT-AWARE SYSTEMS

In human conversation, mutual understanding is reached not solely by the information that is conveyed through the actual communication itself, but by means of its interpretation in the environment - the context. Natural language as a means of human-human conversation, for instance, is interpreted according to the situation in which it is used. Thus the significance of the word 'small', to take an example, is different in the context of macrocosm (e.g. 'small planet') and zoology (e.g. 'small bug'). For humans this is implicitly inferred due to their common understanding of the context. The potential of human-computer dialogue, in contrast, is greatly limited due to lack of context-awareness. A cell phone, for instance, ought to react differently to incoming calls in different environments. If the user is outside, it may ring; if the user is in a meeting, it should vibrate silently. Whereas this context is obvious to the user, it is not to the computer (i.e. the mobile phone). By providing the computer with the necessary context information, the usefulness of applications can be improved significantly.

According to Dey and Abowd (2000) [11], there are two ways to make the computer aware of the context. The first one is to have the user convey all the required context information directly in the course of the interaction. However, it would be tedious to transmit all additional context information, and it might also be difficult in some situations for the user to decide on the relevance of a particular piece of information. The second approach is therefore preferred: Here the system is provided with means to automatically determine the context itself (e.g. sensors, network information), thus putting the decisions which information is relevant into the hands of the application developer while keeping the actual conversation with the user simple. This approach leads to the development of so-called context-aware systems, but let us first discuss the meaning of the word 'context'.

Although there were many attempts to specify what context is, there is no general definition. Schilit and Theimer (1994) [34], for instance, define context as location, identification of users, objects and object changes while Brown (1996) [7] specifies context as all the elements of the user's environment that the computer knows about. Since context-awareness became more important at the emerge of mobile devices and applications, which have to be able to adapt to rapid context changes, location is an attribute often to be found in context definitions (e.g. Ryan et al., (1997) [29]). However, especially in recent years many attempts were made to include other information as well (e.g. Hull et al., (1997) [21]). As we will see, context - as information that is considered as relevant - always depends on the application itself.

But what are context-aware systems? Again there are many attempts to describe the term context-awareness. One good definition can be found in Dey and Abowd (2000) [11]:

*A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.*

Especially as a field of ubiquitous computing [47], context-aware systems become more and more important as a means to realise pervasive computing. According to Baldauf et al., (2007) [3] context-aware systems can be classified in various ways:

◇ **Context Acquisition**

- *Direct Sensor Access.* The application directly accesses and retrieves required information from sensors.
- *Middleware Infrastructure.* The application is built on top of a middleware, which handles the sensor management and therefore hides low-level sensor details from the application.
- *Context Server.* Instead of accessing sensors, the application queries a context managements system which holds the required context information and can be accessed by multiple clients.

◇ **Context Management Models**

- *Widgets.* Widgets are software components that hide the low-level sensor access logic from the application by providing a public interface. They are managed by a global widget manager.
- *Network Services.* Instead of a global widget manager, discovery mechanisms are used to find sensing-services during runtime.
- *Blackboard Model.* In this approach, retrieved data is posted to a common store - a blackboard - and can be retrieved by applications via subscription mechanisms.

◇ **Sensor Types**

- *Physical Sensors.* Aspects of the physical world (e.g. location, volume) are measured with hardware sensors (e.g. GPS, microphone)
- *Virtual Sensors.* Context data is collected by using services or software applications (e.g. electronic calendar, email)

- *Logical Sensors*. New information is inferred by combining data from one or more sensors with additional data from databases or other data source (e.g., resolving city or region name from GPS coordinates)

As described in Ailisto et al., (2002) [1], context-aware systems often show a layered structure.

- ◇ **Sensor layer**. This is the lowest layer and it is responsible for accessing the data sources (which might but must not necessarily be sensors). An example is the driver for a hardware sensor. This layer represents some kind of API to the next layer which can be used to retrieve data from the source.
- ◇ **Raw data extraction layer**. Using the API from the first layer, data is gathered. This layer is used to define some kind of abstract component that provides context data while hiding internal low-level details. If the definitions are held abstract enough, it should be possible to exchange components.
- ◇ **Preprocessing layer**. In case that the raw data needs pre-processing, this layer can be used. It transforms the raw data to a more usable form by methods of combining, reasoning and inferring. Example: Resolve GPS coordinate to region/city name.
- ◇ **Management and storage layer**. Here data is stored and published via a common interface, so it can be queried by applications. This is often realised with a context management server that does not only provide a standard query interface but also useful functions such as history, etc. Example: Store context in an OWL<sup>1</sup> database.
- ◇ **Application layer**. This last layer realises the client application which adapts its behaviour to the context given.

## 2.2 COLLABORATION AND COLLABORATIVE WORKING ENVIRONMENTS

Recent years have seen a shift from individual work to cooperative work in an assembled group. Working together in a group to solve a given task might pose several

---

<sup>1</sup><http://www.w3.org/TR/owl-features/> (last access: 2009-03-31)

challenges like different locations and working times, access to common resources, sharing of knowledge, decision finding, etc. Above that, a worker might not only be member of one group but of several, each with different settings. In order to work cooperatively and efficiently in such a complex environment, collaboration is needed.

In Pallot et al., (2004) [26] and Pallot et al., (2005) [25] collaborations are described in three layers:

- ◇ **Communication** - In the lowest layer the parties have a collaborative partnership. Although they still work individually, they share parts of information with each other. What and how data is exchanged is defined in interface specifications.
- ◇ **Coordination** - The next step is to coordinate goals and tasks and share common object such as calendars, agendas, etc. On this level the parties still work separately, but they synchronise tasks with each other, all of which is managed by a coordinator or project manager.
- ◇ **Co-operation** - Here at last the parties share a common workspace and work in a cooperative manner. This does not only refer to a technical solution for a shared workspace, but the members also need the vision and a common understanding of cooperative work.

A collaboration working environment (CWE) [49] must support the participants on all of these layers and enables them to efficiently perform collaboration work. Examples of existing CWE are TeamSCOPE [35], MOSAIC [30] [32] [31] or ECOSPACE [27] [28].

A very interesting CWE project is inContext [39] [23], which tackles the problem of dynamic team coordination. In collaborations, workers often belong to different organisations, frequently change their locations and are part of teams that are dynamically assembled and dissolved - all this while using a common environment (resources, infrastructure, etc.). inContext is a CWE that efficiently supports collaboration and their workers in such dynamic environments.

## 2.3 SERVICE ORIENTED ARCHITECTURE (SOA) AND SERVICE COMPOSITION

SOA is a software architecture that evolved from distributed computing. In contrast to modular programming (software is split into different modules to realise separation of concern [14]) and distributed computing (a program is divided into parts that run on different machines often in a heterogeneous environment), in SOA system functionality is packed into interoperable services. The services are unassociated and loosely coupled. The heterogeneity as well as other implementation details of the services are hidden behind a service-interface. Applications in SOA are then built out of services, often using some form of composition [50].

Design Principles of SOA: [16] [17]

- ◇ **Standardised Service Contract:**

Services have a technical interface description that describes the purpose and the capabilities of the service as well as other functional and non-functional requirements, depending on the technology used. The interface must be public.

- ◇ **Loose Coupling:**

Coupling describes how much one component relies on another or what kind of assumption it makes about it. Loose coupling therefore refers to a maximum of loosening the relation and the dependencies between software parts, i.e. services in SOA.

- ◇ **Abstraction:**

Abstraction stands for hiding - except for the public interface - internal logic and implementation details from the outside world.

- ◇ **Reusability:**

Reusability plays an important role in SOA. Functionality is packed as service with the intention to be reused.

- ◇ **Autonomy:**

For services to fulfil their task reliably and consistently, they need to have a certain degree of control over the software they encapsulate.

- ◇ **Statelessness:**

Especially in SOA, scalability of software is an important requirement. To keep

scalability at its highest, services are to be designed in a way that requires them to keep no (or only a minimum of) internal state.

◇ **Discoverability:**

To promote reusability, services need to be easily found by discovery mechanisms. To realise this, an elaborate description of what the service does and how it is accessed is important.

◇ **Composability:**

As complex software applications are built by composing services, services must be designed in a way that allows easy composition.

One possible and widely used implementation of SOA are Web services. The most basic technologies that make up the Web service model are:

- ◇ **Simple Object Access Protocol (SOAP)** <sup>2</sup> - as communication protocol between service provider and service requester.
- ◇ **Web Service Description Language (WSDL)** <sup>3</sup> - as specification language to describe the interface of a Web service.
- ◇ **Universal Description, Discovery and Integration (UDDI)** <sup>4</sup> - as service registry.

An introduction to Web services and related standards will not be given in this thesis. For more profound information about Web services, the reader might refer to Alfonso et al., (2004) [2].

Web-applications often consist of many diverse Web services. To master complexity and to encapsulate extensive functionality, services can be composed [15]. According to Alfonso et al., (2004) [2] composition models can be categorised as follows:

- ◇ **Component model:** It describes what kind of assumptions can be made about the elements composed. Do the elements comply with a Web service standard (e.g. HTTP, SOAP, WSDL) or do they just exchange data in form of XML messages?

---

<sup>2</sup><http://www.w3.org/TR/soap/> (last access: 2009-03-31)

<sup>3</sup><http://www.w3.org/TR/wsdl/> (last access: 2009-03-31)

<sup>4</sup>[http://uddi.org/pubs/uddi\\_v3.htm](http://uddi.org/pubs/uddi_v3.htm) (last access: 2009-03-31)

- ◇ **Orchestration model:** It describes the language and abstractions that are used to realise composition. The language, in turn, then specifies how, in which order and under which conditions services are to be invoked.
- ◇ **Data and data access model:** It specifies how data in a composition is defined, and how it is exchanged between services. Many systems nowadays also support XML Schema types apart from their own system specific types (e.g. BPEL <sup>5</sup>).
- ◇ **Service selection model:** It defines how a particular service is selected for invocation and how static or dynamic binding takes place.
- ◇ **Transaction:** It describes how compositions are associated with transactions and how transactions are realised.
- ◇ **Exception handling:** Systems can be distinguished by the way they handle exceptions during the flow. Is there a way to define alternatives or recovery functionality in case of errors?

An example of a Web service composition language is BPEL4WS (Business Process Execution Language for Web Services).

## 2.4 CONTEXT INFORMATION IN COLLABORATIVE WORKING ENVIRONMENTS

In the first part of this chapter, we analysed the term context as well as possible context definitions. Now we are going to identify important context elements in the setting of CWE.

Based on the VieCar project [33] and the comprehensive Context Model [42] developed in the course of the inContext project, the following context information is typically found in CWE:

- ◇ **User** - A CWE involves users/actors. Usually a user is identified by a unique id (e.g., using FOAF <sup>6</sup>), name, contact information as well as organisation, team and project involvement.

---

<sup>5</sup><http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html> (last access: 2009-03-31)

<sup>6</sup><http://www.foaf-project.org/> (last access: 2009-03-31)

- ◇ **Time** - Events in a CWE take place at a certain point in time.
- ◇ **Location** - CWE involves users from different organisations who are physically separated. The location of the user is an important aspect in CWE. It can contain GPS-coordinates as well as the postal address of the place the user is currently located at. The inContext Context Model also includes a concept of mobility and relocation in their location-ontology.
- ◇ **Skill** - Especially for team and project assignment, it is important to know which skills users have.
- ◇ **Team** - Users are organised into teams to work on a particular task. Teams can comprise other teams.
- ◇ **Role** - Users have different roles in an organisation and also different roles in different teams. This describes which roles (and maybe also which responsibilities and privileges) a user has in a given setting.
- ◇ **Activity** - Activities - as the central part of CWE - describe a particular task as a part of work that needs to be done. They involve team members, actions, artefacts and resources. Activities can relate to each other or be parent/child of another activity. They can have a start-time and duration as well as a certain priority.
- ◇ **Action or Context** - This describes atomic actions which were taken by a user as well as his/her current status (e.g. online status, location, etc.). Actions always take place in the setting of an activity and at a certain point of time and involve users. They are used to describe the progress of an activity as well as the interaction between users.
- ◇ **Artefact** - Artefacts are objects that can be created or modified by users (e.g. documents). Artefacts are linked to an activity and are used by users in actions.
- ◇ **Resource** - A resource is a very general concept and describes everything that can be used by a user (e.g. artefacts, vehicles, rooms, tools)



---

### 3 REVIEW OF THE STATE OF ART

This section presents a short overview of the history of context-aware systems. After briefly surveying pioneering systems such as the Active Badge System and giving examples of other major contributions in this field, we will move on to more recent context-aware applications and describe systems like SOCAM, COBRA, CASS, inContext and ESCAPE. These state-of-the-art systems will be analysed with special focus on their important features as well as the drawbacks of their respective designs.

## 3.1 A SHORT HISTORY OF CONTEXT-AWARE SYSTEMS

One of the first context-aware systems was the Active Badge System developed at Olivetti Research Lab [46]. Based on the location of people, which was determined by a special badge they were wearing, phone calls were redirected. Another interesting project was CyberDesk [12], a framework for the dynamic integration of software applications depending on the users' context. The context was extracted from on-screen information like emails, dates, addresses, etc. In later versions, location-awareness and time-information were added to the context, enabling more sophisticated usage [36]. Projects like TEA (Technologies for Enabling Awareness) [44] and mediacups [5] aimed at integrating context sensors into everyday objects. TEA equipped a mobile phone with additional sensors so it would adapt its behaviour (e.g., when to switch to silent mode) according to the user's current situation. In Mediacup, sensors were integrated into an ordinary coffee cup, sensing properties like temperature and movement (static, cup-is-moving, drinking-out-of-the-cup). In this sense we are gradually beginning to realise the pervasive computing vision first described by Weiser (1999) [48].

Another example of a context management system is Kimura [45], which was an attempt to combine physical data of hardware sensors with information of virtual sensors. Solar [8] introduces a framework with a context-sensitive query language to manage context-sensitive name descriptions. The Context-Toolkit [13] is a toolkit to easily create context-processing software components. The components can interact with each other to ultimately deliver usable context-data to the applications.

A quite interesting project was HotTown [22], which aimed at adding context-aware service support to moving locations. In HotTown, users were equipped with a mobile computer that hosted their personal location-aware agent - a GeoBot. A GeoBot was aware of other bots in its surrounding area and had its own knowledge, which he could communicate with others. The user could therefore interact with other bots, with the environment and even with virtual users who were surfing the webpage of HotTown at the same time.

## 3.2 RECENT CONTEXT-AWARE SYSTEMS

The Context Broker Architecture (COBRA) [10] [9] is an agent-based architecture supporting context-aware computation. Intelligent systems, which are part of a so-

called intelligent space (an existent physical place), provide the user with intelligent context-aware services. The centre of the architecture is the Context Broker, which manages the sharing of knowledge between its agents. For knowledge representation OWL is used.

Embedded in the Context Broker is the context-acquisition module, which gathers context-information from various sources (hardware sensors, sensing middleware-infrastructure, context-servers). A drawback of this design is the inability to create composite sensing units. Moreover, all data-processing needs to be done in the Context Broker.

An example of a middleware to prototype context-aware service is SOCAM (Service-Oriented Context-Aware Middleware) [20]. A central server merges the knowledge it receives from distributed context providers and then represents it to the clients. Context providers can be both physical and virtual sensors. They are registered in a service registry and can then dynamically be found by others. Applications built on top of the middleware can then freely access context information and adept their behaviour accordingly. Sensors in SOCAM are directly operated by context providers, which in turn represent the context-information in form of OWL descriptions. SOCAM does not provide any mechanism to compose sensors.

Another centralised middleware approach is realised in CASS (Context-Awareness sub-structure) [18]. It is specifically designed to support context-aware applications on low-performance mobile devices. Another feature is the high context abstraction and the strict separation between context inference and application code. Similar to the previously introduced systems, CASS does not support sensor composition and furthermore lacks both sensor registry and discovery mechanisms.

The CORTEX System [6] is a context-aware middleware approach which is based on the Sentient Object Model [19]. A Sentient Object (Figure 1) is a component that encapsulates a context capture (which receives context and converts it into a suitable format), a context hierarchy (which encapsulates context about actions to be taken) and the inference engine (changing the behaviour according to the context). A Sentient Object can be both consumer and producer of events, which enhances the flexibility and reusability of the system tremendously. Sentient Objects in CORTEX are programmed in STEAM (Scalable Timed Events and Mobility) [24], an event-based middleware for mobile devices.

JCAF (Java Context Awareness Framework) [4] supports the programmer in the development of context-aware applications in the Java framework. Context information can be added as well as retrieved via predefined services. The framework is based on a peer-to-peer architecture.

The ESCAPE framework [38] is a Web service-based context framework spe-

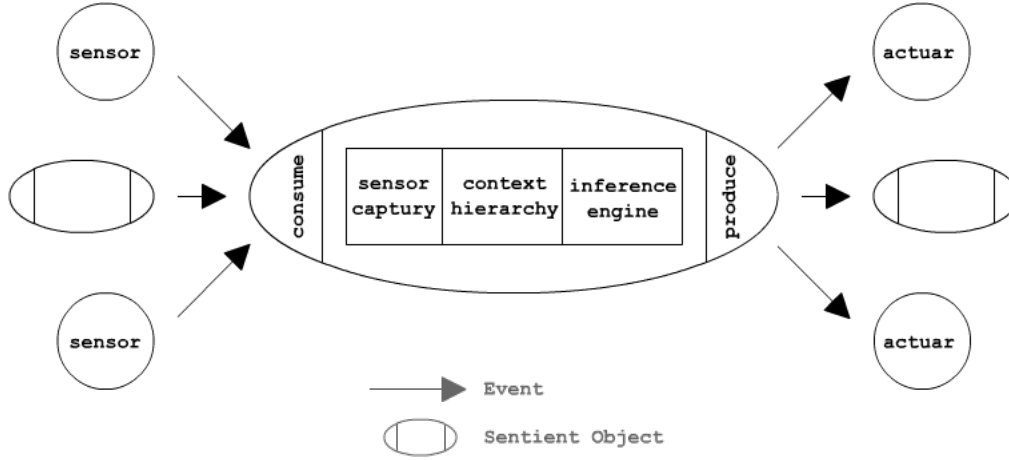


Figure 1: Sentient Object Model

cially developed to add context-sensitive support to tasks in case of an emergency or disaster. The architecture is split into a backend (storing and sharing of context-information) and the frontend (sharing data and context-sensing). The frontend components of ESCAPE are CIMS (context information management services), which autonomously build up peer-networks. In each network, one CIMS will act as super-peer. It is then responsible for gathering context data from the network and pushing it to the back-end as well as for synchronizing context from the backend and distributing the update to the other CIMS. Thus, CIMS can both accept context-information as well act as context-provider. Yet compared to the Sentient Object approach, it lacks the ability for composition.

As already mentioned, the inContext project [39] [23] is also an example of a web-based context management framework specifically designed to support team collaborations in their work. A remarkable feature of this system is that it extracts context from Web service interaction-logs through mining [41].

A comprehensive survey by Truong and Dustdar (2008) [37] analyses and compares state-of-the-art web-based context-aware systems.

---

## 4 PROBLEM STATEMENT

The first part of the chapter identifies shortcomings of state-of-the-art context-aware systems in the area of software-sensing, i.e. the problems of sensor integration and sensor composition, demonstrated in a sample use case. In a next step, a general approach to solve the above challenges will be outlined. Following a definition of the scope of a possible solution based upon this approach, the Context Sensor Development Framework (CSDF) will be presented as an adequate solution to the given problems.

## 4.1 THE SENSING-PROBLEM IN THE SOFTWARE WORLD

As discussed in the previous part, recent years have seen the development of a wide range of context-aware applications. Much effort was put into the process of rendering applications aware of the physical world, thus creating systems capable of sensing time, location, alignment, temperature, humidity and many more properties of the physical world (e.g. TEA, MediaCups). Subsequent systems (e.g. SOCAM) began to integrate virtual sensors as well, which made even more complex context-driven behaviours feasible. Later, with the popularity of Web services, some applications were adapted to be used in SOA (service-of-architecture)-based systems. However, a mere adapter to make applications web-enabled does not address some important aspects that have to be considered when creating web-applications. According to Truong and Dustdar (2008) [37], there are some major differences between ordinary context-aware applications and Web service based ones:

- ◊ In contrast to ordinary applications, which are only used within one organisation, web-based systems are often multi-organisational. As a consequence, open standards rather than proprietary solutions must be used.
- ◊ While early context-systems are often tightly coupled, Web service based systems have to be loosely coupled.
- ◊ In Web service based systems, data-confidentiality, having played a minor role before, becomes a central issue.

Pure Web service oriented applications (e.g. inContext) usually do not present problems like these.

The following part focuses on 'sensing' in context-aware systems. Although many context-aware applications already provide very sophisticated mechanisms in the field of hardware-sensing (e.g. COBRA, SOCAM), fundamental problems remain unsolved in the domain of software-sensing:

- ◊ **Sensor Integration:** In order to become aware of a certain attribute or event, a sensor is needed. Hardware sensors measure properties of the physical world and convert them into a computer-readable format. By either *pulling* (i.e., the system directly queries the sensor and reads the current value) or by *pushing* (i.e., the system is triggered by an event generated by the sensor) the application is rendered aware of the sensor-data. In this sense, the developer of

a system solely has to select and install the proper sensor serving his needs, thus being in control of the sensor. Even if the sensor is pre-installed and cannot be changed, it is - since developed as such - equipped with either a pushing or pulling mechanism that enables the system to sense current values or value-changes in the measured environment.

In the software domain, this proves to be more complicated: Almost any event could be a desired trigger for some kind of action. In order to become aware of such an event, a sensor reactive to it has to be developed. A simple and effective way would be to extend the examined system by attaching a sensor directly to it. Apart from performing its work, the system would therefore also send events to the context-aware application. This approach works fine, provided that the surveyed system is accessible and extendable by the developer. Yet if the system was developed by a 3rd-party and is only offered as a service, it is out of reach of the developer and can therefore not be extended with a sensor. This actually poses a serious problem, which is especially present in Web service based architectures. So far, no practical solution to solve this problem has been found.

- ◇ **Sensor Composition:** In many environments, a sensor might not only measure an atomic value, but sense more complex circumstances. As example serves a mobile phone: If moving at high velocity (location sensor) and engine-sound is present (microphone), the phone is probably located within a car. In order to determine the setting *in-car* with a sensor, data of the location sensor and the microphone must be combined.

Whereas there are many systems which allow such sensor composition in the hardware world (e.g. Kimura), even most recent systems in the Web service domain (e.g. SOCAM, CASS and ESCAPE) still lack the ability to compose sensors in such a way.

## 4.2 THE ROOM RESERVATION USE CASE

A simple use case aims at illustrating the the problems stated above.

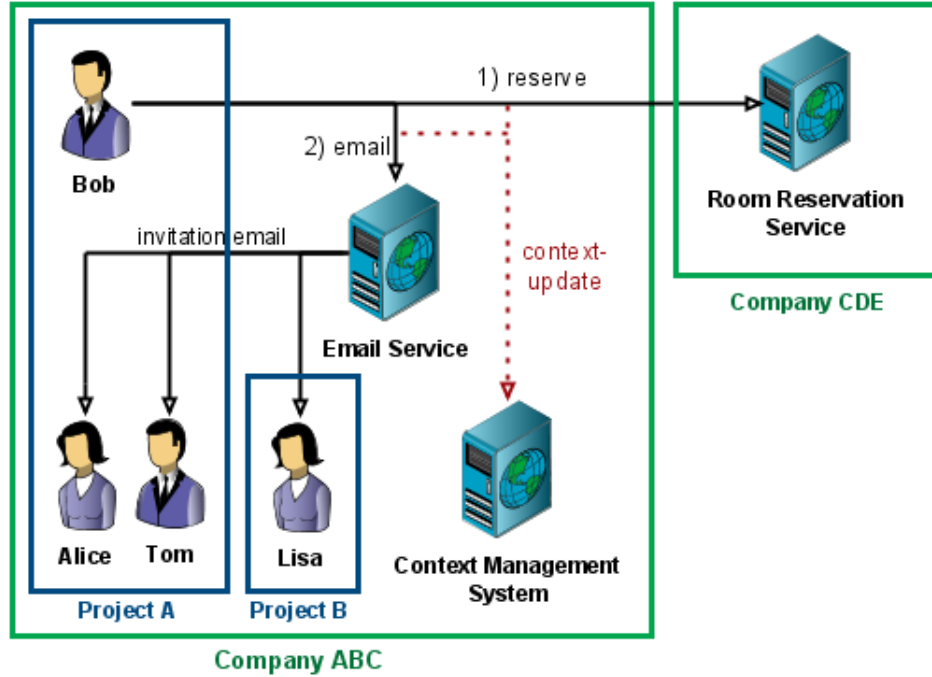


Figure 2: Use Case: Room Reservation

#### 4.2.1 Description

Bob (a worker of company ABC) wants to hold a meeting to discuss and plan the development of project A. As a first step he uses the Room Reservation Service of company CDE to reserve a meeting room on Friday morning. Upon confirmation, he sends out emails using the Email Service to Alice and Tom of Project A to inform them about the time and place of the upcoming meeting. He also invites Lisa, a member of Project B. She is needed to help in the design of a common interface used in both projects.

A sensor-enabled CWE system supports Bob in his work. Apart from Alice and Tom (who are already members of the project), it automatically adds Lisa as an involved-actor to the activity-context of Project A (Figure 2).

#### 4.2.2 Analysis

For a CWE to support such a use case, it needs to be capable of the following:

- ◊ First it has to become aware of events such as *reserve-a-room* and *send-an-email*. Although services in a CWE may partly be developed and therefore be



extendable by the company itself, this is not always the case. In our example, CDE - a company providing conference rooms - has no connection to ABC. As a consequence, it is not possible to attach a software-sensor to the Room Reservation Service.

- ◇ Secondly, events need to be coordinated. In our use case, the CWE action is triggered if emails are sent after a successful room-reservation - all executed by the same user in the same activity-context and probably only within a limited time frame. The sensor for detecting emails and the sensor for room-reservations thus have to be composed to trigger the update-action, i.e. adding Lisa as an involved actor.

## 4.3 DEVELOPMENT OF A POSSIBLE SOLUTION

This chapter is devoted to the question of how a possible solution to the given problems could look like and which additional requirements would have to be met.

### 4.3.1 Sensor Integration

While systems such as SOCAM, CASS, ESCAPE and the like do not yet provide a means to develop software sensors for not directly observable environments, inContext has already proposed a promising approach to solve the problem of sensor integration. It assumes that at least the actual invocation of services is observable [41]. If this is the case, the requests and responses can be logged. Extended with additional data (timestamp, message-id, etc.) it is possible to reconstruct the original series of invocation. Thus we can build a sensor which analyses the log and generates desired events.

Of course, developing code for every single sensor is tedious. Some kind of framework is needed which automates the sensor-base (XML parsing, sensor publishing mechanism, etc.) so that the programmer can focus on developing the business code. inContext, however, still lacks an efficient mechanism to create sensors for analysing invocation logs.

The framework to be developed has to meet some additional requirements, i.e.:

- ◇ **Discovery:** In the area of sensor-development, not all sensors are initially known by the system. Sensors might be added at a later date or may be

updated. Similarly, obsolete sensors might be removed from the system. To support dynamic sensor-management, a registry is required. While systems such as CASS still lack such a registry, others like SOCAM are already providing them in their applications. Still, certain problems remain unsolved. A sensor developer, for instance, sometimes needs to find all sensors which provide output compatible to a given sensor-input. A complex compatibility-operation like this goes far beyond simple parametric search (and can therefore not be done in SOCAM). To make it feasible, the input and output of a sensor must be clearly specified in the first place, and secondly, a comparison algorithm must be developed to verify whether one specification comprises another.

- ◇ **Framework Independence:** The systems discussed so far (e.g. ESCAPE, SOCAM, CASS) are full-fledged context-aware middlewares. As a result, the sensor-layer logic is tightly integrated into the whole application. Sensing, however, is a very complex mechanism that should not be reinvented with every new system. There is clearly a lack of a flexible and powerful sensor development framework that can seamlessly be integrated into any middleware using an open standard.
- ◇ **Heterogeneity:** Particularly in the field of CWE it has to be assumed that sensors will be developed by different companies, using different programming languages and different tools - an aspect not yet discussed in the systems presented so far. In order to support such a heterogeneous environment, a language independent meta-model for sensors is needed which will define the capabilities of the sensor as well as the sensory specification. This approach provides two substantial benefits:
  - The interface is separated from the actual implementation, which can then be done using any technology.
  - The sensor code-base can automatically be generated from the model. This eases the development-process and also simplifies maintenance as the code-base can easily be updated in case the meta-model changes.

#### 4.3.2 Sensor Composition

In order to detect complex events as presented in the use case at the sensor-layer, it is necessary to compose sensors. The Sentient Object Model used in CORTEX is an approach which supports this. In CORTEX, the Sentient Objects are programmed in STEAM, a language adequate in the domain of mobile devices. However, in the

area of Web service based systems, communication has to rely on a more loosely coupled and standardised technology.

Also, the Sentient Object Model itself does neither describe the input and output values of a Sentient Object, nor does it specify how linkage of Sentient Objects takes place. An applicable model for sensors based on the Sentient Object model thus has to solve two challenges. The first one is to define a service-contract, i.e. a specification describing which values the sensor accepts as input and which ones it will produce as output. Using this information, meaningful composition of sensors becomes feasible. Secondly, a mechanism for sensor linkage must be provided, i.e. a means to connect the output of one sensor with the input of another. Depending on how these challenges are solved, additional properties can be inferred:

- ◇ *Exchangeability*. Is it possible to replace a sensor with another without affecting the functionality of the system or sensors dependent on it?
- ◇ *Flexibility*. Can the linkage of sensors be changed without altering the code?

## 4.4 A CONTEXT SENSOR DEVELOPMENT FRAMEWORK

### 4.4.1 The Scope

Before developing a solution, the scope of such has to be defined:

- ◇ **Sensor Development.** The systems discussed so far are all full-fledged context-aware middlewares. Yet a solution needs to be found that will solely deal with the aspect of sensor generation and management. It will not focus on aspects of data storage and data management, reasoning, etc. Thus, it will only realise the *sensor layer* and *raw data extraction layer* in the model of Ailisto et al., (2002) [1].
- ◇ **Observable Message Exchange.** We consider it to be feasible to observe the service interactions. Furthermore, we assume that the log files containing the interactions will not be not encrypted.
- ◇ **Service Invocation Sensors.** Sensors generated by the framework will solely focus on extracting context from service invocation logs. Neither hardware sensors nor other kinds of software sensors will be supported by the solution.

- ◇ **Event-Based.** The solution will be event based. Upon receipt, an invocation will be analysed and respective sensors will be triggered.
- ◇ **SOA-based.** The solution will work in a SOA-based environment, with Web services being used for intercommunication.
- ◇ **Performance.** The solution will be deployed on an average web server. There will be no resource constraint as there would be for applications deployed on mobile devices. The overall performance will be high, yet the goal is not to be capable of handling hundreds of requests per second.
- ◇ **Not realtime.** As the solution bases on analysis of invocation logs, it will not be realtime. At best, it will be near realtime.
- ◇ **Privacy.** Communication will be unencrypted and logs will contain all invocation traffic. Although privacy was mentioned as an important aspect of web-based context-aware systems in Truong and Dustdar (2008) [37], we will not focus on it here.

#### 4.4.2 The Solution

This chapter finally sees the presentation of an approach that solves the given challenges within the scope defined: The Context Sensor Development Framework (CSDF) is a sensor development framework for Web service sensors, which...

- ◇ ...bases on the data mining approach of inContext
- ◇ ...is model driven (using EMF <sup>7</sup>)
- ◇ ...supports powerful filter-techniques for service interactions
- ◇ ...supports automatic databinding for filtered XML-segments (using ADB <sup>8</sup>)
- ◇ ...deploys sensors as Web services
- ◇ ...integrates an elaborate sensor description
- ◇ ...supports powerful sensor-linkage and discovery mechanisms.
- ◇ ...eases integration of third-party Web services (for interaction with the context management system)

---

<sup>7</sup><http://www.eclipse.org/modeling/emf/> (last access: 2009-04-08)

<sup>8</sup>[http://ws.apache.org/axis2/0\\_93/adb/adb-howto.html](http://ws.apache.org/axis2/0_93/adb/adb-howto.html) (last access: 2009-04-24)

Using this framework, the development effort of context sensors in the web-domain is reduced dramatically. Code for sensing and deploying is automatically generated from the sensor model, thus the programmer can focus on the business logic of the sensor.

CSDF solves all the challenges presented earlier:

- ◇ **Sensor Integration:** CSDF bases on the data mining approach of inContext. With the use of a logging service, all interactions of services relayed through it will be copied and forwarded to CSDF. Sensors - previously registered with CSDF with their filter description - will be triggered if the filter matches.

- **Discovery:** CSDF incorporates a sensor registry which contains all sensors currently deployed. A registration record contains, among other information, the input and output specification of the sensor model. Thus it is not only possible to search for sensors by parameters but also based on compatibility to given specifications.

- **Framework Independence:** CSDF is only designed for sensor management, leaving the context management in the hands of the context management system. With the programmer being able to invoke just any kind of action in the logic of the sensor, it does not matter which context management system is used. In addition, required libraries can be easily integrated into the sensor.

Yet, since most of the recent context management systems provide some kind of Web service, CSDF has a special support for such. If listed in the sensor model, service-stubs and methods to invoke the service will be automatically created and added to the code. The effort to perform a context-update in the context management system via Web services is thus significantly reduced.

- **Heterogeneity:** As already mentioned, CSDF is model-based, using EMF. The sensor model describes all parts of the sensor on an abstract level. The actual implementation can therefore be done in any programming language, as long as it complies with the model and implements the required Web service interfaces.

In first version of CSDF, the sensor code-base (generated from the sensor model) is only available for Java. It is possible to re-generate the code-base upon model-changes without overwriting the business logic.

- ◇ **Sensor Composition:** Sensors generated by CSDF are based on the Sentient Object approach and can thus be composed. The service-contract is inherently given by the sensor model. The linkage of sensors is not hardcoded, but dynamically configured after deployment (Flexibility). Replacing a sensor with another one that provides the same or an even stricter service-contract is also possible (Exchangeability).

CSDF solves all given challenges. As will be demonstrated later, the given use case can also be realised using CSDF. It is therefore a successful approach to context-sense Web services that cannot directly be extended with sensor mechanisms.

---

## 5 CONCEPT OF CSDF

This chapter introduces the Context Sensor Development Framework (CSDF), our approach to solve the challenges of the previous chapter. Following a detailed explanation of the components of CSDF, the SensorModel - the formal specification of a Sensor - is presented. Part three and four then discuss the concepts of filtering and sensor composition and their implementation in CSDF. The final part is devoted to the development circle and the message flow between the components of CSDF.

## PREFACE

In this and later chapters, concepts of CSDF will be used throughout the text. The reader will be able to easily recognise a concept by the initial capital letter in its spelling, e.g. Controller, Session Service, Sensor. All concepts will be explained at some point, but not necessarily before their first usage. If required, the reader is advised to look up the respective concept using the index at the end of this thesis.

Some sections also contain references to documents included in the CSDF compilation. In that case, `$CSDF` refers to the root of the compilation.

## 5.1 THE CONTEXT SENSOR DEVELOPMENT FRAMEWORK

As described in the last chapter, the Context Sensor Development Framework (in short, CSDF) is a framework for the creation of software-sensors in a SOA-based system. Based on a data-mining approach, the generated Sensors can extract context from the message exchange between services. The diagram below visualises the concept of CSDF (Figure 3).

When a user calls a Web service, the invocation is relayed through the Service Interceptor. Apart from invoking the actual service, it sends a copy of the service request and response to the Controller. Upon receiving such a notification, the Controller compares it to the input requirements of the Sensors previously registered. If a matching Sensor is found, the interaction data is stored to a session and the Sensor is invoked. Sensors, created via the Generator, extract information, save and load data to and from the session and ultimately perform a context-update in the Context Management System.

Typically, Sensors are not stand-alone. Via Sensor composition, complex context-extracting networks can be created by combining rather simple Sensors.

### 5.1.1 Service Interceptor

The Service Interceptor is a component that intercepts outgoing service invocations: Web services are not invoked directly, but the invocation is relayed through the Interceptor. This design makes it feasible to observe the actual invocation request and response (from now on referred to as *service interaction*). As already mentioned in the last chapter, this might violate privacy ethics, yet is required to enable software sensing in third-party Web services.



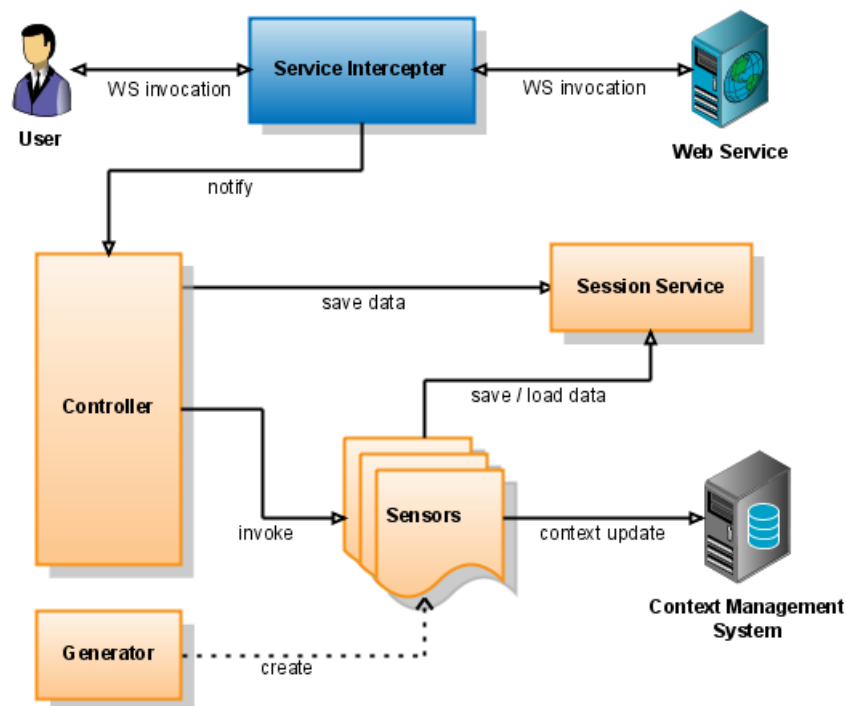


Figure 3: CSDF Concept

The Service Interceptor provides a Web service interface - the *Logging Service*. It contains operations to subscribe for and unsubscribe from service interactions. A client subscribing to the Logging Service needs to implement the *Logging Subscriber* Web service. The latter contains an operation to receive one-way notifications of service interactions.

If a Web service is invoked through the Logging Service, it will send two notifications to the registered logging subscribers - one containing the request and one containing the response. Apart from the actual SOAP document, the interaction also contains message and correlation ids, the message type, the timestamp, etc. A detailed specification of the service interface and the fields of a service invocation is given in inContext D5.3 [43].

The WSDL definitions for both the Logging Service and the Logging Subscriber can be found in \$CSDF/WSDL/Logging/.

**NOTE** The Service Interceptor is not a component of CSDF, but merely a tool used to realise context-sensing with CSDF. Thus CSDF does not contain an implementation of the Service Interceptor. In our design, we used the Service

Interceptor of the inContext framework. Yet, the developer is free to create his own implementation of the Service Interceptor, provided that it follows the specifications.

### 5.1.2 Controller

Being the heart of CSDF, the Controller is responsible for numerous tasks:

- ◇ Registration and interaction with the Service Interceptor
- ◇ Matching service request and response
- ◇ Providing a Sensor registry
- ◇ Management of Sensors
- ◇ Extracting additional context
- ◇ Filtering and invoking Sensors
- ◇ Session and composition management

#### Registration and Interaction with the Service Interceptor

Upon initialization, the Controller automatically registers at the Logging Service interface. The Controller itself implements the Logging Subscriber interface and thus receives all service interactions that are logged at the Service Interceptor.

#### Matching Service Request and Response

Request and response are sent as separate notifications and therefore need to be combined to a complete service interaction. For this reason, the Controller has to temporarily save the request until the matching response is received. Matching is done via the *message-correlation-id* given in one of the fields of the service interaction.

The operation of the Logging Subscriber to receive notifications is defined as one-way. Hence, there is a certain possibility that response-notifications might get lost on the way and thus may never be received by the Controller. If requests without responses pile up, resources of the Controller are wasted and its functionality might get affected eventually. To prevent such a scenario, the *Pending Message Timer* implemented in the Controller deletes pending requests after an adjustable time-period.

*Sensor Registry*

The Controller also acts as Sensor registry. Sensors developed and deployed by CSDF automatically register at the Controller upon initialization. During the registration process the Controller loads certain parts of the Sensor's specification and attaches them to the registration entry of the Sensor. Hence the Controller has comprehensive information about all Sensors registered with it.

Access to this information is provided via a Web service interface. It provides operations to list all Sensors registered and to get the specification of a particular Sensor. In addition, it is possible to query Sensors for compatibility. Given a particular specification, the Controller will list all Sensors compatible to it. This is, for instance, used in the ConfigAssistant as we will see in 8.7 Deployment.

A detailed overview of all Web services provided by the Controller can be found in 7.7 Controller.

*Sensor Management*

In CSDF, Sensors are developed by different teams and deployed on different servers. Older Sensors become obsolete as newer ones are developed. To guarantee stability in such a dynamic environment, the Controller has to keep track of all Sensors registered with it. Outdated registrations of Sensors have to be deleted both to save resources and to keep the registry clean (as other services might rely on it). Yet, as dealing with distributed systems, services might only become temporarily unavailable. The Controller therefore has to distinguish between temporarily (e.g., the internet connection is down) and permanent (e.g., the server hosting the service is removed) unavailability. An example is given in Figure 4.

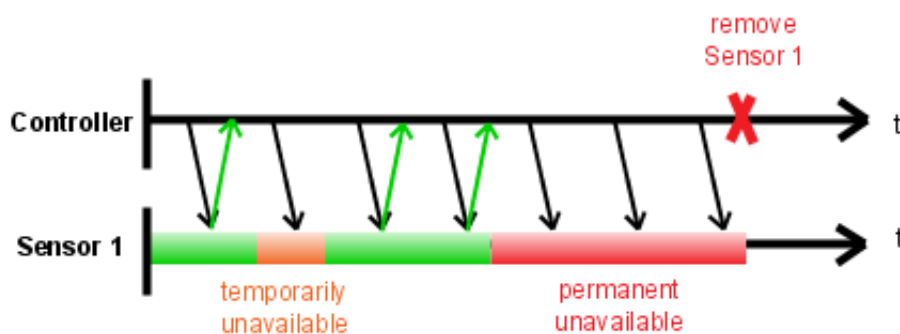


Figure 4: Is-Alive Concept

To realise this concept, the Controller queries its Sensors in certain time intervals. If a Sensor is not available at one point in time, the Controller will not instantly remove it, but rather increase its *service-unavailable counter*. If the service is available at one of the next requests, the counter will be reset. Yet, if the service remains unavailable and therefore the counter reaches a certain predefined value, the Controller will permanently remove it. To reuse the service in such a case, it must be re-registered. This mechanism of the Controller is called the *Is-Alive concept*.

### Extracting additional Context

After combining a request with a response to a complete service interaction, the Controller extracts additional context. In CWE, two context attributes play an important role:

- ◊ **User:** This attribute describes the actor who invoked the service. It is coded as an URI (email-address, URL of owl-node, etc.)
- ◊ **Activity:** This describes the activity in which an action took place. It is coded as an URI (activity-URL, URL of owl-node, etc.)

The Controller will extract those two values, if given, from the service interaction. For encoding, context tunnelling via SOAP-header fields as described in inContext D4.1 [40] is used. An example of encoding an activity and a user in a SOAP document shall be given below:

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <ns1:user_id xmlns:ns1="incontext"
      soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
      soapenv:mustUnderstand="0">
        http://www.vitalab.tuwien.ac.at/projects/incontext/owl/smallcontext.owl#User8
      </ns1:user_id>
    <ns1:activity_id xmlns:ns1="incontext"
      soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
      soapenv:mustUnderstand="0">
        http://www.in-context.eu/activity/Activity#1595
      </ns1:activity_id>
    </soapenv:Header>
    <soapenv:Body>
      ...
    </soapenv:Body>
  </soapenv:Envelope>
```

*Filtering and invoking Sensors*

In the next step, the Controller applies the Filters (see 5.3 Sensor Filtering) of the active Sensors on the service interaction. (The concept of active and passive Sensors will be explained later. Hence an active Sensor is just a special type of registered Sensor). The Filter of a Sensor is given via its sensor description, which is loaded from the Controller during registration. If the Filter matches, the corresponding Sensor will be invoked by the Controller.

Although it puts a considerable strain on the Controller to apply the Filters of all active Sensors onto the service interaction, this is necessary trade-off. The following shall see an explanation of why this approach is more efficient than to leave filtering to the Sensors.

1. *Approach: No pre-filtering done in the Controller:* Below find an analysis of how many service calls are required if the Controller does not use pre-filtering:

```
N ... amount of active Sensors

N x create session
N x save invocation data to session
N x invoke Sensor
N x load Data from Session
-----
4 N x service calls
```

2. *Approach: Pre-filtering done in the Controller:* The Controller will pre-filter Sensors by using the Filter from the Sensor description:

```
N ... amount of active Sensors
Z ... amount of pre-filtered Sensors (Z << N)

Z x create session
Z x save invocation data to session
Z x invoke Sensor
Z x load Data from Session
-----
4 Z x service calls
```

With Z being far smaller than N, pre-filtering drastically reduces the amount of service calls needed. The drawback of this design is that the Controller has to apply

N filters to each interaction received. The higher the number of active Sensors, the greater the strain on the Controller.

After pre-filtering, all successfully selected Sensors will be invoked by the Controller in their own session. The actual interaction sequence can be seen in 5.6.3 Service Interaction and Sensor Invocation.

### Session and Composition Management

An important task of the Controller is the management of invocation-sessions (not to be confused with sessions from the Session Service) and dealing with the invocation-chaining of Sensors. This part of the Controller will be explained in greater detail in 5.3 Sensor Filtering and 5.4 Sensor Composition.

#### 5.1.3 Session Service

In CSDF, data is organised in sessions. The component responsible for data and session management is the Session Service. It implements the following operations:

- ◇ Create a new session
- ◇ Delete an existing session
- ◇ Save variables in a session
- ◇ Load variables from a session
- ◇ Delete variables from a session

**NOTE** There is no operation for listing all variables stored within a session. Whenever a client wants to load data from a session, it must exactly specify the variables to load. This is a protection to prevent spying and data-tampering.

Sessions are accessed via a unique session-id. Also, sessions can only be deleted with the commit-key that is decided upon creation. Hence, only the creator of a session has the right to delete it. This prevents unauthorised deletion via third parties.

Sessions have a *time-lease*. After expiration, the session is automatically deleted by the Session Service. Apart from the lease, sessions also provide a *refresh-time*. On access (e.g., loading or saving data) the lease of the session is extended in case its value is already lower than the refresh-time. Thus only sessions with low or no access at all will be subject to deletion.

**EXAMPLE** Access to a session with a time-lease of 100 and refresh-time of 20:

Access	Time	Session valid until
1	0	100 (open)
2	30	100
3	70	100
4	90	110 (extended)
5	95	115 (extended)
6	110	130 (extended)
7	150	session already expired

Data in sessions is organised in variables. A variable contains the following fields:

- ◊ **Dataid.** The id is used to uniquely identify a variable within a session.
- ◊ **Data.** This is the untyped data of the variable. It may contain anything from atomic values to complete XML documents.
- ◊ **QoS-Attributes.** A variable might be attributed with a list of Quality-of-Service attributes (e.g. confidence, precision).

#### 5.1.4 Sensor

The actual task of context-sensing and processing of CSDF is done in the Sensors. Sensors are Web services that are created using the tools provided by CSDF. They extract information from service interactions, accumulate data and ultimately perform a context-update in the Context Management System.

A schematic black-box model of a Sensor is given in Figure 5. It illustrates the five interfaces of the Sensor:

- ◊ **Input.** This part serves as data input for the Sensor. Additionally, the Sensor specifies requirements on the input data that must be met since the functionality of the Sensor relies on it. If data satisfying the requirements is provided, the Sensor can be invoked.
- ◊ **Output.** After execution, resulting data is delivered at the output of the Sensor. The specification of the Sensor exactly determines what kind of output data the Sensor produces. This data is asserted by the Sensor and is therefore guaranteed.

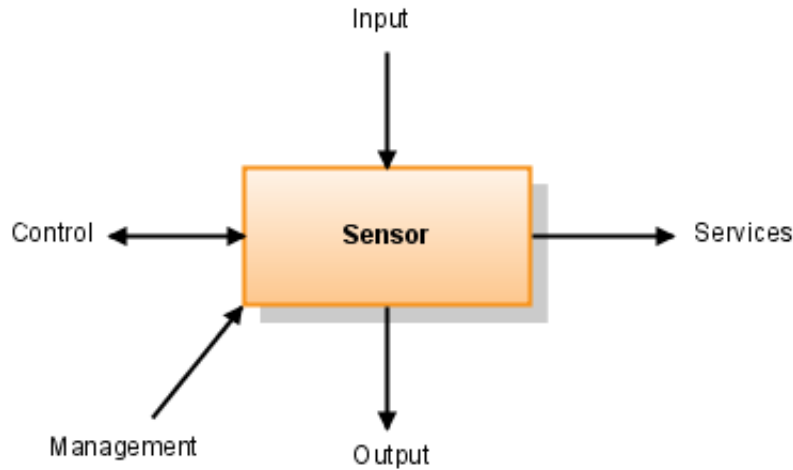


Figure 5: Sensor Interfaces

Both input and output of the Sensor use the same data format. Combined with the concept of linkage, this enables compositions similar to the Sentient Object Model [6]. For more information about this topic, see 5.4 Sensor Composition.

- ◊ **Control.** This port is used to gain global invocation-independent control over the Sensor. It provides access to information about the state of the Sensor (e.g. number of executions) and provides control over its behaviour (e.g. switching on/off a feature).
- ◊ **Service.** The Sensor might integrate other services during its execution. Such a service can either be another Sensor or a Web service.
- ◊ **Management.** This interface is used to initialize the Sensor and manage its activation-state.

This was only a simplified overview of the interfaces of a Sensor. A more comprehensive, schematic overview of the internal structure and external dependencies is shown in Figure 6.

#### Input / Output Ports

Ports are the input and output interfaces of Sensors. A Sensor can have several ports through which it can be invoked. Two different types of port are realised:



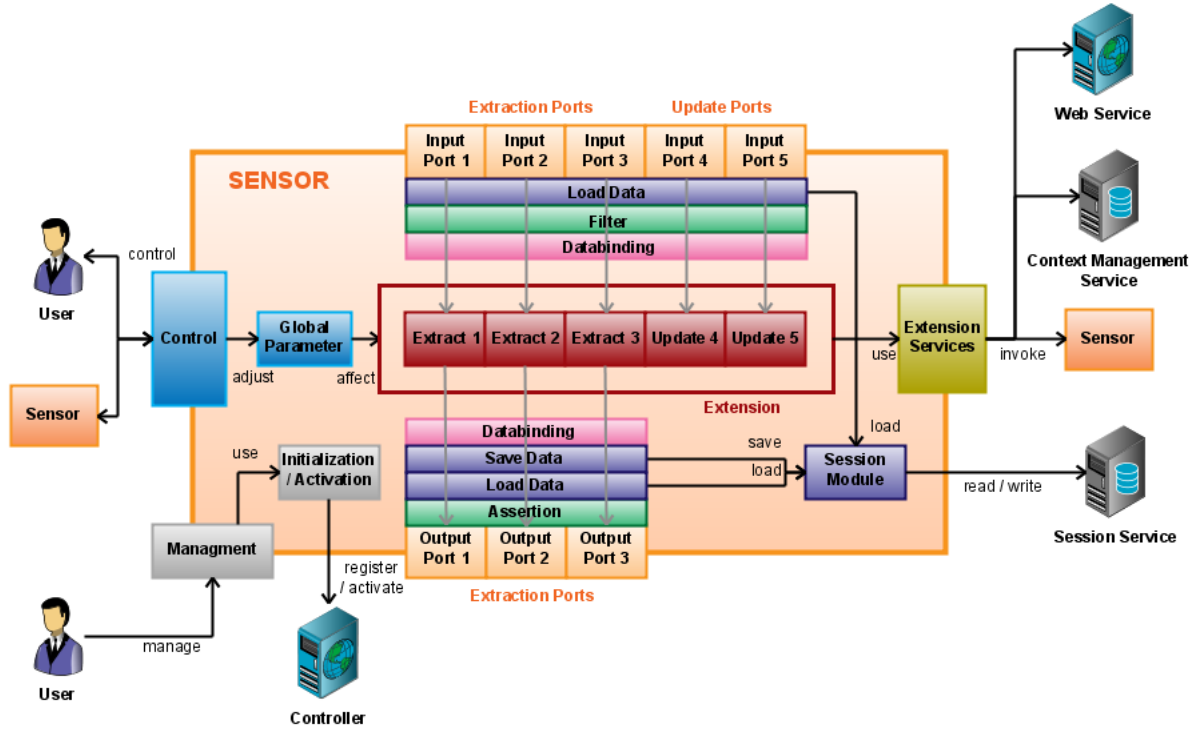


Figure 6: Schematic Overview of the Sensor

- ◇ **Extraction Port.** This port is used for context extraction, i.e. extracting, calculating or inferring additional information from the given input data. The extracted data then serves as the output of this port type. An extraction port provides both a specification for the input data that it requires (input port) and the output data it delivers (output port).
- ◇ **Update Port.** This port is designed for context-updates. In contrast to the extraction port, it only provides an input but no output port and therefore does not deliver any output data. With the data given through the input port, this port performs a context-update, i.e. altering data in a permanent data storage (e.g., reporting a context-change to the Context Management System).

For a detailed overview of the invocation sequence during invocation, please see 5.6.3 Service Interaction and Sensor Invocation.

*Session Module*

Data is not directly transmitted through the ports, as might be suggested by the previous diagram. Rather, all data is managed by the Session Service. Upon invocation, the Sensor loads the data specified in the input port from the Session. After execution, it saves back the result - specified by the output port - to the session.

Compared to the process of directly transmitting data upon invocation, this approach offers the following advantages:

- ◇ The Sensor mainly deals with variables either containing SOAP documents or parts of SOAP or XML documents. Typically, such kinds of documents are very large. In addition, many Sensors might be joined together in a composition in order to extract context.

Without sessions, every invocation must include all variables. Even if a Sensor does not need a particular variable, it still must be transmitted as subsequent Sensors in the invocation chain might require it. This causes extensive data-transfer as long Sensor-chains might contain many variables of considerably large content. Using the session approach, only truly required data needs to be loaded from the Session. So, for instance long XML documents are only transmitted to the Sensor when really needed.

- ◇ Secondly, delivering the minimum of data to the Sensor is a contribution to confidentiality. Without sessions, a Sensor always receives all data, which might also contain confidential information. Using sessions, however, a Sensor only loads data actually needed. It is not aware of the remaining data in the session.

The drawback of this design lies in the number of service-calls. If the data is exchanged via a session, extra service-calls to the Session Service are necessary:

Upon invocation:

```

1 x load input data
1 x save output data
1 x load output data (assertion)
-----
3 x service calls
```

Yet, considering that comparably only little data is transmitted in session service-calls, we believe that the advantages outweigh the drawbacks, for reasons of which the Session Service has been introduced.

### Filter / Assertion

Ports provide a service contract, i.e. certain input data is required and certain output data is guaranteed. The Sensor must be able to rely on the given input and other Sensors might rely on its output. Thus, checking both input (before processing) and output (after processing) is imperative. This is done via Filters (for the input) and Assertions (for the output). If either the input check or the output check fails, the execution is aborted. More information about Filters can be found in 5.3 Sensor Filtering.

### Databinding

Input and output data during Sensor invocation is communicated through the Session Service. As already mentioned before, the Session Service is untyped and can therefore only handle string-coded values. However, the specification of the input and output ports contains a type system. Thus, the developer coding the Sensor logic would have to work with string though the values are actually type checked. This frequently leads to error-prone code, with neither compiler and IDE support given.

To solve this matter, *databinding* is introduced: During the development process of the Sensor, type-classes are generated for all ports. Upon invocation, the Sensor then automatically casts the input to an object of the appropriate type and the resulting object back to string format. The advantages brought about by this approach are as follows:

- ◇ The programmer does not have to deal with string-coded values. Instead, real types can be used even for complex XML-documents, which dramatically reduces the complexity of data handling.
- ◇ The compiler / IDE supports the developer in the coding process. Invalid data assignments result in compiler errors easily noticed by the developer, thus rendering the program more error-proof.

More information about type systems can be found in 5.2.5 Resources and Type System.

### Extension

This part of the Sensor is the actual implementation of the ports' specific logic. Here the programmer codes the business logic of the Sensor: In extraction ports,

new data is generated from existing data. This can be done by combining, inferring or integrating data from other sources. In contrast, an update port performs a permanent change in a different system, i.e. a context-update.

### Global Parameter

Parameters are global attributes that hold information about the state of the Sensor and also affect the behaviour of the extensions. Parameters are independent of particular invocations and are mainly used in the extensions of the Sensor. They can also be accessed from outside using the Web services of the Sensor. Examples of Parameters are:

- ◇ Description of the functionality of the Sensor
- ◇ Number of successful / failed invocations
- ◇ Average execution time
- ◇ Precision of the Sensor
- ◇ Switching on/off a particular feature of the Sensor

Parameters might be *read-only* (e.g. description), *write-only* (e.g. switching on/off a feature) or *read-write* (e.g. precision of a calculation). Sensors already implement some default Parameters, but it is also possible to define Sensor-specific Parameters.

**EXAMPLE** An extension of Sensor X does a mathematical approximation. The global Parameter P controls the precision of the calculation. Decrementing P leads to fewer calculation-iterations, while a high value of P would result in more iterations.

### Integrated Services

Extensions might need to interact with external components, for example with other Sensors or Web services. The Sensor supports integration of such services to a certain extent by including classes required for invocation. In other words, it automatically generates service-client stubs. This, in turn, drastically reduces the programming effort for the developer when using external services.

### Initialization

This module deals with initialization and activation of the Sensor. During initialization, the Sensor loads the SensorModel - the specification describing all elements

of the Sensor - and afterwards registers at the Controller. In addition, Activation / Passivation is also done using this component. Detailed information about the SensorModel can be found in 5.2 Sensor Model, and for information about Activation and Passivation, please see 5.4.5 Active and Passive Sensors.

### 5.1.5 Generator

The Generator is a tool to create new Sensors. It is used during the development phase and generates the Sensor code-base for a given SensorModel. This is done in several steps:

1. Verify the SensorModel
2. Extend the SensorModel
3. Create types for the Sensor
4. Create stubs for integrated services
5. Create Sensor-specific code
6. Backup of implemented code
7. Finalise code-base

More detailed information about the SensorModel can be found in 5.2 Sensor Model.

**NOTE** The current implementation of the Generator does only support the creation of Sensor code-bases for the Java language. Future versions of CSDF will also support generation of Sensors in other programming languages.

#### Verify the SensorModel

In a first step, the SensorModel is verified. This is necessitated because there are some constraints that cannot be checked via the EMF meta-model. In the verification-process, the following things are checked:

- ◇ Check whether the used ids are valid (e.g. portid, standardid)
- ◇ Check whether ids are unique (e.g. port-id)
- ◇ Check if references are valid (e.g. reference to a Standard)

- ◇ Check if certain fields are in a valid format (e.g. type of variables)
- ◇ Check whether there is a circular dependency in the variable definitions

#### Extend the SensorModel

Subsequently, the SensorModel is enhanced and additional information is included in the code-base:

- ◇ Download referenced online resources (e.g. WSDL documents, XML Schemas)
- ◇ Extract XML Schema from WSDL resources
- ◇ Resolve reference-instructions in variable definitions
- ◇ Include default Parameter in SensorModel (e.g. author, name, description)
- ◇ Include default resources in SensorModel (e.g. XML Schema definition)
- ◇ Generate XML Schema file for the Sensor

#### Create Types for the Sensor

The XML Schema file generated in the last step contains a type definition of all input and output ports and variable sets of the Sensor. This document is now used to generate the type classes for a particular programming language. In Java, this is done via the *SchemaCompiler* provided by the Axis2 framework<sup>9</sup>.

#### Create Stubs for integrated Services

In order to conveniently use external services during the coding phase, the Generator creates client-stubs for services specified in the SensorModel. In Java, this is realised via *WSDL2Java*, a program provided by the Axis2 framework.

NOTE In some cases the compiler may be unable to generate code from a given WSDL. Even if generation is possible, the generated code might contain minor errors. Rather than being a bug of CSDF, this is a flaw of WSDL2Java, which does not yet fully support the complete WSDL2 standard.

---

<sup>9</sup><http://ws.apache.org/axis2/> (last access: 2009-04-08)

### Create Sensor-specific Code

In the next step, Sensor-specific code is generated from the SensorModel. This is done using JET<sup>10</sup>, a code generator of the M2T (Model To Text) project of Eclipse. The following documents are created:

- ◇ Extension-classes of the ports
- ◇ Extension controller and supportive methods
- ◇ Axis2 service-description and WSDL files of the Sensor

### Backup of implemented Code

In case that the Sensor was already generated before, the Generator automatically creates a backup of the port-extension classes. This is very useful as the developers can just re-generate the Sensor code-base without worrying about their code being overwritten. Any additional files that were not part of the Sensor code-base will be untouched by the Generator. Yet changes made in the code-base other than in the port-extensions will be overwritten. Therefore, if changes were made in the Sensor core, the developer is advised to perform a manual backup before re-generation.

### Finalise code-base

In a last step, the Generator combines the files generated in the previous steps with the code of Sensor implementation. The result is the code-base of the Sensor implementing the given SensorModel. What remains to be done is to implement the actual business-logic of the ports.

## 5.2 SENSOR MODEL

Sensors are described by the SensorModel. The SensorModel is the formal specification of a Sensor, defining ports, integrated services, resources, etc. in a language independent model. The SensorModel thus reflects the capabilities and interfaces of the Sensor on an abstract level. As CSDF is model-driven, the developer first specifies the SensorModel and then generates the Sensor code-base out of it. We will

---

<sup>10</sup><http://www.eclipse.org/modeling/m2t/?project=jet#jet> (last access: 2009-04-08)

look at the development stages in CSDF at a later section; first, we shall discuss the SensorModel in detail.

CSDF is based on EMF (Eclipse Model Framework) <sup>11</sup>. The SensorModel is a model-instance compliant to an Ecore meta-model. It is developed using the graphical model editor of EMF and serialised using XMI (XML Metadata Interchange) <sup>12</sup>.

The SensorModel comprises the following four parts:

- ◇ Input/Output Specification
- ◇ Control Specification
- ◇ Service Specification
- ◇ Sensor Specification

### 5.2.1 Input/Output Specification

The specification defines the input and output ports of the Sensor. A port comprises a list of variable definitions. In case of an input port, this expresses the data requirements of the port. In case of an output port, it shows the data that is produced by the Sensor. This information will also be used as Filter during Sensor invocation. (For more information, please see 5.3 Sensor Filtering.)

The specification contains:

- ◇ Specification of Filter attributes of the Sensor (see 5.3.4 Filter Techniques)
- ◇ Definition of extraction and update ports
- ◇ Definition of variables in the input and output parts of a port
- ◇ Definition of additional restrictions on the value domain of variables
- ◇ Definition of sets of variables
- ◇ Definition of variable-references - in variable sets, input or output ports

---

<sup>11</sup><http://www.eclipse.org/modeling/emf/> (last access: 2009-04-08)

<sup>12</sup><http://www.omg.org/technology/documents/formal/xmi.htm> (last access: 2009-04-08)



### 5.2.2 Control Specification

This part deals with the definition of Parameters and their access rights. It contains:

- ◇ Specification of a key used for Activation
- ◇ Definition of Standards, i.e. lists that contain Parameters
- ◇ Definition of Parameters
- ◇ Definition of access-rights for Standards

Standards are sets of Parameters that belong together. A default Standard supported by all Sensors is `standard.status`. It specifies the following read-only attributes:

- ◇ Name and description of Sensor
- ◇ Author and time when Sensor was published
- ◇ Full service address
- ◇ Number of successful as well as failed invocations
- ◇ Last error message
- ◇ Average and latest processing time

Apart from the predefined Standard, it is also possible to define your own Standard with Sensor-specific Parameters.

### 5.2.3 Service Specification

Integration of external services is done in this part of the SensorModel. It contains:

- ◇ Specification of the Controller Web service
- ◇ Definition of external Web services to integrate
- ◇ Definition of external Sensors to integrate

External services are used by the Sensor during execution. The Sensors logic might rely on these services, therefore it is *dependent* on them.

### 5.2.4 Sensor Specification

In here, general properties of the Sensors as well as used resources are specified. By including an external resource, its type system is made available for usage in the SensorModel. This specification contains:

- ◇ Specification of author, name, description and service URL of the Sensor
- ◇ Definition of external resources for integration

### 5.2.5 Resources and Type System

As already shown in the previous sections, the SensorModel defines the following data containers:

- ◇ Variables in Input / Output Specification
- ◇ Parameters in the Config Specification

In order to realise such data containers, a typing concept is needed. There are three possible approaches to deal with typing in CSDF:

#### No Type System:

The first approach is to provide no type system at all. All values of the respective data containers are treated as strings. This is a very simple approach and raises the following problems:

- ◇ As there are no types, it is not possible to constraint data containers to a certain domain (e.g. integer, string, particular XML element). As a result, any content is considered as 'valid'.
- ◇ Databinding, as described in 5.1.4 Sensor, is not realisable with this approach.

#### Self-implemented Type System:

The next approach is to implement a fixed set of basic types in CSDF. This solves the problems of the first approach, yet has some drawbacks of its own:

- ◇ It has to be decided how many and what kind of types are needed. This proves to be complicated as it is difficult to a priori estimate what kind of types are necessary to efficiently and conveniently work with CSDF.

- ◊ Is the type system flexible enough to be extended with new types? If types are directly implemented in the SensorModel, new types cannot be added. If the types are implemented on code level, they might vary in implementation in different programming languages.
- ◊ The programming and testing effort for the type system is considerably high.

*Usage of existing Type System:*

A more sophisticated approach is to integrate an already standardised and widely-used type system. The advantages of this approach are that comprehensible and already tested implementations of the type system are available, and that the type system most likely is flexible enough to serve the needs of CSDF.

On the basis of this analysis, CSDF has implemented the third approach. As a type system, *XML Schema* is used. The reason for this is that it is standardised and provides sophisticated support for XML content (which is the main form of data CSDF deals with). In addition, it is easily extendible and several implementations for different programming languages are already available.

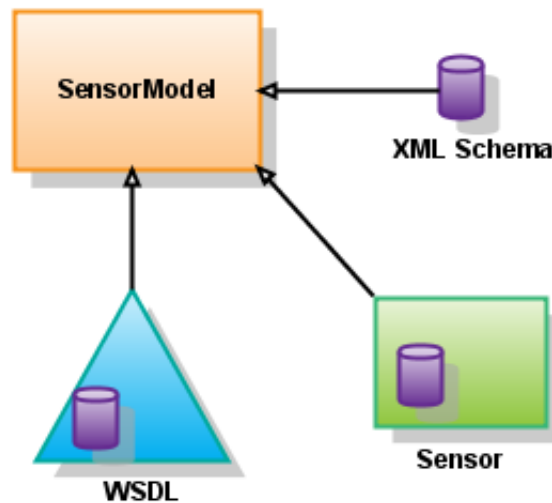


Figure 7: Integration of Resources

Apart from the basic data-types that are an integral part of XML Schema, CSDF also supports the integration of external XML Schema types. As shown in Figure 7, CSDF can integrate the following resources into the SensorModel:

**XML Schema:**

To add additional types to the type system of CSDF, XML Schema documents can be included into the SensorModel. The referenced XML Schema can either be a file or specified via an URL.

**WSDL:**

Upon the integration of a WSDL-file, CSDF parses the document and extracts the schema-part out of it. The result is then automatically saved to a separate file and included as standard XML Schema document. Hence, it is possible to use types for SOAP messages exchanged in Web service invocations.

**Sensor:**

At last, it is also possible to include type systems from other Sensors. This is done by specifying the service address of the Sensor and the namespace of the schema to load. Every deployed Sensor provides the operation `GetResource-ByNamespace`, which can be used to directly load a particular resource from the Sensor. It is therefore possible to either include the main type system of the specified Sensor or one of its resources.

Every resource loaded into the SensorModel is marked with a prefix. The default prefix for the standard XML Schema type is `xsd`. Prefixes uniquely refer to their corresponding type system. The `type`-field for both variables and Parameters has the format `<prefix> ':' <typename>`. The `typename` must refer to a valid type in the type system referred to by `prefix`.

**NOTE** The current implementation of CSDF does only support XSD simple types for Parameters. Complex types cannot yet be processed by CSDF and therefore will be treated as strings without a proper type check. Future versions of CSDF will mitigate this shortcoming by supporting both simple and complex types for Parameters, as is now the case with variables.

## 5.3 SENSOR FILTERING

Sensors extract context from service interactions. But not every service interaction is suitable for every Sensor. A Sensor therefore has to define rules that determine what kind of service interactions it is interested in. This process is referred to as *filtering*. Strictly speaking, filtering does not only apply to service interactions, but generally

to any kind of input of the Sensor. This is important as not all Sensors process service interactions; passive Sensors in particular might deal with rather arbitrary output of other Sensors when used in compositions. Compositions will be discussed in greater detail in the next section; in the following we will focus on the aspect of filtering.

### 5.3.1 What is a Filter?

The Filter of a Sensor specifies the elements (or in terms of CSDF, the variables) a Sensor is interested in. So basically the Filter contains a list of variable definitions. Unlike the variables used in sessions, the variables of filters do only consist of a unique identifier and a type. There is yet no implementation of the QoS-attributes concept for Filter variables. An example of a simple Filter is given in Table 1.

Name	Type
person.name	xsd:string
person.sex	mt:TSex
person.age	xsd:integer
person.items	mt:TItemList

Table 1: Example of a simple Filter

The Filter presented in Table 1 is sensitive to data conglomerations (in short, datasets) that include those four variables. In other words, the Sensor will only react to datasets containing, at a minimum, the variables listed. A Filter thus specifies the *input requirements* of a Sensor. Only if the Filter is satisfied, the Sensor might be invoked. If a dataset fails to meet the Filter requirements, the Sensor cannot be invoked.

Table 2 gives examples of datasets and shows their compatibility to the Filter defined in Table 1. The first dataset does exactly specify all the variables of the Filter and is therefore compatible - it matches the Filter. Dataset 2 specifies an additional attribute, `person.size`, apart from the requested variables, and thus also matches the Filter. In contrast, dataset 3 does not contain the required variable `person.age` and consequently does not match the Filter.

### 5.3.2 Definition of Filter

Filters are indirectly defined in the Input/Output Specification of the SensorModel; indirectly, because they are actually inferred from the port specification. A port

Variable	Dataset 1	Dataset 2	Dataset 3
person.name	Peter	Lisa	Mark
person.sex	male	female	male
person.age	20	24	
person.items	{glasses, book, suitcase}	{magazine}	{dog, cat}
person.size		171cm	182cm
person.weight			70kg
<b>matches?</b>	yes	yes	no

Table 2: Datasets and Filter Matching

usually contains an input and an output port, both specifying a list of variables. The variables of the input port in turn form the Filter for the Sensor. As a Sensor might have more than one port, it can also have more than one Filter. While the variables within a Filter are conjunctive, the Filters of a Sensor are disjunctive. In other words, the Sensor reacts if at least one of its Filters matches a given dataset.

### 5.3.3 Filters and Session Management

The Controller invokes Sensors upon receipt of a service interaction. Via the pre-filtering process (see 5.1.2 Controller), it is possible for the Controller to determine which Sensors are interested in a given interaction. As already mentioned before, the data used during invocations is stored in sessions. In case of at least two simultaneous invocations, the Controller has to decide how to manage the sessions for the Sensor invocation:

**Separate Session for Invocations:** In this approach, a new session is created for every invocation of a Sensor. The advantage of this design is that invocations do not interfere with each other. On the other hand this is also a serious drawback, as there is no way for a Sensor to extract context from two or more sequential service interactions.

**One Session for all Invocations:** Here all invocations of a Sensor are executed in the same session. A Sensor can therefore extract context coded in a series of service interactions. Yet, this approach brings about several serious problems: First, invocations might overlap and thus overwrite each other's result as they access the same session. Second, older and outdated data is not deleted and might interfere with new data. And finally, sessions become very large and use up the resources of the Session Service as they are not deleted.

As we can see, both approaches have some serious drawbacks. That is why we have to introduce some additional parameter for a finer control of the session management.

### 5.3.4 Filter Techniques

To solve the problem of session management, the SensorModel provides three additional parameters:

- ◇ **User-Awareness** - specifies whether the Sensor is reactive to users
- ◇ **Activity-Awareness** - specifies whether the Sensor is reactive to activities
- ◇ **Session-Frame** - specifies the length of the session-time frame

#### User-Awareness

CWE typically involves many actors. Actions performed by one user have a higher correlation than actions of different users, which suggests a distinction between users. With this flag it is possible to make the Sensor aware of different users. In other words, if user-aware, the Sensor will group interactions by users. It is therefore possible to analyse several sequential actions of one user.

The Controller will interpret this attribute as follows: If set to user-aware-mode, the Controller will open one session per user. Anonymous interactions will not result in an invocation. If the flag is not set, the Controller will open only one session for all interactions.

**EXAMPLE** A user-aware Sensor:

Call	User	Session
1	James	1 (open)
2	James	1
3	Mark	2 (open)
4	James	1
5	(anonymous)	-

**EXAMPLE** A Sensor that is not user-aware:

Call	User	Session
1	James	1 (open)
2	James	1
3	Mark	1

*Activity-Awareness*

Similar to the user concept, CWE typically implement some form of activity-concept. Activities comprise actions that are directed to the same goal. We can therefore assume that actions within an activity are more correlated to each other than actions of different activities. Thus it makes sense to implement a mechanism to distinguish between different activities.

Similar to user-awareness, the Controller will open one session per activity if the Sensor is set to activity-aware-mode. In this mode, the Controller will not invoke the Sensor for interactions outside an activity context. If the activity-mode is turned off, the Controller will as usual create only one session for all interactions.

**EXAMPLE** An activity-aware Sensor:

Call	Activity	Session
1	Project1	1 (open)
2	Project1	1
3	Management	2 (open)
4	Project1	1
5	(anonymous)	-

**EXAMPLE** A Sensor that is not activity-aware:

Call	Activity	Session
1	Project1	1 (open)
2	Project1	1
3	Management	1

*Session-Frame*

In CWE, activities contain many actions that are directed to realise a common goal. While activities are usually too coarse for context-extraction, actions are too fine. The intention of a user typically manifests itself in a series of actions that all are directed to the same objective. It is reasonable to assume that such tasks are limited in time. In order to create a layer of discrimination finer than activity and coarser than actions, a so-called session-frame has been introduced.

The session-frame specifies a time frame for grouping service interactions. The first interaction opens a new session. All subsequent interactions falling into the time-frame are executed in the same session. The first interaction that occurs outside the time-frame will open a new session with its own session-frame. A session-frame of 0 will lead to a new session for every interaction. If the Sensor has a session-frame



and is furthermore aware of users and/or activities, the Controller will only group interactions that satisfy the awareness-parameters. In other words, only interactions with the same user and/or the same activity will be grouped in a session frame. Different users and/or activities will lead to the creation of a new session with its own session-frame.

**EXAMPLE** A Sensor with a session-frame of 100, user-aware but not activity-aware:

Call	Time	User	Activity	Session
1	0	James	Project1	1 (open)
2	30	James	Management	1
3	50	Lisa	Project1	2 (open)
4	80	James	Project1	1
	100			1 (close)
5	120	James	Project1	3 (open)
6	120	Lisa	Management	2
	150			2 (close)
7	200	Lisa	Management	4 (open)

NOTE Session 1 spans from 0-100, Session 2 from 50-150,...

**EXAMPLE** A Sensor with a session-frame of 0, both user-aware and activity-aware:

Call	Time	User	Activity	Session
1	0	James	Project1	1 (open, close)
2	10	James	Project1	2 (open, close)

With the help of these attributes, the developer has close control over the management and grouping of service interactions. The approach thus successfully solves the problems presented in the previous section. The only point that still remains to be solved is the problem of interleaving interactions. We will deal with this problem in section 5.4.6 Composition at Runtime.

## 5.4 SENSOR COMPOSITION

### 5.4.1 Composition in General

Sensors are composable. Two Sensors are composed if the output of one Sensor serves as the input of another Sensor. A simple example can be seen in Figure 8. Here the

output of Sensor 1 is delivered to Sensor 2 and Sensor 3. Invocation of Sensor 1 thus also leads to the subsequent invocation of Sensor 2 and 3.

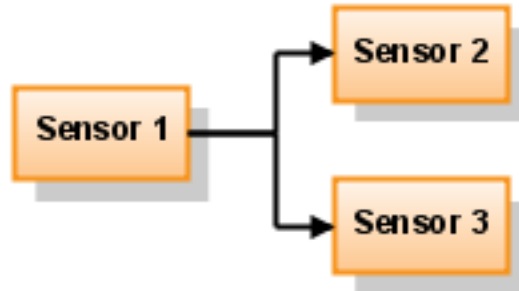


Figure 8: Sensor Composition

Composing Sensors has several advantages. First of all, complex extraction tasks can be split into smaller chunks, each done by a separate Sensor. An example: Sensor A extracts the header of emails, and Sensor B extracts the attachments of emails. We can then build a Sensor C which adds communication-actions to the context-system by using the sender and recipient of A and the attachment of B. Although A, B and C are each very simple Sensors, complex functionality can be created via composition.

This already leads to the second advantage of compositions: Sensors can be reused. Sensors like A and B can be used by various other Sensors as well. For instance, a Sensor D could use the attachment of B, upload it to a document management service and finally add it as a resource of an activity in the Context Management System.

As a Sensor cannot only have one port, functionality can even be split within the Sensor - each port focussing on a special task. As Figure 9 shows, a Sensor can also be linked to itself.



Figure 9: Sensor Composition

When making use of compositions, the actual Sensor logic can often be kept very simple with the Sensor focusing on one task only. This, in turn, results in fewer

bugs. In addition, it is easily possible to test Sensor-functionality with the test tools of CSDF, as we will see in section 5.5 Development Circle.

#### 5.4.2 Compatibility of Sensors

In the context of composition, an important point is the question of compatibility. In order for two Sensors to be successfully linked, the corresponding ports need to be compatible. In the following, we are going to analyse how compatibility can be defined in the setting of CSDF.

As described in the section 5.3 Sensor Filtering, a Sensor can only be invoked if the input data matches the Filter of the port. For a Sensor A to be compatible with a target Sensor S, A has to provide at least all the variables specified in the input requirements of S. If A fails to fully cover the requirements of S, it is not compatible with S. In this sense, we can define two levels of compatibility:

- ◊ **Direct Compatibility.** The output port fully covers the input requirements.
- ◊ **Inferred Compatibility.** The accumulated output of a Sensor chain covers the input requirements.

##### *Direct Compatibility*

This type of compatibility is given if the output specification of the first Sensor contains at least all variables listed in the input specification of the second Sensor. As a result, the first Sensor directly covers all the requirements of the second Sensor.

**EXAMPLE** An example is given in Table 3: Sensor A does specify all variables required by Sensor S and is therefore compatible. Sensor B lacks the required attribute `mail.body` and is therefore not compatible with S.

##### *Inferred Compatibility*

Even if no direct compatibility is given, two Sensors might still be compatible with one other: We assume that a Sensor C, rather than being directly invoked by the Controller, is composed with B, which is, in turn, composed with A. If this is the case, C will be invoked as part of the invocation chain A - B - C. Each invocation of a Sensor adds information to the session, so when C has finally finished the execution, the data of A, B and C is stored in the session. So even if the target Sensor S is not directly compatible with C, it might be compatible to the accumulated data of A, B and C. Inferred Compatibility is therefore given if the required data is not directly

Variables	A	B	S
mail.subject	X	X	X
mail.sender	X	X	
mail.recipient	X		X
mail.body	X		X
mail.attachment		X	
compatible?	yes	no	

Table 3: Examples of Direct Compatibility

generated by the previous Sensor, but as the accumulated result of the invocation chain.

**EXAMPLE** In Table 4 Sensor C is inferred compatible with Sensor S in the invocation chain ABC, as it accumulates all the data requested from Sensor S. On the other hand, Sensor E is not inferred compatible with S because the invocation chain ADE does not produce the required field `leader.name`.

Variables	A	B	C	ABC	A	D	E	ADE	S
person.id	X			X	X			X	
person.name	X			X	X			X	X
person.location	X			X	X			X	X
project.name		X		X		X		X	X
project.person.role						X		X	
project.budged		X		X		X		X	
project.leader		X		X					
leader.name			X	X					X
person.hobbies							X	X	
compatible?	no	no	no	yes	no	no	no	no	

Table 4: Examples of Inferred Compatibility

**NOTE** In the current implementation of CSDF, only the detection of direct compatibility is supported. In theory, the architecture of CSDF allows the detection of inferred compatibility, yet no such algorithm has been implemented so far. This will be the subject of future extensions of CSDF.

### 5.4.3 Loop Detection

One common problem when dealing with composition are loops. A loop is a series of connections forming a circle. In a loop, execution does not come to an end; unless detected, the system is therefore likely to fail or crash.

**EXAMPLE** Some examples of loops can be seen in Figure 10:

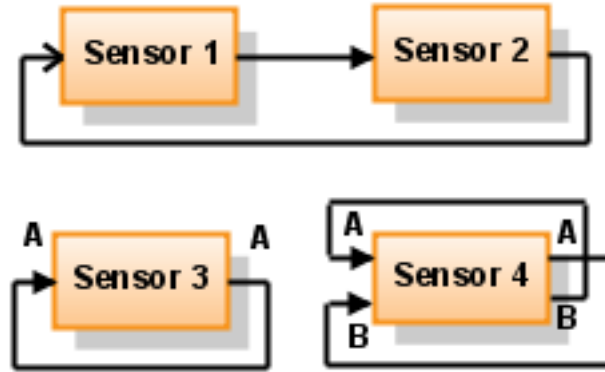


Figure 10: Examples of loops

- ◇ Sensor 3 in the bottom left-hand corner has a link from its output port A to its input port A. If input port A is invoked from some external source, the output of A would be delivered to the input of A again, ... and again, ... and again, ... This is the most basic form of a loop.
- ◇ A more complex example of a loop can be seen at Sensor 2. Here port A links to port B and port B links to port A. An invocation would lead to the execution of A - B - A - B - A, etc. This example shows that, although Sensors can refer to themselves, output ports have to be connected to input ports in such way that no loop occurs.
- ◇ Naturally, loops can also occur in a sequence of Sensors. As seen in the example of Sensor 1 and 2, the output of Sensor 2 is delivered back to Sensor 1. An execution would lead to a never-ending sequence of executions of 1 - 2 - 1 - 2 - 1, etc.

In CSDF, composition is performed by the Controller. Therefore, loop detection is quite simple and efficient, as the centralised Controller can detect loops in advance. The loop-detection algorithm is implemented as follows:

1. In the beginning, the invocation-stack is empty. For a port of a Sensor marked for invocation: Continue at 2.
2. Check if the current port is already listed in the invocation-stack. If yes, loop is detected. If no, continue at 3.
3. Add the port to the invocation-stack.
4. Check for all links connected to the port. For all input ports of Sensors that connect to the specified port: Execute 2-5. Afterwards continue at 5.
5. Delete the port from the execution-stack.

This algorithm can simply be implemented in a recursive function. Upon detection of a loop, the Controller does the following:

#### **Skip element**

The invocation of the port which leads to the loop is aborted. Thus all elements of the loop are executed exactly once. Other non-looping segments of the invocation-chain are executed without interference.

#### **Print warning**

The Controller prints a warning message showing the elements which form the loop. This can easily be reconstructed using the execution-stack.

#### **Delete link**

In a final step, the Controller deletes the link between the top-element of the invocation-stack and the current port. Removal of this link destroys the loop.

In CSDF, links are defined directly at the Sensor. Yet, during the registration process of the Sensor, the linkage-information is downloaded from the Controller, by means of which centralised control of linkage as described in the algorithm becomes feasible.

#### **5.4.4 Types of Links**

In CSDF, there are two different types of links:

- ◇ **Forward-To.** It is defined on an output-port and forwards data to an input-port.
- ◇ **Forward-From.** It is defined on an input-port and inquires data from an output-port.

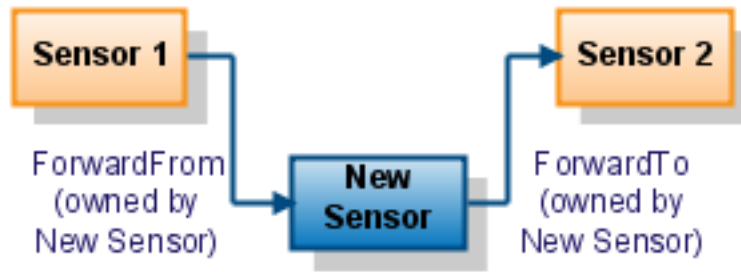


Figure 11: Forward Example 1

The differences between these two link-types will be discussed in a short example:

In Figure 11, a new Sensor is integrated by adding two links to existing Sensors. In this example, the new Sensor uses the output of Sensor 1 as input and delivers its output to Sensor 3. Both links are defined on the new Sensor: A Forward-From is defined on the input port of the new Sensor, requesting data to be forwarded to the port, whenever the specified port of Sensor 1 is invoked. On the output-side of the new Sensor, a Forward-To is defined, automatically forwarding the invocation to Sensor 3 whenever the new Sensor is invoked.

Both links are defined on the new Sensor upon initialization, so there is no need to perform changes in the configuration of Sensor 1 or 2. This dramatically reduces integration effort, as new Sensors can be added to the system seamlessly without affecting the functionality of the overall system.

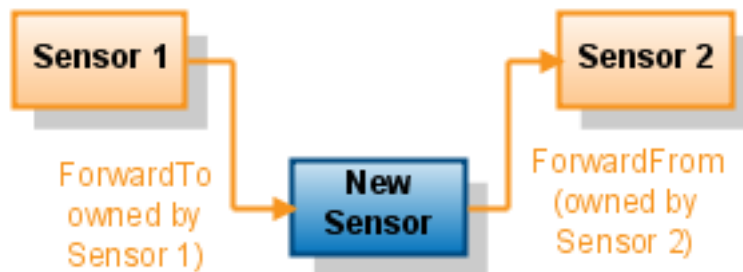


Figure 12: Forward Example 2

In contrast, the links in Figure 12 are defined on Sensor 1 and 2: Sensor 1 defines a Forward-To link to automatically invoke the new Sensor whenever the specified port of Sensor 1 is invoked. Sensor 2 defines a Forward-From on its input port and requests

forwarding of the invocation whenever the specified port of the new Sensor is invoked.

Both examples ultimately lead to the same result. Yet there is a fundamental difference between the link-types, as the owner of the link and the method of integration are different.

#### 5.4.5 Active and Passive Sensors

Using CSDF, the number of Sensors is likely to become considerably high. To be able to use Sensors, they have to be registered at the Controller, which also serves as Sensor registry. Upon receipt of a service interaction, the Controller will execute the pre-filter routine and invoke all matching Sensors. (The concept of pre-filtering has been already discussed in 5.1.2 Controller). Yet, this approach raises the following problems:

- ◇ Firstly, the Controller has to apply a Filter on every Sensor. Even though filtering is a fast process, it is considerably costly performing it many times. For instance, if there are 500 Sensors registered at the Sensor and there are 20 interactions per second, the Controller must execute the filter routine 2000 times per second.
- ◇ In many cases, it is not desired that a Sensor is directly invoked as the result of a service interaction. For instance, a Sensor that converts GSP coordinates to a region name does not need to be invoked by the Controller, but rather is designed to be integrated by other Sensors.

To solve these matters, CSDF provides two types of Sensors:

- ◇ **Active Sensors.** These Sensors can be directly invoked by the Controller. Whenever a service interaction is received, the Controller will check whether active Sensors matches it and hence execute them.
- ◇ **Passive Sensors.** Passive Sensors are not directly invoked by the Controller. They are excluded from the pre-filtering process.

Both active and passive Sensors share the same SensorModel and the same code-base. They can both be used in compositions as well. They are practically identical, the only difference being their activation-status at the Controller. Therefore, it is possible to make passive Sensors active and active Sensors passive. These processes



are respectively called *Activation* and *Passivation*. For details about the WSDL operations, please refer to 7.6.2 Activate and 7.6.4 Passivate.

By distinguishing between active and passive state, the amount of Sensors involved in the pre-filtering process is considerably reduced. This is very important as the Controller is also the bottleneck of CSDF.

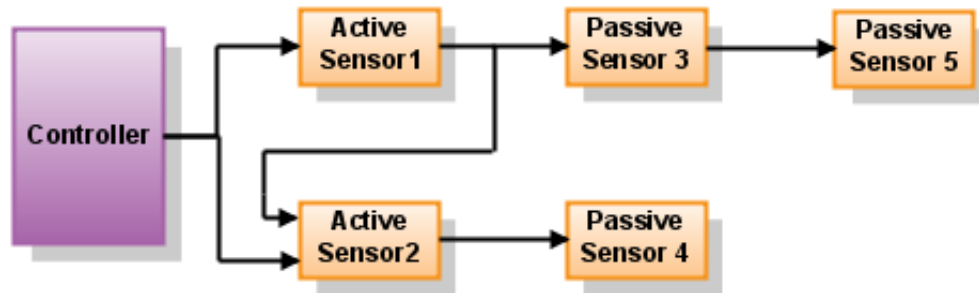


Figure 13: Example of active and passive Sensors

Figure 13 shows an example of a Sensor configuration. There are two active Sensors 1 and 2, which will be directly invoked by the Controller. Sensor 1 forwards invocations to Sensor 2 and 3. Sensor 2, in turn, invokes Sensor 4. In addition, the invocation of Sensor 3 will also lead to the invocation of 5. As we can see, both active and passive Sensors can be freely used in compositions.

#### 5.4.6 Composition at Runtime

As already mentioned, links are directly defined at Sensors. This is done during the initialization phase. (For more details about initialization, please see 7.6.1 Initialize). Yet, more interesting is the actual handling of invocations in composed Sensors during runtime.

There are two important questions that have to be answered:

- I. How to deal with simultaneous invocations of a Sensor?
- II. How to deal with invocations in case that two or more Forwards are defined on the output of a Sensor?

##### Referring to I)

As can be seen in Figure 13, the Controller might receive two service interactions - both matching active Sensor 1 - almost simultaneously. Given that the active Sensor

1 is configured in a way to accept two requests in one session, the Controller must now decide how to execute Sensor 1 twice.

The problem is as follows: The Sensor does not directly receive the interaction data from the Controller, but rather loads it from the session. After execution, it saves back the result to the session. If the Controller decides to invoke it instantly every time a service interaction is received, the following problems arise:

- ◊ First, the Controller has to save the data of the service interaction to the session. As both interactions use the same variables, the second one would immediately overwrite the data of the first. As a result, both executions of Sensor 1 would load the data of the second interaction.
- ◊ Second, the output data of the first execution of the active Sensor 1 will be instantly overwritten by the output data of the second execution.
- ◊ Both the first and the second point were made under the ideal assumption that one service-call takes longer than two and that the execution time of a Sensor is always the same. Yet this is not always the case. Race conditions make the outcome even more unpredictable.

To solve this problem, service interactions that appear within one session must be queued. Only when the first execution is finished, the second one can be started.

#### Referring to II)

As shown in Figure 13, Sensor 1 is linked to both Sensor 2 and 3. It has to be decided how to invoke both Sensors after execution of 1.

The problems are similar to the previous ones. We have two invocations of two different Sensors, yet only one session. If executed simultaneously, the following problems arise:

- ◊ Both Sensors deal with data of the same session. It cannot be predicted which Sensor is faster in loading, processing and saving the data. Results of one Sensor might be unnoticed and directly overwritten by the other Sensor.
- ◊ Especially when considering that one of the Sensors is linked to a third one, this becomes a serious problem. In the example given above, for instance, Sensor 3 is linked to 5, which in return relies on the results of 3. If now

Sensor 2 overwrites the output of 3 before the invocation of 5 takes place, the invocation of 5 might fail due to violated input requirements.

To solve this problem, invocation stacks must be used. The invocation of the second Sensor must be delayed until the invocation of the first and all subsequent invocations are finished.

### The Solution

The solution can now be programmed as follows:

```
UPON RECEIVAL OF SERVICE INVOCATION $si:
  // pre-filtering
  foreach $s in $active_sensors {
    if($s->filter($si) == true) {
      // if there is an open session, use it:
      $sess = find_session($s, $si)
      // otherwise: create session
      if($sess == null)
        $sess = create_session()

      // queue invocation
      $sess->exec_queue->push($si)
    }
  }

INVOCATION WORKER THREAD OF SESSION $sess:
do{
  // if new element in invocation queue
  if($sess->exec_queue->size () != 0) {
    $s = sess->exec_queue->shift();

    // start invocation procedure
    invokeSensor(new stack(), $s);
  }
}while(true)

ROUTINES:
invokeSensor($stack, $s) {
  // implements loop-detection
  if($stack->contains($s) {
    // detection of a loop
    handle_loop($stack, $s)
  }
  else {
    $stack->push($s)
    // execute port of sensor
    execute($s)
    foreach $ls in $s->getForwards() {
      invokeSensor($stack, $ls)
    }
    $stack->pop()
  }
}
```

This approach solves both of the problems presented above. An example can be seen in Figure 14:

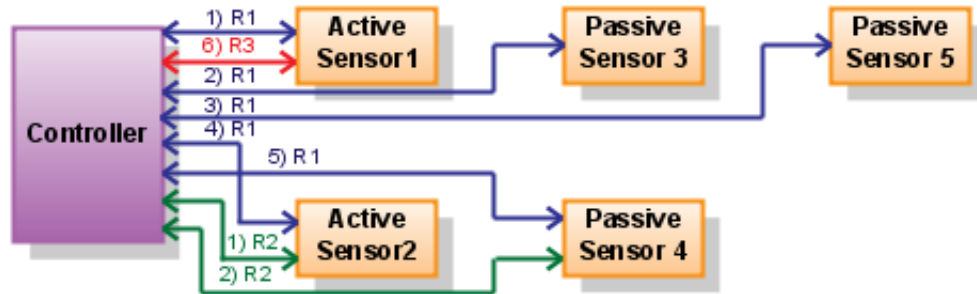


Figure 14: Invocation of active and passive Sensors

Figure 14 shows the actual invocation sequence of the example presented in Figure 13. There are three nearly simultaneous requests R1 (for Sensor 1), R2 (for Sensor 2) and R3 (again for Sensor 1). As the graphic shows, R2 is unrelated to the session of R1 and R3, and can therefore be executed in its own thread. R1 and R3 - according to the configuration of Sensor 1 - must be executed in the same session. Therefore, 6) R3 is queued until 1)-5) R1 and all subsequent invocations are finished. In addition, the invocation of 4) Sensor 2 is delayed until 2) Sensor 3 and linked 3) Sensor 5 is finished.

## 5.5 DEVELOPMENT CIRCLE

In this part we will introduce the actual development cycle of CSDF Sensors. An overview is given in Figure 15. A comprehensive guide on how to develop and deploy an actual Sensor is then given in 8 How To.

### 5.5.1 Create the SensorModel

The first step in the development of CSDF Sensors is the creation of the Sensor-Model. The SensorModel is a model instance of an Ecore meta-model. As it is serialised using XMI, it can be coded in any common text editor. A more convenient way to code the SensorModel is to use the graphical EMF model editor included in the CSDF distribution (Figure 16).

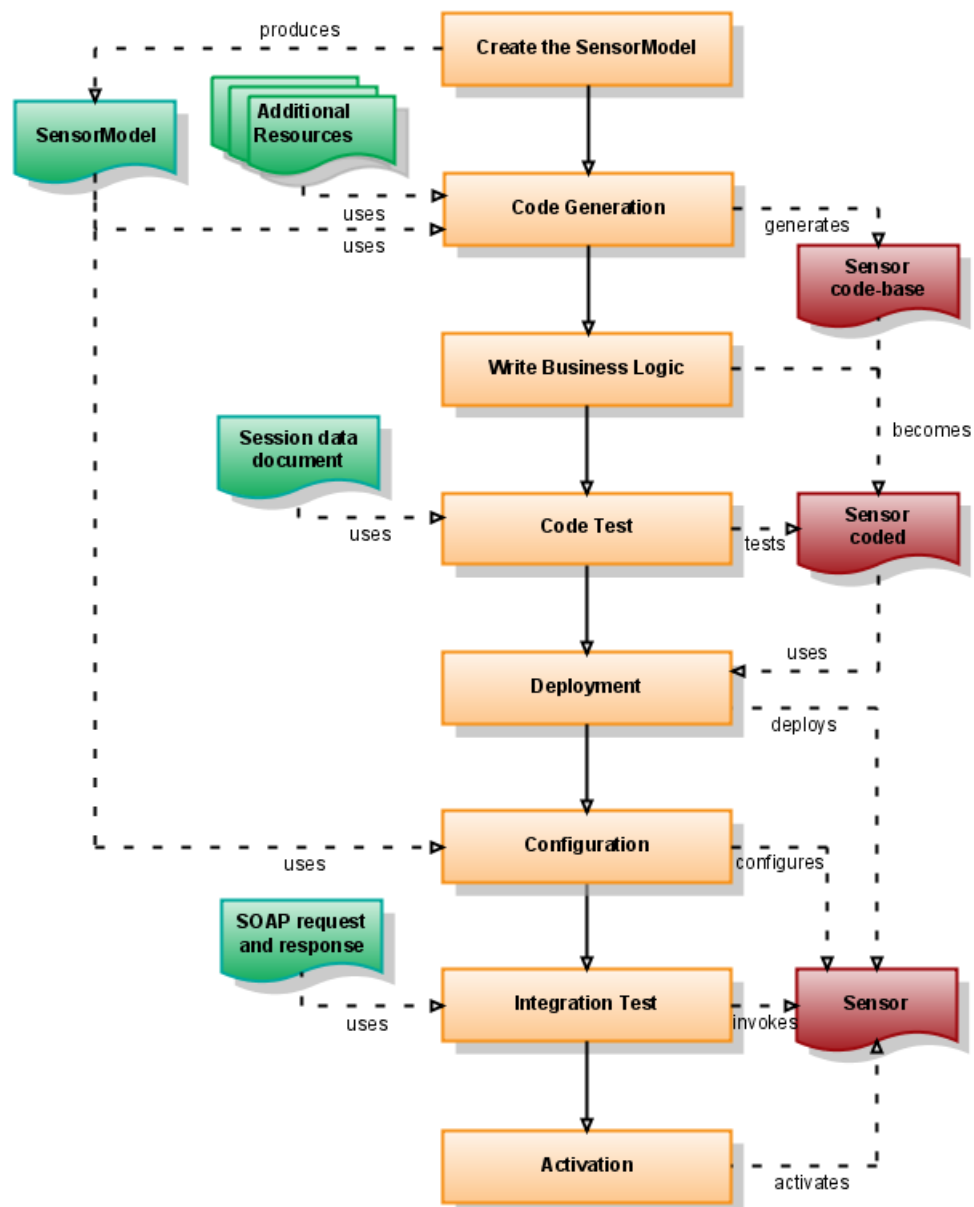


Figure 15: CSDF Development Circle

A detailed overview of what the SensorModel is and what parts of the Sensor it defines is given in 5.2 Sensor Model.

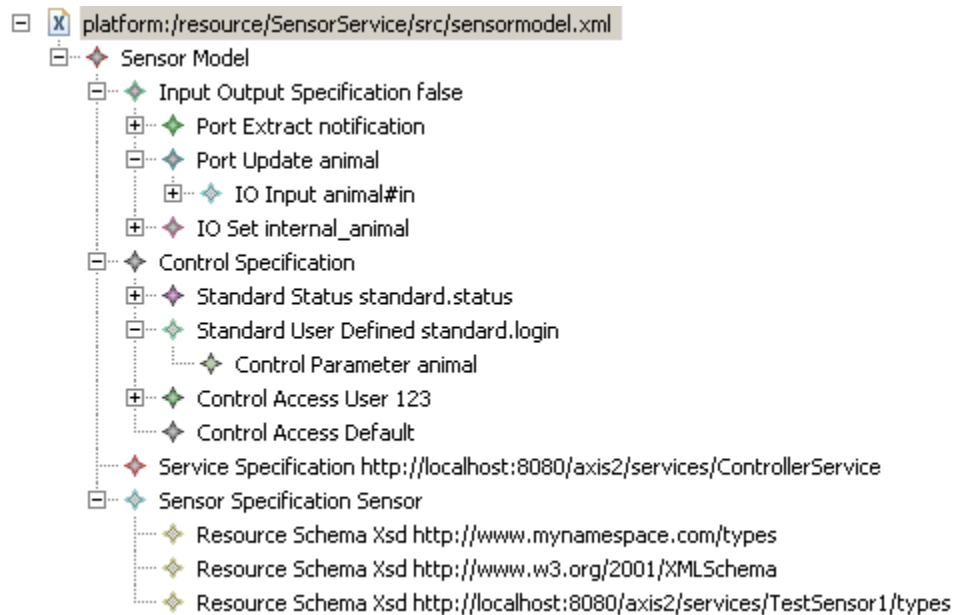


Figure 16: A SensorModel created via the EMF Model Editor

### 5.5.2 Code Generation

The next step is to create the Sensor code-base from the SensorModel. This is done using the Generator. (For detailed information, please see 5.1.5 Generator). The code generator creates the following:

- ◇ Sensor code-base (for Java)
- ◇ Enhanced SensorModel
- ◇ Java type classes for ports and variable-sets
- ◇ Java client-stubs for integrated Web services
- ◇ WSDL files for Sensor and deployment descriptor
- ◇ XML Schemas for included resources and XML Schema for Sensor
- ◇ Batch files for building, packing and deploying Sensor

### 5.5.3 Write Business Logic

After generation of the code-base, the developer has to implement the business logic of the Sensor. In particular, he or she has to code the extensions containing the logic of the respective ports. This can be done using any IDE (e.g. Eclipse).

### 5.5.4 Code Test

Once the ports are fully implemented, the code needs to be tested. A command-line testing tool is already integrated in CSDF, so this task can easily be accomplished.

Before a port can be successfully invoked, the appropriate input data has to be set in the session. This is done with the help of a so-called *session-data document*, an XML document which contains a list of variables and their content, thus reflecting the data that is set in a session.

The XML Schema for session-data documents looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://sensor.csdf.in_context.eu/tester"
  xmlns:tns="http://sensor.csdf.in_context.eu/tester"
  elementFormDefault="qualified">

  <xsd:element name="sessiondata">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="tns:data" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="data">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="xsd:anyType">
          <xsd:attribute name="dataid" type="xsd:string" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

An example session-data might look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:sessiondata xmlns:tns="http://sensor.csdf.in_context.eu/tester"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://sensor.csdf.in_context.eu/tester sessiondata.xsd">
```

```
<!-- specify all the variables that should be set in the session -->
<tns:data dataid="example.text">this is an example text</tns:data>
<tns:data dataid="person.name">mark</tns:data>
<tns:data dataid="person.age">25</tns:data>
<tns:data dataid="person.sex">male</tns:data>
</tns:sessiondata>
```

The variables defined in this file are then saved to a temporary session and the port is executed. The test answers the following questions:

- ◇ Are the input requirements specified correctly and is the port invocable with the given input?
- ◇ Are the output requirements correctly specified and does the extension generate valid output?
- ◇ Is there a bug in the extension code?

### 5.5.5 Deployment

After code verification, the Sensor is ready for deployment. The Sensor is deployed as Axis2 Web service. Therefore it needs to be packed into an .aar archive. The code-base already includes batch files to automatically build and pack the Sensor. Afterwards the finished .aar archive is uploaded to the Axis2 application of the web server, which automatically installs it. The Sensor is then fully deployed. (For more detailed information about the content of an .aar archive, please refer to [http://ws.apache.org/axis2/1\\_4\\_1/quickstartguide.html#services](http://ws.apache.org/axis2/1_4_1/quickstartguide.html#services)).

### 5.5.6 Configuration

Before the Sensor can be used in CSDF, it needs to be initialized and registered at the Controller. There are two ways to initialize a deployed Sensor:

#### *Empty Initialization using a Command-Tool*

The Sensor-code base already integrates a batch file to initialize the deployed Sensor. However, this tool can only be used for empty initialization. To configure Forwards and to reset Web service addresses, the second approach has to be chosen.

#### *Initialization via the ConfigAssistant*

The ConfigAssistant (Figure 17) is a graphical tool to initialize and configure a deployed Sensor. It provides following features:



- ◇ Overview of the input and output ports of the Sensor
- ◇ Configuration of Forwards
- ◇ Resetting of service-addresses of integrated services
- ◇ Saving and loading configuration to and from a file
- ◇ Initializing the Sensor with a given configuration or loading the current configuration from the deployed Sensor

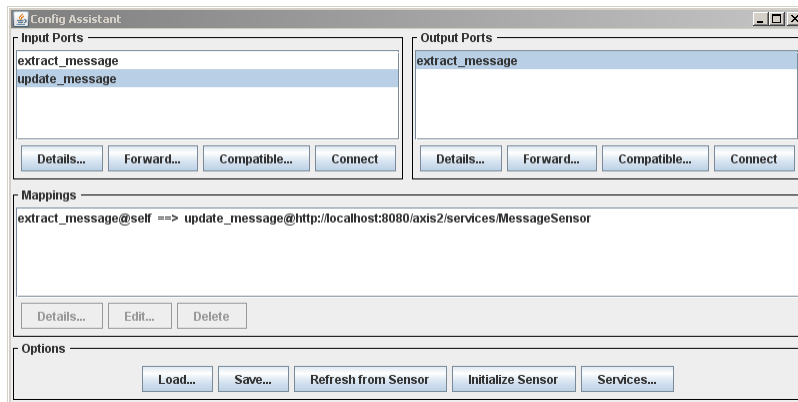


Figure 17: Initialization using the ConfigAssistant

### 5.5.7 Integration Test

Verifying the functionality of the system and the correct integration of the Sensor can be done via integration tests. For this purpose, CSDF provides a testing tool in the Controller. It emulates the Controller and tests the system by feeding a SOAP request and response to it. The Controller combines it to a service interaction and executes Sensors matching it. This test will reveal the following:

- ◇ Does the Filter of a particular Sensor match a given service interaction?
- ◇ Are the Forwards configured properly?
- ◇ Are the linked Sensors really compatible with one another?
- ◇ Is there a loop in the Sensor composition?
- ◇ Does the system correctly extract data and consequently perform the desired context-update?

### 5.5.8 Activation

Once the integration test is passed successfully, the development and integration of the Sensor is completed. As a last step, the Sensor might be activated to directly get invoked from the Controller upon incoming service interactions. This can conveniently be done via an included batch-file, thus successfully completing the development cycle.

## 5.6 SERVICE INTERACTION

To get a better understanding of the communication between the components of CSDF, this section gives an overview of the most important interaction sequences.

### 5.6.1 Initialization of Controller

The Controller is the main part of CSDF. Once it is initialized, the context-extraction with CSDF is running. The start-up sequence of the Controller involves both the Logging Service and the Session Service. A detailed overview is given in Figure 18.

Once the user initiates the initialization-process, the Controller first verifies the functionality of the Session Service. This is necessary because the Session Service is a crucial part of the CSDF. If the Session is either not reachable or does not work properly, CSDF will fail to operate. The Controller tests the Session Service by creating a temporary session and deleting it afterwards. If this operation is successful, the check of the Session Service is finished.

As a next step, the Controller registers at the Logging Service. By registering the Controller will be notified about future service interactions of the Service Interceptor. As a registration parameter the Controller passes the Web service URL of its own implementation of the Logging Subscriber. As a result, the Controller receives the subscription id that is needed in case of unregistration. Hereby, the initialization of the Controller is finished and CSDF is ready for Sensor registration.

The second part of Figure 18 shows the shutdown-sequence of the Controller. Upon shutdown-request, the Controller unregisters at the Logging Service using its subscription id. In addition, it sends unregister-notifications to all registered Sensors to inform them about their enforced removal. A Sensor receiving such a notification switches back to the state before initialization. Finally the Controller will clean up its internal registry and revert back to uninitialized-mode.

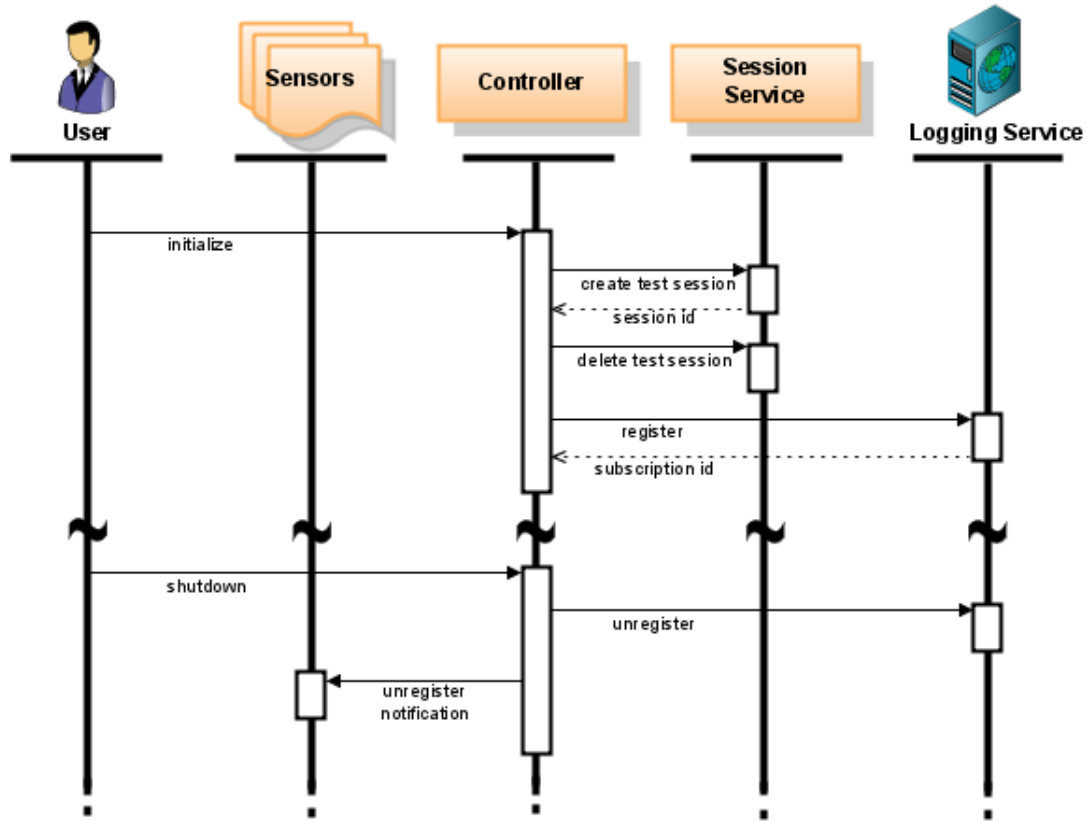


Figure 18: Controller Initialization Sequence

### 5.6.2 Registration of a Sensor

Once CSDF is running, Sensors can be deployed and registered. The registration at the Controller is automatically done in the initialization phase of a Sensor, as shown in Figure 19.

As a first step in the initialization procedure, the Sensor loads the SensorModel, which resides in the web archive of the deployed Sensor. Next, it indexes commonly used parts of the SensorModel for faster access (e.g. ports and variables) and verifies the validity of the Parameter default values.

After this internal initialization, the Sensor attempts to register at the Controller. The Controller then in turn requests parts of the Sensor specification as follows:

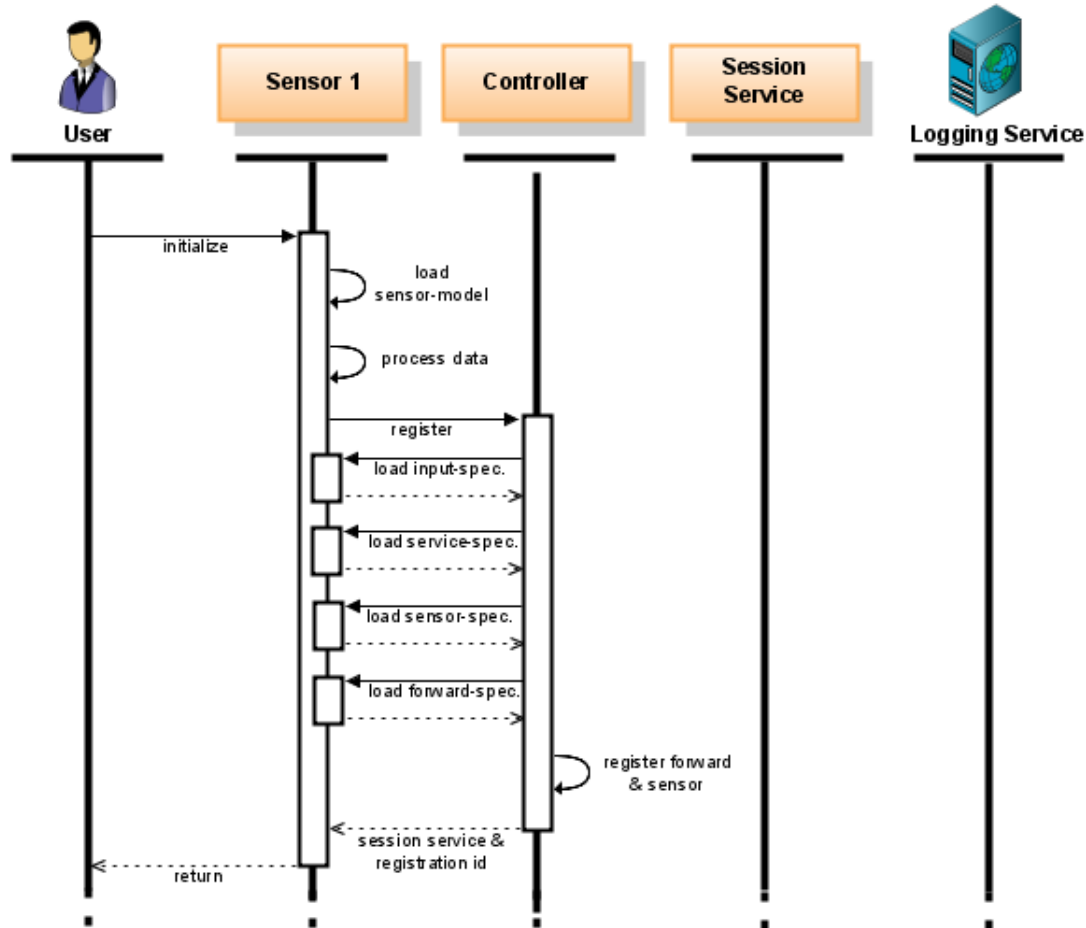


Figure 19: Sensor Initialization Sequence

### Input/Output Specification

This contains the definition of all ports and variable sets specified in the SensorModel. With this information the Controller exactly knows the input requirements and output assertions of the Sensor. This is needed in order to pre-filter (see 5.1.2 Controller) and determine the compatibility of ports.

### Service Specification

This specification contains a list of all services that are to be integrated into the Sensor. In other words, it is an overview of all external services the Sensor uses during execution. This information is useful when querying for Sensors relying on a particular service.

**Sensor Specification**

It contains a general description of the Sensor including name, description, author, service-URL, etc. This information is for example returned when querying the Controller for a list of all registered Sensors.

**Forward Specification**

To realise service composition, the Controller needs to load the linkage information from the Sensor. This contains a list of all Forwards the Sensor defines.

In the next step, the Controller completes the registration of the Sensor and registers the loaded Forwards. Therefore, the Sensor is fully integrated into CSDF and its Sensor compositions. Finally, the Controller returns the location of the Session Service and the registration id of the Sensor. The registration id is needed in case of unregistration from the Controller. As a last step the Sensor can be activated (see 5.4.5 Active and Passive Sensors).

**5.6.3 Service Interaction and Sensor Invocation**

In the following, we are going to analyse the complete workflow of CSDF starting with the invocation of a service by the user. Once the Controller is notified of the service interaction via the Service Interceptor, it will invoke the appropriate Sensors. Those in turn will extract data from the interaction and ultimately perform a context update. The whole communication process is shown in Figure 20 and Figure 21.

As illustrated in Figure 20, Web services are not invoked directly. Rather, the invocation is relayed through the Service Interceptor. The Service Interceptor in turn sends a copy of the request to the Controller and afterwards invokes the real Web service. Once the execution is finished and the result is returned, it sends the copy of the response to the Controller and delivers the result back to the user. This whole operation is transparent for the user, i.e. he or she is not aware of the fact that the Web service is actually not invoked directly.

As requests and response are sent separately, the Controller needs to temporarily save the request until it receives the corresponding response. Upon its receipt, both documents are combined to a complete service interaction (see 5.1.2 Controller). In the next step, the Controller performs the pre-filtering routine to determine all Sensors that match the given interaction and must therefore be invoked (see 5.1.2 Controller).

In the following, we assume that the Filter of a port of Sensor 1 matched the service interaction (Figure 21). In this case the Controller first needs to create a new session via the Session Service and then stores the interaction data to it. Finally the

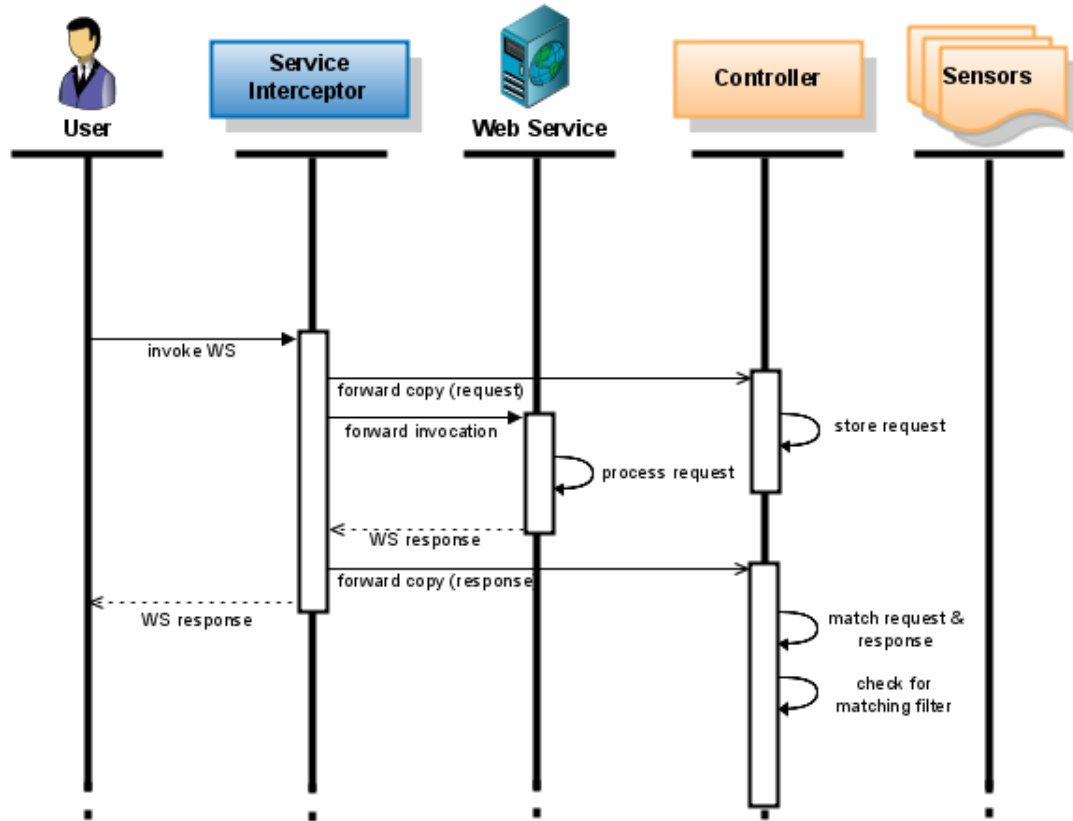


Figure 20: Service Interaction and Sensor Invocation (Part 1/2)

Controller invokes the port of Sensor 1. (In this step, the session id is passed in the invocation request).

Upon invocation, the Sensor first needs to check the input requirements. To do so, it loads all variables specified in the input part of the port from the session. If one of the required fields has not previously been set in the session, the operation will fail and the Sensor will abort the execution. The same applies if the subsequent check of the loaded data fails.

If the input is valid, the execution of the Sensor can be initiated. Prior and posterior to the execution of the extension logic, databinding, i.e. converting string values of variables to appropriate programming language objects and back, is performed. Depending on the port type, the execution differs:

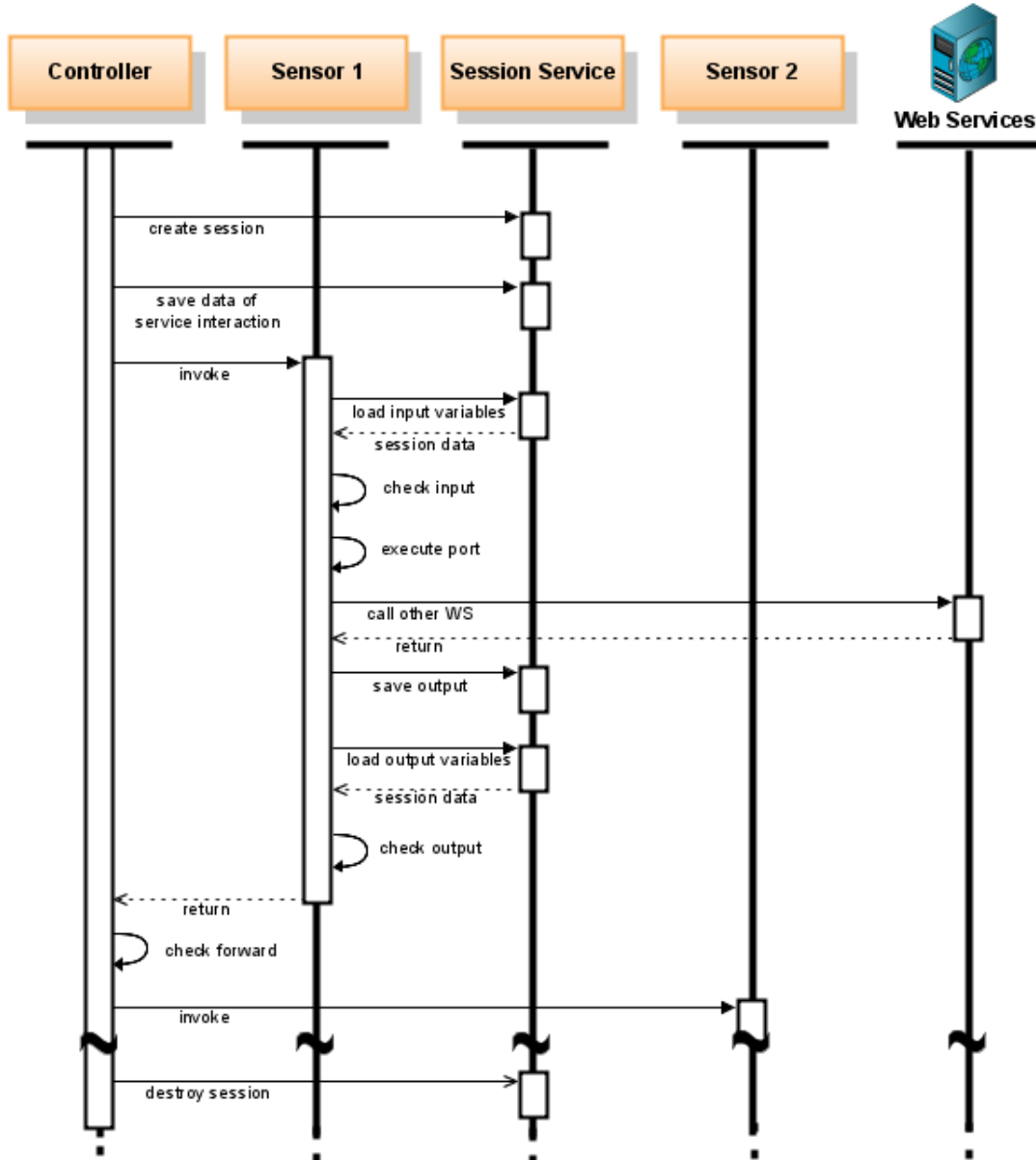


Figure 21: Service Interaction and Sensor Invocation (Part 2/2)

**Extraction ports**

It extracts context-information from the given data. Depending on the Sensor, it might also integrate external Web services to perform this task.

**Update ports**

It is designed to perform a context-update in the Context Management System. This is usually done by invoking an external Web service to report the context-change.

In case of update ports, the invocation of the Sensor is completed with the context-update. Extraction ports, on the other hand, have to assert certain output data defined in the port's specification. In this case, the result of the execution is first saved to the session. In the next step the Sensor loads all variables of the output port. This is necessary as in some cases the output of the extension does not match the variables of the output port. Finally, the loaded data is checked against the port's specification. If the check fails, the execution is aborted and an error is returned. If it is successful, the invocation of the Sensor is complete.

Once the control is passed back, the Controller checks for Forwards involving the executed Sensor. All Sensors found are stored to an execution queue and invoked sequentially. (For more information about execution in compositions, please see 5.4.6 Composition at Runtime.)

After the invocation of all marked Sensors is finished, the Controller proceeds as follows:

**In case of single interactions**

The invocation of the Sensor is complete. Subsequent interactions will be executed in their own session. Thus the Controller can immediately delete the session from the Session Service.

**In case of session-framed interactions**

Although the invocation of the Sensor is complete, subsequent interactions might lead to invocations in the same session. Therefore the Controller cannot delete the Session from the Session Service until the session-frame is closed and all invocations are finished.

More detailed information about filtering and session-framed interactions can be found in 5.3.4 Filter Techniques.



---

## 6 SENSOR MODEL

The following chapter is devoted to the Sensor-Model, a formal specification of all features of the Sensor. With the first step in the creation of a CSDF Sensor being the design of the Sensor-Model, a comprehensive understanding of all its parts is crucial. Thus, this chapter provides an analysis of all classes and fields used in the SensorModel, highlighting their purpose. Selected examples are then used to illustrate the usage of these features.

## PREFACE

This part contains a detailed analysis of the SensorModel. Understanding all elements of the SensorModel is indispensable for developers who create Sensors using CSDF. However, it is not imperative to read this part in order to understand later chapters of this thesis (except for the chapter on CSDF Web Services). Therefore, readers who are more interested in a guide on creating a sample Sensor might skip this and the next chapter and continue reading at 8 How To.

## 6.1 INTRODUCTION

### 6.1.1 Ecore

To understand the following part, fundamental knowledge of EMF and the Ecore model-language is necessary. It would go beyond the scope of this thesis to introduce this complex framework in detail, therefore the reader is advised to read the introduction provided at <http://www.eclipse.org/modeling/emf/docs/> to get a deeper understanding of EMF.

The most important concepts of Ecore are:

- ◇ **Package.** This is the root of every Ecore model. It contains all the classes.
- ◇ **Class.** A class defines a model-element and can contain a list of features. Ecore-classes also support multiple heritage.
- ◇ **Feature.** A feature can either be a field or a reference to another class. Both fields and references support multiplicity. Ecore distinguishes between child-containment (the field contains the actual instance of the class) and reference-containment (only a reference to the instance of the class is saved).

### 6.1.2 SensorModel ID

Both the SensorModel and the services defined by CSDF make use of various ids. For ease of understanding we are going to explain their usage and their scope beforehand:

- ◇ **portid** - It is used to uniquely identify a port within the SensorModel.

- ◇ **ioid** - It is used to uniquely identify a variable-set within the SensorModel. It can either directly refer to a variable-set in the **specs** section of the SensorModel, or it may refer to a port using `<portid> '#in'` or `<portid> '#out'`.
- ◇ **dataid** - It is used to uniquely identify a variable within a session. It is also used in ports to define variables required or asserted by the port.
- ◇ **standardid** - It is used to uniquely identify a Standard within the SensorModel.
- ◇ **controlid** - It is used to uniquely identify a Parameter within a particular Standard.
- ◇ **parameterid** - It is used to uniquely identify a Parameter within the whole SensorModel.
- ◇ **serviceid** - It is used to uniquely identify an integrated service within the SensorModel.
- ◇ **resourceid** - It is used to uniquely identify a resource within the SensorModel.
- ◇ **prefix** - The prefix is used to refer to the type system defined in a resource of the SensorModel. It is mainly used in the **type**-field of variables and Parameters.
- ◇ **sessionid** - It is used to uniquely identify a session of the Session Service.

## 6.2 STATIC DEFINITIONS

The static definition of the SensorModel contains structures which are used by the user when creating a new Sensor. Its one and only root is the element **SensorModel** containing the five parts which comprise the SensorModel. These are as follows:

- ◇ **InputOutput-Specification**: Defines inputs and outputs of the Sensor.
- ◇ **Control-Specification**: Defines global control parameters and access rules.
- ◇ **Service-Specification**: Defines external services used at runtime.
- ◇ **Sensor-Specification**: Defines external resources for generation of the Sensor.

Name	Type	Containment	Multiplicity
iospecification	InputOutputSpecification	child	1-1
controlspection	ControlSpecification	child	1-1
servicespecification	ServiceSpecification	child	1-1
sensorspecification	SensorSpecification	child	1-1

Table 5: Class-Overview: SensorModel

### 6.2.1 SensorModel

This element is the root element of any SensorModel.

#### iospecification

This attribute contains the specification of the input/output ports of the Sensor as well as the declaration of the session-variables.

#### controlspection

This attribute specifies global properties of the sensors. Apart from built-in properties (like execution time, number of failures, etc.) it is possible to declare own properties, defining their domain and finally specifying rules for accessing them.

#### servicespecification

With this attribute external services (like operations of other sensors or other Web services) which are used by the Sensor can be specified.

#### sensorspecification

This specification includes the declaration of required resources like external XML Schemas, WSDL-files, etc. for the code generation of the Sensor.

### 6.2.2 InputOutputSpecification

This part of the specification is used to define ports of a Sensor, which are essential in order to be invocable by the Controller or other Sensors. There are two types of ports:

- ◇ PortExtract
- ◇ PortUpdate

Ports can define both an input and output-part. The input part defines data necessary for invocation. If the required data is not provided upon invocation, the execution of the Sensor will be aborted immediately. On the other hand, the output

part of the port specifies data guaranteed by the sensor upon successful invocation. In this way a port can be seen as a kind of contract between the Sensor and its caller.

`PortExtract` is, as the name suggests, used for extraction of data either from XML-documents or through combination of data from different sources. Since other Sensors might want to make use of existing extraction operations, every extraction port has to define both an input and an output part.

In contrast, `PortUpdate` is primarily used to update data of the environment. This can of course be done in many ways, e.g. execute Web services, update data in databases, write files, etc. Since chances of such kind are most if not all the time very context sensitive, update ports can only be invoked from the Controller or directly from other ports of the Sensor of which they are part. This security mechanism is used to prevent unnoticed and unwanted updates in a complex Sensor composition. For this reason and furthermore to not mistakenly use it for extraction purpose, this kind of port must not define an output part.

Apart from the port definition, this specification also includes the declaration of session variables. Those can first of all be used to directly write or read data from the session, but also to define common sets of variables referred to by different ports.

At last global settings for the session management are also specified here. In order for the Controller to determine whether to open a new session or to use an existing session to invoke a given Sensor, three parameters are used:

- ◇ `isuseraware` - execute only in same session if user is the same.
- ◇ `isactivityaware` - execute only in same session if activity is the same.
- ◇ `sessiontime` - execute only in the same session if subsequent invocations occur within a given time frame.

NOTE These parameters are only used by the Controller for session management.

There is no restrictive mechanism which would prevent an invocation among Sensors, even though it might violate the specified rules.

#### `isuseraware`

This flag variable is used to declare the Sensor as user-aware. If it is set to `false`, the Sensor is insensible to users and therefore creates only one session for all requests. If set to `true`, requests of the same user are grouped and invoked in the same session (see 5.3.4 Filter Techniques).

Name	Type	Containment	Multiplicity
isuseraware	boolean	attribute	1-1
isactivityaware	boolean	attribute	1-1
sessiontime	boolean	attribute	1-1
ports	PortAbstract	child	0-*
defs	IOSet	child	0-*

Table 6: Class-Overview: InputOutputSpecification

NOTE Although 'users' can be any kind of information which serves to identify a particular user as such, in this framework the 'user' is defined as URI. It is extracted by the Controller from the SOAP-header of service interactions (see 5.1.2 Controller).

#### isactivityaware

Similar to the userware-flag, this attribute declares the Sensor as activity-aware. If it is set to `false`, the Controller will create only one session for all invocations, regardless of the activity. In contrast, if set to `true`, every session of Sensor the is limited to one and only one activity. If no activity is provided, the Sensor will not be invoked by the Controller (see 5.3.4 Filter Techniques).

NOTE Similar to the concept of the 'user', an 'activity' can basically be anything that serves to identify a task that users are involved in. It is up to the developer to decide on an appropriate granularity of tasks. In the context in which the framework has been developed, a URI is used to identify activities. It is automatically extracted by the Controller upon receipt of SOAP-notifications (see 5.1.2 Controller).

#### sessiontime

This attribute controls the time frame which is used for a session. The first call of a Sensor, which opens the session, also initializes the session-frame. Any subsequent call occurring within the given time frame and satisfying both awareness-settings, will be executed in the same session. Both calls outside the session frame and those not satisfying the awareness-parameters will lead to opening a new session with its own session frame. The value is to be interpreted in seconds. If set to 0, any call will open a session, regardless of the values of the other awareness-flags (see 5.1.2 Controller).

#### ports

This variable contains a list of the ports of the Sensor. Since the type PortAbstract

is an abstract concept, only `PortExtract` and `PortUpdate` elements can be added. Ports which are defined here serve as entry-point to execute the Sensor. Every port defines its own business logic, which has to be implemented by the developer after Sensor generation.

#### **defs**

This is used to specify sets of variables that can be used to read and write data from and to the session. Therefore it can be used as a means to exchange data between ports in case that data cannot be passed using the input and output-variables of the port. In other words, these definitions provide a way to read and write data not specified in the input and output-parts of the port.

#### **6.2.3 PortAbstract**

This type is the abstract base class of any port-type. Realised port-types are:

- ◊ `PortExtract` - extract / combine data
- ◊ `PortUpdate` - update / write data persistently

A port serves as a contract between the Sensor and the caller. For more information please see 6.2.2 `InputOutputSpecification`.

Name	Type	Containment	Multiplicity
portid	string (alphanumeric, ., -)	attribute	1-1
input	IOInput	child	0-1
output	IOOutput	child	0-1

Table 7: Class-Overview: `PortAbstract`

#### **portid**

Any port has to be uniquely identifiable by its id.

#### **input**

Using this variable, data required upon invocation is specified. The data has to be set in the session by the caller before invoking the Sensor. If not present in the session, the execution will be aborted immediately. The data specified here will be compiled to a type-class of the programming language and used in the extensions.

**NOTE** All the data which is required for execution should be specified here. Optional data, on the other hand, should be specified using the `defs`-field from

InputOutputSpecification and loaded from the session manually during execution.

#### output

This field specifies the output assertions. In other words, the Sensor (in particular the business logic of the port) has to ensure that the data specified here is properly set in the session at the end of the execution. The Sensor itself will check its own output assertions after execution. In case of violation, it will cancel execution prematurely with an error. Similar to the `input`-field, the data specified here will be compiled to a type-class.

NOTE Only data which can be guaranteed after execution should be specified here. Optional data should be specified using the `defs` field from InputOutputSpecification and saved to the session manually during execution.

NOTE Only PortExtract ports might define an output part. PortUpdate must not use this field.

#### 6.2.4 PortExtract -> PortAbstract

This port-type is used to gather data and provide it for other ports or Sensors by saving it to the session. Gathering data can be done in many ways, e.g. extracting data from XML-documents, querying Web services, combining data already stored in the session, etc.

Name	Type	Containment	Multiplicity
no fields defined			

Table 8: Class-Overview: PortExtract

For more information about the fields, please see 6.2.3 PortAbstract.

NOTE Extraction ports can be called by the Controller, the Sensor itself, and any other Sensor.

#### 6.2.5 PortUpdate -> PortAbstract

The update port is used to update data in the context in which CSDF is used. Possible ways to do so are, for instance, to update values in a database, execute update operations in Web services, save data to files, etc.

For more information about the fields, please see 6.2.3 PortAbstract.



Name	Type	Containment	Multiplicity
no fields defined			

Table 9: Class-Overview: PortUpdate

NOTE Since updates might be very context sensitive, update-ports can be only called by the Controller and by ports of the Sensor itself. Sensors cannot directly execute update-ports of other Sensors.

### 6.2.6 IOInput -> IODefinition

This type describes the input part of a port.

Name	Type	Containment	Multiplicity
no fields defined			

Table 10: Class-Overview: IOInput

For more information about the fields, please see 6.2.9 IODefinition.

### 6.2.7 IOOutput -> IODefinition

This type describes the output part of a port.

Name	Type	Containment	Multiplicity
no fields defined			

Table 11: Class-Overview: IOOutput

For more information about the fields, please see 6.2.9 IODefinition.

### 6.2.8 IOSet -> IODefinition

This class defines a set of variables which can be used to exchange data with the Session Service as well as a common definition that can be included by other ports and/or other Sensors.

For more information about the fields, please see 6.2.9 IODefinition.

#### readable

If this flag variable is set to `true`, the Generator will generate methods to load instances of the data defined here from the session. Using these, the developer can

Name	Type	Containment	Multiplicity
readable	boolean	attribute	1-1
writable	boolean	attribute	1-1

Table 12: Class-Overview: IOSet

freely load data from the session during the execution of the business logic. If the variable is set to **false** no such methods will be generated.

#### **writable**

If this flag variable is set to **true**, the Generator will generate methods to save instances of the data defined here to the session. The developer can then manually save data to the session during the execution of the business logic. If the variable is set to **false** no such methods will be generated.

### 6.2.9 IODefinition

This abstract type defines a set of variables which will be compiled by the Generator to a type-class of the chosen programming language. This type will then be used by the developer to exchange data with the Session Service.

There are two ways to add variables:

- ◇ *Explicit definition* - variables are defined directly
- ◇ *Include instruction* - variables are loaded from another IODefinition

Name	Type	Containment	Multiplicity
id	string (alphanumeric, ., -, #)	attribute	1-1
specs	DataSpecification	attribute	0-*
includes	IOReference	attribute	0-*

Table 13: Class-Overview: IODefinition

#### **id**

Any IODefinition needs to have a unique id, the so-called **id**. While the developer might omit this id when defining input and output-parts of ports (the Generator will automatically alter it to properly fit the **portid**), it must be specified when creating IOSets.

**specs**

This field is used to define variables in the set directly.

**includes**

Using this attribute, variable-definitions from other sets can be loaded. Instead of defining the same sets of variables over and over again, it is recommended to define them once and include them when needed.

**6.2.10 DataSpecification**

This type is used for variable specification within the SensorModel. A variable consists of the following parts:

- ◇ *Id* - unique identifier of variable
- ◇ *Type* - a domain that defines what is to be considered as a valid value
- ◇ *Assertions* - additional constraints on the domain of the variable
- ◇ *QoS-attributes* - quality of service attributes

Variables are used to hold atomic pieces of data. Although the Session Service allows exchange of single variables, the developer is urged to use the predefined methods for data-exchange of whole variable-sets defined via IODefinition. A comprehensive introduction to the type system of variables is given in 5.2.5 Resources and Type System.

Name	Type	Containment	Multiplicity
dataid	string (alphanumeric, ., -)	attribute	1-1
datatype	string (prefix ':' type)	attribute	1-1
description	string	attribute	0-1
assertion	Assertion	child	0-*
qos	QoSAttribute	child	0-*

Table 14: Class-Overview: DataSpecification

NOTE The DataSpecification is a variable specification and DataValue is the actual value of a variable.

**dataid**

This id is used to uniquely identify a DataSpecification within an IODefinition. Furthermore it is used in DataValue to refer to the definition of variables.

**datatype**

This field specifies the type of the variable. The format of this value is as follows: `<prefix> ':' <typename>`. `prefix` has to be declared in a resource in SensorSpecification, while `typename` has to refer to a valid type within that resource. The only prefix which need not and must not be specified as a resource is `xsd`, which refers to the standard XML Schema datatypes. It will be automatically added to the SensorModel during code generation.

- ◊ When referring to types from XML Schemas, `typename` has to refer to a valid type within that schema.
- ◊ When using WSDL-resources, `typename` has to match an operation of that WSDL-Schema. It is important to use an AssertionWSOperation when using types from a WSDL-resource.
- ◊ When using Sensor-resources, `typename` has to refer to a valid IODefinition within that Sensor.

**EXAMPLE** Examples of valid types, given the following resources:

Namespace	Prefix	Type
http://mytypes.com	mt	ResourceSchemaXsd
http://mailinglist.in_context.eu	ml	ResourceWSDL
http://types.othersensor.com	tos	ResourceSensor

Type	Valid	Explanation
xsd:string	true	simple XML Schema 'string'-type
xsd:anyType	true	complex XML Schema 'anyType'-type
mt:Project	true	XML-Schema 'Project'-type in namespace 'http://mytypes.com'
ml:CreateList	true	operation 'CreateList' in given WSLD document
tos:DocumentData	true	IOSet 'DocumentData' in given Sensor
tos:PortA#out	true	IOOutput of port 'PortA' in given Sensor
tos:PortSend#in	true	IOInput of port 'PortSend' in given Sensor
mx2:Project	false	unbound prefix 'mx2'
tos:PortA#xyz	false	undefined sequence '#xyz'
mt:xsd:abc	false	wrong format
string	false	missing prefix

**description**

This is an optional field describing the variable specification.

**assertion**

Using assertions it is possible to restrict the domains of variables even further, e.g. by introducing additional checks with regular expressions or evaluating XPath-expressions on XML instance data.

**qos**

QoS-attributes are used to express metadata about variables. It is possible to define additional requirements such as reliability or timeliness. Unfortunately, the current databinding of CSDF does not support QoS-attributes in extensions. This will be adapted in future version of CSDF.

**6.2.11 Assertion**

This abstract type is the base of all assertion-classes. By means of assertions it is possible to further restrict the domain of variables defined by DataSpecification. The different types of assertions are as follows:

- ◇ AssertionXPath - evaluate XPath statements on XML data
- ◇ AssertionWSOperation - check for specific operation in SOAP document
- ◇ AssertionRegex - perform regular expression check on strings

**NOTE** Though DataSpecification defines types for variables, the Sensor core handles variable instances as DataValue, therefore the actual data is treated as string when applying the assertions. Only later, before and after execution of the extensions, databinding is performed.

Name	Type	Containment	Multiplicity
description	string	attribute	0-1

Table 15: Class-Overview: Assertion

**description**

This is an optional field describing the assertion.

### 6.2.12 AssertionExpression -> Assertion

This is an abstract base class for expression-based assertions.

Name	Type	Containment	Multiplicity
no fields defined			

Table 16: Class-Overview: AssertionExpression

For more information about the fields, please see 6.2.11 Assertion.

### 6.2.13 AssertionXPath -> AssertionExpression

This assertion type is used to evaluate XPath statements on variables. In order to realise that, the data is first converted to an XML document and the compiled XPath statement is executed thereafter. This assertion fails if the data does not contain valid XML or if the XPath does not match any node of the XML document.

Name	Type	Containment	Multiplicity
xpath	string	attribute	1-1
nsaware	boolean	attribute	1-1
nsmap	NamespaceDefinition	child	0-*

Table 17: Class-Overview: AssertionXPath

For more information about the fields, please see 6.2.12 AssertionExpression -> Assertion.

**xpath**

This field holds the actual XPath statement.

**nsaware**

This flag indicates whether the XPath expression is namespace-aware or not. If set to `true`, the statement is interpreted as namespace-aware. If set to `false`, the namespace-prefix of nodes in the XPath expression is treated as part of the node name itself.

**nsmap**

With this field, namespaces can be defined for the XPath statement. In case that no prefix is given, the namespace is treated as default namespace.

**6.2.14 AssertionRegex -> AssertionExpression**

With this type, regular expressions can be performed on variables. The assertion fails if the regular expression does not match the given data.

Name	Type	Containment	Multiplicity
regex	string	attribute	1-1

Table 18: Class-Overview: AssertionRegex

For more information about the fields, please see 6.2.12 AssertionExpression -> Assertion.

**regex**

This field holds the regular expression.

**6.2.15 AssertionWSOperation -> Assertion**

This assertion is used to check whether a given input data parsed as SOAP XML document contains a particular Web service call. Although this could also be done using an XPath statement, it is recommended to use this assertion in case of SOAP data, as it is also responsible for cutting out the SOAP request/response relevant data from the SOAP envelope. If a variable uses a type from a WSDL-document without using this assertion, the data cannot properly be converted during the databinding-process and therefore the execution might fail.

**NOTE** For that reason always use `AssertionWSOperation` when using types from WSDL-documents.

This assertion fails if the data cannot be successfully parsed as XML document or if it does not contain the specified SOAP request or response.

Name	Type	Containment	Multiplicity
operation	string	attribute	1-1
request	boolean	attribute	1-1

Table 19: Class-Overview: AssertionWSOperation

For more information about the fields, please see 6.2.11 Assertion.

#### **operation**

This field describes the WSDL-operation which should be checked for when evaluating the SOAP document.

#### **request**

This flag has to be set to `true` in case of a SOAP request and `false` in case of a SOAP response.

### **6.2.16 NamespaceDefinition**

This class is used to define a namespace (e.g. in `AssertionXPath`). Namespaces are defined with a prefix and a URL of the namespace. In case that no prefix is given, the namespace is assumed to be the default namespace.

Name	Type	Containment	Multiplicity
namespace	string	attribute	1-1
prefix	string	attribute	0-1

Table 20: Class-Overview: NamespaceDefinition

#### **namespace**

This field describes the namespace URL.

#### **prefix**

This optional field defines a prefix for the namespace.

### **6.2.17 QoSAttribute**

QoS-attributes are used to annotate data instances with metadata and/or to specify functional and non-functional requirements. To simplify matters, the domain of



QoS-attributes is limited to values between 0 and 1, expressing the degree the QoS attribute is served. A value close to 0.0 means no strong connection to the given QoS, while a value near 1.0 expresses a tight relationship. Thus both discrete and continuous domains have to be resampled to a value between 0 and 1 before usage.

Name	Type	Containment	Multiplicity
qosid	string (alphanumeric, ., -)	attribute	1-1
value	double (0.0-1.0)	attribute	1-1

Table 21: Class-Overview: QoSAttribute

**qosid**

Any QoSAttribute needs to be uniquely identifiable within the given DataSpecification.

**value**

This value expresses the degree of congruence resampled to values between 0 and 1. 0 expresses the lowest value in the original domain, while 1 expresses the highest possible value.

**6.2.18 IOReference**

This type refers to the input or output part of a port or a variable set. The reference can be either local (reference within the Sensor) or external (reference to another Sensor). It is used to include variable definitions and therefore supports the concept of reusability.

NOTE The Generator tries to resolve references on best effort base. If all references form a non-circular graph, they will be resolved correctly regardless of the order they appear in. In case of circular references, the Generator produces an error and aborts the generation.

Name	Type	Containment	Multiplicity
ioid	string (alphanumeric, ., -, #)	attribute	1-1
nsprefix	string	attribute	1-1

Table 22: Class-Overview: IOReference

**ioid**

This id specifies which IODefinition should be included. For IOSet, the id is simply

copied. To refer to parts of port, the id of the port appended by `#in` for the input part and respectively `#out` for the output part is used.

**EXAMPLE** Examples of valid references for the given Sensor:

Port-ID	IOSet-ID
SendMessage	
StoreData	MessageData

Ioid	Valid	Explanation
MessageData	true	references IOSet 'MessageData'
SendMessage#in	true	references input part of port 'SendMessage'
StoreData#out	true	references output part of port 'StoreData'
MessageData#in	false	IOSets do not have input and output parts
Project	false	'Project' is not defined
StoreData#all	false	'#all' is not a valid suffix

### nsprefix

The prefix is used to identify the Sensor which the variable definitions should be included from. In case of a local inclusion it has to be set to `self`. If the developer intends to include definitions from other Sensors, the desired Sensor first has to be included as a ResourceSensor in the SensorSpecification and is then referenced via its prefix.

**EXAMPLE** Examples of valid references for the given Sensor-resources:

Type	location	namespace	prefix
ResourceSensor	http://.../Sen1		s1
ResourceSensor	http://.../Sen2	http://.../Sen2/types	s2a
ResourceSensor	http://.../Sen2	http://mytypes.com	s2b

Nsprefix	Valid	Explanation
self	true	local reference
s1	true	references 'http://.../Sen1'
s2a	true	references 'http://.../Sen2'
s2b	false	's2b' does not include the type system of the Sensor itself but another resource
s3	false	unbound prefix 's3'

### 6.2.19 ControlSpecification

Sensors can have global Parameters which provide status information about the Sensor or a means to influence the behaviour of the Sensor. For more information about Parameters, please see 5.1.4 Sensor.

Apart from Parameter definition, this specification also enables the developer to define access rules on Parameters. In some cases it might be reasonable to allow only specific users - identified via a secret key - to read or write sensitive Parameters of a Sensor.

Name	Type	Containment	Multiplicity
standard	Standard	child	0-*
access	ControlAccess	child	0-*
activationkey	string	attribute	1-1

Table 23: Class-Overview: ControlSpecification

#### standard

This field defines the Standards which are used by the Sensor. A Standard is a set of Parameter definitions. In the current stage of development, there are two different kinds of Standards:

- ◇ StandardStatus - a common set of Parameters for all Sensors
- ◇ StandardUserDefined - an extension base for user-defined Parameter definitions

#### access

As mentioned before, it is possible to control access on Standards. This field enables the developer to define both default and specific access rules on Standards. There are currently two ways to define access:

- ◇ ControlAccessDefault - default access for any user (no authentication required)
- ◇ ControlAccessUser - access for authenticated users (via password)

**NOTE** Though there is no limit to the amount of access definitions, this specification must not contain more than one ControlAccessDefault definition.

#### activationkey

Although in most cases it might not be necessary to restrict Activation (see 5.4.5 Active and Passive Sensors), it is possible to define a key for activating/passivating a Sensor. If this security mechanism is not required, the field should be left blank.

### 6.2.20 Standard

This is the abstract base class for all Standards. A Standard is a set of Parameter definitions which can either be programmatically generated or user defined. A standard can be seen as a grouping mechanism for Parameters that belong together and for which common access rules apply.

Name	Type	Containment	Multiplicity
standardid	string (alphanumeric, ., -)	attribute	1-1
description	string	attribute	0-1
parameter	ControlParameter	child	0-*

Table 24: Class-Overview: Standard

#### standardid

Each standard has to be identifiable by a unique id. This id then is also used as a prefix for Parameters to uniquely identify them within the whole Sensor.

**EXAMPLE** Standard `standard.status` with Parameter `no_of_invocations` becomes  
`standard.status.no_of_invocations`.

#### description

This field contains the description of the Standard.

#### parameter

Here the Parameters of the Standard are defined.

### 6.2.21 StandardStatus -> Standard

This type provides a set of common Parameters which will automatically be assimilated by the Generator and are already integrated in the logic of the Sensor. The id of this Standard is `standard.status` and it contains the following Parameters:

- ◇ `name` - name of Sensor (taken from SensorSpecification)
- ◇ `author` - author of Sensor (taken from SensorSpecification)
- ◇ `published` - time and date when Sensor was initialized
- ◇ `description` - description of Sensor (taken from SensorSpecification)

- ◇ `service` - service address from Sensor (taken from `SensorSpecification`)
- ◇ `no_of_invocations` - number of successful invocations of Sensor since start-up
- ◇ `no_of_errors` - number of invocations that failed because of an internal or external error
- ◇ `last_error` - descriptive message of last error
- ◇ `avg_processtime` - average processing time of an invocation
- ◇ `latest_processtime` - processing time of last invocation

Name	Type	Containment	Multiplicity
no fields defined			

Table 25: Class-Overview: `StandardStatus`

For more information about the fields, please see 6.2.20 Standard.

NOTE Developers must not extend this Standard with additional Parameters in the `SensorModel`.

### 6.2.22 `StandardUserDefined` -> `Standard`

This type is intended to be used by developers to introduce their own Sensor specific Parameters.

Name	Type	Containment	Multiplicity
no fields defined			

Table 26: Class-Overview: `StandardUserDefined`

For more information about the fields, please see 6.2.20 Standard.

### 6.2.23 `ControlParameter`

A Parameter is a means to retrieve status information from the Sensor and to set control flags to influence the behaviour of a Sensor on a level independent from single invocations. Parameters, which can be both read and written via the Sensors Web service, are typically tightly integrated into the business logic of the Sensor. The

Name	Type	Containment	Multiplicity
controlid	string (alphanumeric, ., -)	attribute	1-1
description	string	attribute	0-1
type	string (prefix ':' type)	attribute	1-1
default	string	attribute	1-1
readable	boolean	attribute	1-1
writable	boolean	attribute	1-1

Table 27: Class-Overview: ControlParameter

value domain of a Parameter is defined via XML Schema Types. At the current stage of development only simple types are allowed, though.

#### controlid

This is a unique id to identify a Parameter within the Standard. To uniquely identify Parameters on Sensor-level the `standardid` of the Standard and this `controlid` are combined with ':' to form a so-called `parameterid`.

**EXAMPLE** Standard `standard.status` with Parameter `no_of_invocations` becomes  
`standard.status.no_of_invocations`.

#### description

This optional field is intended to describe the purpose of the Parameter and explains how it affects the working process of the Sensor.

#### type

This field specifies the type of the Parameter. The format of this value is as follows: `<prefix> ':' <typename>`. `prefix` has to be declared in a resource in `SensorSpecification`, while `typename` has to refer to a valid type within that resource. The prefix `xsd` might be used to refer to standard XML Schema datatypes.

**NOTE** Unlike the `datatype` field in `DataSpecification`, this type can only refer to XML Schema simple types.

**EXAMPLE** Examples of valid types given the following resources:

Namespace	Prefix	Type
http://mytypes.com	mt	ResourceSchemaXsd

Type	Valid	Explanation
xsd:string	true	simple XML Schema 'string'-type
xsd:anyType	false	no complex types allowed
mt:MySimpleType	true	XML-Schema 'MySimpleType'-type in namespace 'http://mytypes.com'
mt:MyComplexValue	false	no complex types allowed
mx2:Project	false	unbound prefix 'mx2'
mt:xsd:abc	false	wrong format
string	false	missing prefix

**default**

With this field, the default value, which is assigned to the Parameter during the initialization phase, can be assigned.

**NOTE** It is crucial that the default value is valid according to the domain specified via `type`.

**readable**

This attribute sets the readable-flag of the Parameter. If set to `true`, the Parameter can be read via the Web service. If set to `false`, it is not possible to load the value of the Parameter via the Web service.

**writeable**

This attribute sets the writeable-flag of the Parameter. If set to `true`, the Parameter can be written via the Web service. If set to `false`, it is not possible to set the value of the Parameter via the Web service.

**6.2.24 ControlAccess**

This class is the abstract base for access restriction definitions. It specifies read and write-access to Standards on either a default or a user-specific setting.

Name	Type	Containment	Multiplicity
no fields defined			

Table 28: Class-Overview: ControlAccess

### 6.2.25 ControlAccessDefault -> ControllAccess

In case that no specific access rules for Standards are needed, this class can be used. It grants read and write-access of all defined Standards to anyone. In case that special rules are required, the use of ControlAccessUser is recommended. Of course it is also possible to combine both settings, with the only limitation that there must not be more than one ControlAccessDefault defined in the ControlSpecification of a Sensor.

Name	Type	Containment	Multiplicity
no fields defined			

Table 29: Class-Overview: ControlAccessDefault

### 6.2.26 ControlAccessUser -> ControllAccess

If special access rules for Standards are needed, this class should be used. It defines access rules on Standards using a password for authentication. In case that default access to Parameters without authentication is needed, the use of ControlAccessDefault is recommended.

Name	Type	Containment	Multiplicity
standardaccess	ControlStandardAccess	attribute	0-*
key	string	attribute	1-1

Table 30: Class-Overview: ControlAccessUser

#### standardaccess

This list defines specific access rules for Standards.

NOTE Although there is no limit to access rules for Standards, there must not be two rules which refer to the same Standard.

#### key

This is the authentication key which must be provided by the caller upon access of Parameters. In case that no ControlAccessDefault is defined, the key might be left blank to define default access, otherwise it must be a non-empty string.



Name	Type	Containment	Multiplicity
standard	Standard	reference	1-1
readable	boolean	attribute	1-1
writeable	boolean	attribute	1-1

Table 31: Class-Overview: ControlStandardAccess

### 6.2.27 ControlStandardAccess

This type is used to specify access for a specific, previously defined Standard.

#### **standard**

This element is a reference to a Standard which was previously defined in Control-Specification.

#### **readable**

This attribute indicates whether Parameters of the referred Standard can be read accessed or not. If set to **true**, read access to Parameters is possible. If set to **false**, Parameters cannot be read using this access definition.

NOTE It is understandable that only Parameters that are marked as readable in their ControlParameter definition can be accessed this way.

#### **writeable**

This attribute indicates whether Parameters of the referred Standard can be write accessed or not. If set to **true**, write access to Parameters is possible. If set to **false**, Parameters cannot be written using this access definition.

NOTE It is understandable that only Parameters, which are marked as writeable in their ControlParameter definition can be accessed this way.

### 6.2.28 ServiceSpecification

To perform its task, the Sensors might integrate third party services e.g. Web services. For more detailed information about integrated services, please see 5.1.4 Sensor.

This specification furthermore specifies the location of the Controller Web service. Upon initialization the Sensor will attempt to register itself at the Controller. If the Controller service cannot be accessed, the Sensor will fail to initialize.

Name	Type	Containment	Multiplicity
controllerservice	string	attribute	1-1
services	ServiceDescription	child	0-*

Table 32: Class-Overview: ServiceSpecification

**controllerservice**

This field specifies the address of the Controller Web service.

**services**

This list contains all the services that are to be integrated by the Generator and which might be used in the logic of the Sensor.

**6.2.29 ServiceDescription**

This class is the abstract base for all service types.

Name	Type	Containment	Multiplicity
serviceid	string (alphanumeric, _)	attribute	1-1
description	string	attribute	0-1

Table 33: Class-Overview: ServiceDescription

**serviceid**

Every service specification must have a unique id.

**description**

This optional field is intended to describe the kind of service and the way it is integrated into the Sensor logic.

**6.2.30 ServiceWS**

With this type Web services can be integrated into the Sensor. The Generator will automatically generate client-stubs for the service specified via the WSDL file.

Name	Type	Containment	Multiplicity
wSDL	URL	attribute	1-1

Table 34: Class-Overview: ServiceWS

For more information about the fields, please see 6.2.29 ServiceDescription.

**wsdl**

This is the location of the WSDL-document of the Web service.

### 6.2.31 ServiceSensor

This specification is used to integrate other Sensors as services. It allows the developer to specify both the service address of the Sensor and the port that should be executed. The Generator will then automatically generate a method for invocation of the specified Sensor.

**NOTE** It is possible to leave one or even both fields blank, as they can be set during the initialization phase of the Sensor. Thus it is possible to react to changes of the location of Sensors without altering the SensorModel.

Name	Type	Containment	Multiplicity
serviceurl	URL	attribute	0-1
portid	string (alphanumeric, ., -)	attribute	0-1

Table 35: Class-Overview: ServiceSensor

For more information about the fields, please see 6.2.29 ServiceDescription.

**serviceurl**

This attribute specifies the location of the SensorCore Web service of the Sensor for integration.

**portid**

This field is used to specify which port of the addressed Sensor should be used.

### 6.2.32 SensorSpecification

This type contains general information about the Sensor, for instance the service address at which it will be deployed, a description, the author, etc. Furthermore, it provides fields to integrate external resources into the SensorModel. Resources might appear similar to the services of ServiceDescription, but they are both fundamentally different. Services are used to integrate real services (like Web services, database-services, etc.) into the Sensor and are invoked at runtime. Resources on the other enrich the Sensor and SensorModel, e.g. by adding a type system, which might then be referred to within the SensorModel. For more information about type systems, see 5.2.5 Resources and Type System.

Name	Type	Containment	Multiplicity
name	string (alphanumeric, _)	attribute	1-1
description	string	attribute	0-1
serviceurl	URL	attribute	1-1
author	string	attribute	1-1
resources	Resource	child	0-*

Table 36: Class-Overview: SensorSpecification

**name**

This is the name of the Sensor, which will also be used as service name when publishing the Sensor as Web service. For this reason it might only contain alphanumeric characters and underscores.

**NOTE** It is crucial that the `name` matches the last part of the `serviceurl` that describes the deployment address of the Sensor.

**description**

This optional field is used to describe the purpose of the Sensor.

**serviceurl**

This field specifies the location at which the Sensor will be available after deployment.

**NOTE** It is important that the last part of the `serviceurl` matches the `name` that will be used as name of the Web service.

**EXAMPLE** If a Sensor named `MySensor` is published on a host `http://myhost.com:8080`, the `serviceurl` might look like `http://myhost.com:8080/axis2/MySensor`.

**author**

This is the name or the email of the developer of the Sensor and might be used in times of failures, updates or maintenance to determine who is responsible for developing the Sensor.

**resources**

Resources are external documents which are integrated into the SensorModel and might be referred to from within the model itself. Currently supported resources are identified via a namespace and a unique prefix, but future versions of the CSDF might support other kinds of resources as well.

NOTE A Sensor must not have two resources with the same id or the same prefix.

The following types of resources are supported at the current stage of development:

- ◇ ResourceSchema - include a schema (e.g. XML Schema) into the SensorModel
- ◇ ResourceWSDL - include a WSDL resource and its types
- ◇ ResourceSensor - include types or resources of another Sensor

### 6.2.33 Resource

This abstract type is the root of all resource-type classes. A resource is a data source that provides additional information or extends the SensorModel.

Name	Type	Containment	Multiplicity
resourceid	string (alphanumeric, _)	attribute	1-1
location	string	attribute	1-1
local	boolean	attribute	1-1

Table 37: Class-Overview: Resource

#### resourceid

Any service must have a unique id that is used for identification.

#### location

This is the location of the service. It can either be either a URL or a local file.

#### local

This flag indicates whether the `location` has to be interpreted as URL or as relative local file. If set to `true`, the resource is loaded as a file relatively from the location of the SensorModel-file. If set to `false`, the `location` is interpreted as URL.

### 6.2.34 ResourceWithNamespace -> Resource

This abstract type provides the base for all namespace based resources. A resource which resides in a particular namespace can be addressed within the SensorModel by its unique prefix.

For more information about the fields, please see 6.2.33 Resource.

#### namespace

This is the namespace of the resource.

Name	Type	Containment	Multiplicity
namespace	string	attribute	1-1
prefix	string	attribute	1-1

Table 38: Class-Overview: ResourceWithNamespace

**prefix**

Using this prefix the resource can be identified within the SensorModel.

NOTE The SensorModel must not contain two resources with the same prefix.

**6.2.35 ResourceSchema -> ResourceWithNamespace**

This resource is the abstract base for all schema resource types. Schemas provide additional types which can be used in, for instance, DataSpecification or ControlParameter.

Name	Type	Containment	Multiplicity
no fields defined			

Table 39: Class-Overview: ResourceSchema

For more information about the fields, please see 6.2.34 ResourceWithNamespace -> Resource.

**6.2.36 ResourceSchemaXsd -> ResourceSchema**

This resource is used to include an XML Schema into the SensorModel.

Name	Type	Containment	Multiplicity
no fields defined			

Table 40: Class-Overview: ResourceSchemaXsd

For more information about the fields, please see 6.2.34 ResourceWithNamespace -> Resource.

**6.2.37 ResourceSensor -> ResourceWithNamespace**

With this resource type, the data types of another Sensor or one of its resources can be loaded and integrated into the SensorModel. In order to do so, the `location` of the

resource has to be set to the address of the Sensor from which a resource should be included. To include the type system of the Sensor itself (in other words, the types which are generated from the InputOutputSpecification of the Sensor), the `namespace` field has to be set to the namespace of the type system (`<serviceurl> '/types'`) or can just be left blank. To load an external resource, the `namespace` is set to the namespace of the resource which should be included.

Name	Type	Containment	Multiplicity
no fields defined			

Table 41: Class-Overview: ResourceSensor

For more information about the fields, please see 6.2.34 ResourceWithNamespace -> Resource.

**EXAMPLE** Given a Sensor at `http://x.com/axis2/Sen1` with the following resources:

Type	Namespace	Prefix
ResourceSchemaXsd	<code>http://mytypes.com</code>	typ
ResourceWSDL	<code>http://mailinglist.in_context.eu</code>	mail

The following ResourceSensor could be defined at another Sensor:

Location	Local	Namespace	Valid
http://x.com/axis2/Sen1	false	http://x.com/axis2/Sen1/types	true
or:			
http://x.com/axis2/Sen1	false		true
http://x.com/axis2/Sen1	false	http://mytypes.com	
http://x.com/axis2/Sen1	false	http://mailinglist.in_context.eu	true
http://x.com/axis2/Sen1	false	http://notype.com	false

### 6.2.38 ResourceWSDL -> ResourceWithNamespace

This resource type is used if the Sensor has to deal with SOAP requests of a Web service. If included as a resource, the Generator will automatically extract the schema of the WSDL document, store them as a separate XML Schema resource and generate type-classes for it. Therefore at runtime the data in SOAP requests and response can be directly converted to a typed object. This does not only reduce the implementation effort of the developer, since real objects can be used instead of manually extracting information of a SOAP envelope, but also eliminates possible error sources.

Name	Type	Containment	Multiplicity
convertSchemaElementToSchemaType	boolean	attribute	1-1

Table 42: Class-Overview: ResourceWSDL

For more information about the fields, please see 6.2.34 ResourceWithNamespace -> Resource.

#### **convertSchemaElementToSchemaType**

Messages in the WSDL document always refer to XML Schema elements, but the SensorModel only uses XML Schema types. For this reason an automatic element-to-type conversion can be performed in the Generator. If this flag is set to **true**, all global elements in the WSDL schema part are converted to XML Schema types. If set to **false**, no such conversion takes place.

## 6.3 DYNAMIC DEFINITIONS

The dynamic definitions are not directly used when creating a SensorModel, but are rather needed for communication purpose (especially as transfer-objects) between Sensor, Controller and Session Service.

### 6.3.1 DataSet

A dataset can be viewed as a collection of DataValue objects. It is used to exchange a set of variables between services.

Name	Type	Containment	Multiplicity
data	DataValue	child	0-*

Table 43: Class-Overview: DataSet

#### **data**

This field contains the variables that belong to the collection.



### 6.3.2 DataValue

The `DataValue` can be seen as an instance of the variable-type defined by `DataSpecification`. The instance holds the data in string representation as well as a `dataid`, thus it can be matched with the corresponding `DataSpecification`.

Name	Type	Containment	Multiplicity
<code>dataid</code>	string (alphanumeric, ., -)	attribute	1-1
<code>value</code>	string	attribute	1-1
<code>qos</code>	<code>QoSAttribute</code>	child	0-*

Table 44: Class-Overview: `DataValue`

#### `dataid`

The `id` is used to identify the `DataSpecification` to which the variable belongs.

#### `value`

This field holds the actual value of the variable. It is stored in string representation and must conform to the type that is defined by the corresponding `DataSpecification`.

#### `qos`

Although not used in the current state of CSDF, variables can be annotated with metadata. For instance, the reliability or the precision of the data can be coded in QoS-attributes attached to the actual data.

The `DataSpecification` defines which QoS-attributes are required and how high their values have to be. In contrast, the `qos` here describes the actual value of a QoS-parameter.

### 6.3.3 ParameterValue

The `ParameterValue` is the actual instance of a `Parameter` defined by `ControlParameter`. Via the `parameterid`, it can be matched with its definition.

Name	Type	Containment	Multiplicity
<code>parameterid</code>	string (alphanumeric, ., -)	attribute	1-1
<code>value</code>	string	attribute	1-1

Table 45: Class-Overview: `ParameterValue`

#### `parameterid`

The id is used to identify both the Standard and the ControlParameter defining the Parameter.

#### value

This is the actual value of the Parameter, stored in string representation. It must conform to the type defined in ControlParameter.

### 6.3.4 PortReference

This type is used to refer to a particular port of a Sensor using the address of the Sensor service and the name of the port.

Name	Type	Containment	Multiplicity
serviceuri	string	attribute	1-1
portid	string (alphanumeric, ., -)	attribute	1-1

Table 46: Class-Overview: PortReference

#### serviceuri

This is the address of the Sensor - the location where the Sensor is deployed.

#### portid

This is the name of the port. It should refer to a valid port of the Sensor that is addressed via `serviceuri`.

### 6.3.5 SensorInfo

The SensorModel is too complex and contains sensitive data (such as passwords), therefore it cannot be exchanged via Web services. To be able to exchange important aspects of the SensorModel in a compact way, the SensorInfo is used. It describes the ports as well as the Forward definitions of a Sensor.

Name	Type	Containment	Multiplicity
sensor	ServiceSensor	child	1-1
ports	SensorInfoPort	child	0-*
services	string	attribute	0-*

Table 47: Class-Overview: SensorInfo

#### sensor

This field contains an instance of ServiceSensor which does not describes an inte-

grated service of the Sensor, but rather the Sensor itself. The `portid` field is left blank and the `serviceid` is set to the name of the Sensor defined in `SensorSpecification`.

#### **ports**

This field contains a collection of all ports of the Sensor.

#### **services**

This is a list of all integrated services of the Sensor.

### **6.3.6 SensorInfoPort**

This type is a compact version of the port definition via `PortAbstract`, enhanced with data concerning the Forward specification of the port. For more information about Forwards, see 5.4.4 Types of Links.

Name	Type	Containment	Multiplicity
update	boolean	attribute	1-1
portid	string (alphanumeric, ., -)	attribute	1-1
inputs	SensorInfoIO	child	0-*
outputs	SensorInfoIO	child	0-*
forwardto	PortReference	child	0-*
forwardfrom	PortReference	child	0-*

Table 48: Class-Overview: `SensorInfoPort`

#### **update**

This flag determines the kind the port. If it is set to `true`, the port is an update port (`PortUpdate`). In contrast, `false` indicates a port used for data extraction (`PortExtract`).

#### **portid**

This field is used to identify the port. It corresponds to the `portid` in `PortAbstract`.

#### **inputs**

This field is a collection of all input variable definitions of the described port.

#### **outputs**

This field is a collection of all output variable definitions of the described port.

#### **forwardto**

A Forward that links the Sensor's output with another Sensor's input port is stored in this field.

#### **forwardfrom**

A Forward that links another Sensor's output with this' Sensor's input is stored in this field.

### **6.3.7 SensorInfoIO**

This type is a simplified version of `DataSpecification` and describes an input or output variable of a port. Since the complex types of `DataSpecification` could not be interpreted, as they refer to resources of the `SensorModel`, the `datatype` is split into the namespace and the actual typename.

Name	Type	Containment	Multiplicity
ns	string	attribute	1-1
type	string	attribute	1-1
dataid	string (alphanumeric, ., _)	attribute	1-1

Table 49: Class-Overview: `SensorInfoIO`

#### **ns**

This describes the namespace of the type being used. It is retrieved by looking up the prefix in the resources in `SensorSpecification`.

#### **type**

The second part of the original `datatype` is stored in this field and it describes the actual type within the given namespace.

#### **dataid**

This id is used as unique identifier of the data in the current session. It is obtained from `DataSpecification`.

---

## 7 CSDF WEB SERVICES

The Controller, the Session Service and the Sensor communicate via Web services. To comprehend the actual message exchange between these components, a fundamental knowledge of these services is necessary. This chapter introduces all Web service interfaces in detail. It contains a list of all operations, the data being transmitted as well as possible faults. Additionally, it gives an insight into how the respective operations are used in the workflow of CSDF.

## PREFACE

This chapter deals with the Web services of CSDF. Although it gives a deep insight into the workflow and the communication patterns of CSDF, it is not imperative to read this part in order to understand later chapters of this thesis. Therefore, readers who are more interested in a guide on creating a sample Sensor might skip this chapter and continue reading at 8 How To.

Web services of CSDF commonly make use of parts of the SensorModel. To understand the data which is actually conveyed in a service call, it is suggested to study chapter 6 Sensor Model before reading this section. Furthermore, global ids described in 6.1.2 SensorModel ID are also used throughout the text.

## 7.1 SENSOR SERVICES

This part will introduce the Web services of the Sensor. A Sensor provides five service interfaces which are as follows:

- ◇ SensorIO - operations to query port and Forward definitions of Sensor
- ◇ SensorControl - operations to get and set Parameters
- ◇ SensorService - operations to query integrated services
- ◇ SensorCore - operations for invocation and is-alive queries
- ◇ SensorManagement - operations to initialize, activate and passivate Sensor

## 7.2 SENSORIO

This service deals with port and Forward definitions of the Sensor. It provides operations to query input requirements and output assertion of a port and to list a detailed specification of all Forwards defined at the Sensor.

This functionality is mainly used by the Controller upon registration of the Sensor. To construct an adequate Filter for the registering Sensor, the Controller loads

both port specifications and Forward definitions. Though not yet implemented, this information could be used to realise *dynamic service composition* and to *visualise Sensor composition*. The first stands for an algorithm that automatically identifies an adequate chain of Sensors capable of processing a service interaction, given only the specification of the input and the desired output. The latter describes a tool providing functionality to visualise the actual Sensor composition in a graph.

### 7.2.1 GetIOSpecification

This operation is used to load the whole InputOutputSpecification of a Sensor that contains all information about port and variable definitions.

This service is used...

- ◇ ...by the Controller to retrieve the input and output specifications of the Sensor upon registration.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	specification	InputOutputSpecification	1-1

Table 50: Service-Overview: GetIOSpecification

Output:

`specification`

This field contains the port and variable specification of the Sensor.

### 7.2.2 GetPort

Given a `portid`, this operation returns the port definition in an instance of `PortAbstract`. It can be used to query information about a particular port.

	Name	Type	Multiplicity
<b>Input</b>	portid	string	1-1
<b>Output</b>	update	boolean	1-1
	port	PortAbstract	1-1
<b>Fault</b>	UnknownIdentifierFault		

Table 51: Service-Overview: GetPort

Input:**portid**

This is the id of the port that should be loaded.

Output:**update**

This flag is set to **true**, if the loaded port is an update port (PortUpdate) and **false**, if the port is an extraction port (PortExtract).

**port**

This contains the definition of the port being requested.

Fault:**UnknownIdentifierFault**

If no port with the given id is defined, this fault will be returned.

**7.2.3 ListAllForwards**

This operation lists all Forward definitions of a Sensor.

This service is used...

- ◇ ...by the Controller to load the Forwards of all ports of the Sensor. The Controller then uses this information to integrate the Sensor in the compositions of CSDF.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	forwards	tForward	0-*

Table 52: Service-Overview: ListAllForwards

Output:**forwards**

This field contains all Forwards defined at the Sensor.

**7.2.4 GetPortForwards**

This operation lists all Forwards that are defined on a particular port of the Sensor.



	Name	Type	Multiplicity
<b>Input</b>	portid	string	1-1
<b>Output</b>	input	tForward	0-1
	output	tForward	0-1
<b>Fault</b>	UnknownIdentifierFault		

Table 53: Service-Overview: GetPortForwards

Input:**portid**

This id specifies the port from which the Forwards should be loaded.

Output:**input**

This field contains a list of all Forward-Froms defined on the input port, if any.

**output**

This field contains a list of all Forward-Tos defined on the output port, if any.

Fault:**UnknownIdentifierFault**

If no port with the given id is defined, this fault will be returned.

**7.2.5 Type: tForward**

This type is used to describe the data of a Forward.

Name	Type	Multiplicity
input	boolean	1-1
portid	string	1-1
forward	PortReference	1-*

Table 54: Type-Overview: tForward

**input**

This flag indicates the type of Forward being defined. If set to **true**, the Forward is defined on the output port and references an input port (Forward-To). If set to **false**, the Forward is defined on the input port and references an output port (Forward-From).

**portid**

This is the id of the port at which the Forward is defined.

**forward**

This is a list of references to ports of other Sensors. Combined with **input** and **portid** of the local port, the Forward definition can be completed.

## 7.3 SENSORCONTROL

This service deals with Standards, Parameters, access rules and resources: It provides operations to query all Standards of a Sensor and to list all Parameters of a selected Standard. Also, it is possible to retrieve access rules defined for a particular access key.

The service furthermore offers operations to read and values of Parameters. Thus, control over the workflow of a Sensor during runtime by a user or even by another Sensor becomes feasible. For instance: Sensor A invokes Sensor B and receives the result of a calculation. Yet, according to the quality-criteria of A, the result is not precise enough. In this case, A could adjust the precision-Parameter of B using this service and re-execute the calculation.

The last part of this service deals with resources. On request the Sensor can return a list of all resources that are defined in the SensorModel. Moreover, it is even possible to load the content of a resource directly via a Web service call. This tremendously enhances the flexibility of the system, as resources need not have to be shared between Sensors beforehand - Sensors can just load resources on the fly.

### 7.3.1 ListAllStandards

This operation lists all Standards including their Parameters.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	standards	Standard	0-*

Table 55: Service-Overview: ListAllStandards

Output:**standards**

This is a list of all Standards defined in the SensorModel.

**7.3.2 GetStandard**

With this operation a particular Standard and its Parameters can be queried.

	Name	Type	Multiplicity
<b>Input</b>	standarid	string	1-1
<b>Output</b>	standard	Standard	1-1
<b>Fault</b>	UnknownIdentifierFault		

Table 56: Service-Overview: GetStandard

Input:**standarid**

This id specifies the Standard to be returned.

Output:**standard**

This variable contains the Standard that was specified by the id.

Fault:**UnknownIdentifierFault**

If no Standard with the given id is defined, this fault will be returned.

**7.3.3 ListAccessForKey**

Using this operation access rules for a particular key can be queried.

	Name	Type	Multiplicity
<b>Input</b>	controlkey	string	1-1
<b>Output</b>	standards	ControlStandardAccess	0-*
<b>Fault</b>	UnknownIdentifierFault		

Table 57: Service-Overview: ListAccessForKey

Input:**controlkey**

This key is used for authentication of a user. It must be identical to **key** of **ControlAccessUser** for which the access rules should be listed.

Output:**standards**

This list contains all rules defined for the given key.

Fault:**UnknownIdentifierFault**

If no **ControlAccessUser** with the given key is defined, this fault will be returned.

**7.3.4 GetParameterValue**

This operation returns the current values of **Parameters**.

**NOTE** In special cases it might be desired to calculate the value of **Parameters** at the time queried instead of returning a pre-saved value (e.g. time the Sensor is running in seconds). This can be realised in the extension part of the Sensor. The developer can add code which dynamically calculates the value of a **Parameter** upon request. It has to be made sure that the returned value is valid according to the type of the **Parameter**.

	Name	Type	Multiplicity
<b>Input</b>	controlkey	string	1-1
	parameterid	string	1-*
<b>Output</b>	parameters	ParameterValue	1-*
<b>Fault</b>	InvalidAccessKeyFault		
	ParameterAccessFault		
	NoAccessFault		
	UnknownIdentifierFault		

Table 58: Service-Overview: GetParameterValue

Input:**controlkey**

This key is used for authentication. In case of default access via **ControlAccess-**

Default, the field must be left blank. To access via a particular `ControlAccessUser`, the respective key has to be set.

**parameterid**

This field specifies all Parameters of which the values should be loaded.

*Output:*

**parameters**

This array contains the current values of the requested Parameters.

*Fault:*

**InvalidAccessKeyFault**

If no access definition matches the provided key, this fault will be returned.

**ParameterAccessFault**

If a Parameter to be set is not defined as readable, this fault will be returned.

**NoAccessFault**

If a Parameter is marked as readable, but the loaded access definition does not include the necessary read-access rights, this fault will be returned.

**UnknownIdentifierFault**

If one of the given ids refers to a non-existent Parameter, this fault will be returned.

### 7.3.5 SetParameterValue

This method is used to set new values of Parameters.

**NOTE** If an error occurs due to invalid input (e.g. non-existent parameter ids, invalid new values, insufficient access rights), the operation will be aborted and none of the Parameters will be set to a new value.

**NOTE** In some cases it might be desired to react to new values of Parameters instantaneously. Again, this can be realised by writing code in the extension of the Sensor, which will then automatically be executed upon `SetParameterValue`-requests.

*Input:*

**controlkey**

This key is used for authentication. In case of default access via `ControlAccess-`

	Name	Type	Multiplicity
<b>Input</b>	controlkey	string	1-1
	parameters	ParameterValue	1-*
<b>Output</b>	-		
<b>Fault</b>	InvalidAccessKeyFault		
	ParameterAccessFault		
	NoAccessFault		
	UnknownIdentifierFault		
	ValueInvalidFault		

Table 59: Service-Overview: SetParameterValue

Default, the field must be left blank. To access via a particular ControlAccessUser, the respective key has to be set.

#### parameters

This field contains a list of the new values of the Parameters. It is important that all ids refer to existing Parameters and that the values are valid according to the types of the Parameters.

#### Fault:

##### InvalidAccessKeyFault

If no access definition matches the provided key, this fault will be returned.

##### ParameterAccessFault

If a Parameter to be set is not defined as writeable, this fault will be returned.

##### NoAccessFault

If a Parameter is marked as writeable, but the loaded access definition does not include the necessary write-access rights, this fault will be returned.

##### UnknownIdentifierFault

If one of the given ids refers to a non-existent Parameter, this fault will be returned.

##### ValueInvalidFault

If the new value of a Parameter is not valid according to the type of the Parameter, this fault will be returned.

### 7.3.6 ListResources

This operation returns all resource definitions specified in the SensorModel.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	resources	Resource	0-*

Table 60: Service-Overview: ListResources

#### Output:

**resources**

This field contains the resource definitions.

### 7.3.7 GetResourceByNamespace

This operation returns the content of a resource. The resource to be loaded is identified by its namespace. No wrapping XML elements are used, so it is possible to directly process the content of the resource (especially when using REST).

NOTE When developers use ResourceSensor in their SensorModel, the XML Schema of the Sensor will contain *include*-instructions that use the REST interface to load resources from the referenced Sensors.

This service is used...

- ◇ ...by Sensors integrating resources of other Sensors. Rather than sharing resources beforehand, they are loaded dynamically via this interface when needed.

	Name	Type	Multiplicity
<b>Input</b>	namespace	string	1-1
<b>Output</b>	content of the resource		
<b>Fault</b>	ResourceUnknownFault		
	ResourceFault		

Table 61: Service-Overview: GetResourceByNamespace

#### Input:

**namespace**

The namespace specifies which resource of the Sensor should be loaded.

Output:

The content of the resource is output directly without any wrapping XML element.

Fault:**ResourceUnknownFault**

If no resource for the given namespace is defined, this fault will be returned.

**ResourceFault**

If the resource cannot be found or cannot be loaded for any reason, this fault will be returned.

**7.3.8 GetNamespaceByPrefix**

This operation is used to resolve a prefix into the namespace of a resource. For instance, it might be used when looking up the type prefixes of DataSpecification, which were loaded using the services of SensorIO.

	Name	Type	Multiplicity
<b>Input</b>	prefix	string	1-1
<b>Output</b>	namespace	string	1-1
<b>Fault</b>	PrefixUnknownFault		

Table 62: Service-Overview: GetNamespaceByPrefix

Input:**prefix**

This is the prefix to be resolved into a namespace.

Output:**namespace**

This is the namespace defined for the given prefix.

Fault:**PrefixUnknownFault**

If no resource for the given prefix is defined, this fault will be returned.



## 7.4 SENSORSERVICE

This Web service handles the integrated services of the Sensor. Moreover, it provides functionality to retrieve a description of the Sensor itself.

### 7.4.1 ListAllServices

This operation returns a list of all integrated services of the Sensor which are defined in the SensorSpecification of the SensorModel.

This service is used...

- ◊ ...by the Controller in registration process to load the integrated services of the Sensor.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	controllerservice	string	1-1
	sessionservice	string	1-1
	services	ServiceDescription	0-*

Table 63: Service-Overview: GetParameterValue

#### Output:

**controllerservice**

This is the location of the Controller Web service.

**sessionservice**

This is the location of the Session Service Web service.

**services**

This field contains all integrated services of the Sensor.

### 7.4.2 GetSelf

Using this operation a short description of the Sensor can be retrieved.

This service is used...

- ◊ ...by the Controller to get the description of the registering Sensor.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	sensor	ServiceSensor	1-1

Table 64: Service-Overview: GetSelf

Output:**sensor**

This field contains the description of the Sensor. The `portid` field is left blank and `serviceid` is set to the name of the Sensor.

## 7.5 SENSORCORE

The interfaces for an actual invocation of the business logic are provided in this Web service. Given a session containing the required variables specified by the `InputOutputSpecification`, the port of the Sensor can be invoked and executed. Depending on the type of port, results might be written back to the session.

Apart from invocation, this service offers an operation to query whether a Sensor is still available and a notification method to inform about unregistration from the Controller.

### 7.5.1 Invoke

This is the most important operation of the Sensor. Using it, it is possible to invoke a particular port of a Sensor and execute its business logic. In order to do so, two requirements must be met. First of all, an open session is needed. Second, the variable requirements of the `InputOutputSpecification` of the chosen port must be met. Thus, all required variables must previously be set in the session. Upon invocation, the port will then load the variables from the session and check them for validity. If either a variable is not set or its value is not valid, the execution of the port will be cancelled prematurely. The same will happen if the Sensor has not been initialized beforehand or if the provided session is invalid or closed. For more information, please refer to 5.6.3 Service Interaction and Sensor Invocation.

This service is used...

- ◇ ...by the Controller to invoke active Sensors in case that their Filter matches an incoming service interaction.
- ◇ ...by the Controller to invoke Sensors being part of a composition.
- ◇ ...by Sensors invoking other Sensors as integrated service.

	Name	Type	Multiplicity
<b>Input</b>	sessionid	string	1-1
	portid	string	1-1
	userid	string	0-1
	activityid	string	0-1
<b>Output</b>	success	boolean	1-1
<b>Fault</b>	ControllerServiceFault		
	SessionServiceFault		
	InputRequirementFault		
	OutputAssertionFault		
	ProcessFault		
	NotInitializedFault		
	DependentServiceFault		

Table 65: Service-Overview: Invoke

Input:

**sessionid**

The session-id has to refer to a valid and already opened session of the Session Service.

**portid**

With this field the port to be executed is specified. It has to refer to a valid port of the Sensor.

**userid**

This optional field enables the caller to pass a user as context information.

**activityid**

This optional field enables the caller to pass an activity as context information.

*Output:***success**

This flag indicates whether the invocation was successful or not. It will be `true` because in cases of errors, respective faults will be returned.

*Fault:***ControllerServiceFault**

This is a reserved fault for errors encountered in the Controller. In the current development stage of CSDF, a Sensor does not communicate with the Controller during invocation, so this fault will never be returned.

**SessionServiceFault**

If the Session Service cannot be reached or the given session is not valid, this fault will be returned.

**InputRequirementFault**

If either the specified port is invalid or the required input data is not set in the session or not valid, this error will be returned.

**OutputAssertionFault**

If the output assertions of the Sensor cannot be met after invocation, this error will be returned. This usually means that the developer failed to properly set the output data in the extension.

**ProcessFault**

If problems occur during the execution of the business logic, this error will be returned. The reason for this usually is faulty code or insufficient exception handling.

**NotInitializedFault**

If the Sensor has not been initialized prior to the invocation, this error will be returned.

**DependentServiceFault**

If an error occurs not in the Sensor code itself, but in services on which the Sensor depends on, this fault will be returned.

### 7.5.2 UnregistrationNotification

This interface is used to inform the Sensor about unregistration from the Controller. If successful, the Sensor will revert to uninitialized-status.

**NOTE** It is necessary to provide the key received during registration. This is a security mechanism to prevent arbitrary shutdown of Sensors by third parties. By providing the registration key the authenticity of the Controller is guaranteed.

This service is used...

- ◇ ...by the Controller upon shutdown-request and following Sensor removal.

	Name	Type	Multiplicity
<b>Input</b>	registerkey	string	1-1
	managementservice	string	0-1
<b>Output</b>	-		
<b>Fault</b>	RegisterKeyFault		

Table 66: Service-Overview: UnregistrationNotification

Input:

**registerkey**

This is the key received by the Controller during Initialize.

**managementservice**

This is the location of the service which was responsible for registration management, i.e. the Controller.

Fault:

**RegisterKeyFault**

If the provided key does not match the key used during registration, this fault will be returned.

### 7.5.3 IsAlive

This operation is mainly used by the Controller to check whether the Sensor is still available. If the Sensor fails to reply IsAlive-requests for a certain number of times, it will get removed by the Controller. For more information about this mechanism, please see 5.1.2 Controller

This service is used...

- ◇ ...by the Controller to perform the Is-Alive algorithm.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	-		

Table 67: Service-Overview: IsAlive

## 7.6 SENSORMANAGEMENT

This Web service of the Sensor is intended to be directly operated by the developer. After deploying a Sensor, it has to be initialized in order to be used. During the process of initialization the Sensor will itself register at the Controller and can then optionally be activated. For more information about registration please refer to 5.6.3 Service Interaction and Sensor Invocation.

### 7.6.1 Initialize

After deployment, the Sensor has to be initialized, otherwise it cannot be invoked. The initialization of a Sensor will lead to its registration at the Controller. For more information about initialization, see 5.6.3 Service Interaction and Sensor Invocation.

The initialization-request also contains the Forward specification as well as optional binding entries of integrated services. For more information about forwards, please refer to 5.4.4 Types of Links. Service location and port of integrated services must either be specified in the SensorModel or contained in the initialization-request, otherwise the initialization of the Sensor will fail. Entries in the initialization-request will override default values of the SensorModel.

**NOTE** Since both the Forward mappings and the Sensor-bindings might change, they can easily be altered simply by specifying new values and invoking this operation again. This mechanism enhances the flexibility of the system tremendously, as there is no need to alter the SensorModel on changes of the linkage of Sensors. Every time the Initialize operation is invoked, the Controller will reset the Sensor to passive mode, so it might have to be activated again.

This service is used...

- ◇ ...by the user to initialize the Sensor.

	Name	Type	Multiplicity
<b>Input</b>	services	tServiceType	0-*
	forwardfroms	tForward	0-*
	forwardtos	tForward	0-*
<b>Output</b>	-		
<b>Fault</b>	ControllerServiceFault		
	InitializeFault		

Table 68: Service-Overview: Initialize

Input:**services**

This field specifies the actual binding of all ServiceSensor. At minimum all services not fully specified in the SensorModel have to be listed here.

**forwardfroms**

This field describes all Forward-Froms of the Sensor.

**forwardtos**

This field describes all the Forward-Tos of the Sensor.

Fault:**ControllerServiceFault**

If the Controller is not initialized or if there are other problems on the part of the Controller, this fault will be returned.

**InitializeFault**

If not all the services are fully specified, this error will be returned. It usually expresses that **services** did not contain all necessary services bindings.

**7.6.2 Activate**

With this operation a Sensor can be marked as active on the Controller. Active Sensors are directly invoked by the Controller in case that an adequate service interaction is received. For more information about this topic, please refer to 5.4.5 Active and Passive Sensors.

This service is used...

- ◊ ...by the user to activate the Sensor.

	Name	Type	Multiplicity
<b>Input</b>	activationkey	string	1-1
<b>Output</b>	-		
<b>Fault</b>	NotInitializedFault		
	ControllerServiceFault		
	ActivationKeyFault		

Table 69: Service-Overview: Activate

Input:**activationkey**

This field contains the activation key specified in ControlSpecification. This is a security mechanism to prevent arbitrary activation of Sensors by third parties. Only the developer of a Sensor should know the activation key specified in the SensorModel.

Fault:**NotInitializedFault**

If the Sensor is not initialized, this fault will be returned.

**ControllerServiceFault**

If the Controller is not initialized or if there are other problems on the part of the Controller, this fault will be returned.

**ActivationKeyFault**

If the provided activation key does not match the one specified in the SensorModel, this fault will be returned.

**7.6.3 IsActive**

This operation returns the actual activation status of the Sensor.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	isactive	boolean	1-1
<b>Fault</b>	NotInitializedFault		

Table 70: Service-Overview: IsActive



Output:**isActive**

This field indicates whether the Sensor is marked as active or not. If **true**, the Sensor is active and might be invoked by the Controller directly, if **false**, the Sensor is passive.

Fault:**NotInitializedFault**

If the Sensor is not initialized, this fault will be returned.

**7.6.4 Passivate**

This operation is used to set the Sensor back to passive on the Controller. A passive Sensor can only be invoked in the course of a Sensor composition or via a direct service call from another Sensor. For more information, please see 5.4.5 Active and Passive Sensors.

This service is used...

- ◇ ...by the user to passivate the Sensor.

	Name	Type	Multiplicity
<b>Input</b>	activationkey	string	1-1
<b>Output</b>	-		
<b>Fault</b>	NotInitializedFault		
	ControllerServiceFault		
	ActivationKeyFault		

Table 71: Service-Overview: Passivate

Input:**activationkey**

This field contains the activation key which is specified in `ControlSpecification`. This is a security mechanism to prevent arbitrary passivation of Sensors by third parties. Only the developer of a Sensor should know the activation key specified in the `SensorModel`.

*Fault:***NotInitializedFault**

If the Sensor is not initialized this fault will be returned.

**ControllerServiceFault**

If the Controller is not initialized or if there are other problems on the part of the Controller, this fault will be returned.

**ActivationKeyFault**

If the provided activation key does not match the one specified in the SensorModel, this fault will be returned.

**7.6.5 Type: tServiceType**

This type is used to specify the actual binding of the ServiceSensor entries defined in ServiceSpecification.

Name	Type	Multiplicity
serviceid	string	1-1
serviceuri	string	1-1
portid	string	1-1

Table 72: Type-Overview: tServiceType

**serviceid**

With this id the ServiceSensor to be bound is specified.

**serviceuri**

This is the actual service location of the Sensor.

**forward**

This field specifies the port to be used.

**7.6.6 Type: tForward**

This type is used to define Forwards on the local Sensor.

**portid**

This is the id of the local port of the Sensor.

Name	Type	Multiplicity
portid	string	1-1
forward	PortReference	1-*

Table 73: Type-Overview: tForward

**forward**

This field contains a list of all the Forwards defined on the local port. It specifies both service location and port of the external Sensors.

## 7.7 CONTROLLER

This service interface is used to interact with the Controller. It contains methods to manage the registration of Sensors, search and browse registered Sensors, find compatible ports as well as management routine for the Controller itself. For a general overview of the tasks of the Controller, see 5.1.2 Controller.

### 7.7.1 Register

This operation is used to register a Sensor at the Controller. It is invoked by the Sensor automatically upon initialization via Initialize. During the registration process, the Controller will query the Sensor for the following data:

- ◇ InputOutputSpecification - definition of ports, variables and Forwards
- ◇ ServiceSpecification - definition of integrated services
- ◇ ServiceSensor - a description of the Sensor

For more information about the registration of a Sensor, please refer to 5.6.2 Registration of a Sensor.

**NOTE** If a Sensor with the given service address is already registered at the Controller, any subsequent invocation of this operation will overwrite previous registrations. This is necessary, as the Sensor might be re-initialized in order to update mappings and bindings. For more information, please see 7.6.1 Initialize

**NOTE** Sensors are always set to passive-status after registration.

This service is used...

- ◊ ...by the Sensor upon initialization.

	Name	Type	Multiplicity
<b>Input</b>	ioservice	string	1-1
	controlservice	string	1-1
	serviceservice	string	1-1
	coreservice	string	1-1
<b>Output</b>	sessionservice	string	1-1
	sensorkey	string	1-1
<b>Fault</b>	ServiceFault		
	NotInitializedFault		
	ConfigurationFault		

Table 74: Service-Overview: Register

#### Input:

##### **ioservice**

This is the location of the SensorIO Web service of the Sensor. It usually has the value `<sensor-core> 'IO'`.

##### **controlservice**

This is the location of the SensorControl Web service of the Sensor. It usually has the value `<sensor-core> 'Control'`.

##### **serviceservice**

This is the location of the SensorService Web service of the Sensor. It usually has the value `<sensor-core> 'Service'`.

##### **coreservice**

This is the location of the SensorCore Web service of the Sensor.

#### Output:

##### **sessionservice**

This is the location of the Session Service Web service.

##### **sensorkey**

This is the key which the Sensor will use to authenticate itself when using other operations of the Controller.

Fault:**ServiceFault**

If the Controller cannot request and retrieve the above data from the Sensor, this fault will be returned.

**NotInitializedFault**

If the Controller is not initialized, this fault will be returned.

**ConfigurationFault**

If there is an error in the configuration of the Sensor, this fault will be returned.

**7.7.2 Unregister**

This is used to unregister a Sensor from the Controller. By unregistering it, the Controller will delete all data in relation to the Sensor (e.g. Forwards, ports).

	Name	Type	Multiplicity
<b>Input</b>	coreservice	string	1-1
	sensorkey	string	1-1
<b>Output</b>	-		
<b>Fault</b>	SensorKeyInvalidFault		
	ServiceNotRegisteredFault		
	NotInitializedFault		

Table 75: Service-Overview: Unregister

Input:**coreservice**

This is the location of the SensorCore Web service of the Sensor.

**sensorkey**

This is the key that the Sensor received during registration via Register.

Fault:**SensorKeyInvalidFault**

If the provided key does not match the one received during registration, this fault will be returned.

**ServiceNotRegisteredFault**

If the specified Sensor is not registered at the Controller, this fault will be returned.

**NotInitializedFault**

If the Controller is not initialized, this fault will be returned.

**7.7.3 SetActiveStatus**

Using this method it is possible to activate and passivate a Sensor. Passive Sensors will only be invoked within Sensor compositions while active Sensors will also be invoked by the Controller if an adequate service interaction matches its Filter. For more information, please see 5.4.5 Active and Passive Sensors.

This service is used...

- ◊ ...by the Sensor upon Activation and Passivation via Activate and Passivate.

	Name	Type	Multiplicity
<b>Input</b>	coreservice	string	1-1
	sensorkey	string	1-1
	active	boolean	1-1
<b>Output</b>	-		
<b>Fault</b>	SensorKeyInvalidFault		
	ServiceNotRegisteredFault		
	NotInitializedFault		

Table 76: Service-Overview: SetActiveStatus

*Input:***coreservice**

This is the location of the SensorCore Web service of the Sensor.

**sensorkey**

This is the key that the Sensor received during registration via Register.

**active**

This field specifies the activation-status of the Sensor. If set to **true**, the Sensor will be activated, if set to **false**, it will be passivated.

*Fault:***SensorKeyInvalidFault**

If the provided key does not match the key received during registration, this fault will be returned.

**ServiceNotRegisteredFault**

If the specified Sensor is not registered at the Controller, this fault will be returned.

**NotInitializedFault**

If the Controller is not initialized, this fault will be returned.

**7.7.4 ListAllServices**

This operation returns a list of all Sensors currently registered at the Controller, regardless of their activation-status.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	services	ServiceSensor	0-*

Table 77: Service-Overview: ListAllServices

Output:**services**

This field contains a list of all currently registered Sensors.

**7.7.5 ListAllServicesDetails**

This operation returns detailed specifications of all Sensors currently registered at the Controller, regardless of their activation-status.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	services	SensorInfo	0-*

Table 78: Service-Overview: ListAllServicesDetails

Output:**services**

This field contains a list of all currently registered Sensors.

**7.7.6 GetServiceByCore**

Given service location, this operation returns a detailed specification of the respective Sensor.

	Name	Type	Multiplicity
<b>Input</b>	coreservice	string	1-1
<b>Output</b>	services	SensorInfo	1-1
<b>Fault</b>	ServiceUnknownFault		

Table 79: Service-Overview: GetServiceByCore

Input:**coreservice**

This is the location of the SensorCore Web service of the Sensor to be loaded.

Output:**services**

This field contains a list of all currently registered active Sensors.

Fault:**ServiceUnknownFault**

If no Sensor with the given service location is registered at the Controller, this fault will be returned.

**7.7.7 GetServiceByRequirements**

This operation makes it possible to search for registered Sensors by particular requirements. This means that all Sensors will be returned which meet the requirements and not only those that exactly match them. The `inputids` and `outputids` are applied pairwise to each port, but it is already sufficient that only one port meets the requirements in order for the Sensor to be included in the resulting list.

	Name	Type	Multiplicity
<b>Input</b>	inputids	string	0-*
	outputids	string	0-*
	services	string	0-*
<b>Output</b>	services	SensorInfo	0-*

Table 80: Service-Overview: GetServiceByRequirements



Input:**inputids**

This is a list of dataids that must appear in the input part of the PortAbstract.

**outputids**

This is a list of dataids that must appear in the output-part of the PortAbstract.

**services**

This is a list of all service locations that must appear in the ServiceSpecification.

Output:**services**

This field contains all Sensors that meet the given requirements.

**7.7.8 ListAllActiveServices**

This operation returns a list of all Sensors currently activated at the Controller.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	services	SensorInfo	0-*

Table 81: Service-Overview: ListAllActiveServices

Output:**services**

This field contains all active Sensors.

**7.7.9 Initialize**

To start context extraction with CSDF, the Controller must be initialized first. This is done using this operation. An overview of the initialize-procedure of the Controller can be seen at 5.6.1 Initialization of Controller.

This service is used...

- ◇ ...by the user to initialize the Controller.

	Name	Type	Multiplicity
<b>Input</b>	-		
<b>Output</b>	-		
<b>Fault</b>	SubscriptionServiceFault		
	SessionServiceFault		

Table 82: Service-Overview: Initialize

*Fault:***SubscriptionServiceFault**

If there is a problem during the registration at the Logging Service, this fault will be returned.

**SessionServiceFault**

If the Session Service cannot be reached or sessions cannot be created, this fault will be returned.

**7.7.10 Shutdown**

This operation will shutdown the Controller and set it back to un-initialized status. 5.6.1 Initialization of Controller gives an overview of the shutdown-procedure of the Controller.

This service is used...

- ◇ ...by the user to shutdown the Controller.

	Name	Type	Multiplicity
<b>Input</b>	shutdownkey	string	
<b>Output</b>	-		
<b>Fault</b>	ShutdownKeyInvalidFault		

Table 83: Service-Overview: Shutdown

*Input:***shutdownkey**

This is the shutdown-key of the Controller. It must match the key specified in the properties file of the Controller.

Fault:**ShutdownKeyInvalidFault**

If the provided key does not match the key specified in the properties file of the Controller, this fault will be returned.

**7.7.11 GetCompatibleInputPorts**

This operation is used to find compatible input ports for a given output port description. For an input port to be compatible the following requirements must be fulfilled:

- ◇ The assertions given in `specs` cover at least the requirements of the input port.

A detailed introduction concerning the concept of compatibility is given at 5.4.2 Compatibility of Sensors.

This service is used...

- ◇ ...by the ConfigAssistant to identify compatible input ports.

	Name	Type	Multiplicity
<b>Input</b>	specs	SensorInfoIO	1-*
<b>Output</b>	direct	PortReference	0-*
	inferred	PortReference	0-*

Table 84: Service-Overview: GetCompatibleInputPorts

Input:**specs**

This field contains the variable specifications. They are interpreted as assertions of an output port.

Output:**direct**

This is the list of direct compatible input ports.

**inferred**

This is the list of inferred compatible input ports. (This feature has not yet been implemented so this list will always be empty).

### 7.7.12 GetCompatibleOutputPorts

This operation is used to find compatible output ports for a given input port description. For an output port to be compatible the following requirements must be fulfilled:

- ◊ The output port assertions must at least cover all requirements given by **specs**.

A detailed introduction concerning the concept of compatibility is given at 5.4.2 Compatibility of Sensors

This service is used...

- ◊ ...by the ConfigAssistant to identify compatible output ports.

	Name	Type	Multiplicity
<b>Input</b>	specs	SensorInfoIO	1-*
<b>Output</b>	direct	PortReference	0-*
	inferred	PortReference	0-*

Table 85: Service-Overview: GetCompatibleOutputPorts

#### Input:

##### **specs**

This field contains a list of variable specifications. They are interpreted as requirements of an input.

#### Output:

##### **direct**

This is the list of direct compatible output ports.

##### **inferred**

This is the list of inferred compatible output ports. (This feature has not yet been implemented so this list will always be empty).

## 7.8 SESSION SERVICE

Sessions are used as common data store to enable information exchange between Controller and Sensors. They are short lived and will be deleted either on request

or automatically after a certain time lease. For more detailed information about sessions and the tasks of the Session Service, see 5.1.3 Session Service.

### 7.8.1 Get

This operation returns the current values of selected variables and is used to read data from a session.

This service is used...

- ◇ ...by the Sensor to load input data of a port from the session.
- ◇ ...by the Sensor to load output data of a port from the session for a final check of the output assertions.

	Name	Type	Multiplicity
<b>Input</b>	sessionkey	string	1-1
	required	boolean	1-1
	dataid	string	0-*
<b>Output</b>	data	DataSet	1-1
<b>Fault</b>	SessionInvalidFault		
	UnknownIdentifierFault		

Table 86: Service-Overview: Get

#### Input:

##### **sessionkey**

This is the identifier of the session.

##### **required**

This flag indicates whether all requested values are required or not. If set to **true**, a fault will be generated if one or more requested variables are not set in the session. If set to **false**, not defined variables will simply not be contained in the result.

##### **dataid**

This field specifies the variables to be loaded from the session.

#### Output:

##### **data**

This field contains the requested data from the session.

*Fault:***SessionInvalidFault**

If no session is defined for the provided session key, this fault will be returned. The id might either be wrong or the lease ran out and the session therefore expired.

**UnknownIdentifierFault**

If `required` is set to `true`, this fault will be generated if one or more requested variables are not defined in the session.

**7.8.2 Set**

With this operation data of a session can be set. This method works with incremental data changes.

**EXAMPLE** Example of changes of the data in a session upon invocation of this operation:

Session (before)		Set		Session (after)	
ID	Value	ID	Value	ID	Value
name	Oliver			name	Oliver
age	23	age	24	age	24
		hair	brown	hair	brown
work	student	work	teacher	work	teacher
subject	English			subject	English
		income	2300	income	2300

This service is used...

- ◇ ...by the Controller to save data of the service invocation to the session.
- ◇ ...by the Sensor to save output data of a port to the session.

	Name	Type	Multiplicity
<b>Input</b>	sessionkey	string	1-1
	data	DataSet	1-1
<b>Output</b>	-		
<b>Fault</b>	SessionInvalidFault		

Table 87: Service-Overview: Set

Input:**sessionkey**

This is the identifier of the session.

**data**

This field contains all variables to be set in the session.

Fault:**SessionInvalidFault**

If no session is defined for the provided session key, this fault will returned. The id might either be wrong or the lease ran out and the session therefore expired.

**7.8.3 Delete**

Using this operation, variables can be deleted from the session. This method works with incremental data changes.

**EXAMPLE** Example of changes of the data in a session upon invocation of this operation:

Session (before)		Delete ID	Session (after)	
ID	Value		ID	Value
name	Oliver	age	name	Oliver
age	23			
work	student		work	teacher
income	2300	income		

	Name	Type	Multiplicity
<b>Input</b>	sessionkey	string	1-1
	dataid	string	0-*
<b>Output</b>	-		
<b>Fault</b>	SessionInvalidFault		
	UnknownIdentifierFault		

Table 88: Service-Overview: Delete

Input:**sessionkey**

This is the identifier of the session.

**dataid**

This list specifies variables to be deleted from the session.

*Fault:*

**SessionInvalidFault**

If no session is defined for the provided session key, this fault will returned. The id might either be wrong or the lease ran out and the session therefore expired.

**UnknownIdentifierFault**

If one or more variables that are specified are not defined in the session, this fault will be generated.

#### 7.8.4 SessionCreate

This operation opens a new session. It is possible to define the lifespan of the session as well as the refresh time. The latter specifies to which point the lifespan should be extended in case of access. Both parameters are just treated as suggestions. It is up to the Session Service itself to decide on values for both parameters. For more details about session leases, see 5.1.3 Session Service.

This service is used...

- ◇ ...by the Controller to create a new session before invoking Sensors matching an incoming service interaction.

	Name	Type	Multiplicity
<b>Input</b>	lease	integer	1-1
	refresh	integer	1-1
<b>Output</b>	sessionkey	string	1-1
	commitkey	string	1-1
	lease	integer	1-1

Table 89: Service-Overview: SessionCreate

*Input:*

**lease**

This field specifies the suggested lifespan in seconds.

**refresh**

This field specifies the suggested refresh time in seconds.



Output:**sessionkey**

This is the id of the new session.

**commitkey**

This key is needed for deletion of a session via SessionDestroy.

**lease**

This is the actual lease of the session in seconds.

**7.8.5 SessionDestroy**

This operation is used to manually close a session and discard all its data. To successfully delete a session, the **commitkey** is needed which was obtained upon creation. This is a security mechanism, so only the creator of the session can actually delete it.

This service is used...

- ◇ ...by the Controller to delete a session.

	Name	Type	Multiplicity
<b>Input</b>	sessionkey	string	1-1
	commitkey	string	1-1
<b>Output</b>	-		
<b>Fault</b>	SessionInvalidFault		
	CommitKeyInvalid		

Table 90: Service-Overview: SessionDestroy

Input:**sessionkey**

This is the identifier of the session.

**commitkey**

This is the key provided upon creation of the session.

*Fault:***SessionInvalidFault**

If no session is defined for the provided session key, this fault will be returned. The id might either be wrong or the lease ran out and the session therefore expired.

**CommitKeyInvalid**

If the provided key does not match the one generated upon creation of the session, this fault will be returned.

---

## 8 How To

After the successful installation of CSDF, we can start to create our first Sensor. The first part of this chapter is devoted to the design of a SensorModel using the graphical model editor. In a next step the Sensor code-base is generated via the Generator. After coding the business logic of the ports, their functionality is tested in simple code tests. The Sensor is then deployed and initialized using the ConfigAssistant before undergoing a final integration test.

## PREFACE

This part provides a very detailed tutorial explaining how to work with CSDF. Readers who are more interested in the results might skip this chapter and continue reading at 9 Evaluation.

In order to follow the instructions of this section the installation of CSDF is required. A comprehensive guide dealing with the setup of CSDF can be found in A Installation Guide.

## 8.1 INTRODUCTION

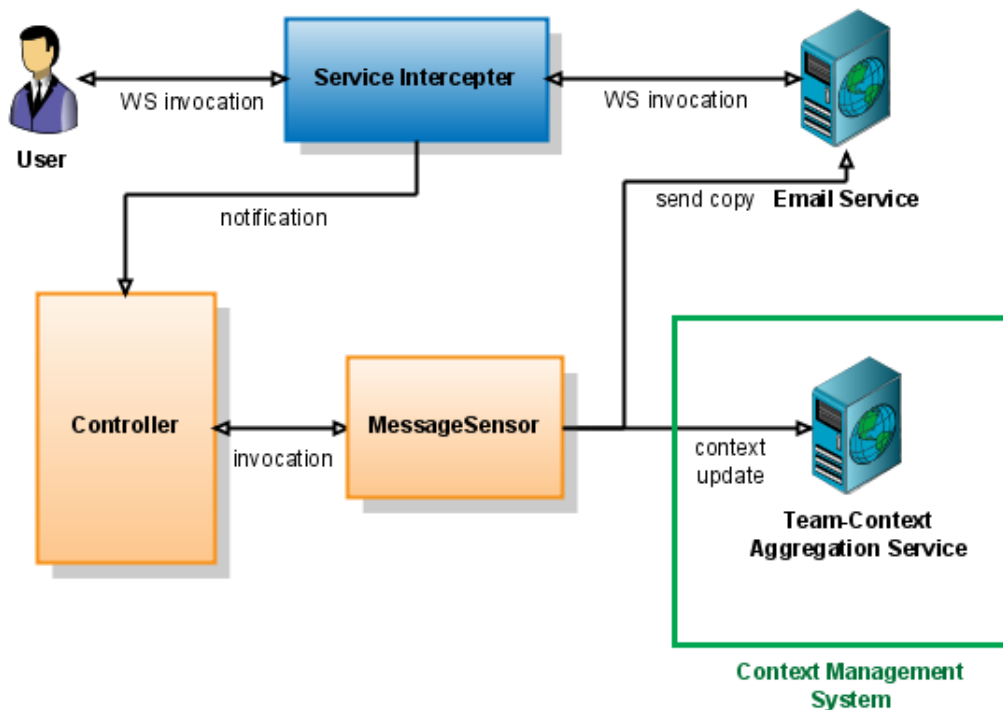


Figure 22: MessageSensor - Overview

In this part we will create a simple Sensor which will be sensitive to messages sent via the Email Service. Figure 22 shows how the Sensor will work. (This diagram

is simplified. Components like the Session Service, other Sensors as well as the components of the Context Management System are omitted). The Sensor will be both user aware and activity aware. If it receives an email invocation, it will use the Team Context Aggregation Service to add a new communication action and append it to the activity context it is executed in. Furthermore the Sensor will have a copy function, which can be enabled via a control parameter.

The source code of our Sensor, the SensorModel as well as the WSDL specifications can be found in `$CSD/Examples/MessageSensor/`.

## 8.2 THE SENSORMODEL

The SensorModel contains all the vital information of the Sensor. Its structure is clearly defined by an EMF specification and the model itself is serialised using XML. Thus it is possible to code the SensorModel in any standard text editor, because it is - after all - only XML. Still, there is a much more convenient and failure proof method to do so. The SensorModel-plugin for Eclipse provides all the tools to load, edit and save SensorModels in a richly featured graphical editing environment. Using this editor creating and altering a SensorModel becomes an easy task.

### 8.2.1 Creating a new SensorModel

We will use Eclipse to create a new SensorModel.

1. Start Eclipse and create a new Project or select an existing Project in which we will create the SensorModel.
2. Click *File / New / Others...* and choose *Example EMF Model Creation Wizards / Sensormodel Model* from the dialog and click *Next >* (Figure 23).
3. Specify a folder for the SensorModel, name it `MySensorModel.sensormodel` and click *Next >* (Figure 24).
4. For *Model Object* choose *Sensor Model* and for *XML Encoding* choose *UTF-8*, then click *Finish* (Figure 25).

Eclipse will now generate a new SensorModel-file and open it in the graphical editor (Figure 26).

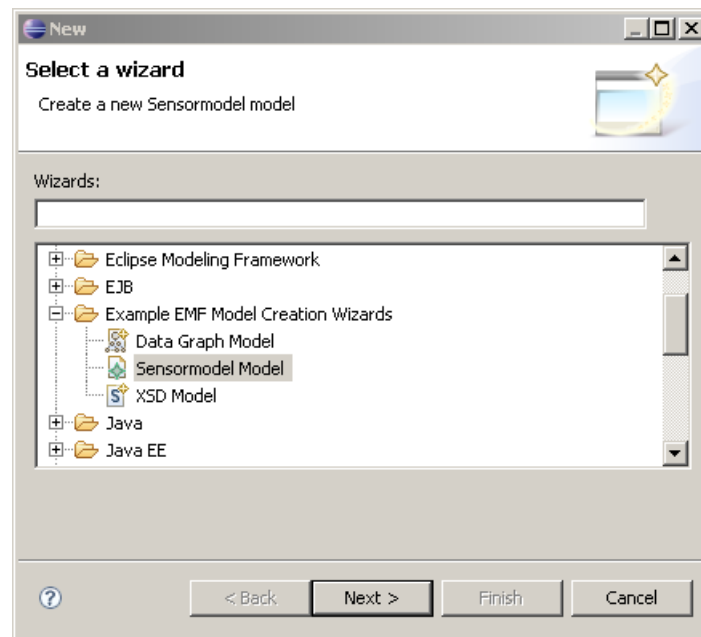


Figure 23: SensorModel Wizard (Page 1)

### 8.2.2 Edit the SensorModel

The editor is a graphical tool to display, alter and extend a SensorModel:

- ◇ Right-click on a node in the tree and choose *New Child / xxx* to create a new sub-element.
- ◇ Right-click on a node in the tree and choose *New Sibling / xxx* to create a sibling element.
- ◇ Right-click on a node in the tree and choose *Show properties view* to view and edit the fields of an element.

For detailed information about the elements and fields, please refer to SensorModel.

### 8.2.3 Adding the Input/Output Specification

Now we will create two ports for our Sensor. The first one will be an extraction port to receive and extract information from an email invocation. The second one will be an update port to realise the context-update and the copy-function.

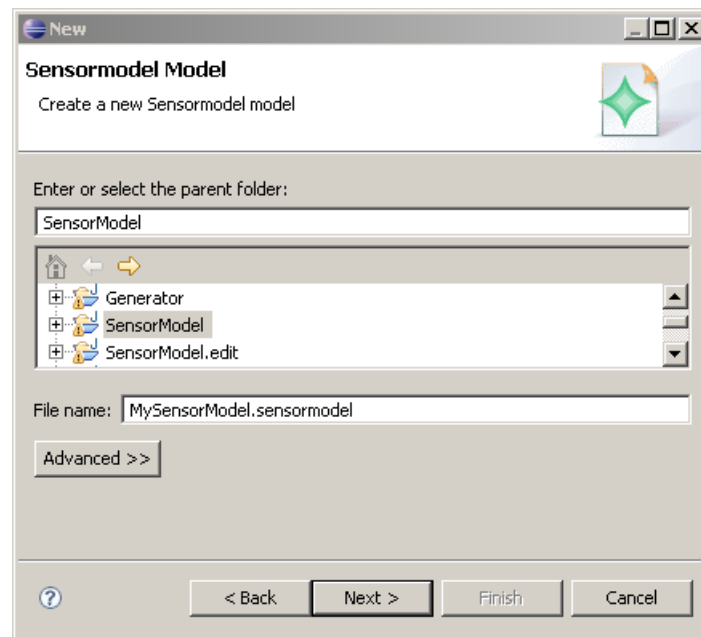


Figure 24: SensorModel Wizard (Page 2)

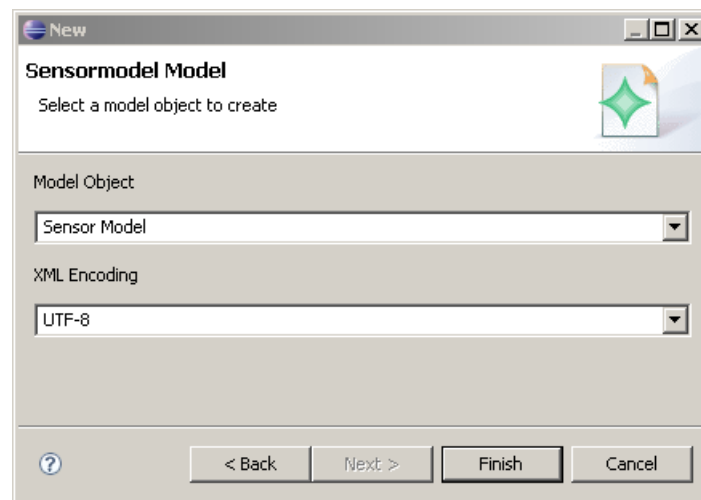


Figure 25: SensorModel Wizard (Page 3)

1. Add a new child **Input Output Specification** to **Sensor Model**.
  - ◇ Set *Isactivityaware* to "true". This will cause the Sensor to be sensitive

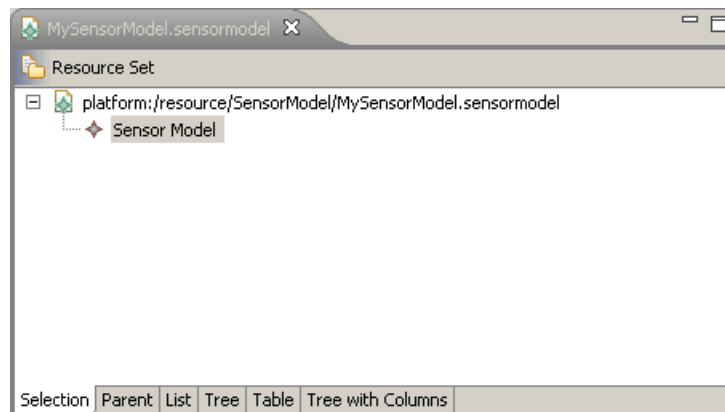


Figure 26: SensorModel Editor

to activities.

- ◇ Set *Isusweraware* to "true". This will make the Sensor sensitive to users.
- 2. Add a new child **Port Extract** to **Input Output Specification**. This port will be used to extract the receiver, the subject and the body of a `sendMessage-request`. Every port needs a unique id, so set its *Portid* to "extract\_message".
- 3. Add a new child **IO Input** to **Port Extract**. This is the input part of the port. The field *Ioid* can be left blank, because it will be automatically filled in by the Generator.
- 4. Add a new child **Data Specification** to **IO Input**.
  - ◇ Set *Dataid* to "request.message". Although ids can be chosen freely, this is a special id, which is used to receive the SOAP request message from the Controller.
  - ◇ Set *DataType* to "mes:sendMessage". This field will hold a SOAP request from the type `sendMessage`. We will later define the prefix `mes` as Resource-Sensor.
  - ◇ Set *Description* to "sendMessage request".
- 5. Add a new child **Assertion WS Operation** to **Data Specification**.
  - ◇ Set *Description* to "sendMessage operation".



- ◇ Set *Operation* to "sendMessage". This will cause the Controller to only filter out messages of the type `sendMessage`. Furthermore it enables the Sensor to cast SOAP requests to typed object in Java.
  - ◇ Set *Request* to "true".
6. Add a new child **IO Output** to **Port Extract**.
  7. Add a new child **Data Specification** to **IO Output**.
    - ◇ Set *Dataid* to "message.receiver". This field will store the receiver of the `sendMessage`-request.
    - ◇ Set *DataType* to "xsd:string".
    - ◇ Set *Description* to "message receiver".
  8. Add a new child **Data Specification** to **IO Output**.
    - ◇ Set *Dataid* to "message.subject". This field will store the subject of the `sendMessage`-request.
    - ◇ Set *DataType* to "xsd:string".
    - ◇ Set *Description* to "message subject".
  9. Add a new child **Data Specification** to **IO Output**.
    - ◇ Set *Dataid* to "message.body". This field will store the message body of the `sendMessage`-request.
    - ◇ Set *DataType* to "xsd:string".
    - ◇ Set *Description* to "message body".
  10. Add a new child **Data Specification** to **IO Output**.
    - ◇ Set *Dataid* to "message.type". This field will describe the kind of message.
    - ◇ Set *DataType* to "mt:tMessageType". We will later declare a `ResourceSchemaXsd` for this prefix and use it to include our own XML Schema.
    - ◇ Set *Description* to "message type".
  11. Add a new child **Port Update** to **Input Output Specification**. This port will initiate the context-update. The data we extracted in the previously defined port will serve as input. Set *Portid* to "update\_message".

12. Add a new child **IO Input** to **Port Update**.
13. Add a new child **IO Reference** to **IO Input**. We have already defined all necessary variables in the extraction port, so we just include them here.
  - ◇ Set *Ioid* to "extract\_message#out". This is a reference to the output part of the extraction port.
  - ◇ Set *Nsprefix* to "self". self is a special prefix and used for local references i.e. addressing within the SensorModel itself.

We have now defined all ports needed for our Sensor. The SensorModel should now look like this (Figure 27).

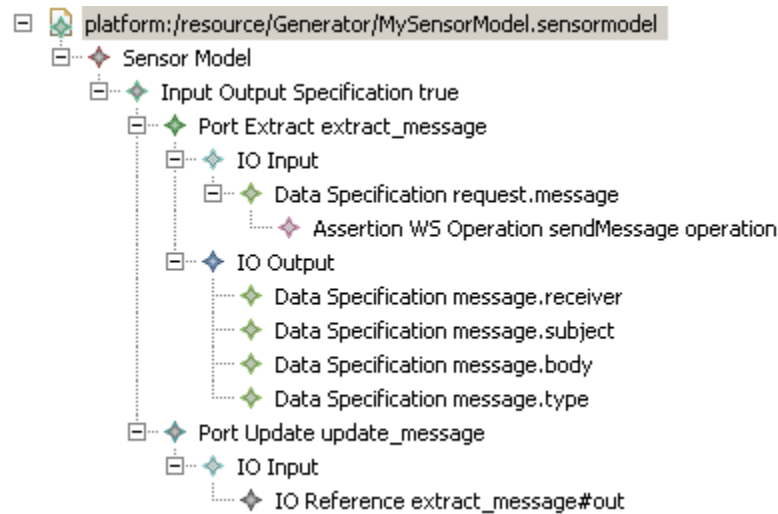


Figure 27: Input and Output Specification of SensorModel

### 8.2.4 Adding the Control Specification

To control our Sensor, we will define three Parameters. The first one will simply be used to count the number of messages that the Sensor processes. With the second one the copy-function can be enabled/disabled. Finally the last one will specify the email address to which the copies should be sent.

1. Add a new child **Control Specification** to **Sensor Model**. Leave the fields to their default values.

2. Add a new child **Standard Status** to **Control Specification**. This adds the basic Parameters for general information about the Sensor.
  - ◇ Set *Standardid* to "standard.message".
  - ◇ Set *Description* to "message parameters".
4. Add a new child **Control Parameter** to **Standard User Defined**. This is the Parameter to count messages.
  - ◇ Set *Controlid* to "count".
  - ◇ Set *Default* to "0".
  - ◇ Set *Description* to "number of messages".
  - ◇ Set *Readable* to "true".
  - ◇ Set *Writeable* to "false".
  - ◇ Set *Type* to "xsd:integer".
5. Add a new child **Control Parameter** to **Standard User Defined**. This is the Parameter to enable and disable the copy-functionality.
  - ◇ Set *Controlid* to "copy".
  - ◇ Set *Default* to "false".
  - ◇ Set *Description* to "switch copy function on or off".
  - ◇ Set *Readable* to "true".
  - ◇ Set *Writeable* to "true".
  - ◇ Set *Type* to "xsd:boolean".
6. Add a new child **Control Parameter** to **Standard User Defined**. This Parameter specifies the email address where copies of messages should be sent to.
  - ◇ Set *Controlid* to "copy-to".
  - ◇ Set *Default* to "webmaster@localhost.com".
  - ◇ Set *Description* to "recipient of copy".

- ◇ Set *Readable* to "true".
  - ◇ Set *Writeable* to "true".
  - ◇ Set *Type* to "xsd:string".
7. Add a new child **Control Access Default** to **Control Specification**. This will grant any user access to Parameters.

The SensorModel should now look like this (Figure 28).

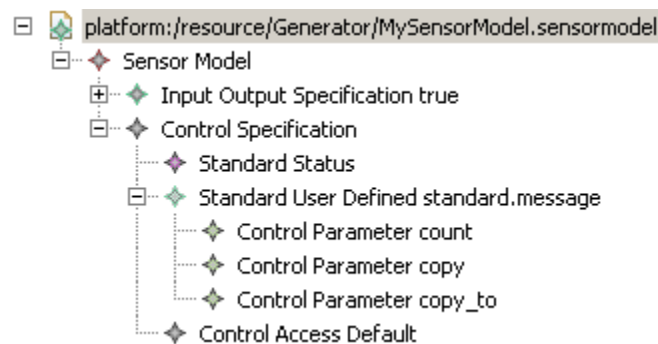


Figure 28: Control Specification of SensorModel

### 8.2.5 Adding the Service Specification

In the next step we can add integrated service to the Sensor. In our case we will include the Email Service to send copies as well as the Team-Context Aggregation service to perform the context-update.

1. Add a new child **Service Specification** to **Sensor Model**. Set *Controllerservice* to the location of the Controller. If the Controller is deployed on the same machine, the location might look like this:  
"http://localhost:8080/axis2/services/ControllerService".
2. Add a new child **Service WS** to **Service Specification**. Here we add the Email Service.
  - ◇ Set *Description* to "email service".
  - ◇ Set *Serviceid* to "service.email".

- ◇ Set *WSDL* to "<http://localhost:8080/axis2/services/ContextAwareEmailService?wsdl>".

NOTE This refers to your own implementation of the service. The WSDL for can be found in `$CSDF/Examples/MessageSensor/wsd1/`. Instead of using the WSDL of the deployed service, you can alternatively use the local WSDL file, but please be careful to use `file://` as prefix and the absolute path if you refer to local files.

3. Add a new child **Service WS** to **Service Specification**. Here we add the Team Context Aggregation Service.

- ◇ Set *Description* to "`team context aggregation service`".
- ◇ Set *Serviceid* to "`service_teamcas`".
- ◇ Set *WSDL* to "<http://localhost:8080/axis2/services/TeamContextAggregationService?wsdl>".

NOTE This refers to your own implementation of the service. The WSDL for can be found in `$CSDF/Examples/MessageSensor/wsd1/`. Instead of using the WSDL of the deployed service, you can alternatively use the local WSDL file, but please be careful to use `file://` as prefix and the absolute path if you refer to local files.

The SensorModel should now look like this (Figure 29).

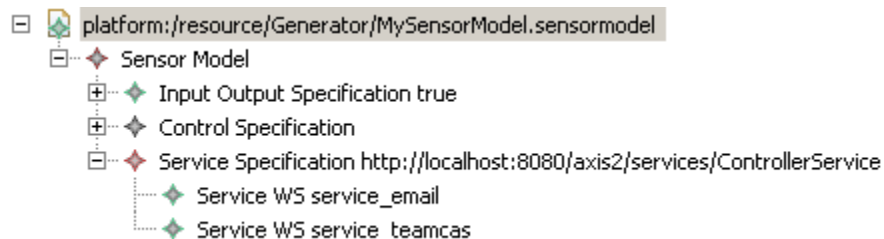


Figure 29: Service Specification of SensorModel

### 8.2.6 Adding the Sensor Specification

The last step is to enter information about the Sensor itself and include additional resources (e.g. external XML Schema files).

1. Add a new child **Sensor Specification** to **Sensor Model**.
  - ◇ Set *Author* to something like "test@localhost.com".
  - ◇ Set *Description* to "adds emails as communication-messages".
  - ◇ Set *Name* to "MessageSensor".
  - ◇ Set *Serviceurl* to the location where the Sensor will be finally deployed. If you are deploying on your local machine, the location might look like this:  
`http://localhost:8080/axis2/services/MessageSensor`.

NOTE The *Name* and the last part of *Serviceurl* must be the same.
2. Add a new child **Resource Schema Xsd** to **Sensor Specification**. Here we will include a local XML Schema with the prefix mt. We referred to this schema earlier by using `mt:tMessageType` in the output variable `message.type`.
  - ◇ Set *Local* to "true".
  - ◇ Set *Location* to "MessageSchema.xsd". This refers to the XML Schema at `$CSD/Examples/MessageSensor/MessageSchema.xsd`. We will copy the file to the appropriate directory later.
  - ◇ Set *Namespace* to "http://localhost/types/messages". The namespace has to be exactly the same as the one defined in the XML Schema.
  - ◇ Set *Prefix* to "mt".
  - ◇ Set *Resourceid* to "schema\_messagetype".
3. Add a new child **Resource WSDL** to **Sensor Specification**. We will now include the Email Service as resource. This is necessary because we used the type `mes:sendMessage` in the input variable `request.message`.
  - ◇ Set *Local* to "false".
  - ◇ Set *Location* to "http://localhost:8080/axis2/services/ContextAwareEmailService?wsdl".

NOTE This refers to your own implementation of the service. The WSDL for can be found in `$CSD/Examples/MessageSensor/wsdl/`. Instead of using the WSDL of the deployed service, you can alternatively use the local WSDL file, but please be careful to set *Local* to "true" in that case.

- ◇ Set *Namespace* to "`http://service.emailservice.ns.www.in_context.eu`". The namespace has to be exactly the same as the one used in the types-section of the WSDL document.
- ◇ Set *Prefix* to "`mes`".
- ◇ Set *Resourceid* to "`wsdl.messageservice`".
- ◇ Set *Convert Schema Element To Schema Type* to "`true`". This will automatically convert XML Schema element definitions to XML Schema type definitions.

The SensorModel should now look like this (Figure 30).

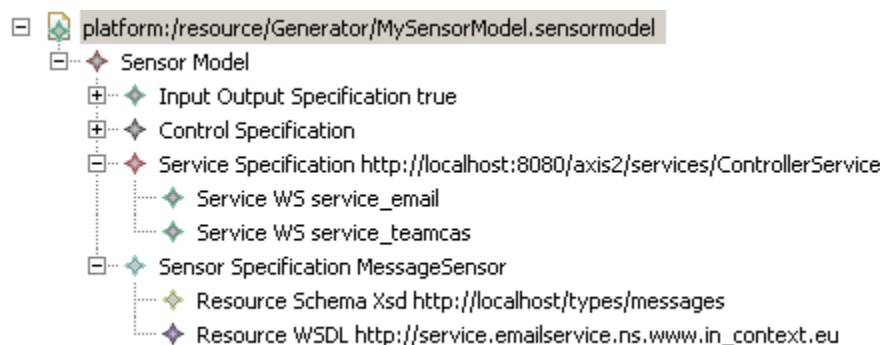


Figure 30: Sensor Specification of SensorModel

### 8.2.7 The finished SensorModel

The source code of your SensorModel should now look similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<sm:SensorModel xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sm="http://sensormodel.csd.f.in_context.eu">
  <iospecification isuseraware="true" isactivityaware="true">
    <ports xsi:type="sm:PortExtract" portid="extract_message">
      <input>
        <specs dataid="message.request" datatype="mes:sendMessage" description="sendMessage request">
          <assertions xsi:type="sm:AssertionWSOperation" description="sendMessage operation"
            operation="sendMessage" request="true"/>
        </specs>
      </input>
      <output>
        <specs dataid="message.receiver" datatype="xsd:string" description="message receiver"/>
      </output>
    </ports>
  </iospecification>
</sm:SensorModel>
```

```

    <specs dataid="message.subject" datatype="xsd:string" description="message subject"/>
    <specs dataid="message.body" datatype="xsd:string" description="message body"/>
    <specs dataid="message.type" datatype="mt:tMessageType" description="message type"/>
  </output>
</ports>
<ports xsi:type="sm:PortUpdate" portid="update_message">
  <input>
    <includes ioid="extract_message#out" nsprefix="self"/>
  </input>
</ports>
</iospecification>
<controls specification>
  <standards xsi:type="sm:StandardStatus"/>
  <standards xsi:type="sm:StandardUserDefined" standardid="message.parameters"
    description="standard.message">
    <parameter controlid="count" description="number of messages" type="xsd:integer"
      default="0" readable="true"/>
    <parameter controlid="copy" description="switch copy function on or off"
      type="xsd:boolean" default="false" readable="true" writeable="true"/>
    <parameter controlid="copy_to" description="recipient of copy" type="xsd:string"
      default="webmaster@localhost.com" readable="true" writeable="true"/>
  </standards>
  <access xsi:type="sm:ControlAccessDefault"/>
</controls specification>
<services specification controllerservice="http://localhost:8080/axis2/services/ControllerService">
  <services xsi:type="sm:ServiceWS" serviceid="service_email" description="email service"
    wsdl="http://localhost:8080/axis2/services/ContextAwareEmailService?wsdl"/>
  <services xsi:type="sm:ServiceWS" serviceid="service_teamcas"
    description="team context aggregation service"
    wsdl="http://localhost:8080/axis2/services/TeamContextAggregationService?wsdl"/>
</services specification>
<sensors specification name="MessageSensor" description="adds emails as communication-messages"
  serviceurl="http://localhost:8080/axis2/services/MessageSensor" author="test@localhost.com">
  <resources xsi:type="sm:ResourceSchemaXsd" resourceid="schema_message_type"
    location="MessageSchema.xsd" local="true" namespace="http://localhost/types/messages"
    prefix="mt"/>
  <resources xsi:type="sm:ResourceWSDL" resourceid="wsdl_messageservice"
    location="http://localhost:8080/axis2/services/ContextAwareEmailService?wsdl"
    namespace="http://service.emailservice.ns.www.in_context.eu" prefix="mes"
    convertSchemaElementToSchemaType="true"/>
</sensors specification>
</sm:SensorModel>

```

## 8.3 CODE GENERATION

In this section we will deal with the code generation, which is performed by the Generator.



### 8.3.1 Setup

Before we can invoke the Generator, the files which are referred from the SensorModel have to be gathered. The directory of the SensorModel will from now on be referred as `$MODEL_DIR`.

1. Copy `$CSDF/Examples/Resources/MessageSchema.xsd` to `$MODEL_DIR`. If you have included local WSDL documents in your SensorModel, copy them as well.

### 8.3.2 Code Generation

Now we can start the Generator. This is done via an Ant script. Note that the Eclipse Ant runner is needed to execute the script. The normal Ant runner is not able to handle JET instructions and therefore would fail.

To make it easier to invoke the Generator, we will write a short shell script. This can be used again, for instance, if the developer makes changes in the SensorModel and wants to regenerate the code-base.

1. Create the file `$CSDF/Development/Generator/MessageSensor.bat` and write the following code:

```
$ECLIPSE_HOME/eclipsec -data ../.. -consoleLog -application org.eclipse.ant.core.antRunner  
-Dmodelfile=$MODEL_DIR/MySensorModel.sensormodel -Dtargetdir=../MessageSensor setup
```

Make sure to set the correct paths to the Eclipse- and the SensorModel-directory.

2. Execute `$CSDF/Development/Generator/MessageSensor.bat`

This should invoke the Eclipse Ant runner, which will start the JET transformation and the code generation. Furthermore resources will be loaded both online and offline and stubs will be generated for included WSDL services, so the whole code generation might take a few minutes. Once the generation is finished, the code-base will be available at `$CSDF/Development/MessageSensor`.

If the code generation fails, check the following list:

- ◇ Check the paths of your shell script.
- ◇ Check the referred resources and services of your SensorModel. All local resources are relative to `$MODEL_DIR`.

- ◇ Check whether you forgot to fill in some values of the SensorModel. Some fields are required and must be filled in, otherwise the SensorModel is not valid. Refer to the SensorModel-documentation for details.
- ◇ If you included online services or resources, check whether they are really available. Check if internet connection is available.
- ◇ Check if the setup of the development machine is correct. Have all required plugins been installed and has the Generator environment been properly configured? Refer to the Installation-guide for details.
- ◇ Check the Eclipse error-log. You will find it in the in `.metadata/.log` at your workspace directory.

## 8.4 THE GENERATED CODE

### 8.4.1 Eclipse Project

The Sensor code-base was now generated by the Generator. It is available at `$CSDF/Development/MessageSensor`. We will include it as a project in Eclipse and take a look at the generated code.

1. Start Eclipse and create a new *Java Project* and name it *MessageSensor*. The location for this project will be `$CSDF/Development/MessageSensor`.  
After project creation you will notice that Java fails to compile the project. The reason for this is that required libraries are still missing, thus we will include them now.
2. Right-click on the Project *MessageSensor* / *Properties*, change to *Java Build Path* and open the tab *Libraries*.
3. Click *Add Library...* / *User Library*, check both **Emf** and **Axis2** and click *Finish*. It should now look like this (Figure 31).

After closing the dialog, Java should recompile the project and the errors should disappear. If there are still errors in the project, check the following list:

- ◇ Check if the setup of the development machine is correct. Have all required plugins been installed and has the user library setup been done properly? Refer to the Installation-guide for details.

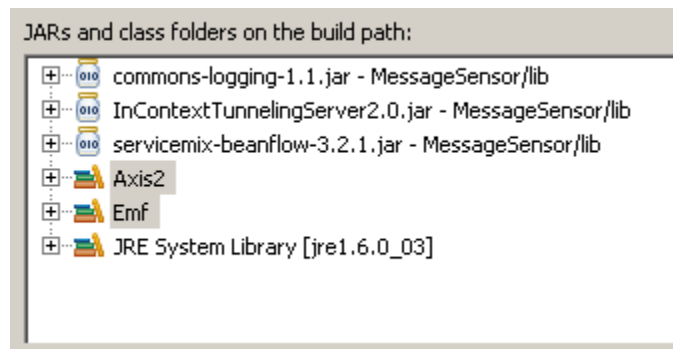


Figure 31: Libraries in the Build Path of Project

- ◇ *WSDL2Java*, which compiles the WSDL document of the integrated services to Java stubs might produce a slightly faulty code (especially if the type `xsd:anyType` is used). Check whether it is just a small error in one of the generated stubs.

#### 8.4.2 Generated Packages

Now we will take a short look at the generated packages. For details about the content of each package, refer to the API-documentation of CSDF available at [\\$CSDF/Documentation/javadoc](#).

`eu.in_context.csdf.axis.sensormodel`

This contains the *SensorModel*-classes which were generated by Axis2. They are used for ADB with Axis.

`eu.in_context.csdf.configassistant`

`eu.in_context.csdf.configassistant.gui`

These packages contain the *ConfigAssistant* - a GUI tool for initializing the Sensor, as we will see later.

`eu.in_context.csdf.conversion`

Here you will find classes for conversion between Axis2 and EMF model objects and furthermore the class for ADB conversion.

`eu.in_context.csdf.sensor`

This class contains the main application and the startup-code of the Sensor.

`eu.in_context.csdf.sensor.exception`

This class defines exceptions classes used by the Sensor.

`eu.in_context.csdf.sensor.extension`

This is the part of the Sensor which is to be extended by the developer. We will later work with the classes in here.

`eu.in_context.csdf.sensor.resource`

This package deals with the integrated resources as well as XML Schema management.

`eu.in_context.csdf.sensor.services`

In here you will find the actual implementation of the Web services offered by the Sensor.

`eu.in_context.csdf.sensor.sensor.types`

These classes were generated from the port and variable definitions. They are used for databinding.

`eu.in_context.csdf.sensor.sensormodel`

`eu.in_context.csdf.sensor.sensormodel.impl`

`eu.in_context.csdf.sensor.sensormodel.util`

These are the SensorModel-classes generated by EMF.

`eu.in_context.csdf.sensor.services.controller`

In here are the Axis2 stub classes for the Controller Web service.

`eu.in_context.csdf.sensor.services.sensorcontrol`

In here are the Axis2 stub and server classes for the SensorControl Web service.

`eu.in_context.csdf.sensor.services.sensorcore`

In here are the Axis2 stub and server classes for the SensorCore Web service.

`eu.in_context.csdf.sensor.services.sensorio`

In here are the Axis2 stub and server classes for the SensorIO Web service.

`eu.in_context.csdf.sensor.services.sensormanagement`

In here are the Axis2 stub and server classes for the SensorManagement Web service.

`eu.in_context.csdf.sensor.services.sensorservice`

In here are the Axis2 stub and server classes for the SensorService Web service.

`eu.in_context.csdf.sensor.services.session`

In here are the Axis2 stub classes for the Session Service Web service.

`eu.in_context.csdf.tools`

This package contains useful functions used in various parts of the Sensor.

`eu.in_context.www.ns.emailservice.service`

`eu.in_context.www.ns.emailservice.service.xsd`

These packages were generated by Axis2 and contain the stub classes for the integrated Email Service.

`eu.in_context.www.ns.emailservice.service.teamcontextaggregationservice`

These packages were generated by Axis2 and contain the stub classes for the integrated Team-Context Aggregation Service.

`localhost.axis2.services.messagesensor.types`

`localhost.types.messages`

This package was generated by Axis2 for ADB.

### 8.4.3 Generated Files

Next we will take a look at the kind of files included in the Project. You might want to open them to gain some insight into their content.

`src/0MessageSchema.xsd`

This is the XML Schema we included as resource.

`src/1wsdl_mes.xsd`

This file contains the XML Schema, which was extracted from the Email Service. This was done because we included the Email Service as resource.

`src/self_local.xsd`

This file contains the type definitions for the SensorModel. It was generated from the port and variable definitions.

`src/self.xsd`

This file is strikingly similar to `self_local.xsd`, except for the fact that the XSD *include*-instructions are made 'online'. It is possible to refer to this XML Schema from external sources because the include-instructions can be resolved once the Sensor is deployed.

`src/sensormodel.xml`

This file contains the altered SensorModel. Later we will explain the difference to the originally defined SensorModel.

`src/xmlschema.dtd`

`src/xmlschema.xsd`

These are the DOCTYPE and the XML Schema definition for XML Schema itself.

`test/sessiondata.xsd`

`test/example.xml`

These are files used for testing the Sensor as we will see later.

`ant.properties`

This file contains the paths to the libraries which are used to compile the Sensor with Ant.

NOTE You might want to adjust the settings in here, so Ant can compile and deploy the Sensor.

`build.xml`

This file contains the Ant tasks to build and deploy the Sensor.

`*.bat`

These files are shellscripts to execute Ant tasks.

#### 8.4.4 Commands

The Sensor comes with a few command-files which help to build, pack and deploy the Sensor.

`clean.bat`

This command is used to delete all the files that were generated during the build.

`create-jar.bat`

With this command the Sensor will be compiled and packed as .aar-archive, which can be deployed in Axis2.

`deploy.bat`

This command will deploy and pack the Sensor and copy it to the hot-deployment directory of Axis2. Understandably, this can only be done if the web server is running on the same machine or is directly accessible.

**undeploy.bat**

This command will delete the packed Web service archive from the Axis2 deployment directory and thus undeploy the Sensor. Understandably, this can only be done if the web server is running on the same machine or is directly accessible.

**sensor-init.bat**

Upon execution this will invoke the Initialize-operation of the already deployed Sensor.

**NOTE** The initialization will fail if parts of the SensorModel were not fully specified. In such cases it is suggested to use the ConfigAssistant for initialization.

**config-assistant.bat**

This will start the ConfigAssistant, which is a GUI application for configuring and initializing the deployed Sensor.

**sensor-activate.bat**

After initialization, this command might be used to mark the deployed Sensor active at the Controller.

**sensor-passivate.bat**

After initialization, this command might be used to set the deployed Sensor passive at the Controller.

**8.4.5 The altered SensorModel**

As mentioned before the CodeGenerator has slightly altered the SensorModel:

- ◇ The `ioId` of the input- and output-parts have been set.
- ◇ The `IOReference`-instructions have been resolved and have been replaced by `DataSpecification`, as can be seen in the `specs` of `PortUpdate`.
- ◇ The Parameters of `StandardStatus` have been included.
- ◇ The previously defined resources are slightly altered and some additional resources (like the type definition of the Sensor itself or the XML Schema definition) have been added.

```

<?xml version="1.0" encoding="ASCII"?>
<sm:SensorModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sm="http://sensormodel.csf.in_context.eu">
  <iospecification isuseraware="true" isactivityaware="true">
    <ports xsi:type="sm:PortExtract" portid="extract_message">
      <input ioid="extract_message#in">
        <specs dataid="request.message" datatype="mes:sendMessage" description="sendMessage request">
          <assertions xsi:type="sm:AssertionWSOperation" description="sendMessage operation"
            operation="sendMessage" request="true"/>
        </specs>
      </input>
      <output ioid="extract_message#out">
        <specs dataid="message.receiver" datatype="xsd:string" description="message receiver"/>
        <specs dataid="message.subject" datatype="xsd:string" description="message subject"/>
        <specs dataid="message.body" datatype="xsd:string" description="message body"/>
        <specs dataid="message.type" datatype="mt:tMessageType" description="message type"/>
      </output>
    </ports>
    <ports xsi:type="sm:PortUpdate" portid="update_message">
      <input ioid="update_message#in">
        <specs dataid="message.receiver" datatype="xsd:string" description="message receiver"/>
        <specs dataid="message.subject" datatype="xsd:string" description="message subject"/>
        <specs dataid="message.body" datatype="xsd:string" description="message body"/>
        <specs dataid="message.type" datatype="mt:tMessageType" description="message type"/>
      </input>
    </ports>
  </iospecification>
  <controlspecification activationkey="">
    <standards xsi:type="sm:StandardStatus" standardid="standard.status"
      description="Standard for default control parameter, as well as authoring information">
      <parameter controlid="name" description="Name of Sensor"
        type="xsd:string" default="MessageSensor" readable="true"/>
      <parameter controlid="author" description="Author of Sensor"
        type="xsd:string" default="test@localhost.com" readable="true"/>
      <parameter controlid="published" description="Published Date"
        type="xsd:dateTime" default="2009-01-14T11:14:01" readable="true"/>
      <parameter controlid="service" description="Address of service"
        type="xsd:anyURI" default="http://localhost:8080/axis2/services/MessageSensor"
        readable="true"/>
      <parameter controlid="description" description="Description of Service"
        type="xsd:string" default="receives messages and copies them" readable="true"/>
      <parameter controlid="no_of_invocations"
        description="Indicates number of successful invocations"
        type="xsd:nonNegativeInteger" default="0" readable="true"/>
      <parameter controlid="no_of_errors"
        description="Indicates how often the service failed during execution"
        type="xsd:nonNegativeInteger" default="0" readable="true"/>
      <parameter controlid="last_error" description="Last Error Message occurred during Invocation"
        type="xsd:string" default="" readable="true"/>
      <parameter controlid="avg_processtime"
        description="Average Process-Time of Service in Milliseconds"
        type="xsd:nonNegativeInteger" default="0" readable="true"/>
      <parameter controlid="latest_processtime" description="Process-Time of last Invocation"
        type="xsd:nonNegativeInteger" default="0" readable="true"/>
    </standards>
    <standards xsi:type="sm:StandardUserDefined" standardid="standard.message"
      description="message parameters">

```



```

    <parameter controlId="count" description="number of messages"
      type="xsd:integer" default="0" readable="true"/>
    <parameter controlId="copy" description="switch copy function on or off"
      type="xsd:boolean" default="false" readable="true" writeable="true"/>
    <parameter controlId="copy_to" description="recipient of copy"
      type="xsd:string" default="webmaster@localhost.com" readable="true" writeable="true"/>
  </standards>
  <access xsi:type="sm:ControlAccessDefault"/>
</controlsSpecification>
<servicesSpecification controllerservice="http://localhost:8080/axis2/services/ControllerService">
  <services xsi:type="sm:ServiceWS" serviceid="service_email" description="email service"
    wsdl="http://localhost:8080/axis2/services/ContextAwareEmailService?wsdl"/>
  <services xsi:type="sm:ServiceWS" serviceid="service_teamcas"
    description="team context aggregation service"
    wsdl="http://localhost:8080/axis2/services/TeamContextAggregationService?wsdl"/>
</servicesSpecification>
<sensorsSpecification name="MessageSensor" description="adds emails as communication-actions"
  serviceurl="http://localhost:8080/axis2/services/MessageSensor" author="test@localhost.com">
  <resources xsi:type="sm:ResourceSchemaXsd" resourceid="schema_messagetype"
    location="0MessageSchema.xsd" local="true" namespace="http://localhost/types/messages"
    prefix="mt"/>
  <resources xsi:type="sm:ResourceWSDL" resourceid="wsdl_messageservice"
    location="http://localhost:8080/axis2/services/ContextAwareEmailService?wsdl"
    namespace="http://service.emailservice.ns.www.in_context.eu" prefix="mes_wsdl"
    convertSchemaElementToSchemaType="true"/>
  <resources xsi:type="sm:ResourceSchemaXsd" resourceid="1wsdl_mes.xsd" location="1wsdl_mes.xsd"
    local="true" namespace="http://service.emailservice.ns.www.in_context.eu/xsd" prefix="mes"/>
  <resources xsi:type="sm:ResourceSchemaXsd" resourceid="xmlschema.xsd" location="xmlschema.xsd"
    local="true" namespace="http://www.w3.org/2001/XMLSchema" prefix="xsd"/>
  <resources xsi:type="sm:ResourceSchemaXsd" resourceid="self.xsd" location="self.xsd"
    local="true" namespace="http://localhost:8080/axis2/services/MessageSensor/types" prefix="self"/>
</sensorsSpecification>
</sm:SensorModel>

```

#### 8.4.6 Extension Points

There are two places for the developer to extend the generated code.

- ◇ **Ports** - For every defined port a Java source code file is generated in `eu.in_context.csdf.sensor.extension`. The developer writes the code there that should be executed upon invocation of that port. In case of an extraction port, data should be gathered or calculated and saved to the session. In case of an update port, the developer is free to update data or execute data-manipulation services.
- ◇ **Global Configuration** - In here the Parameters of the Sensor are managed. Parameters should usually have an impact on the business logic of the Sensor. They are independent of particular invocations and are used to configure the global behaviour of the Sensor.

## 8.5 CODE EXTENSION

In this section we will write the business logic of the sensor. The fully programmed Sensor can be found at `$CSDF/Examples/MessageSensor/MessageSensor_source.zip`.

### 8.5.1 Writing the Extraction Port Code

First we will write the code for extracting data after invocation by the Controller. The input of this port is - as defined in the SensorModel - a `sendMessage` SOAP request and the output is a list of message variables.

1. Open `eu.in_context.csdf.sensor.extension.ExtractExtract_message` in Eclipse. It contains the logic for the extraction port.

If you look at the input and output parameters of the `invoke()` method you will immediately notice that the Generator already converted the SOAP input request to appropriate Java types. It is thus an easy task to extract data from the request and save it to the output variable.

**NOTE** Always set all fields of the output variable! The Sensor execution will fail if you forget to fill in fields as they are required. The reason is that those fields are assertions and must be guaranteed by the Sensor, because Sensors later in the invocation-chain will depend on them.

2. Fill in the following code into the `invoke()` method:

```
public void invoke(Extract_messageIn in, Extract_messageOut out)
    throws ProcessException, DependentServiceException {
    System.out.println("Extract: 'http://localhost:8080/axis2/services/
        MessageSensor' - Operation: 'extract_message'");
    // extract data from SOAP request and save to out
    out.setMessage_body(in.getRequest_message().getBody());
    out.setMessage_receiver(in.getRequest_message().getTo());
    out.setMessage_subject(in.getRequest_message().getSubject());
    out.setMessage_type(TMessageType.EMAIL);

    // increase message counter
    int counter = Integer.parseInt(GlobalConfiguration.getConfiguration()
        .getParameter("standard.message.count"));
    counter++;
    GlobalConfiguration.getConfiguration().setParameter(
        "standard.message.count", "" + counter);
}
```

This script does nothing else but copy the data from the SOAP messages stored in the variable `in` to the output variable `out`. This will then automatically be

saved to the session and can be used by other ports. Furthermore we increase the message counter by one for every message we receive. Although values of Parameters are type checked we have to work with strings here. Maybe later versions of CSDF will support real Java types for Parameters.

As we can see, thanks to the Generator and the data binding, it becomes really easy to access even complex data structures of SOAP documents. Moreover, the developer usually does not even have to access the session directly, as the input and output data is automatically loaded from and saved to the session.

### 8.5.2 Writing the Update Port Code

In the next step we will code the logic of the update port. First it should add a command-action to the activity-context (using the Team-Context Aggregation Service). Furthermore it should send a copy of the message to a specifiable email address.

1. Open `eu.in_context.csdf.sensor.extension.UpdateUpdate_message` in Eclipse. It contains the logic of the update port.

NOTE The port only contains an input variable, as update ports are not supposed to save data to the session.

2. Fill in the following code into the `invoke()` method:

```
public void invoke(Update_messageIn in)
    throws ProcessException, DependentServiceException {
    System.out.println("Update: 'http://localhost:8080/axis2/services/
        MessageSensor' - Operation: 'update_message'");

    try {
        // add action to team context aggregation service
        TeamContextAggregationServiceStub stub =
            new TeamContextAggregationServiceStub();
        AddAction action = new AddAction();
        TActionData data = new TActionData();
        action.setIn(data);

        TCommunicationAction tcom = new TCommunicationAction();
        tcom.setActionURI(new URI("message:" + new Date().toString()));
        tcom.setExecutedByFoafAgent(new URI[] {new URI(this.getUser())});
        tcom.setNotificationType(TNotificationType.Unknown);
        tcom.setToFoafAgent(new URI[] {new URI("mailto:"
            + in.getMessage_receiver())});
        tcom.setFromFoafAgent(new URI(this.getUser()));
        tcom.setDescribesActivityURI(new URI(this.getActivity()));
        tcom.setTimestamp(new GregorianCalendar());
        data.setAction(tcom);
```

```

        stub.addAction(action);
    } catch (AxisFault e) {
        throw new DependentServiceException(
            "[Update Port] Action-Service failed - Axis Fault", e);
    } catch (RemoteException e) {
        throw new DependentServiceException(
            "[Update Port] Action-Service failed - Remote Exception", e);
    } catch (MalformedURLException e) {
        throw new ProcessException(
            "[Update Port] Malformed URL Exception", e);
    }

    // if the copy function is enabled ...
    if("true".equals(GlobalConfiguration.getConfiguration()
        .getParameter("standard.message.copy"))) {
        // ... send copy of message to specified receiver
        String receiver = GlobalConfiguration.getConfiguration()
            .getParameter("standard.message.copy_to");

        try {
            // send email via Email-WebService
            EmailServiceStub stub = new EmailServiceStub();
            SendMessage message = new SendMessage();
            message.setTo(receiver);
            message.setSubject("[MessageSensor] - Copy of '"
                + in.getMessage_subject() + "'");
            message.setBody(
                "ORIGINAL MESSAGE:\n" +
                "Receiver: " + in.getMessage_receiver() + "\n"+
                "Subject: " + in.getMessage_subject() + "\n"+
                "Type: " + in.getMessage_type() + "\n"+
                "Content: " + in.getMessage_body());

            stub.sendMessage(message);
        } catch (AxisFault e) {
            throw new DependentServiceException(
                "[Update Port] EMail-Service failed - Axis Fault", e);
        } catch (RemoteException e) {
            throw new DependentServiceException(
                "[Update Port] EMail-Service failed - Remote Exception", e);
        } catch (SendMessageFault e) {
            throw new DependentServiceException(
                "[Update Port] EMail-Service failed - SendMessageFault", e);
        }
    }
}

```

In the first part, it uses the service-stubs to add a command-action to the activity-context. The current user and activity context can easily be loaded via `this.getUser()` and `this.getActivity()`. This is because we declared our Sensor as user and activity aware (in the SensorModel).

In the next part, the email is sent in case the copy-function is enabled. The recipient is loaded from the Parameter `standard.message.copy_to`.

As we can see, the stubs for the Web service were automatically generated by the Generator, because we defined them as integrated services. In this example of course the execution of the *sendMessage*-operation will always fail, because we do not have the required access rights to execute the service. (For details refer to the actual documentation of the Email Service).

## 8.6 CODE TEST

After writing the Sensor-logic, the next step is to test the code. Once the Sensor is deployed, finding errors via remote debugging is really cumbersome, therefore local testing should be done wherever possible. This saves a lot of time during Sensor development.

### 8.6.1 Session-data Files

Port can only be invoked if a session filled with proper data is available. For local testing, we need to emulate such a session. This is done via *session-data files*. You will find an example at `test/example.xml`.

1. Make a copy the file `test/example.xml` and name it `test/sendMessage.xml`.

We will now write a session configuration which will be similar to the one which is present before the invocation of the extraction port. In order words: We will imitate the Controller and directly invoke the extraction port of the Sensor. In order to do so, we first need to set the data in the session just like the Controller would do. The actual configuration of the session-data is done with this file.

2. Fill in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:sessiondata
  xmlns:tns="http://sensor.csdf.in_context.eu/tester"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://sensor.csdf.in_context.eu/tester sessiondata.xsd">

  <tns:data dataid="request.message">
    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsd="http://service.emailservice.ns.www.in_context.eu/xsd">
      <soapenv:Header/>
```

```

        <soapenv:Body>
            <xsd:sendMessage>
                <xsd:to>test@example.com</xsd:to>
                <xsd:subject>Test message</xsd:subject>
                <xsd:body>The content of the message</xsd:body>
            </xsd:sendMessage>
        </soapenv:Body>
    </soapenv:Envelope>
</tns:data>
<tns:data dataid="response.message">
    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
        <soapenv:Header/>
        <soapenv:Body>
            <ns:sendMessageResponse
                xmlns:ns="http://service.emailservice.ns.www.in_context.eu/xsd">
                <ns:return>true</ns:return>
            </ns:sendMessageResponse>
        </soapenv:Body>
    </soapenv:Envelope>
</tns:data>
</tns:sessiondata>

```

This file contains the SOAP request and the SOAP response. The *dataids* are set to `request.message` and `request.response` - the same ids the Controller uses to save SOAP documents to the session.

Normally we would observe a sample invocation of the service and use the request and response for the session file. In doing so, the code test becomes more valid.

### 8.6.2 Test of the Extraction Port

Next, we will run the test for the extraction port.

1. Start the Tomcat Apache server if it is not already running.
2. Deploy the Session Service if it is not already deployed.
3. In Eclipse, click *Run / Run Configurations...* and add a new Java Application.

- ◇ Set *Name* to '*MessageSensor Test Extract*'.
- ◇ Set *Project* to '*MessageSensor*'.
- ◇ Set *Main class* to '*eu.in\_context.csdf.sensor.Tester*'.
- ◇ Switch to the tab *Arguments*.
- ◇ Set *Program arguments* to the following value:

```

http://localhost:8080/axis2/services/SessionService
-a http://www.in-context.eu/activity/Activity#1595
-u http://www.vitalab.tuwien.ac.at/projects/incontext/owl/smallcontext.owl#User8
-o test/portExtract.xml
test/sendMessage.xml extract_message

```

The first parameter is the address of the Session Service. Alter it according to your environment. The next two parameters define the environment the Sensor is executed in: The activity is set to `http://.../Activity#1595` and the user is set to `http://.../#User8`. The next parameter specifies that the output should be saved to the file `portExtract.xml`. The last two parameters belong together and specify the session file to load and the port to execute.

#### 4. Apply the changes and run the application.

If the port is successfully executed, something like this should be printed to the console:

```

Loaded file://E:\Diplomarbeit\Development\System\MessageSensor\src\sensormodel.xml
Extract: 'http://localhost:8080/axis2/services/MessageSensor' - Operation: 'extract_message'

```

The result is written to `portExtract.xml` and it should look like this:

```

<tns:sessiondata xmlns:tns="http://sensor.csdf.in_context.eu/tester"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://sensor.csdf.in_context.eu/tester sessiondata.xsd">
  <tns:data dataid="message.receiver">test@example.com</tns:data>
  <tns:data dataid="message.subject">Test message</tns:data>
  <tns:data dataid="message.body">The content of the message</tns:data>
  <tns:data dataid="message.type">EMail</tns:data>
</tns:sessiondata>

```

This file describes the session variables after the port was executed. As we can notice, it only contains the variables directly set by the output. The reason for this is that there is - for security reasons - no method in the Session Service to list all variables stored in the session. So it is just possible to load known variables. In our case, these are the output-variables.

This file is all we need to execute the update port in the next step.

### 8.6.3 Test of the Update Port

We can now test the update port of the Sensor, which would usually be invoked after the extraction port via a Forward.

1. Start the Tomcat Apache server if it is not already running.
2. Deploy the Session Service if it is not already deployed.
3. In Eclipse, click *Run / Run Configurations...* and add a new Java Application.

- ◇ Set *Name* to '*MessageSensor Test Update*'
- ◇ Set *Project* to '*MessageSensor*'
- ◇ Set *Main class* to '*eu.in-context.csd.f.sensor.Tester*'
- ◇ Switch to the tab *Arguments*
- ◇ Set *Program arguments* to the following value:

```
http://localhost:8080/axis2/services/SessionService
-a http://www.in-context.eu/activity/Activity#1595
-u http://www.vitalab.tuwien.ac.at/projects/incontext/owl/smallcontext.owl#User8
test/sendMessage.xml extract_message
test/portExtract.xml update_message
```

The first parameter is the address of the Session Service. Alter it according to your environment. The second and the third parameter specify the context. The next two parameters specify the session file and the port `extract_message` for the first invocation. After the execution is finished, the second session file will be loaded and the `update_message` port will be executed.

4. Apply the changes and run the application.

If the port is successfully executed, something like that should be printed to the console:

```
Loaded file://E:\Diplomarbeit\Development\System\MessageSensor\src\sensormodel.xml
Extract: 'http://localhost:8080/axis2/services/MessageSensor' - Operation: 'extract_message'
Update: 'http://localhost:8080/axis2/services/MessageSensor' - Operation: 'update_message'
```

As you can see, you can specify a series of invocations for a Sensor. The required session files can either be written manually or be generated from the output as we did before.

**NOTE** As the update port was executed, a context-update was performed. Be careful that the updates of the tests do not interfere with relevant data of the Context Management System.

Next we can also test the invocation of the Email Service. For this we first need to set the Parameter `standard.message.copy` to `true`. This is done via the option `-p`.



1. Change the previously defined run environment or create a new and enter the following program arguments:

```
http://localhost:8080/axis2/services/SessionService
-a http://www.in-context.eu/activity/Activity#1595
-u http://www.vitalab.tuwien.ac.at/projects/incontext/owl/smallcontext.owl#User8
-p "standard.message.copy=true|standard.message.copy_to=my@email.com"
test/sendMessage.xml extract_message
test/portExtract.xml update_message
```

2. Save the changes and run the application.

This should generate an exception in the `invoke()` method of the update port and the following should be printed to the console:

```
Loaded file://E:\Diplomarbeit\Development\System\MessageSensor\src\sensormodel.xml
Extract: 'http://localhost:8080/axis2/services/MessageSensor' - Operation: 'extract_message'
Update: 'http://localhost:8080/axis2/services/MessageSensor' - Operation: 'update_message'
Exception in thread "main" java.lang.Exception: Failed during Invocation
    at eu.in_context.csdf.sensor.Tester.invoke(Tester.java:417)
    at eu.in_context.csdf.sensor.Tester.<init>(Tester.java:221)
    at eu.in_context.csdf.sensor.Tester.main(Tester.java:143)
Caused by: eu.in_context.csdf.services.sensorcore.Invoke_DependentServiceFault:
[Update Port] EMail-Service failed - Axis Fault
...
Caused by: org.apache.axis2.AxisFault: Not Authorized
...
```

As we are not authorised an exception is returned upon invocation of the Email Service.

**NOTE** Although this result would of course not be satisfactory under normal circumstances, we will leave it like that. After all, this is just a tutorial explaining the possibilities of CSDF. In normal working environments, Sensors with such a fundamental flaw should never be deployed.

## 8.7 DEPLOYMENT

Finally, we will deploy and initialize the Sensor on the Controller.

### 8.7.1 Deploying the Sensor

The necessary build files for deployment are already included in the Sensor package, so deployment is quite easy.

If the web server directory is directly accessible, we can directly deploy the sensor:

1. Start Apache Tomcat, if not yet running.
2. Open `ant.properties` and set the correct deployment directory for Axis2.
3. Execute `deploy.bat`

In case the web server is on another server and cannot be accessed directly:

1. Start Apache Tomcat, if not yet running.
2. Execute `create-jar.bat`
3. Copy the generated `MessageSensor.aar` to the deployment directory of your Axis2 web application.

If deployment is successful, Apache Tomcat should generate the following message in the console:

```
[INFO] Deploying Web service: MessageSensor.aar
```

On the service-listing page of Axis2, usually found at `http://YOUR_HOST/axis2/services/listServices`, you will find the following new services:

- ◇ **MessageSensorIO** - the SensorIO Web service
- ◇ **MessageSensorControl** - the SensorControl Web service
- ◇ **MessageSensorService** - the SensorService Web service
- ◇ **MessageSensorManagement** - the SensorManagement Web service
- ◇ **MessageSensor** - the SensorCore Web service

### 8.7.2 Empty Initialization

Before the Sensor can be used in CSDF, it needs to be initialized. At first we will discuss the initialization without configuration.

1. Start Apache Tomcat, if not yet running.
2. Initialize the Controller (see A.4 Configuration).
3. Execute `sensor-init.bat`.

If the setup was done properly and the Sensor is deployed, the following message will be printed to the console:

```
[input] Press any key to continue...
```

The Sensor is now successfully registered at the Controller. For instance, a query via the `ListAllServices`-operation will return this Sensor amongst others.

If the script fails and the Sensor cannot be initialized, check the following:

- ◇ Is Tomcat Apache running and Axis2 properly configured with all required libraries?
- ◇ Are the Controller, the Session Service and the Sensor deployed?
- ◇ Is the Controller initialized?
- ◇ Does the `SensorModel` contain unspecified fields (e.g. addresses of services)? If so, it cannot be initialized. In such cases the `ConfigAssistant` has to be used.

Although the Sensor is registered at the Controller now, the Forward between the extraction and the update port is not yet set. To do that, we need a more sophisticated tool - the `ConfigAssistant`.

### 8.7.3 Overview of ConfigAssistant

A more flexible approach to initialize the Sensor is provided with the `ConfigAssistant`. It is a graphical tool used not only to initialize the Sensor, but also to configure the Forwards and reset service addresses.

1. Start Apache Tomcat if it is not already running
2. Initialize the Controller (see A.4 Configuration)
3. Execute `config-assistant.bat`. This will start the `ConfigAssistant` for this Sensor.

The `ConfigAssistant` will open in a new window (Figure 32).

#### Input Ports

In the top left corner you will see the input ports for this Sensor. In our case we have two ports which can be used for input:

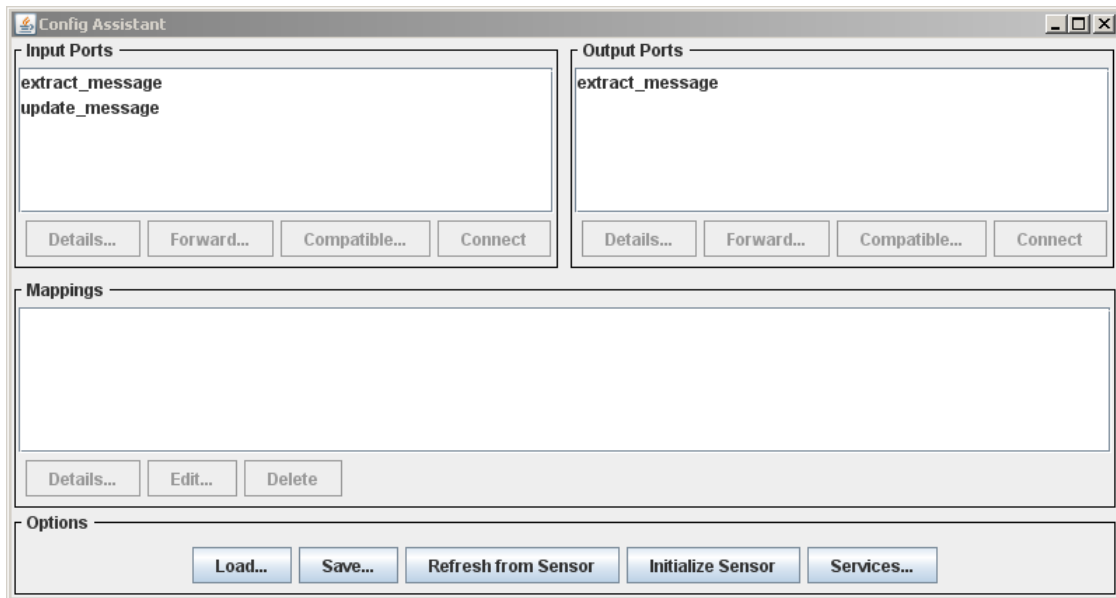


Figure 32: The ConfigAssistant

- ◇ `extract_message` - The PortExtract-port of the Sensor
- ◇ `update_message` - The PortUpdate-port of the Sensor

Here we can select input ports and define *Forward-Froms* on them:

#### Input Ports / Details...

This action will show the details of the selected input port. If you select `extract_message` and click details, a new window with the exact specification will open (Figure 33)

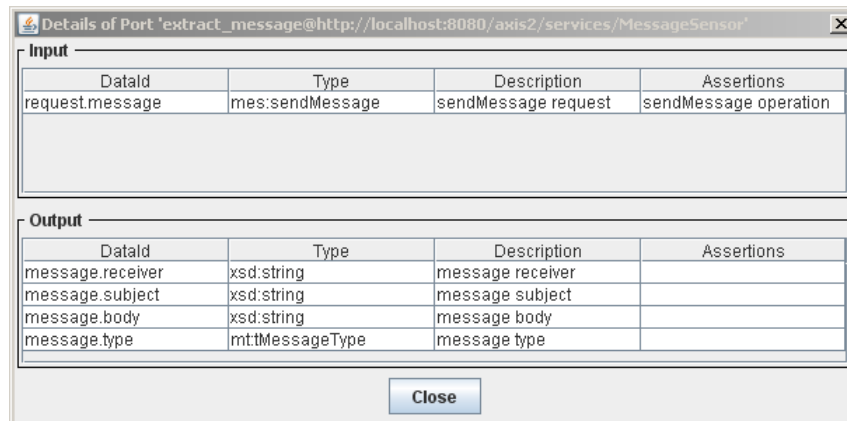
You can see the variable requirements of the input part of the port, as well as the asserted variables for the output part.

#### Input Ports / Forward...

This will let you define your own Forward-From via input of an address. The input format of the target address is `<port> '@' <service>` (Figure 34).

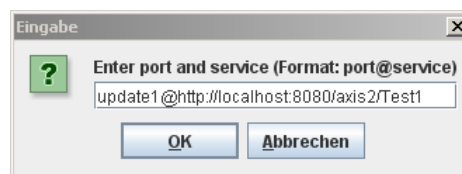
#### Input Ports / Compatible...

This is a very useful feature of the ConfigAssistant, as it allows the user to find compatible output ports of the selected input port (Figure 35).



Details of Port 'extract_message@http://localhost:8080/axis2/services/MessageSensor'			
<b>Input</b>			
DataId	Type	Description	Assertions
request.message	mes:sendMessage	sendMessage request	sendMessage operation
<b>Output</b>			
DataId	Type	Description	Assertions
message.receiver	xsd:string	message receiver	
message.subject	xsd:string	message subject	
message.body	xsd:string	message body	
message.type	mt:MessageType	message type	

Figure 33: ConfigAssistant - Port Details



**Eingabe**

Enter port and service (Format: port@service)

update1@http://localhost:8080/axis2/Test1

OK Abbrechen

Figure 34: ConfigAssistant - Enter Forward

Direct compatibility is given when the outputs of the found Sensor provide at least all required input variables for the input port. Inferred compatibility means that the Controller found compatibility through accumulating output variables in a chain of linked Sensors. For more information about compatibility, please refer to 5.4.2 Compatibility of Sensors.

**NOTE** The inferred compatibility is yet not realised at the current development of CSDF. It is a subject of future extensions.

It is then possible to view details of a compatible port or add a mapping to the port - a Forward-From. (To find compatible ports, the ConfigAssistant queries the Controller. In the beginning no Sensor is registered at the Controller, so both lists will be empty).

### Input Ports / Connect...

This operation creates a Forward-From using the selected port in *Output Ports* as source and the selected *Input Port* as destination.

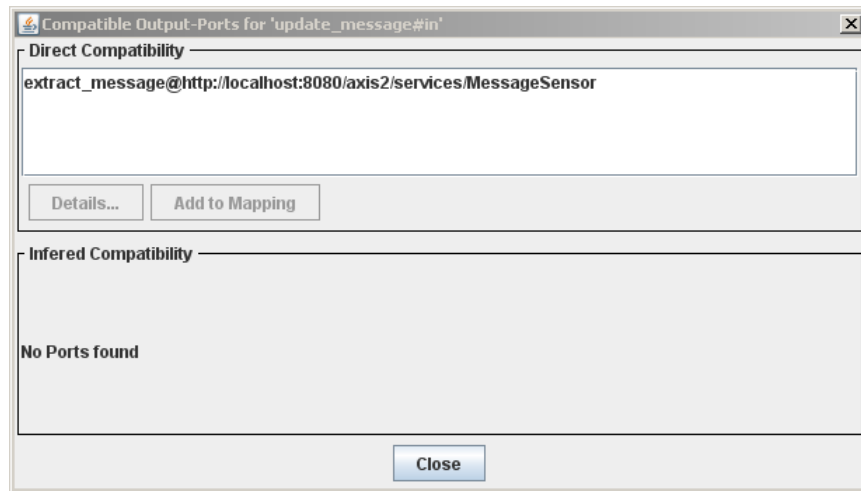


Figure 35: ConfigAssistant - Compatible Ports

### Output Ports

Here the output ports are listed. PortUpdate-ports do not have an output part so only the PortExtract-ports are shown here. Using the options here, it is possible to define *Forward-Tos*.

#### **Output Ports / Details...**

This shows the detailed specification of the selected output port.

#### **Output Ports / Forward...**

Analogously to the input side, this lets the developer define a Forward-To by entering the address and the port of the Sensor to which data should be forwarded.

#### **Output Ports / Compatible...**

Analogously to the input side, compatible input ports for the selected output port will be listed. The details can be viewed and Forward-To mappings can be created.

#### **Output Ports / Connect**

This automatically creates a Forward-To using the selected input port as source and the selected output-port as destination.

### Mappings

Here the current mappings of Forward-Froms and Forward-Tos are listed. For more information about Forwards, please refer to 5.4.4 Types of Links.

- ◇ **Forward-To** - The local port is listed on the left side and indicated by @self. Data is forwarded to the port on the right side.
- ◇ **Forward-From** - The local port is listed on the right side and indicated by @self. Data is inquired from the port on the left side.

NOTE Two different mappings might lead to the same result, but will still be listed separately here for this is the logic of the Forwards:

```
extract_message@self ==> update_message@http://localhost:8080/axis2/services/MessageSensor
```

...will produce the same result as...

```
extract_message@http://localhost:8080/axis2/services/MessageSensor ==> update_message@self
```

The first is a Forward-To which forwards data to another port of the same Sensor. The second is a Forward-From which inquires data from another port of the same Sensor. The Controller will of course detect such mappings and will only forward the data once. Yet the logic behind it is different, so the mappings are handled separately in the ConfigAssistant.

### **Mappings / Details...**

This will list the detailed specification of the referred external port.

### **Mappings / Edit...**

Here the address of the referred port can be changed.

### **Mappings / Delete**

This will delete the currently selected mapping.

### Options

In the bottom of the window you will find the general options.

### **Options / Load...**

Load a mapping configuration of the Sensor from a file.

### **Options / Save...**

Save the current mapping configuration of the Sensor to a file.

**Options / Refresh from Sensor**

Query the Sensor and load its current configuration.

**Options / Initialize Sensor**

Initialize the Sensor with the current mapping.

**Options / Services...**

Set or override the addresses and ports of services of the SensorModel. This option can be used in case `ServiceSensor` are included in the SensorModel.

**8.7.4 Initialization via the ConfigAssistant**

After this short overview, we will use the ConfigAssistant to create the mapping between the extraction and the update port.

1. Select 'update\_message' in *Input Ports*.
2. Select 'extract\_message' in *Output Ports*.
3. Create a Forward-From by clicking *Connect* in *Input Ports*.

A new entry will appear in mappings:

```
extract_message@http://localhost:8080/axis2/services/MessageSensor ==> update_message@self
```

This Forward-From connects the output of the extraction port with the input of the update port. The update port will now inquire data from the extraction port and therefore always be invoked after a successful invocation of the extraction port.

We could now use this mapping to initialize the Sensor. But instead, we will try another approach which will probably be used in CSDF environments with already many Sensors deployed:

1. Delete all mappings.
2. Click *Initialize Sensor*. The Sensor will now be initialized without any mapping. You should see a message box which says that the Sensor has been successfully initialized.
3. Now select the `update_message` of the *Input Ports* and click *Compatible....* In the new window, you should see the following entry in *Direct Compatibility*.



```
extract_message@http://localhost:8080/axis2/services/MessageSensor
```

If you look closely at the address of the Sensor, you will notice that this is the Sensor we are currently configuring. The Controller found that the selected update port is compatible to the `extract_message` port of the given - this - Sensor.

4. Create a mapping by clicking *Add to Mapping* and close the window. You will now see that a new mapping is listed in *Mappings*.
5. Now click *Initialize Sensor* to configure the Sensor.

As we can see, it is easy to check whether ports are really compatible by using the feature described above. If non-compatible ports are connected with Forwards, the Controller will automatically delete the mapping upon invocation. Although this will be reported in the console, the developer is likely to fail to notice such messages. Therefore configuration should be done with care, so there will be no need for the Controller to delete faulty mappings afterwards.

## 8.8 INTEGRATION TEST

The Sensor is already deployed, configured and registered at the Controller. Now it is time to do integration testing. CSDF supports integration tests for Sensors which are directly invoked by the Controller - so-called activated Sensors. For such cases a testing environment is directly included in the Controller. It enables the developer to locally emulate the Controller and simulate service interactions to test the system. For Sensors which are not directly linked to the Controller (and are not supposed to be set to active), no such environment exists. In such cases the developer might use any active Sensor linked to the one to be tested and perform the integration test with it instead. In the invocation chain the desired Sensor will be invoked eventually and can thus be tested too.

### 8.8.1 SOAP Request and Response

The first step to integration testing is creating the SOAP request and response messages for the simulated service interaction:

1. In `$CSDF/Development/ControllerService/serviceinteractions/` create a new file and name it `'sendmessage_request.xml'`.
2. Fill it with the following code:

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <ns1:user_id xmlns:ns1="incontext"
      soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
      soapenv:mustUnderstand="0">
      http://www.vitalab.tuwien.ac.at/projects/incontext/owl/smallcontext.owl#User8
    </ns1:user_id>
    <ns1:activity_id xmlns:ns1="incontext"
      soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
      soapenv:mustUnderstand="0">
      http://www.in-context.eu/activity/Activity#1595
    </ns1:activity_id>
  </soapenv:Header>
  <soapenv:Body>
    <ns1:sendMessage
      xmlns:ns1="http://service.emailservice.ns.www.in_context.eu/xsd">
      <ns1:to>test@example.com</ns1:to>
      <ns1:subject>Test message</ns1:subject>
      <ns1:body>The content of the message</ns1:body>
    </ns1:sendMessage>
  </soapenv:Body>
</soapenv:Envelope>
```

Normally you would get the code of this service interaction by actually invoking the service, for example via *soapUI* <sup>13</sup>.

3. In `$CSDF/Development/ControllerService/serviceinteractions/` create a new file and name it `'sendmessage_response.xml'`.
4. Fill it with the following code:

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header />
  <soapenv:Body>
    <ns:sendMessageResponse
      xmlns:ns="http://service.emailservice.ns.www.in_context.eu/xsd">
      <ns:return>true</ns:return>
    </ns:sendMessageResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

We have now created the request and the response which will be used to simulate a service interaction at the Controller.

<sup>13</sup><http://www.soapui.org/> (last access: 2009-04-19)

### 8.8.2 Start the integration test

Now we will feed the request and the response to the test environment of the Controller, which will process it like a real service interaction received by the logging service. We will thus be able to check two things:

- ◇ Firstly, the test will immediately show whether the port configuration was done properly. If the input specification is already slightly different from the actual SOAP request, the Sensor will not be invoked by the Controller.
- ◇ Secondly, we will be able to see whether the Forward configuration is working properly:
  - The Controller will produce errors for non existing references or circular references.
  - The Controller will show if invocation failed because of incompatible ports.
  - We can deduce from the console output which port was actually invoked and which was not. Maybe some important mappings were forgotten during configuration.

NOTE It must be clear that although the Controller is emulated offline, the Sensors are not. This means that any service interaction, which leads to an invocation of the Sensor, might result in unwanted context-updates. The developer has to keep that in mind when doing integration tests.

1. Make sure that Tomcat Apache is running and all services including the Sensors are deployed. The Controller must be initialized and the Sensor must be configured and initialized with the actual mapping. The Sensor should not be marked active.
2. In Eclipse, click *Run / Run Configurations...* and add a new Java Application.
  - ◇ Set *Name* to '*I-Test MessageSensor*'
  - ◇ Set *Project* to '*Controller*'
  - ◇ Set *Main class* to '*eu.in\_context.csdf.controller.TesterController*'
  - ◇ Switch to the tab *Arguments*
  - ◇ Set *Program arguments* to the following value:

```
-a http://localhost:8080/axis2/services/MessageSensor
-aonline
-ronline
serviceinteractions/sendmessage_request.xml
serviceinteractions/sendmessage_response.xml
```

With `-ronline` all Sensors registered at the real Controller are imported. The option `-aonline` then also sets all Sensors to active which are marked active on the real Controller. Additionally, the first parameter sets the *MessageSensor* as active. This is the service we want to test. It is not marked active on the real Controller, so we have to activate it in the test environment. The last two parameters specify the SOAP requests and responses which should be wrapped in service interactions and sent to the Controller. Of course it would be possible to specify not only 2, but also 4, 6, ... documents, which will always be processed pairwise.

NOTE Not only the Sensors but also their current Forward configuration is automatically loaded. So with both options `-ronline` and `-aonline`, the emulated Controller is exactly configured as the real Controller.

3. Apply the changes and run the application.

If the test is successfully executed, the following output should be generated

```
Received Request
Received Response
=> 'extract_message@http://localhost:8080/axis2/services/MessageSensor' matched requirements
+ extract_message@http://localhost:8080/axis2/services/MessageSensor
+ update_message@http://localhost:8080/axis2/services/MessageSensor
Finished
```

The lines with + indicate that the given port has been executed. If not indented, the Sensor was directly invoked by the Controller. If the line is indented compared to the previous one, it expresses that the port is executed through a Forward as in the case of `update_message`.

Of course we can now also change the Parameters of the Sensor and execute the Test again. To change the Parameters, tools like *soapUI* might be used.

All in all, we can see that the Sensors reacted to the service interaction and that the Forward was properly configured. With this the integration test is over and the Sensor can be activated.

### 8.8.3 Final Activation

The integration of a new Sensor is usually completed through its activation. Of course this does not apply to Sensors which are not supposed to be directly invoked by the Controller but are instead executed through Forwards.

We will now activate the Sensor:

1. Execute `sensor-activate.bat`.

The following message will be printed to the console:

```
[input] Press any key to continue...
```

The Sensor is now activated and will be invoked from the Controller if suitable service interactions are received.

If the script fails and the Sensor cannot be activated, check the following:

- ◇ Is Tomcat Apache running and has Axis2 been properly configured with all required libraries?
- ◇ Are the Controller, the Session Service and the Sensor deployed?
- ◇ Has the Controller been initialized?
- ◇ Has the Sensor been properly initialized?

---

## 9 EVALUATION

This section looks at three situations common in the setting of CWE and analyses how a context-aware application could support the user in his work: The first and the second use case deal with context-extraction from two separate service interactions, the third one, on the contrary, directly processes data extracted from a single service invocation. In a next step, a possible solution to the given problem and its implementation in CSDF is presented.

## PREFACE

This part introduces use cases which are likely to occur in the context of CWE. The WSDL for all services used in this section can be found at [\\$CSDF/WSDL](#). The reader is advised to study them for a deeper understanding of the scenarios presented.

## 9.1 USE CASE: MAILINGLIST

### 9.1.1 Description

At a regular meeting, Tom gets elected to be the coordinator of the new subproject A1 of project A. After getting back to his workplace, Tom immediately sets up a new workspace for project A1. He does that by creating a new activity X which should contain all information, tasks and events regarding project A1. As a next step, he sends out emails to all members of the project, in this case Lisa, Bob and Alice, to inform them about the new workspace he created.

A CWE supports Tom in his work. As it is very likely that notification-emails like this one are sent again in the course of the project, the CWE automatically creates a mailing-list for project A1 containing all its members (Tom, Lisa, Bob and Alice). The mailing list is created via the Mailinglist Service and finally added as a resource to the new activity X (Figure 36).

### 9.1.2 Sensor Design

To solve this scenario, we could create two small Sensors sensitive to the event create-activity and the event send-email respectively. Then, we could create a third Sensor that combines them and performs the context-update. Yet, for demonstration purposes we choose a simpler approach: We will create one Sensor with two ports, one for create-activity events and the other for send-email events and the context-update. We therefore need:

- ◇ *One Sensor with 2 Ports* - one extraction port (create-activity) and one update port (send-email)
- ◇ *Session Framing* - 5 minute session-frame between create-activity and send-email-event

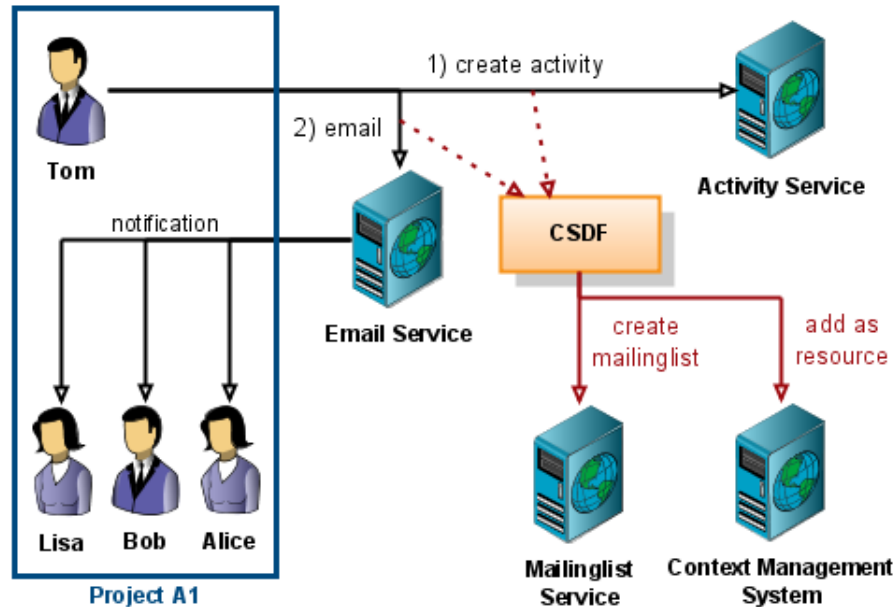


Figure 36: Use Case: Mailinglist

- ◇ *User-awareness* - both actions are performed by the same user
- ◇ *No activity-awareness* - the actions do not take place in a common activity-scope

The SensorModel now looks like this:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <sm:SensorModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xmlns:sm="http://sensormodel.csd.in_context.eu">
5    <iospecification isuseraware="true" isactivityaware="false" sessiontime="300">
6      <ports xsi:type="sm:PortExtract" portid="addactivity">
7        <input>
8          <specs dataid="request.message" datatype="act:addActivities" description="...">
9            <assertions xsi:type="sm:AssertionWSOperation" description="..."
10              operation="addActivities" request="true"/>
11          </specs>
12          <specs dataid="response.message" datatype="act:addActivitiesResponse" description="...">
13            <assertions xsi:type="sm:AssertionWSOperation" description="..."
14              operation="addActivities" request="false"/>
15          </specs>
16        </input>
17        <output>
18          <includes nsprefix="self" ioid="activity" />
19        </output>

```



```

20 </ports>
21 <ports xsi:type="sm:PortUpdate" portid="sendmail">
22   <input>
23     <specs dataid="request.message" datatype="ema:sendMessage" description="...">
24       <assertions xsi:type="sm:AssertionWSOperation" description="..."
25         operation="sendMessage" request="true"/>
26     </specs>
27   </input>
28 </ports>
29 <defs ioid="activity" readable="true" writeable="true">
30   <specs dataid="activity.url" datatype="xsd:string" description="activity url" />
31   <specs dataid="activity.statusnew" datatype="xsd:boolean"
32     description="flag variable, only set if activity was created" />
33 </defs>
34 </iospecification>
35 <controlspecification>
36   <standards xsi:type="sm:StandardStatus"/>
37   <access xsi:type="sm:ControlAccessDefault"/>
38 </controlspecification>
39 <servicespecification controllerservice="http://localhost:8080/axis2/services/ControllerService">
40   <services xsi:type="sm:ServiceWS" serviceid="mailinglistservice"
41     description="Mailinglist-Service"
42     wsdl="http://localhost:8080/axis2/services/MailingList?wsdl" />
43   <services xsi:type="sm:ServiceWS" serviceid="activityservice"
44     description="Activity-Service"
45     wsdl="http://localhost:8080/axis2/services/activityservice?wsdl" />
46 </servicespecification>
47 <sensorspecification name="TestScenario1" description="mailinglist usecase"
48   serviceurl="http://localhost:8080/axis2/services/TestScenario1" author="florian">
49   <resources xsi:type="sm:ResourceWSDL" resourceid="activityservice"
50     location="http://localhost:8080/axis2/services/activityservice?wsdl"
51     local="false" namespace="http://www.in-context.eu/activityservice/"
52     prefix="act" convertSchemaElementToSchemaType="true" />
53   <resources xsi:type="sm:ResourceWSDL" resourceid="emailservice"
54     location="http://localhost:8080/axis2/services/EmailService?wsdl"
55     local="false" namespace="http://service.emailservice.ns.www.in_context.eu/xsd"
56     prefix="ema" convertSchemaElementToSchemaType="true" />
57 </sensorspecification>
58 </sm:SensorModel>

```

**Line 5:** This is the definition of a session-framed user-aware Sensor.

**Line 8-15:** This is the input-port of the create-activity event. It defines variables for the SOAP request and response. Both variables already use the types of the Active Service WSDL via prefix `act`.

**Line 18:** This line defines the output of the create-activity port. It loads the variable definition from line 29-33.

**Line 23-26:** Here is the input-port of the send-email event. It defines a variable for the SOAP request already using the proper WSDL type via prefix `ema`.

**Line 29-33:** Here is the definition of two variables for exchanging activity data. It is used in the output of the extraction port and will also be loaded in the update port.

**Line 40-45:** As we want to use the Mailinglist Service (to create a mailinglist) and the Activity Service (to add the mailinglist as a resource), we add them as integrated services.

**Line 49-56:** Here we include both the Activity Service and the Email Service as resources to use their WSDL types in the input and output ports.

### 9.1.3 Sensor Logic

After generating the Sensor code-base from the SensorModel given above, we can start to implement the code for both ports:

**ExtractAddactivity :** In this port we just need to set the properties of the activity in the output variable.

**UpdateSendmail :** First we load the activity-data from the session. Then we create a new mailinglist using the service-stubs from the Mailinglist Service. Finally, we add the new mailinglist to the activity as a resource using the service-stubs of the Activity Service.

In the next step we deploy the Sensor and initialize it. The Sensor does not need to be configured. Both ports will directly be invoked by the Controller, so we do not need to set any Forwards. The final step is to activate the Sensor, by means of which the scenario is solved.

Files for this scenario can be found at `$CSDF/Examples/Scenario1`.

## 9.2 USE CASE: ROOM RESERVATION

### 9.2.1 Description

**NOTE** This is the use case that was presented earlier in the Problem Statement (see 4.2 The Room Reservation Use Case). For the sake of convenience, the use case description is included here again.

Bob (a worker of company ABC) wants to hold a meeting to discuss and plan the development of project A. As a first step he uses the Room Reservation Service of company CDE to reserve a meeting room on Friday morning. Upon confirmation, he sends out emails using the Email Service to Alice and Tom of Project A to inform them about the time and place of the upcoming meeting. He also invites Lisa, a member of Project B. She is needed to help in the design of a common interface used in both projects.

A sensor-enabled CWE system supports Bob in his work. Apart from Alice and Tom (who are already members of the project), it automatically adds Lisa as an involved-actor to the activity-context of Project A (Figure 37).

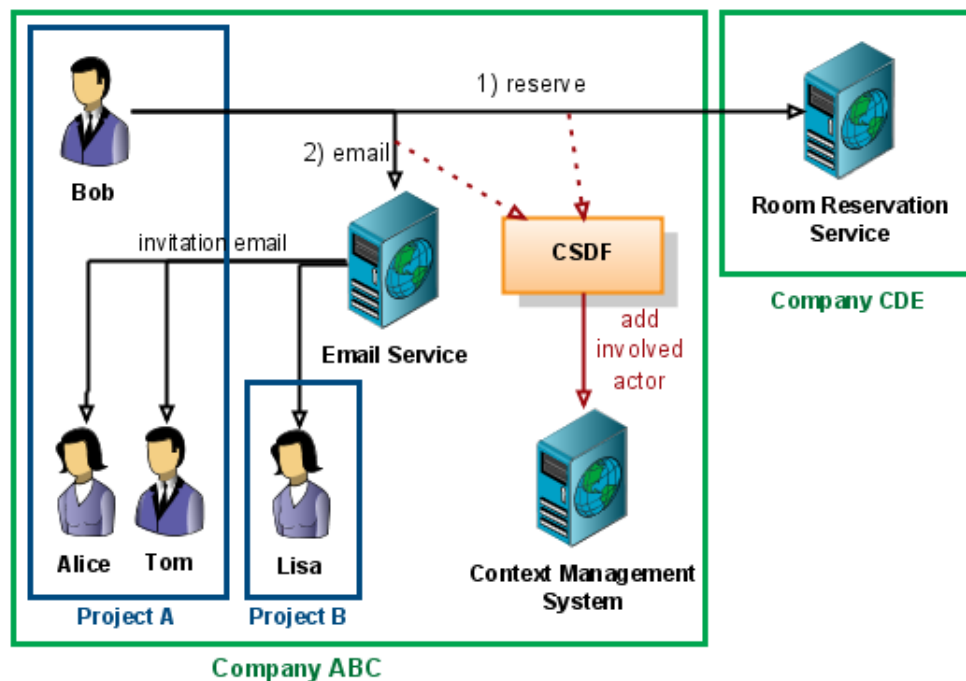


Figure 37: Use Case: Room Reservation

### 9.2.2 Sensor Design

The design of this Sensor is similar to the previous one. We will create a Sensor with two ports, the first one sensitive to room-reservations and the second one to react to send-email events. Thus we need:

- ◇ *One Sensor with 2 Ports* - one extraction port (room-reservation) and one update port (send-email)
- ◇ *Session Framing* - 3 minute session-frame between room-reservation and email-event
- ◇ *User-awareness* - both actions are performed by the same user
- ◇ *Activity-awareness* - both actions take place in the same activity-context

The SensorModel looks like this:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <sm:SensorModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xmlns:sm="http://sensormodel.csd.f.in_context.eu">
5    <iospecification isuseraware="true" isactivityaware="true" sessiontime="180">
6      <ports xsi:type="sm:PortExtract" portid="reserveroom">
7        <input>
8          <specs dataid="request.message" datatype="met:ReserveRoom" description="...">
9            <assertions xsi:type="sm:AssertionWSOperation" description="..."
10              operation="ReserveRoom" request="true"/>
11          </specs>
12          <specs dataid="response.message" datatype="met:ReserveRoomResponse" description="...">
13            <assertions xsi:type="sm:AssertionWSOperation" description="..."
14              operation="ReserveRoom" request="false"/>
15          </specs>
16        </input>
17        <output>
18          <includes nsprefix="self" ioid="meetingroom" />
19        </output>
20      </ports>
21      <ports xsi:type="sm:PortUpdate" portid="sendmail">
22        <input>
23          <specs dataid="request.message" datatype="ema:sendMessage" description="send mail request">
24            <assertions xsi:type="sm:AssertionWSOperation" description="send message"
25              operation="sendMessage" request="true"/>
26          </specs>
27        </input>
28      </ports>
29      <defs ioid="meetingroom" readable="true" writeable="true">
30        <specs dataid="meetingroom.id" datatype="xsd:string" description="room id" />
31        <specs dataid="meetingroom.statusreservation" datatype="xsd:boolean"
32          description="flag variable, only set if rooms is reserved" />
33      </defs>
34    </iospecification>
35    <controls specification>
36      <standards xsi:type="sm:StandardStatus"/>
37      <access xsi:type="sm:ControlAccessDefault"/>
38    </controls specification>
39    <services specification controllerservice="http://localhost:8080/axis2/services/ControllerService">
40      <services xsi:type="sm:ServiceWS" serviceid="activityservice" description="Activity-Service"
41        wsdl="http://localhost:8080/axis2/services/activityservice?wsdl" />
42    </services specification>

```

```

43 <sensorspecification name="TestScenario2" description="..."
44   serviceurl="http://localhost:8080/axis2/services/TestScenario2" author="florian">
45   <resources xsi:type="sm:ResourceWSDL" resourceid="roomservice"
46     location="http://localhost:8080/axis2/services/MeetingRoom?wsdl" local="false"
47     namespace="http://meetingroom.www.in_context.eu" prefix="met"
48     convertSchemaElementToSchemaType="true" />
49   <resources xsi:type="sm:ResourceWSDL" resourceid="emailservice"
50     location="http://localhost:8080/axis2/services/EmailService?wsdl"
51     local="false" namespace="http://service.emailservice.ns.www.in_context.eu/xsd"
52     prefix="ema" convertSchemaElementToSchemaType="true" />
53 </sensorspecification>
54 </sm:SensorModel>

```

**Line 5:** This is the definition of a session-framed user- and activity-aware Sensor.

**Line 8-15:** This is the input-port of the room-reservation event. It defines variables for the SOAP request and response using the appropriate WSDL-types via prefix `met`.

**Line 18:** This line defines the output of the room-reservation port. It loads the variable-definition from line 29-33.

**Line 23-26:** Here is the input-port of the send-email event. It defines a variable for the SOAP request using the proper WSDL-type via prefix `ema`.

**Line 29-33:** This is the definition of two variables for exchanging room-reservation data. We use it in the extraction port and will also load it in the logic of the update port.

**Line 40-41:** We add the Activity Service as integrated service because we will use it to add involved actors to the activity.

**Line 45-52:** In this part we include the Room Reservation Service and the Email Service as resources in order to use their types in the input and output ports of the Sensor.

### 9.2.3 Sensor Logic

In a next step, we generate the code-base from the SensorModel. After generation has finished, we start to code the extensions of the Sensor:

**ExtractReserveroom :** We just copy the reservation-data to the output variable.

**UpdateSendmail** : First, we load the reservation-data from the Session. Then we use the Activity Service to add the recipients of the email as involved actors to the current activity-context.

After coding the extension logic, we can deploy and initialize the Sensor. The Sensor does not need any extra configuration, so we can directly activate it at the Controller and thus successfully solve the scenario.

Files for this scenario can be found at `$CSDF/Examples/Scenario2`.

## 9.3 USE CASE: LOAD DOCUMENT

### 9.3.1 Description

Alice is a worker of company ABC and involved in project A. The goal of project A is to develop the new product X. Alice now needs a few technical specifications from company CDE, which delivers some of the parts of product X. For this purpose, she uses the Document Service. This web-interface is provided by CDE for its customers to enable them to load technical specifications of all their products.

A context-aware CWE system supports Alice in this task: Whenever a document is loaded using the Document Service, the CWE automatically adds it as a resource to the activity-context of Project A (Figure 38).

### 9.3.2 Sensor Design

This is a rather simple use case that can easily be solved with one Sensor. Again, we will use two ports, one to react to load-document events and the other to perform the desired context-update.

- ◇ *One Sensor with 2 Ports* - one extraction port (load-document) and one update port (add-resource)
- ◇ *No Session-framing* - this Sensor only supports single interactions

The SensorModel looks like this:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <sm:SensorModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

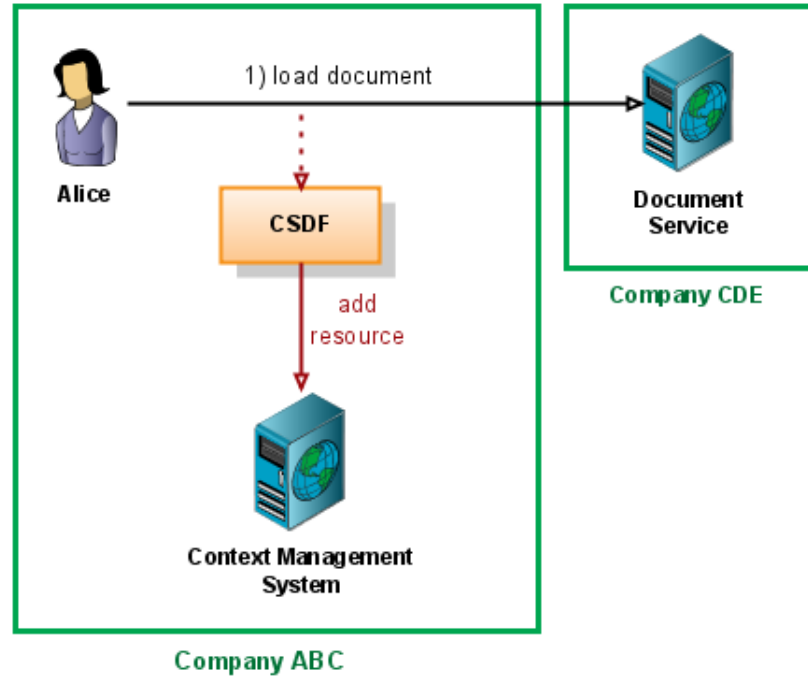


Figure 38: Use Case: Load Document

```

4   xmlns:sm="http://sensormodel.csd.f.in_context.eu">
5   <iospecification isuseraware="true" isactivityaware="true" sessiontime="0">
6     <ports xsi:type="sm:PortExtract" portid="loaddocument">
7       <input>
8         <specs dataid="request.message" datatype="doc:LoadDocument" description="...">
9           <assertions xsi:type="sm:AssertionWSOperation" description="..."
10             operation="LoadDocument" request="true"/>
11         </specs>
12         <specs dataid="response.message" datatype="doc:LoadDocumentResponse" description="...">
13           <assertions xsi:type="sm:AssertionWSOperation" description="..."
14             operation="LoadDocument" request="false"/>
15         </specs>
16       </input>
17       <output>
18         <specs dataid="document.url" datatype="xsd:string" description="room id" />
19         <specs dataid="document.statusload" datatype="xsd:boolean"
20           description="flag variable, only set if document is loaded" />
21       </output>
22     </ports>
23     <ports xsi:type="sm:PortUpdate" portid="adddocumentasresource">
24       <input>
25         <includes nsprefix="self" ioid="loaddocument#out" />
26       </input>
27     </ports>
28   </iospecification>
29   <controls specification>

```

```

30     <standards xsi:type="sm:StandardStatus"/>
31     <access xsi:type="sm:ControlAccessDefault"/>
32 </controls specification>
33 <services specification controller service="http://localhost:8080/axis2/services/ControllerService">
34     <services xsi:type="sm:ServiceWS" serviceid="activityservice" description="Activity-Service"
35         wsdl="http://localhost:8080/axis2/services/activityservice?wsdl" />
36 </services specification>
37 <sensors specification name="TestScenario3" description="..."
38     serviceurl="http://localhost:8080/axis2/services/TestScenario3" author="florian">
39     <resources xsi:type="sm:ResourceWSDL" resourceid="documentservice"
40         location="http://localhost:8080/axis2/services/DocumentManagement?wsdl" local="false"
41         namespace="http://documentmanagement.www.in_context.eu" prefix="doc"
42         convertSchemaElementToSchemaType="true" />
43 </sensors specification>
44 </sm:SensorModel>

```

**Line 5:** This is the definition a single-interaction Sensor.

**Line 8-15:** Here is the input port of the load-document event defining variables for both SOAP request and response. The variables use WSDL-types via the prefix doc.

**Line 18-20:** The output of extraction port delivers information about the document being loaded.

**Line 25:** The variables of the input of the update port are imported from the output of the extraction port.

**Line 34-35:** We add the document as a resource, therefore we must add the Activity Service as integrated service.

**Line 39-42:** We used WSDL-types in the port definition and thus have to include the Document Service as a resource.

### 9.3.3 Sensor Logic

After generating the code-base of the Sensor, we can code the business logic of the extensions:

**ExtractLoaddocument :** This port just saves the data from the document to the output variable.

**UpdateAdddocumentasresource :** Here, we use the data from the input variable and add a new resource to the activity-context using the Activity Service client stubs.



After coding the extension logic, we can deploy the Sensor. The initialization is done using the ConfigAssistant, which we use to define the following mapping:

```
loaddocument@self ==> adddocumentasresource@http://localhost:8080/axis2/services/TestScenario3
```

This Forward-To connects the output of the extraction port to the input of the update port. At last, we can activate the Sensor at the Controller and thus successfully solve the scenario.

Files for this scenario can be found at `$CSDF/Examples/Scenario3`.

---

## 10 CONCLUSION AND OUTLOOK

This final chapter summarises the main aspects of this thesis and suggests ideas for possible future research. The first part provides a concluding overview of the primary concepts of CSDF. Part two focuses on the contributions of this thesis for scientific research in that particular field of study. The last section analyses shortcomings in the current implementation of CSDF and proposes extensions for possible future releases.

## 10.1 CONCLUSION

This section provides a summary of the results of this thesis:

- I. *Problem solved*: The problems described in chapter 4 have been solved. CSDF, based on the datamining-approach described in section 5.1, makes it feasible to context-sense Web services in SOA-based environments, where context sensors cannot directly be attached to the Web service of interest.
- II. *Fully automated Generation*: The framework provides a computer-supported approach to develop Sensors for web environments. Using the Generator presented in section 5.1.5, a fully functional code-base for Sensors can be generated. The only task that remains to be done is to code the actual business logic of the Sensors and deploy it.
- III. *Model-driven*: The creation of new Sensors is model-driven. The SensorModel, described in chapter 6, is used to define Sensors on an abstract level. With this specification being language-independent, Sensors can be implemented in any programming language. However, the current version of CSDF only provides a code generation mechanism for the Java language.
- IV. *Sensors as Web Services*: The entire communication between Controller, Session Service and Sensors relies on the loosely coupled SOA-architecture. As described in chapter 7, Sensors implement a comprehensive Web service interface. Thus, they can be developed and deployed on any server using any programming language.
- V. *Dynamic Sensor Management*: The Controller as the central part of CSDF manages Sensor registration. As illustrated in section 5.1.2, Sensors can be added and removed dynamically during runtime. In addition, as also shown in 5.1.2, the Controller implements an algorithm to automatically remove a Sensor in case that it cannot be reached for a certain period of time.
- VI. *Flexible Sensor Composition*: Sensors are typically designed in a way to focus on one simple task only. Even so, complex functionality can be achieved by using Sensor composition as described in 5.4. Similar to the Sentient Object Model presented in 3.2, Sensors can be both producers and consumers of data. Linkage information, as shown in section 5.4.4, is defined directly on the Sensor itself. Thus, seamless integration, i.e. integration without the need to alter the configuration of existing Sensors, becomes feasible.

- VII. *Composition Support*: As illustrated in section 5.4.2 and 5.4.3, the Controller offers additional mechanisms to support Sensor composition. First, it automatically detects loops in existing Sensor networks. Second, it provides an interface to query all Sensors compatible with a given specification.
- VIII. *Extraction from several Interactions*: CSDF also enables context extraction from a series of Web service invocations. As presented in 5.3.4, special attributes in the SensorModel control how incoming service interactions are grouped in sessions for context extraction.

## 10.2 SUMMARY OF CONTRIBUTIONS

This part describes the contributions for scientific research made in the course of this thesis:

- I. *Context Sensors in SOA-based Environments*: As outlined in chapter 3, many context-aware applications have been developed in recent years. Although considerable effort has gone into research in the area of perceiving aspects of the physical world, technologies on software-sensors are a rather new branch of study. As analysed in chapter 4, one fundamental problem in web based context-aware applications is the integration of the sensor: Third party Web services cannot be modified and thus not directly extended with a sensing unit. In this thesis, we proposed an approach that mitigates this shortcoming. Upon the assumption that at minimum the service request and response are observable, we presented a framework for the creation of sensors capable of analysing and extracting context solely from messages exchanged in a service invocation.
- II. *Flexible Sensor Composition*: Context-aware applications use sensor-devices to become aware of aspects of the physical or virtual world. As presented in an example in chapter 4, 'context' often comprises several such facts. Many context-sensitive applications introduce a reasoning layer to overcome this problem. However, if complex aspects could already be detected on sensory level, the overlying logic would be drastically reduced in complexity. Our approach introduced an architecture which allows for flexible sensor composition: Sensors can be reused, similar to the Sentient Object Model described in 3, and complex context-extraction networks can be built by combining many small sensing units.

III. *Computer-supported Development*: Though software-sensors measure a wide range of possible aspects of their environment, they typically share a common infrastructure; for instance, mechanisms to publish data, a description for integration into a service registry and an interface to query common and specific parameters of the sensor. As these parts and others are practically identical, regardless of the actual logic of the sensor, they can automatically be generated by a framework rather than being coded by the developer. This tremendously reduces the development effort of sensors. In this thesis, we proposed a model-driven architecture which enables automated sensor generation via the SensorModel, the language-independent model description of a Sensor. Although currently only implemented in Java, any kind of sensor base can theoretically be generated from a given SensorModel. Especially in the setting of CWE, which involves multiple companies and different technologies, this proves to be an important requirement.

## 10.3 OUTLOOK

With this being the first release of CSDF, there is still room for improvement. This section gives a short overview of aspects of CSDF that have not been implemented yet or parts that could be enhanced:

**Inferred Compatibility:** A feature of CSDF that has not yet been implemented is the detection of inferred-compatible ports, as described in section 5.4.2. All necessary information already resides in the Controller and an adequate Web service is implemented too, thus only the implementation of the actual algorithm remains to be done. An algorithm solving the given problem could look as follows:

1. Initially, the accumulated context is empty. For all active-nodes: Execute 2-4.
2. Add the output of the current node to the accumulated context.
3. If the current node is the target node, save the node-path.
4. Identify all Sensors compatible with the current accumulated context. For every compatible node: Execute 2-4.

All paths leading to the target node indicate compositions with inferred compatibility. Of course some kind of loop detection is needed, otherwise the

execution of the algorithm might not come to an end. With regard to processing, this algorithm is rather costly if executed on request-basis. An alternative would be to identify all inferred compatibility paths during start-up and save the results in a list. Naturally, the algorithm must then be executed again every time the Sensor composition changes.

**Dynamic Service Composition:** Based on the inferred compatibility detection, dynamic service composition can be implemented. Thus, the developer just specifies the given input and the required output, upon which the Controller will try to automatically find a suitable combination of Sensors that would satisfy both requirements. This would be useful for CSDF with a high number of primitive Sensors deployed. The developer could just implement the context-update and leave the routing of an adequate context extracting composition in the hands of the Controller.

**Visualisation of Composition:** Although the current version of CSDF allows for flexible service compositions, it does not provide any tools to monitor the linkage of Sensors. With the number of registered Sensors at the Controller tending to become considerably high, a tool visualising the actual configuration would be very helpful. The tool could load all Sensor specifications and Forward definitions of the Controller via its Web service and then use this information to generate a graph displaying the actual composition of Sensors. Especially in cases of faulty linkage, this tool would prove highly beneficial to quickly locate errors in the Forward configurations.

**Stress Test of Controller:** The bottleneck of CSDF is the Controller. For this reason, a stress test would be very valuable as it would provide information about the number of Sensors the Controller can effectively work with and the number of simultaneous service interactions it can cope with.

**Typed Session Service:** In the current design of CSDF, the Session Service is untyped. Therefore, data must be parsed and cast to the adequate types upon execution of a port of the Sensor. A future extension of CSDF could introduce a typed version of the Session Service or even replace it by another technology, for instance, a shared data store or a tuple space.

**Extension of Parameter Types:** The definition of Parameters is currently limited to simple types of the type system. In addition, no databinding has yet been implemented for Parameters.

**Quality-of-Service Attributes:** The designs of the SensorModel and the Session Service already include a concept of QoS attributes. However, the current implementation of the databinding algorithm does not yet support it. QoS parameters might prove very useful to annotate extracted context with additional information, therefore the databinding mechanism should be extended in a way to be able to handle such data.

**Programming Language Support:** Given a particular SensorModel, the Generator of CSDF is used to generate the code-base of a Sensor. In the current version, only Java is supported as target language. Future extensions of CSDF might also support generation of other programming languages, e.g. C#.

**Analysing Complex Flows of Interactions:** CSDF was developed to ease the development of Sensors analysing the message flow between Web services. It already provides some mechanisms to extract context from a series of interactions joined together by either a user, an activity and/or by the time of invocation. This might prove to be sufficient in most cases, yet sometimes context is coded in a more complicated series of interactions (e.g. workflow). To identify such complex patterns, a sophisticated description mechanism beyond the capabilities of the SensorModel is required. Basic patterns of such workflows are analysed in inContext D1.2 [\[41\]](#).

---

# A INSTALLATION GUIDE

This chapter sees a step-by-step installation of CSDF. Following an overview of the system requirements, the manual provides instructions on how to install additional and required software components. Part three then covers the set-up of CSDF for both the development machine as well as the web server used for deployment. The final part focuses on the deployment and initialization of CSDF and guides through the necessary configuration of Eclipse.



## PREFACE

This part contains a step-by-step installation guide for CSDF. To simplify matters, basic terminology will be introduced at this stage. The following terms should be kept in mind when reading this section:

<b>Term</b>	<b>Explanation</b>
development machine	The machine at which new Sensors for CSDF will be developed.
web server	The machine deploying the Controller, the Session Service and the Sensors of CSDF.
<code>\$JAVA_HOME</code>	The base directory of the Java JDK installation. Example: <code>c:/java/jdk1.6.0.03</code>
<code>\$APACHE_HOME</code>	The base directory of the Apache Tomcat installation. Example: <code>c:/apache-tomcat-6.0.14</code>
<code>\$APACHE_WEB</code>	The directory with the web applications of Apache Tomcat. Example: <code>\$JAVA_HOME/webapps</code>
<code>\$AXIS2_HOME</code>	The base directory of the Axis2 installation. Example: <code>c:/axis2-1.3</code>
<code>\$AXIS2_WEB</code>	The directory where Axis2 is deployed as web application on Apache Tomcat. Example: <code>\$APACHE_WEB/axis2</code>
<code>\$ANT_HOME</code>	The base directory of the Ant installation. Example: <code>c:/ant-1.7.1</code>
<code>\$EMF_LIB</code>	The location of the .jar-libraries of the EMF installation. Example: <code>c:/emf-libs</code>
<code>\$JET_LIB</code>	The location of the .jar-libraries of the JET installation. Example: <code>c:/jet-libs</code>
<code>\$ECLIPSE_HOME</code>	The base directory of the Eclipse installation. Example: <code>c:/eclipse</code>
<code>\$CSDF</code>	The base directory of the CSDF compilation.
<code>\$DEV</code>	The abbreviation for <code>\$CSDF/Development</code> .

Table 91: Common Terms

## A.1 SYSTEM REQUIREMENTS

CSDF was both developed and tested on a desktop computer. Although it works fine even on normal computers, it is recommended to use a more powerful machine for deployment. There is no stress test for CSDF available, so it is not possible to say how many resources it actually needs and how well it performs and scales. For testing purposes it is sufficient to install it on an average desktop machine, but if many service interactions are expected, a real web server would probably be the better choice.

As a Windows platform was chosen to develop CSDF, this whole guide deals with the installation on a Windows platform. Installation on other platforms might slightly differ. The software has only been tested on Windows platforms so far, but there are no platform specific components, so it should theoretically work on any platform. This will be a matter of tests in future.

## A.2 PREREQUISITES

There are some prerequisites before beginning the setup of CSDF.

### A.2.1 Java JDK

CSDF has been developed in Java, thus a Java JDK has to be installed on the development machine. Furthermore an environment variable needs be set: The variable must have the name 'JAVA\_HOME' with the value of `$JAVA_HOME`.

**Version:** 1.6 or higher (developed using: 1.6.0.03)

**Download:** <http://java.sun.com/javase/downloads/index.jsp>

**Windows:** The `$JAVA_HOME/bin` directory must be added to the class path. This is necessary to be able to directly invoke Java commands from the command shell.

### A.2.2 Ant

Some of the components of CSDF rely on Ant, so it has to be installed on the development machine.

**Version:** Any version should work (developed using: 1.7.1)

**Download:** <http://ant.apache.org/bindownload.cgi>

**Windows:** The `$ANT_HOME/bin` directory must be added to the class path. This is necessary to be able to directly invoke Ant from the command shell.

### A.2.3 Logging Service

In order for CSDF to work properly, a Logging Service must be installed, running and available. All the services to be processed by CSDF must be relayed through the Service Interceptor and logged by the Logging Service. CSDF will then on start-up automatically register at the Logging Service and therefore receive copies of all service interactions.

The Logging Service was initially designed in the course of the inContext [23] project. For more detailed information, see 5.1.1 Service Interceptor. The WSDL specification of the Logging Service can be found at `$CSDF/WSDL/Logging/`.

## A.3 INSTALLATION

### A.3.1 Tomcat Apache

Tomcat Apache is a popular web server by Apache. CSDF is a collection of Web services which are deployed under an Axis2 web application. To deploy Axis2 we use Tomcat Apache, so the first step is to install it on the web server. It is available as .zip archive and can be used immediately after unpacking it.

*Steps on the web server:*

1. Download Apache Tomcat
2. Unpack it to `$APACHE_HOME`
3. Start the web server via `$APACHE_HOME/bin/startup.bat`. The server should be running on port 8080 on localhost.
4. Test it by loading `http://localhost:8080/` in any browser. You should see the welcome page of Apache Tomcat.

**Version:** 3.4 (developed using: 6.0.14)

**Download:** <http://tomcat.apache.org/download-60.cgi>

### A.3.2 Eclipse

Eclipse is a state-of-the-art programming environment for the Java language. Although the reader might use another environment to code the Sensor logic, the installation of Eclipse is still required. The reason for this is that the Generator uses the JET technology to perform Model-to-Text transformations. Unfortunately JET is tightly integrated into Eclipse and cannot be used without an operable Eclipse installation. Furthermore the graphical interface for creating and editing SensorModels is only available as Eclipse plugin.

*Steps on the development machine:*

1. Download Eclipse
2. Unpack it to `$ECLIPSE_HOME/..`

**Version:** 3.4.x (developed using: `ganymede-SR1-win32`)

**Download:** <http://www.eclipse.org/downloads/>

### A.3.3 Eclipse Add-ons

In case the downloaded Eclipse compilation does not include all the necessary plugins, these can be installed separately. If the reader chooses a higher version than the suggested Eclipse compilation, he/she has to be careful to select compatible versions of the add-ons.

#### EMF-SDO-XSD

This packet includes the Eclipse Modeling Framework (EMF) as well as classes for XSD.

*Steps on the development machine:*

1. Download EMF-SDO-XSD
2. Unpack it to `$ECLIPSE_HOME/..`
3. Temporarily unpack it and copy `eclipse/plugins/*.jar` to `$EMF_LIB`

**Version:** 2.4.x (developed using: `All-in-One SDK 2.4.1`)

**Download:** <http://www.eclipse.org/modeling/emf/downloads/>

GEF

This is the Graphical Editing Framework (GEF) of Eclipse. It is used to create and use graphical user interfaces to work with Eclipse models.

*Steps on the development machine:*

1. Download GEF
2. Unpack it to `$ECLIPSE_HOME/..`

**Version:** 3.4.x (developed using: All-in-One SDK 3.4.1)

**Download:** <http://www.eclipse.org/gef/downloads/>

JET

JET is a part of the Model-to-Text (M2T) project of Eclipse. It is used to generate code from an EMF model.

*Steps on the development machine:*

1. Download JET
2. Unpack it to `$ECLIPSE_HOME/..`
3. Temporarily unpack it and copy `eclipse/plugins/*.jar` to `$JET_LIB`

**Version:** 0.9.x (developed using: SDK 0.9.1)

**Download:** <http://www.eclipse.org/modeling/m2t/downloads/?project=jet>

WTP

The Web Tools Platform (WTP) contains tools for creating and publishing Web services in Eclipse.

*Steps on the development machine:*

1. Download WTP
2. Unpack it to `$ECLIPSE_HOME/..`

**Version:** 3.0.x (developed using: 3.0.3-20081113)

**Download:** <http://download.eclipse.org/webtools/downloads/>

### *SensorModel*

To be able to load and edit SensorModels, the following plugins have to be installed.

*Steps on the development machine:*

1. Copy \$CSDF/Plugins/\*.jar to \$ECLIPSE\_HOME/plugins

**Version:** 1.0.0

#### A.3.4 Axis2

Axis2 is a Web service engine deployed as a web application on Apache Tomcat. It will be used to host the Sensors, the Controller and the Session Service of CSDF.

*Steps on the development machine:*

1. Download Axis2 as .zip
2. Unpack it to \$AXIS2\_HOME

*Steps on the web server:*

1. Download Axis2 as .war
2. Copy the .war-file to \$APACHE\_WEB
3. Start Apache Tomcat - it will automatically install Axis2 as web application.
4. Copy \$EMF\_LIB/\*.jar in \$AXIS2\_WEB/WEB-INF/lib - This will enable Axis2 to work with EMF classes.

**Version:** 1.3 or higher (developed using: 1.3)

**Download:** <http://ws.apache.org/axis2/download.cgi>

**Windows:** The \$AXIS2\_HOME/bin directory must be added to the class path. This is necessary to be able to directly invoke `Wsd12Java` from the command shell.

## A.4 CONFIGURATION

### A.4.1 Eclipse Settings

In order to work with the different components of CSDF and to develop Sensors, Eclipse must be configured. We will set up a new workspace which we will use for Sensor development and testing.

*Steps on the development machine:*

1. Start Eclipse.
2. Click *File / Switch Workspace / Other...* and enter `$DEV` for *Workspace*. Once you click *OK*, Eclipse should restart and load an empty workspace.
3. Open the *Windows / Preferences* dialog in the menu.
4. Navigate to *Java / Build Path / User Libraries*.
5. Create a new User Library, name it 'Axis2' and add `$AXIS2_HOME/lib/*.jar`
6. Create a new User Library, name it 'Emf' and add `$EMF_LIB/*.jar`
7. Create a new User Library, name it 'Jet' and add `$JET_LIB/*.jar` and close the dialog.
8. Click *File / Import...*, choose *General / Existing Project into Workspace* and enter `$DEV` as root directory. A list with projects should appear. Check the following projects:
  - ◇ Common
  - ◇ ConfigAssistant
  - ◇ ControllerService
  - ◇ SensorService
  - ◇ SessionService
  - ◇ Generator

The imported project should compile without any errors. If there are any errors, check the following list:

- ◇ Have all the required plugins been installed for Eclipse?

- ◇ The projects might refer to a wrong or non-existing version of the Java runtime environment. In this case correct the setting in the '*building paths*'-page of the project.
- ◇ Do the user libraries contain the correct libraries and are their names identical to the ones specified above?

### A.4.2 Session Service

First we configure and deploy the Session Service to the web server.

*Steps on the web server:*

1. Start Apache Tomcat, if not yet running.

*Steps on the development machine:*

1. Edit `$DEV/SessionService/ant.properties` and set the correct paths.
2. Edit `$DEV/SessionService/session.properties` and adapt the properties to your needs.
3. Execute `$DEV/SessionService/create-jar.bat`. This will compile the component as `.aar` Web service.
4. Copy `$DEV/SessionService/SessionService.aar` to `$AXIS2_WEB/WEB-INF/services -`. The service should automatically be deployed from Axis2. You can check this by looking at the output messages in the console of Apache Tomcat.

### A.4.3 Controller

Next the Controller has to be deployed.

*Steps on the development machine:*

1. Edit `$DEV/ControllerService/ant.properties` and set the correct paths.
2. Edit `$DEV/ControllerService/controller.properties` and set the correct locations of the Controller service itself, the subscription service, the logging service and the Session Service.
3. Execute `$DEV/ControllerService/create-jar.bat`, which will compile the component as `.aar` Web service.



4. Copy `$DEV/ControllerService/ControllerService.aar` to `$AXIS2_WEB/WEB-INF/services` - The service should automatically be deployed from Axis2. You can check this by looking at the output messages in the console of Apache Tomcat.
5. Execute `$DEV/ControllerService/controller-init.bat`. This will invoke the Initialize operation on the deployed Controller. The initialization was successful if you can see the message 'Press Return key to continue...'.

In case you experience problems initializing the Controller:

- ◇ Check if Apache Tomcat is running.
- ◇ Check if the settings in `$DEV/ControllerService/controller.properties` are correct. If the location of one of the services is faulty, the Controller will not start or not work properly.
- ◇ Check if the Session Service is deployed properly and is accessible.
- ◇ Check if the Logging Service is running and accessible.

**NOTE** After initialization the Logging Service will forward copies of service invocations to the Controller. If both services are not deployed on the same server, the Controller must be reachable from outside. In case, check your firewall settings to allow access to the web server from outside.

---

## B SOURCE CODE LISTING

## B.1 ECORE META-MODEL

```

<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="sensormodel"
  nsURI="http://sensormodel.csd.f.in_context.eu" nsPrefix="sm">
  <eClassifiers xsi:type="ecore:EClass" name="SensorModel">
    <eStructuralFeatures xsi:type="ecore:EReference" name="iospecification" lowerBound="1"
      eType="#//InputOutputSpecification" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="controlspecification" lowerBound="1"
      eType="#//ControlSpecification" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="servicespecification" lowerBound="1"
      eType="#//ServiceSpecification" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="sensorspecification" lowerBound="1"
      eType="#//SensorSpecification" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="QoSAttribute">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="qosid" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="value" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EDouble"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Assertion" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AssertionExpression" abstract="true">
    eSuperTypes="#//Assertion"/>
  <eClassifiers xsi:type="ecore:EClass" name="AssertionXPath">
    eSuperTypes="#//AssertionExpression">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="xpath" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="nsaware" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="nsmap" upperBound="-1"
      eType="#//NamespaceDefinition" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AssertionRegex">
    eSuperTypes="#//AssertionExpression">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="regex" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"
      defaultValueLiteral=""/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AssertionWSOperation">
    eSuperTypes="#//AssertionExpression">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="operation" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="request" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="NamespaceDefinition">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="namespace" lowerBound="1"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="prefix"
      eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>

```

```

</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="InputOutputSpecification">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="isuseraware" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="isactivityaware" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="sessiontime" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EInt"
    defaultValueLiteral="0"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="ports" upperBound="-1"
    eType="#//PortAbstract" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="defs" upperBound="-1"
    eType="#//IOSet" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="PortAbstract" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="portid" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="input" eType="#//IOInput"
    containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="output" eType="#//IOOutput"
    containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="PortExtract" eSuperTypes="#//PortAbstract"/>
<eClassifiers xsi:type="ecore:EClass" name="PortUpdate" eSuperTypes="#//PortAbstract"/>
<eClassifiers xsi:type="ecore:EClass" name="IODefinition" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="ioid" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="specs" upperBound="-1"
    eType="#//DataSpecification" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="includes" upperBound="-1"
    eType="#//IOReference" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="IOReference">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="ioid" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"
    defaultValueLiteral=""/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="nsprefix" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="IOInput" eSuperTypes="#//IODefinition"/>
<eClassifiers xsi:type="ecore:EClass" name="IOOutput" eSuperTypes="#//IODefinition"/>
<eClassifiers xsi:type="ecore:EClass" name="IOSet" eSuperTypes="#//IODefinition">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="readable" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="writeable" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="DataSpecification">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="dataid" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="datatype" lowerBound="1"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="assertions" upperBound="-1"
    eType="#//Assertion" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="qos" upperBound="-1"
    eType="#//QoSAttribute" containment="true"/>

```

```

</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ControlSpecification">
  <eStructuralFeatures xsi:type="ecore:EReference" name="standards" upperBound="-1"
    eType="#//Standard" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="access" upperBound="-1"
    eType="#//ControlAccess" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="activationkey" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ControlParameter">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="controlid" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="type" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="default" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="readable" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="writeable" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ControlAccess" abstract="true"/>
<eClassifiers xsi:type="ecore:EClass" name="ControlAccessDefault"
  eSuperTypes="#//ControlAccess"/>
<eClassifiers xsi:type="ecore:EClass" name="ControlAccessUser" eSuperTypes="#//ControlAccess">
  <eStructuralFeatures xsi:type="ecore:EReference" name="standardaccess" upperBound="-1"
    eType="#//ControlStandardAccess" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="key" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ControlStandardAccess">
  <eStructuralFeatures xsi:type="ecore:EReference" name="standard" lowerBound="1"
    eType="#//Standard"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="readable" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="writeable" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ServiceSpecification">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="controllerservice" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="services" upperBound="-1"
    eType="#//ServiceDescription" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ServiceDescription" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="serviceid" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ServiceWS" eSuperTypes="#//ServiceDescription">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="wsdl" lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ServiceSensor" eSuperTypes="#//ServiceDescription">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="serviceuri"

```

```

    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="portid"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Standard" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="standardid" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="parameter" upperBound="-1"
        eType="#//ControlParameter" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="StandardStatus" eSuperTypes="#//Standard"/>
<eClassifiers xsi:type="ecore:EClass" name="StandardUserDefined" eSuperTypes="#//Standard"/>
<eClassifiers xsi:type="ecore:EClass" name="SensorSpecification">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="serviceurl" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="author" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"
        defaultValueLiteral=""/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="resources" upperBound="-1"
        eType="#//Resource" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Resource" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="resourceid" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="location" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"
        defaultValueLiteral=""/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="local" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ResourceWithNamespace" abstract="true"
    eSuperTypes="#//Resource">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="namespace" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="prefix" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ResourceSchema" abstract="true"
    eSuperTypes="#//ResourceWithNamespace"/>
<eClassifiers xsi:type="ecore:EClass" name="ResourceSchemaXsd" eSuperTypes="#//ResourceSchema"/>
<eClassifiers xsi:type="ecore:EClass" name="ResourceSensor"
    eSuperTypes="#//ResourceWithNamespace"/>
<eClassifiers xsi:type="ecore:EClass" name="ResourceWSDL"
    eSuperTypes="#//ResourceWithNamespace">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="convertSchemaElementToSchemaType"
        lowerBound="1" eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="DataSet">
    <eStructuralFeatures xsi:type="ecore:EReference" name="data" upperBound="-1"
        eType="#//DataValue" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="DataValue">

```

```

    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dataid" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="value" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="qos" upperBound="-1"
        eType="#//QoSAttribute" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ParameterValue">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="parameterid" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="value" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="PortReference">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="serviceuri" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="portid" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="SensorInfo">
    <eStructuralFeatures xsi:type="ecore:EReference" name="sensor" lowerBound="1"
        eType="#//ServiceSensor" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ports" upperBound="-1"
        eType="#//SensorInfoPort" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="services" upperBound="-1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="SensorInfoPort">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="update" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="portid" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="inputs" upperBound="-1"
        eType="#//SensorInfoIO" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="outputs" upperBound="-1"
        eType="#//SensorInfoIO" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="forwardto" upperBound="-1"
        eType="#//PortReference" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="forwardfrom" upperBound="-1"
        eType="#//PortReference" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="SensorInfoIO">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="ns" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dataid" lowerBound="1"
        eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#/EString"/>
</eClassifiers>
</ecore:EPackage>

```

## REFERENCES

- [1] AILISTO, H., ALAHUHTA, P., HAATAJA, V., KYLLOENEN, V., AND LINDHOLM, M. Structuring context aware applications: Five-layer model and example case, 2002. 7, 23
- [2] ALONSO, G., CASATI, F., KUNO, H., AND MACHIRAJU, V. *Web Services. Concepts, Architectures and Applications*. Springer-Verlag Berlin Heidelberg, 2004. 10
- [3] BALDAUF, M., DUSTDAR, S., AND ROSENBERG, F. A survey on context-aware systems. *Int J. Ad Hoc and Ubiquitous Computing, Vol.2, No.4* (2007), 263–277. 6
- [4] BARDRAM, J. The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. *Pervasive* (2005), 98–115. 15
- [5] BEIGL, M., GELLERSEN, H., AND SCHMIDT, A. Mediacups: Experience with design and use of computer-augmented everyday objects. *Computer Networks, Vol. 35. No. 4. Elsevier* (2001), 401–409. 14
- [6] BIEGEL, G., AND CAHILL, V. A framework for developing mobile, context-aware applications. *Proceedings of the 2nd IEEE Conference on Pervasive Computing and Communication* (2004), 361–365. 15, 36
- [7] BROWN, P. The stick-e document: a framework for creating context-aware applications. *Proceedings of the Electronic Publishing, Palo Alto* (1996), 259–272. 5
- [8] CHEN, G., AND KOTZ, D. Context-sensitive resource discovery. *First IEEE International Conference on Pervasive Computing and Communications* (2003), 243–252. 14
- [9] CHEN, H. Phd-thesis - an intelligent broker architecture for pervasive context-aware systems, 2004. 14
- [10] CHEN, H., FININ, T., AND JOSHI, A. An ontology for context-aware pervasive computing environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review, Vol. 18. No. 3* (2004), 197–207. 14



- [11] DEY, A., AND ABOWD, G. Towards a better understanding of context and context-awareness, 2000. 5, 6
- [12] DEY, A., ABOWD, G., AND WOOD, A. Cyberdesk: a framework for providing self-integrating context-aware services, 1998. 14
- [13] DEY, A., SALBER, D., AND ABOWD, G. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction Journal*, Vol. 16. No. 2-4 (2001), 97–166. 14
- [14] DIJKSTRA, E. On the role of scientific thought. *Selected writings on Computing: A Personal Perspective*, New York, USA (1982), 60–66. 9
- [15] DUSTDAR, S., AND SCHREINER, W. A survey on web service compositions. *Int. J. Web and Grid Services*, Vol. 1, No. 1, 2005 (2005), 1–30. 10
- [16] ERL, T. *Service-Oriented Architecture: Concepts, Technology and Design*. Prentice Hall/PearsonPTR, 2006. 9
- [17] ERL, T. Soa principles, 2009. <http://www.soapprinciples.com/>. 9
- [18] FAHY, P., AND CLARKE, S. Cass: Middleware for mobile context-aware applications. *ACM MobiSys Workshop on Context Awareness* (2004). 15
- [19] FITZPATRICK, A., BIEGEL, G., CLARKE, S., AND CAHILL, V. Towards a sentient object model, 2002. 15
- [20] GU, T., PUNG, H., AND ZHANG, D. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications (JNCA)*. Vol. 28. No. 1. (2005), 1–18. 15
- [21] HULL, R., NEAVES, P., AND BEDFORD-ROBERTS, J. Towards situated computing. *Proceeding of the First International Symposium of Wearable Computers (ISWC '97)* (1997), 146. 5
- [22] KANTER, T. Attaching context-aware services to moving locations. *IEEE Internet Computing*, Vol. 7. No. 2. (2003), 43–51. 14
- [23] KENDAL, P. Effective team collaboration via collaborative software - incontext, 2009. <http://www.in-context.eu>. 8, 16, 215
- [24] KUHN, T. Webservices, steam und iceberg, 2005. 15

- [25] PALLOT, M., PRINZ, W., AND SCHAFFERS, H. Future workplaces, towards the 'collaborative' web, 2005. 8
- [26] PALLOT, M., SALMINEN, V., PILLAI, B., AND PAWAR, K. Business semantics: Collaboration? the magic instrument enabling plug & play, 2004. 8
- [27] PRINZ, W., LÖH, H., SCHAFFERS, H., SKARMETA, A., AND DECKER, S. Ecospace - towards an integrated collaboration space for eprofessionals, 2008. 8
- [28] PRINZ, W., AND PALLOT, M. Ecospace - ami@work communities wiki, 2009. <http://www.ami-communities.eu/wiki/ECOSPACE>. 8
- [29] RYAN, N., PASCOE, J., AND MORSE, D. Enhanced reality fieldwork: the context-aware archaeological assistant. *Proceeding of the 25th Anniversary Computer Applications in Archaeology* (1997). 5
- [30] SCHAFFERS, H., BRODT, T AND PALLOT, M., AND PRINZ, W. *The Future Workspace - Perspectives on Mobile and Collaborative Working*. Telematica Instituut, 2006. 8
- [31] SCHAFFERS, H., AND PALLOT, M. Mosaic - ami@work communities wiki, 2009. <http://www.ami-communities.eu/wiki/MOSAIC>. 8
- [32] SCHAFFERS, H., PRINZ, W., PALLOT, M., AND FERNANDO, T. Mobile and collaborative workplaces: An agenda for innovation, 2005. 8
- [33] SCHALL, D., DORN, C., DUSTDAR, S., AND DADDUZIO, I. Viecar - enabling self-adaptive collaboration services, 2007. 11
- [34] SCHILIT, B., AND THEIMER, M. Disseminating active map information to mobile hosts. *IEEE Network*, Vol. 8, No. 5 (1994), 22–32. 5
- [35] STEINFELD, C., JANG, C.-Y., AND PFAFF, B. Supporting virtual team collaboration: The teamscope system, 1999. 8
- [36] STRIMPAKOU, M., ROUSSAKI, I., PILS, C., ANGERMANN, M., ROBERTSON, P., AND ANAGNOSTOU, M. Context modelling and management in ambient-aware pervasive environments, 2005. 14
- [37] TRUONG, H., AND DUSTDAR, S. A survey on context-aware web service systems. *International Journal of Web Information Systems* (2008). 16, 18, 24

- 
- [38] TRUONG, H., JUSZCZYK, L., MANZOOR, A., AND DUSTDAR, S. Escape - an adaptive framework for managing and providing context information in emergency situations. *Smart Sensing and Context, Second European Conference, EuroSSC 2007* (2007), 207–222. 15
- [39] TRUONG AT AL., H.-L. incontext: a pervasive and collaborative working environment for emerging team forms, 2008. 8, 16
- [40] TUV (VIENNA UNIVERSITY FOR TECHNOLOGY). incontext - d4.1 - principles and mechanisms for 'context tunneling', 2006. <http://www.in-context.eu/uploads/files/20061215D4.1v1.0Principles20and20Mechanisms20for20Context20Tunnelling.pdf>. 32
- [41] TUV (VIENNA UNIVERSITY FOR TECHNOLOGY). incontext - d1.2 - discovering service-interaction patterns - methods and mining algorithms, 2007. <http://www.in-context.eu/uploads/files/20061106D1.2v1.1Service20Interaction20Patterns.pdf>. 16, 21, 211
- [42] TUV (VIENNA UNIVERSITY FOR TECHNOLOGY). incontext - d2.2 - design and proof-of-concept implementation of the incontext context model version 1, 2007. <http://www.in-context.eu/uploads/files/20070627D2.2v1.0Context20model20design20and20prototype20implementation.pdf>. 11
- [43] TUV (VIENNA UNIVERSITY FOR TECHNOLOGY). incontext - d5.3 - design and implementation of pcsa intermediate prototype, 2007. 29
- [44] VAN LAERHOVEN, K. Technology for enabling awareness (tea), 1998. <http://www.teco.edu/tea/>. 14
- [45] VOIDA, S., MYNATT, E., MACINTYRE, B., AND CORSO, G. Integrating virtual and physical context to support knowledge workers. *IEEE Pervasive Computing, Vol. 1. No. 3.* (2002), 73–79. 14
- [46] WANT, R., HOPPER, A., FALCAO, V., AND GIBBONS, J. The active badge location system. *ACM Transactions on Information Systems. Vol. 10. No 2.* (1992), 91–102. 14
- [47] WEISER, M. Ubiquitous computing, 1996. 6
- [48] WEISER, M. The computer for the 21st century, 1999. 14

- 
- [49] WIKIPEDIA. Collaborative working environment, 2009. [http://en.wikipedia.org/wiki/Collaborative\\_Working\\_Environment](http://en.wikipedia.org/wiki/Collaborative_Working_Environment). 8
- [50] WIKIPEDIA. Service-oriented architecture, 2009. [http://en.wikipedia.org/wiki/Service-oriented\\_architecture](http://en.wikipedia.org/wiki/Service-oriented_architecture). 9

# INDEX

- Activate, 131
- Activation, 60
- Active Sensor, 60
- Activity Service, 195
- Activity-Awareness, 52
- Assertion, 89
- AssertionExpression, 90
- AssertionRegex, 91
- AssertionWSOperation, 91
- AssertionXPath, 90
  
- CASS, 15
- COBRA, 14
- Collaborative Working Environment, *see*  
CWE
- Composition, 53
  - Algorithm, 63
- ConfigAssistant, 68
- Context Management System, 28
- Context Sensor Development Framework,  
*see* CSDF
- Control Specification, 45
- ControlAccess, 99
- ControlAccessDefault, 100
- ControlAccessUser, 100
- Controlid, 78
- Controller, 30
  - Context, 32
  - Filtering, 33
  - Initialization, 70
- ControlParameter, 97
- ControlSpecification, 95
- ControlStandardAccess, 101
- CORTEX, 15
- CSDF, 28
  
- CWE, 8
  
- Dataid, 78
- DataSet, 108
- DataSpecification, 87
- DataValue, 109
- Delete, 147
- Development Circle, 64
- Direct Compatibility, 55
- Document Service, 202
  
- Ecore, 78
- Email Service, 20, 195, 199
- ESCAPE, 15
- Extraction Port, 36
  
- Filter, 49
- Filtering, 48
- Forward, 58
- Forward-From, 58
- Forward-To, 58
  
- Generator, 41, 66
- Get, 145
- GetCompatibleInputPorts, 143
- GetCompatibleOutputPorts, 144
- GetIOSpecification, 115
- GetNamespaceByPrefix, 124
- GetParameterValue, 120
- GetPort, 115
- GetPortForwards, 116
- GetResourceByNamespace, 123
- GetSelf, 125
- GetServiceByCore, 139
- GetServiceByRequirements, 140
- GetStandard, 119

- inContext, 8, 16
- Inferred Compatibility, 55
- Initialize, 130, 141
- Input/Output Specification, 44
- InputOutputSpecification, 80
- Integrated Service, 45
- Invoke, 126
- IODefinition, 86
- Ioid, 78
- IOInput, 85
- IOOutput, 85
- IOReference, 93
- IOSet, 85
- Is-Alive Concept, 31
- IsActive, 132
- IsAlive, 129
  
- JCAF, 15
  
- ListAccessForKey, 119
- ListAllActiveServices, 141
- ListAllForwards, 116
- ListAllServices, 125, 139
- ListAllServicesDetails, 139
- ListAllStandards, 118
- ListResources, 123
- Logging Service, 28
- Logging Subscriber, 28
- Loop Detection, 57
  
- Mailinglist Service, 195
  
- NamespaceDefinition, 92
  
- Parameterid, 78
- ParameterValue, 109
- Passivate, 133
- Passivation, 60
- Passive Sensor, 60
- Pending Message Timer, 30
  
- PortAbstract, 83
- PortExtract, 84
- Portid, 78
- PortReference, 110
- PortUpdate, 84
  
- QoSAttribute, 92
  
- Register, 135
- Resource, 46, 105
- ResourceSchema, 106
- ResourceSchemaXsd, 106
- ResourceSensor, 106
- ResourceWithNamespace, 105
- ResourceWSDL, 107
- Room Reservation Service, 20, 199
  
- Sensor, 35
  - Activation, 70
  - Compatibility, 55
  - Composition, *see* Composition
  - Databinding, 39
  - Extension, 39
  - Filtering, *see* Filtering
  - Integrated Services, 40
  - Invocation, 73
  - Parameter, 40
  - Port, 36
  - Registration, 71
  - Session Module, 38
- Sensor Specification, 46
- SensorInfo, 110
- SensorInfoIO, 112
- SensorInfoPort, 111
- SensorModel, 43, 80
- SensorSpecification, 103
- Sentient Object Model, 15
- Service Interceptor, 28
- Service Oriented Architecture, *see* SOA

---

Service Specification, 45  
ServiceDescription, 102  
Serviceid, 78  
ServiceSensor, 103  
ServiceSpecification, 101  
ServiceWS, 102  
Session Service, 34  
Session-Data File, 67  
Session-Frame, 52  
SessionCreate, 148  
SessionDestroy, 149  
Sessionid, 78  
Set, 146  
SetActiveStatus, 138  
SetParameterValue, 121  
Shutdown, 142  
SOA, 9  
SOCAM, 15  
Standard, 96  
StandardStatus, 96  
StandardUserDefined, 97  
  
Test  
    Code, 67  
    Integration, 69  
Type System, 46  
  
Unregister, 137  
UnregistrationNotification, 128  
Update Port, 36  
User-Awareness, 51  
  
Variable, 87