Service Request Oriented Architecture

vorgelegt von
Diplom-Informatiker
Ilja Radusch
aus Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr. Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Hans-Ulrich Heiß
Berichter:	Prof. Dr. Dr. h.c. Radu Popescu-Zeletin
Berichter:	Prof. Dr. Peter Pepper

Tag der wissenschaftlichen Aussprache: 13. November 2008

Berlin 2009

D 83

Abstract

The vision of Ambient Intelligent Systems describes human-centric environments that are able to adapt intelligently to the user situation. However, the growing complexity of implementing such Ambient Intelligent Systems has now outpaced even the most ambitious research efforts due to the ever increasing number of communication devices and the corresponding increase in the number of difficult choices each user is faced with. Implementing this vision of Ambient Intelligent Systems in a world of heterogeneous devices, networks, and services available to, and required by, users has thus proved harder than was previously expected. Against this backdrop, this thesis presents a novel approach that introduces a non-disruptive, scalable architecture for Ambient Intelligent Systems.

In line with recent research projects, this thesis starts by analyzing typical single and multi-user scenarios in varying environments, deriving the requirements for an appropriate architecture on an abstract level. This requirements analysis gives a novel layer model for Ambient Intelligent Systems that decouples the interpreting and processing of user needs – generally referred to as context-awareness – from service interaction, and loosely coupled service execution from low-level information access across varying networks and devices. This layer model is then further extended to the *Service Request Oriented Architecture* by identifying service requests as connecting entities between the layers. Thus service request orientation offers a convenient abstraction for translating abstract user requests to specific service calls in an extendable and loosely coupled manner that allows for integration not only of existing context-agnostic services but also of a wide range of devices, from small, resource-bounded sensor nodes to mobile devices or high-end computers.

As work on this Service Request Oriented Architecture was a joint effort between the TU Berlin and Fraunhofer FOKUS, the findings of this thesis have been fed into a number of national and international projects such as Autarke Verteilte Mikrosysteme (AVM) funded by the German Federal Ministry for Education and Research (BMBF) and e-Sense, funded by the European Commission.

German Abstract

Die Vision der Ambient Intelligent Systems beschreibt Umgebungen, die sich unsichtbar und eingebettet in der Benutzerumgebung auf intelligente Weise der aktuellen Situation des Benutzers anpassen können. Leider wuchs die Komplexität bei der Umsetzung dieser Vision durch die rasche Einführung neuer Kommunikationsgeräte schneller als die Forschung in diesem Bereich bewältigen konnte. Die Vergangenheit zeigte, dass in dieser wachsenden Welt von unterschiedlichen Geräten, Netzwerken und Diensten, deren Möglichkeiten nur teilweise von den Benutzern verstanden und angenommen werden, neuartige Systeme, die nicht auf bestehende Infrastrukturen aufsetzen, nicht durchzusetzen sind. Diese Arbeit beschreibt dementsprechend einen neuen Ansatz einer skalierbaren Architektur für Ambient Intelligent Systems, die ausdrücklich bestehende Infrastrukturen nutzt und erweitert und so in der Lage ist, bereits existierende Dienste einfach einzubinden.

Analog zu aktuellen Forschungsprojekten beginnt die Arbeit mit der Beschreibung von Szenarien mit einzelnen und mehreren Benutzern, die in der darauf folgenden Analyse genutzt werden, um die notwendigen Anforderung zu ermitteln. Aus diesen Anforderungen wird dann ein neuartiges Schichtenmodell für Ambient Intelligent Systems entwickelt, welches die Interpretation und Bearbeitung von Benutzerbedürfnissen von der Interaktion der Dienste und die lose gekoppelte Dienstausführung vom allgemeinen Informationszugriff auf verschiedenste Geräte und Netzwerke trennt. Dieses Schichtenmodell wird im weiteren Verlauf der Arbeit zu einer vollständigen Referenzarchitektur ausgebaut, bei der Dienstanfragen (*Service Requests*) als Bindeglied für die Kommunikationseinheiten der verschiedenen Schichten fungiert. Die Ausrichtung auf Dienstanfragen (*Service Request Orientation*) hat sich als eine adäquate Abstraktion für die Umwandlung von generischen Benutzeranfragen (*User Requests*) in spezifische Dienstaufrufe bewährt. Diese erweiterbare lose Koppelung von Diensten erlaubt es existierende Dienste auf einfache Weise zu integrieren sowie Informationen von einer großen Breite von Geräten abzufragen und in das System einzubeziehen, angefangen von kleinsten Sensorknoten über mobile Geräte bis hin zu hochwertigen Computern.

Die Arbeit an dieser Dienstanfragen-orientierten Architektur (*Service Request Oriented Architecture*) ist das Ergebnis gemeinsamer Forschungen von der TU Berlin und Fraunhofer FOKUS. Die Ergebnisse dieser Arbeit sind in eine Reihe von nationalen und internationalen Forschungsprojekte eingeflossen wie z.B. Autarke Verteilte Mikrosysteme (AVM) gefördert vom BMBF sowie e-Sense, welches von der Europäischen Union gefördert wurde.

Acknowledgements

First of all I would like to thank Radu Popescu-Zeletin for giving me the opportunity to embark on this thesis and for providing expert guidance in the inspiring environments of university work at the TU Berlin and applied research at Fraunhofer Institute FOKUS. I am grateful for his tremendous patience and continuous support throughout all my work on this thesis.

Furthermore my heartfelt thanks go out to all university students who graduated in the field of Service Request Oriented Architectures. I would particularly like to express my deepest gratitude to Witold Drytkiewicz, David Linner, and Carsten Jacob for their truly exceptional work.

My thesis would have been impossible without the support of all my former and present colleagues in both institutions who have provided invaluable moral and technical support.

Finally, I would like to thank my parents, Jenny and Ralf-Dieter Radusch, for their unflagging support and assistance in each and every area of my life.

Contents

1. Introduction

Recent developments in mobile communication and small computing devices have had a tremendous impact on our world. They have brought the dream of Ubiquitous Computing and Communication closer to reality. These Ambient Intelligence concepts help to moderate the predicted user information overload, where applications utilize advanced anticipation algorithms to adapt their interaction towards the user according to his or her current situation. As the computer becomes ever more pervasive, the number of tasks assigned to computers increases and the complexity of user interaction with the machines increases as well, since all these tasks must hide behind simple interfaces. In this regard, Ambient Intelligence, inspired by the vision of Ubiquitous Computing, proposes the simplest user interface which is basically no visible interface at all. In this vision the user does not need to interact directly with the computer anymore since the current user intention can, and should be, derived from data gathered invisibly via various sensors in the environment.

Given the various previous approaches to Ambient Intelligence, we will first propose a general overview of the different aspects in terms of both user and device interaction and corresponding technical requirements. We will first analyze and categorize different scenarios describing a world of ubiquitous networks. We argue that invisible computers may not be quite as desirable as the vision usually suggests, and we try to identify important aspects for satisfying user interaction. In terms of the initial problem of user information overload, we propose that future user-centric computers must be able to deliver the right information at the right time in the right amount of detail to the user without neglecting the currently deployed communication systems and infrastructures. Therefore, we present a non-disruptive and scalable architecture for providing user centric applications in arbitrary network environments.

1.1. Methodology and Structure

Developing frameworks for Ambient Intelligent Systems has been and still is a long ongoing effort. Although the general vision of Ambient Intelligence is rather clearly defined as helping future users to cope better with the technology available to them, the necessary steps and intermediate goals are harder to define. Past research into Ambient Intelligence was driven by specific stakeholders such as consumer electronics manufacturers, (i.e. Philips), or the telecommunication industry, (i.e. Motorola, and Nokia to mention but two). Interestingly, after many years of research no approach by either stakeholder has yet managed to completely fulfill the hopes of Ambient Intelligence. Even so, in the process a specific research methodology primarily utilizing descriptive scenarios has been tested and has proven beneficial. This thesis will apply such accepted research methodology and starts by describing vari-

ous defining ambient intelligent scenarios. Subsequently we analyze the technical implications and extrapolate requisite technologies or devices, applications, and issues. These implications are then used to structure the various issues involved in the development of Ambient Intelligent Systems in a comprehensive layer model. This is followed by the introduction of the Service Request Oriented Architecture, comprising a universal access layer, pREST, with SEDS, a semantic data space implementation on top. Furthermore, we will learn how to integrate existing legacy services not specifically build for this architecture.

2. Ambient Intelligence

Since the emergence of modern communication technologies, individuals have quickly realized that their actions and general opportunities are greatly influenced not only by physical interaction with other individuals in their vicinity, but more and more by events within their whole expanding communication range. The novel experience for individuals was that information about events, possibly involving people they are not even familiar with, could influence their daily lives. At this point communication – or information exchange – gained a new quality because previously information was basically related to individuals and things directly known to the sender or receiver. The new concept of *News* describes all information disseminated by print, radio, television, Internet or even by word of mouth. Yet the acceleration of data transmission and the exponential expansion of the communication range have now led to our current situation where virtually every event in any place on earth can be instantly communicated across the whole world. Individuals quickly realized that they must learn to deal with the information overload emanating from mere advances in communication technology. And even though whole industries soon began to emerge for such information confining and filtering, in the end individuals still need to adapt and learn how to cope with much more information than those who had gone before them.

In addition, the evolution of computers into ever more powerful and smaller devices, has wrought a second change on the nature and effects of communication. On the one hand the degree of automation we now have allows remote control of physical objects through communication while on the other the computerization and virtualization of former physical transactions has led to a situation where information can directly and literally change the physical world without human interaction. This development is partly described and introduced by the term *Ubiquitous Computing* coined by Mark Weiser in 1991 as the computer paradigm of the 21st century. Weiser et al. [02][01][04] envisioned the disappearance of computer devices as we know them, together with all their dedicated monitors and input devices. According to Weiser, in the future computing power will be embedded in every object surrounding humans. The introduction of new input mechanisms such as speech recognition or other advanced mechanisms for detecting the current user needs will mean that no human will ever need to use ordinary input devices such as a keyboard or a mouse. Subscribing to Mark Weiser's seminal vision, Philips along with other researchers developed the paradigm of *Ambient Intelligence* (AmI) in the late nineties, which has now been widely adopted throughout Europe. Other terms such as *Pervasive Computing* or *Ubiquitous Networking* describe similar approaches in the US or the Japanese research community. Against this backdrop *Ambient Intelligence* (and similar such approaches) is set to evolve the vision of vanishing computers still further by implementing the following five basic paradigms:

- The computer will become embedded, disappearing into customary objects
- The computer will become context-aware, recognizing the user and his or her current situation
- The computer will become personalized, customized specifically for each user
- The computer will become adaptive, capable of running in almost any environment
- The computer will become anticipatory, guessing human needs without user interaction

2.1. Background

This vision has been brought closer to reality by programs like the 5th Framework Program from the European Information Society Technologies Advisory Group (ISTAG) that started in May 2000 and by actions and other efforts driven by non-profit consortia such as the Wireless World Research Forum (WWRF). Synchronous to these programs in Europe and the U.S., researchers in Japan have developed their own similar vision of a *ubiquitous network society* in programs such as "e-Japan" and the follow-up "u-Japan".

Critics of world-wide research into Ambient Intelligence (AmI) have long pointed out the potential for abuse as Ambient Intelligence Systems need to gather vast amounts of information without the user even noticing. Addressing such fears is not just a matter of security and privacy or applying well-known cryptographic algorithms to the information thus gathered, since this basically just secures the transportation channels. Rather it is a question of bringing transparency to the whole system design without thereby losing the envisioned benefits. [99] Accordingly, the vision of human-centric communication and the *Service Request Oriented Architecture* proposed here is not only able to scale and include distributed information sources from every node in the network, but also allows the individual to stay within this flow of information and identify exactly how and where the information is accessed, stored, and evaluated. Moreover, enabling transparency through an open and extendable architecture – from information gathering and information network transport through to information evaluation – not only creates trust in Ambient Intelligence but also increases robustness as each component of the proposed architecture is easily replaceable. Each component can be accessed and controlled by the user on every layer, thereby avoiding former black-box approaches that act invisibly on behalf of the individual.

2.1.1. I-centric Communication Reference Model

The WWRF adopted a further evolution of the vision of Ambient Intelligence for their third generation and beyond (3Gb) service architectures and coined the term *I-Centric Communications*. Now focusing on services rather than devices, this vision basically sought to free up the individual communication space of humans, allowing them to interact with all objects in their environment regardless of physical restrictions. As depicted in Figure 1 below, the *Individual Communication Space* includes devices as well as abstract concepts such as Knowledge, Food, or People. All the entities humans (or individuals) will interact with – and even users themselves – are referred to as abstract I-centric objects serving as a logical representation [66]. Thus these I-centric objects carry specific properties:

- I-centric objects represent entities surrounding the user
- I-centric objects are addressable,
- I-centric objects provide well-defined services to individuals
- I-centric objects can be activated and deactivated
- I-centric objects can act by themselves according to the specific needs of an individual
- I-centric objects can wrap arbitrary legacy entities

Figure 1: The *Individual Communication Space* as defined by *I-Centric Communications*

Furthermore, according to the I-Centric Communications paradigm individuals communicate through sharing common I-centric objects. I-centric objects then can also communicate with each other, following a specific interaction model for I-centric objects. As interaction of I-centric objects always occurs with respect to one or several specific I-centric contexts, these contexts refer to the relationships and circumstances of the individual concerned and are independent of the actual execution environment.

Together with this I-centric service architecture, the WWRF also developed a complete I-centric reference model as depicted in Figure 2.

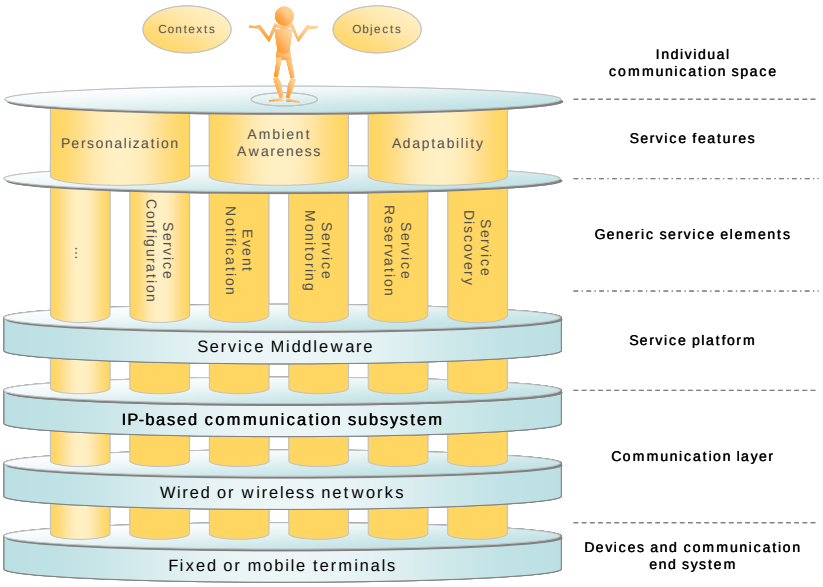

Figure 2: I-centric reference model

The I-centric reference model defines the building blocks needed for implementing I-centric services. A service middleware on a basic IP-based communication layer provides generic service blocks or service elements as they are called in the reference model. These service blocks include support for the discovery of nodes and services, interfaces for monitoring and event notification as well as interfaces for the configuration and reservation of resources. The advanced service features cited above such as personalization, ambient awareness, and adaptability, utilize these generic service elements to provide ambient intelligent – or I-centric – services to the user.

The efforts of the WWRF were mainly driven by players from the telecommunication industry. Thus the I-centric reference model, relies heavily on fixed or mobile terminals as do the underlying scenarios leading to this model.

2.1.2. Conclusion

The various research programs mentioned above testify to the fact that deriving a universal architecture for a generic Ambient Intelligence framework is no easy task and one that still has not been accomplished. Accordingly, we will use their results to deduce a new architecture that describes the requisite entities. Another finding of past research programs is that utilizing comprehensive scenario descriptions is the best way to identify the technologies, applications, issues and drivers needed for

implementing a widespread Ambient Intelligence System. We will adapt this methodology and describe its various defining scenarios in the following sections.

2.2. Scenarios for an Ambient Intelligent World

This section describes the major scenarios guiding the development of the framework proposed in this thesis. We shall introduce scenarios developed in previous research projects, such as e-Sense, WINNER, MAGNET, and MobiLife. These scenarios are then analyzed and refined down to three basic scenarios for further investigation in this thesis. This way we neither have to introduce as yet unknown and unchecked scenarios, nor do we risk being overwhelmed by the sheer volume of available scenarios.

Considered as an elaborate form of story-telling, scenarios are well known in the area of future technologies analysis. First used by the military to plan various attacks and counter-attacks, this methodology was quickly taken up by a variety of future studies. Scenarios help to identify key aspects, like technologies, drivers, and upcoming issues in a comprehensible fashion. A scenario describes several aspects of a specific future which can be further elaborated by varying major parameters or preconditions. Scenarios also found their place in modern development paradigms usually in the form of use-case description, i.e. in UML. However, the main difference between such use-case descriptions and scenarios is that use-cases visualize the interaction between already known actors and components whereas scenarios help to identify and picture the actors and components that are needed.

Scenarios can be distinguished according to their specific type. They can either be trend extrapolating or contrasting, either supporting current technologies or helping to visualize the - possibly negative - consequences of emerging ones. Scenarios can also be descriptive or normative, explorative or anticipatory. Descriptive scenarios simply describe a possible future whereas normative scenarios also try to convey specific positive or negative judgments. Since we do not intend to evaluate the possible human impacts of this technology in this thesis, we will only deal with descriptive scenarios in the analysis below. Similarly we will not use explorative scenarios which try to forecast the consequences of circumstances observable in the present but rather utilize anticipatory scenarios that can be used to “backcast” what effects or technologies are needed to reach a specific future already described by Ambient Intelligence.

The vision of Ambient Intelligence aims to support the user in coping with an ever more computerized world. For the analysis of the scenarios we will suppose three basic roles for the user in guiding the scenario analysis. Although every user takes on several roles on a daily basis, these can essentially be reduced to three basic ones – the public participant, the professional, and the private individual. Note that these

basic roles apply equally to groups of users or even situations. The currently active role has a great influence on the behavior of the Ambient Intelligent System in terms of the way data about the environment and the user can be used by the system and made available to others. These roles also influence all other aspects of the Ambient Intelligent System introduced above such as personalization and context-aware adaptation on behalf of the user.

As private individuals or groups we all have everyday tasks to do such as housework, shopping, etc. AmI holds out the promise of relieving the tedium of such activities through the broad use of personal information, acting autonomously on behalf of the user and even foreseeing needs and wishes before they are formulated.

The professional also utilizes AmI to relieve his or her working life according to their personal preferences. This can include helping in organizing and executing work tasks, and filtering and categorizing information to avoid the now notorious information overload. However, unlike the private individual's data the user data gathered and utilized always belongs to a specific work environment. Nor should all private data generally available about the user be used by the AmI system at work to avoid privacy concerns. User-related company data must be protected from theft and misuse. This is obvious, and easy to implement for specific types of data such as company documents or spreadsheets. Even so, a personal filter list for important phone numbers may indeed include valuable company information.

Finally, the public participant represents the biggest challenge for the future of AmI. Current privacy laws, although different in each country, all outline quite strictly what kind of data can be utilized by the AmI systems in the public domain. Although most users welcome more personalized service offers, they are also aware of the implications of the loss of privacy these entail. Additionally, leaving aside the intentional and consensual gathering of personal user data, it is indeed unintentional and accidentally acquired user data that usually proposes a bigger threat to user privacy. Although privacy issues are not the immediate concern of this thesis, the architecture proposed below will be able to accommodate them.

Similar to the roles of the user we can further categorize the scenarios according to the environment the user is currently in and what the respective goals of AmI can be. This is particular interesting as the deployment of Ambient Intelligent Systems is not, and cannot be, centralized or enforced. The advantages of Ambient Intelligence are now emerging through various bottom-up approaches initiated, driven, and financed by users themselves. Thus every scenario describing the future of Ambient Intelligence must feature a unique benefit for the user that cannot otherwise easily be achieved. A parallel approach to the part-achievement of features promised by AmI is for service providers to offer these as part of a better user experience. Accordingly personalization and personalized advertisements are the most common

features implemented today. However, as noted above, the privacy issues implied by those services are still likely to loom large in the future.

We thus distinguish two basic settings in our scenarios - Smart Environments [03] and Smart Personal Mobility. Smart Environments mainly describe the user environments envisioned by Mark Weiser with almost invisible computing devices acting for, and on behalf of, the user. We assume that the goal of the AmI system is to maintain the user environment as constantly and consistently as possible - ideally without any user interaction at all. This is in contrast to the latter setting, Smart Personal Mobility. Here typically no AmI enabled devices are available, and the user is rather equipped with an advanced personal digital assistant which is expected to help with user-initiated requests or tasks. Of course, we expect future AmI systems to generally support both settings but this distinction will now help us to better classify the scenarios on a abstract level. It will also help to better differentiate later between user-initiated and ubiquitous AmI services and their respective requirements.

2.2.1. Smart Environments

As outlined above, invisible computing devices and anticipatory services are not only the goal of AmI, they are also needed for managing the complexity of future interactions between humans and machines. Similar to the above classification of user roles, we can further divide smart environments into three different locations, each with their specific technical requirements and characteristics – smart homes, smart office, and smart public services. These are further described in this section.

In general smart environments implement the vision of always on, networked, and highly distributed computing devices, ranging from small sensors – for e.g. measuring the temperature – and actuators - e.g. lights - to more resourceful devices such as music players or cell phones, and even beyond to full-fledged computers either in the vicinity of the user or as hidden servers. Of course, all these devices are supposed to cooperate regardless of whatever applications are deployed and without any prior knowledge of the specifics of such cooperation. The following sub-settings will help us to derive the technical requirements for a framework supporting Ambient Intelligence.

2.2.2. Smart Homes

Freeing people of the burden of housework through modern devices engineered by technology has long been a major driving force behind innovation. In the past elaborate devices such as power systems, lights, or washing machines increased the convenience and value of homes. The notion of prolonging this success story explains the great efforts manufacturers of household devices such as Philips are

putting into the area of Ambient Intelligence. Scenarios set in Smart Homes are thus common and easy to follow since they readily apply to most people in modern society. Most scenarios given in research projects quickly identify annoyances found in modern homes and provide appropriate solutions. Well-known examples include picture frames that always display the favorite image of the user standing in front of it. Or the incoming phone call that automatically mutes the music playing and follows the user from room to room. Unfortunately, these scenarios and their implementations have proven difficult to implement in a general, transferable, and extendable manner.

Even without a complete presentation of all the scenarios described in the various projects and without a specific analysis including a special focus on economic aspects, it is still safe to say that future smart homes are more driven by the services provided to the user than by the new devices themselves. In other words, users will tend to change and replace services utilizing existing devices rather than install new ones. Installing a new device almost always means introducing at least one new service to the smart home.

The services genuinely belonging to smart homes can be roughly divided into two basic groups: monitoring services, such as health or pet care, and home automation services such as automatic lighting or a simplified configuration of devices. These services are ubiquitously available without explicit user-interaction. Furthermore, the characteristics of these two basic service groups are special to the setting of smart homes, given the less strict requirements on privacy usually involved with user monitoring. By choosing the setting of the private home we can therefore focus our analysis on requirements for technical implementation.

The analysis of the above scenarios shows that the devices deployed remain static whereas the services installed comparatively interchangeable. Specifically, in terms of the kind of devices deployed we should be prepared to deal with a very broad spectrum ranging from small sensor nodes right through to in-house servers. The devices most probably come from a variety of vendors, but somehow must be able to function with each other. In general, the above requirements represent the basic technical requirements for AmI systems. The scenarios described below build upon these basic requirements and utilize their capabilities. Specific scenarios described in the setting of smart homes, can be, and are already being implemented, using conventional off the shelf sensors and actuators. However, these implementations require custom work to bring together the various parts and lack the universality of our approach. In this way they form the starting point for our framework.

Accordingly, we start with a trivial scenario for smart homes and its common implementation as now found in homes to analyze the issues that need to be targeted and to identify differences to future AmI systems. In this scenario motion sensors are used as input, and a simple light is used as an actuator which switches on if the

sensors detect movements in the room they are installed in. Figure 3 below depicts a centralized implementation of this scenario. As shown, several motion sensors are used in a sensor network for redundancy and increased accuracy. Even though this scenario is extremely simple on the drawing board, its actual implementation is highly complex as the aim is to give it maximum user-friendliness, e.g. sparing users all the hassle of configuration details.

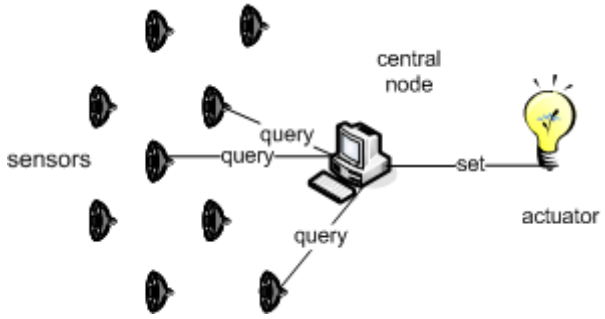

Figure 3: Simple scenario for Smart Homes with centralized implementation

Generally, we can observe that the application logic needed for this scenario, i.e. connecting the input sensor information with a simple rule to switch on the light, can be implemented on the motion sensors, the light actuator, or on a central external node monitoring and controlling all devices. The latter seems to be the pragmatic choice, as it offers flexibility of communication interfaces to access the sensors and the actuator, no resource constraints and a convenient way of interaction with the user.

However, centralizing the functionality on specific hardware and degrading other participating devices to peripherals renders the service unrealizable should this central node fail to operate. This approach also squanders vital energy in transferring sensor and control messages to and from the central node, and disproportionately binds up resources for a relatively trivial task. Nor does it scale with the number of interacting devices that have to be coordinated. Such an approach requires central knowledge about the sensors and actuators currently deployed in the network with wide ranging implications, i.e. each node must be directly addressable by the central node, requiring a form of address assignment with additional consumption of energy.

In this context, we may assert that the general value of smart homes increases with the number of sensors - i.e. input devices - available to the monitoring services. Thus the value of home automation increases with the number of services available to the user so that the possibilities for new and additional services increase not only with the number of sensors but in line with the number of actuators i.e. output devices.

Alternatively, this scenario can also be realized through direct cooperation of the devices involved in the service, i.e. the sensors and actuators, or intermediate nodes transmitting data in a multi-hop network. This means that much fewer resources, for e.g. communication and computing, need to be utilized than for service provision. Failure of one of the devices can easily be offset by a nearby equivalent device, and scalability is improved since spatially disjoint interactions do not affect each other, and sensor information can be aggregated within the network, again saving communication energy. This is especially important since not all sensors within a smart home network are likely to be connected to a power line but run on batteries instead. The implications of such networked sensor nodes have been extensively studied in the research field of Sensor Networks which we will also try to apply to our architecture.

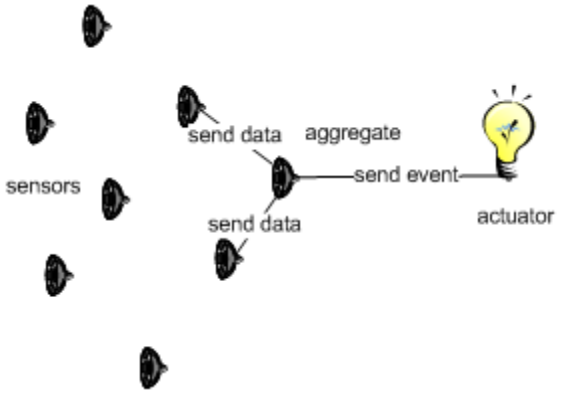

Figure 4: the decentralized implementation approach

Although the decentralized approach depicted in Figure 4, is more scalable and extendable and therefore more suited for future AmI environments, distributing the service logic in this way poses a number of new challenges to the implementation and the overall system itself. Firstly, general network connectivity is needed for a successful implementation of either approach. Most previous research assumed that connectivity is based at least on the IP protocol. With regard to the above scenario we do not need to be as specific as that. However, we do need to identify the minimum properties for network connectivity. These are:

- Nodes must be able to identify and address neighbors
- Nodes must be able to transmit messages to neighbors

Note that those properties are rather minor and comparable to the requirements of the MAC layer. The IP protocol, on the other hand, has much higher requirements, e.g. network-wide unique IP addresses. However, beyond this general network connectivity we also need a standardized access protocol for requesting and sending information such as sensor data or events from node to node. And since current

nodes just 'know' their direct neighbors, we also require a standard for addressing or ways to identify and discover objects, i.e. either nodes or services.

2.2.3. Smart Self-Configuration

Bearing in mind the AmI vision of ubiquitous computing with minimal user intervention, we can now extend the above scenario to address another important issue: self-configuration. As described above, neither can the sensor directly address the light nor can the light directly request the sensors, since both can only address their respective neighbors. This scenario also includes the (temporary) removal or replacement of either device. What we want is to allow additional lights or sensors to be easily included in the scenario. This can result in several lights being controlled through the motion sensors. Or other kinds of sensors, such as simple switches, being able to control the available lights.

Therefore this addressing scheme must be flexible enough to support those scenarios. However, for the configuration and interconnection of these nodes a flexible way of lose coupling is needed to accommodate new or changing nodes. This is especially true if these devices are not fixed in terms of the services they can provide. In this case, the overall system has to find an appropriate replacement for the now missing functionality. To achieve that, the overall system must be able to specify in a general manner what the user currently wants to be done and on a much lower level, what kind of service is currently requested.

2.2.4. Smart Services

The previous scenario introduced the notion of user and service requests, which we will discuss later in more detail, to dynamically accommodate device changes. Such service requests are used to address current system goals independently of specific devices. The following scenario further details the benefits of such loosely coupling via requests.

What we do here is simply extend the above scenario from Section 2.2.3 Smart Self-Configuration. Instead of lights switching on or off, we replace these actuators with jukebox devices that can play specific music files. We are now able to include the personalization aspects of Ambient Intelligence and require the system to dynamically select appropriate music instead of simply working off an a-priori playlist. Furthermore, the selection should depend not just on the taste of a single user but of all users present as well as taking account of the time of day. Since by its very nature such automatic music selection is not perfect, users can provide feedback that will also be included in this and further selections.

The analysis of this scenario yields several additional requirements. In addition to the rather technical requirements identified in the previous scenario, we now need a

new layer for addressing user-related issues like personalization, which generally subsumes all necessary service adjustments to better accommodate user preferences. Users not only must be able to specify and adjust their likes and dislikes, but the system itself must be able to combine several of such user preferences in order to correctly adjust the system's behaviour to all users present. Since we are still in the setting of smart private homes, the privacy issues touched on above are still very crucial. We therefore include an additional setting, the office, which will more explicitly bring this and other additional aspects to the requirement analysis.

2.2.5. Smart Offices

As outlined above, another important setting for scenarios is the office. First, in offices we are likely to encounter several users acting in new roles. Second, businesses have a higher demand for enterprise features such as redundancy and service recovery.

We thus extend the above scenarios to include such feature requests. Redundancy is implemented by several devices offering the same or a similar service. So instead of several devices fulfilling the same service request, i.e. lights switching on, we now want precisely one device to handle the request. However, in case the device is no longer available or unable to process the request, we also want the other devices to automatically take over. Utilizing the jukebox devices introduced in Section 2.2.4 Smart Services this scenario now includes several jukebox devices which are requested to play specific songs. To avoid confusion by the user, only one jukebox device is supposed to be playing songs. Of course, the jukebox device mentioned here can be easily replaced with any another arbitrary device offering this specific service.

The requirements for this scenario are similar to those of Section 2.2.3 Smart Self-Configuration. However, instead of addressing all the devices, we need to dynamically address just one device and include appropriate mechanisms for service recovery in case the active jukebox device becomes unavailable.

Of course, this scenario can also feature the personalized music selection introduced above. However, in this office setting additional privacy aspects must be respected. The following scenario not only tries to take account of this, but also gives an example of how to include services not specifically designed to work within this AmI system.

2.2.6. Smart Public Services

One important aspect of the previous smart homes and smart offices scenario settings is that they are user-controlled environments. This means that within the home, users can choose and select the devices and service they deploy. Likewise,

within an office environment, it is possible to ensure that compatibility between devices. In contrast to these environments at the other end of the spectrum stand service offered through a web interface. The next scenario will outline why and how these must be incorporated within a complete AmI architecture.

The previous scenarios generally differentiated between input devices and output devices. We have already exemplified the range of input devices, from simple motion sensors or switches to components delivering more advanced input information such as user preferences in music. Likewise, output devices can range from lights to jukeboxes. One common requirement we can derive here is that the basic information entity is rather a data or message, as opposed to a service or code. This has a number of advantages in terms of the technical requirements since we do not require a common execution environment and the security that comes with code-oriented approaches such as agent systems. Since we only use and transform data on the devices within the AmI network, we are able to include existing legacy services with little effort. The details of such integration are outline below. The following section describes a scenario that utilizes this property.

2.2.7. Smart Search

This scenario aims at providing context-aware recommendations to a group of users using today's internet search services. It is set in an office where several users in their professional roles are engaged in a business meeting. The goal is to use an existing legacy service that is advanced enough to make duplication of the service within the AmI framework unnecessary. The most obvious service now offered is the Internet search for web sites.

Over the past few years great efforts have gone into the indexing and ranking of search results. Search providers collect vast amounts of data from web sites and users to select and rank the most appropriate search results. Such efforts now include personalized approaches where users are required to log in and have their search history monitored and analyzed. However, the limits of such approaches due to privacy issues as well as to incomplete knowledge have long been widely discussed. In our scenario, therefore, devices in the user domain are used to do such personalization in a transparent manner.

In this scenario the users in the business meeting are searching for a place to eat in the evening. In this search the system is expected to include the personal food preferences as well as the business-related circumstances of the meeting. For instance, for a meeting including external partners or important customers the choice of restaurants will be clearly different from the kind of restaurants selected for internal meetings.

Once more, this rather simple scenario description allows us to picture various high-level implementation approaches. First, one could implement an own recommendation database that retrieves and evaluates local information such as the number and kind of people involved. Second, instead of costly implementing and maintaining such a database, one could try to extend existing databases. Unfortunately, this raises a number of privacy issues as well as insurmountable technical issues in terms of external access to internal, i.e. local and private, sensor information. We need instead the overall system to make a bridge between the internal collection and evaluation of sensor information and the external service invocation. While this is rather easy to implement for specific scenarios, we need to find a generic approach suitable for all kinds of services and information.

2.3. Layer Model for Ambient Intelligence

This chapter has outlined previous approaches to Ambient Intelligent Systems and pictured various scenarios whose generalizations are used to derive the technical requirements for the following specification of our architecture. Previous efforts highlight two important aspects - the use of scenario descriptions for a comprehensive analysis of the technical requirements and the need for layering the different problems when designing and implementing AmI systems. In short, we have derived the following five general layers as depicted in Figure 5 with the user above all as the focal point of this architecture.

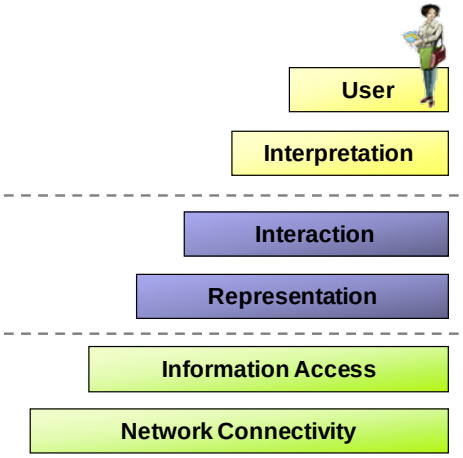

Figure 5: Required layering derived from scenario descriptions

The lowest layer, *Network Connectivity*, allows devices to discover and connect to their neighbors by sending and receiving messages – as introduced in section 2.2.2 above. On top of this we also require a generic information access protocol for retrieving data and manipulating the devices. The representation layer allows us to specify and address service requests without unique identification of nodes. The

interaction layer implements a loosely coupled service provisioning. In the top two layers we encapsulate the user and the interpretation of user demands and requests. We also need to specify, evaluate and possibly transform abstract user requests to specific service requests. User requests can be a search request for restaurants in the vicinity that take into account user eating preferences or a request for favorite music to be played on currently available jukebox devices .

The following chapters will use these requirements and explain for each layer which solution is best suited for implementing all of the above scenarios. We will start with a theoretical discussion of coupling in time, space, and representation, and the respective interaction models. This is followed by a discussion of the respective service models that includes an analysis of existing technologies. We will learn how to express semantic information for service description as well as how to formalize user needs and requests in semantic descriptions. We will then use this theoretical background to introduce a Service Request Oriented Architecture and an exemplary implementation of the same.

3. Theoretical Background and Related Work

With reference to the technical requirements identified in Chapter 2.2 Scenarios for an Ambient Intelligent World and the layer model introduced in conclusion, we now discuss the theoretical background and state-of-the-art technologies for each layer. For better understanding, we will start with the two middle service related layers – interaction and representation, given the new decomposition of traditional middleware approaches called for by the requirements analysis for Ambient Intelligent Systems. Traditional middleware systems tightly couple access protocol, data representation, and the service interaction model for the sake of application transparency. However, this brings with it inflexibility and a lack of scalability and flexibility for heterogeneous environments. Nonetheless, the need to deal with heterogeneous environments has spawned a number of new middleware protocols that enable invocation across system boundaries, and provide infrastructure services and a unified, abstract view of the system. We thus start by a general analysis of service interaction models that will lead us to a new interaction model and allow for truly loose coupling in space, time, and representation.

Against this background, we will shortly review respective service models for loosely coupled service interaction. We then describe various technologies for describing semantic information and appropriate mechanisms for the interpretation and classification of semantic data. As indicated in the requirements analysis, we will need semantic descriptions for the analysis of user goals and demands as well as for the decoupling of services in representation to enable ubiquitous and heterogeneous Ambient Intelligent Systems.

3.1. Coupling and Interaction Models

The interaction model specifies how data is exchanged among several participants in an interaction. It abstractly describes an interaction medium and the ways participants communicate via this medium. The medium defines how the handling of data is organized, i.e., it describes the conditions for a data exchange between two participants and enforces communication ways and base structures for the representation of data without naming contents and purpose. Control of the medium by participants is realized through a finite set of interaction directives. These directives are directly related to the functionality of the medium and allow the participants to publish data on the medium, to define how this data needs to be handled and even to administrate it within the medium. Take, for instance, a simple messaging system: the basic data structure is a message, the medium is a virtual channel between two participants and the interaction directives are *send*, to emit data to the medium, and *receive*, to indicate readiness for receiving data.

Cabri et al. classify interaction models in [17] according to the strength of coupling between the interacting parties, whereby time and space are regarded as the most significant characteristics. Given the heterogeneous setting of the computing environment, the following sections will also include considerations on the representation of data as drivers for loosely coupled interactions.

3.1.1. Coupling in Time

Temporal coupling typically requires all parties involved in an interaction to be synchronized in time. Delays in data exchange are taken as minimal and thus may be disregarded. In contrast, decoupling in time explicitly disregards temporal dependencies, i.e., delays in data exchanges are expected.

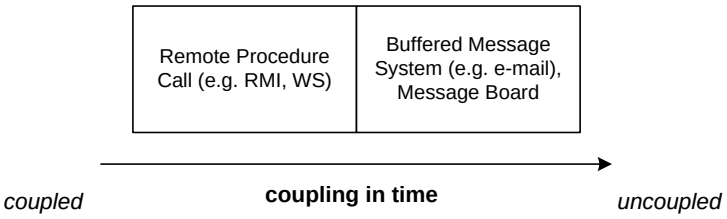

Figure 6: Classification of interaction models in terms of temporal coupling

A common example for a temporally coupled interaction is the Remote Procedure Call (RPC). Here one party performs a procedure that was previously requested by another one. Until the procedure is complete, the caller is usually blocked while awaiting results. RPC is especially applied in middleware intended to support the interoperability of heterogeneous systems while hiding the distribution. Technologies like Java Remote Method Invocation (RMI) [18] and Web Services are modern relatives of RPC.

Buffered message systems or message boards are examples of temporally uncoupled interaction models. The former includes ordered message storage system that is temporally coupled with the interacting parties and consequently prevents them from being directly coupled in time. Implementations of these systems include the Java Message Service (JMS) [19] and the common e-mail system. Message boards are similar to buffered message systems, only they do not provide a fixed order for the stored messages. Moreover, unlike common message systems, they require the addressee to poll a message instead of delegating it automatically.

3.1.2. Coupling in Space

Spatial coupling implies that the parties involved in an interaction know each other, at least by name. In contrast, a spatially decoupled interaction allows parties to stay anonymous. They are neither supposed to know each other, nor how many other

parties are involved in an interaction. Interactions are usually based on the contents of the data exchanged through the interaction medium. Spatially decoupled interactions are especially favorable to dynamic computing environments that do not rely on the addressability of their resources or need to support one-to-many communications.

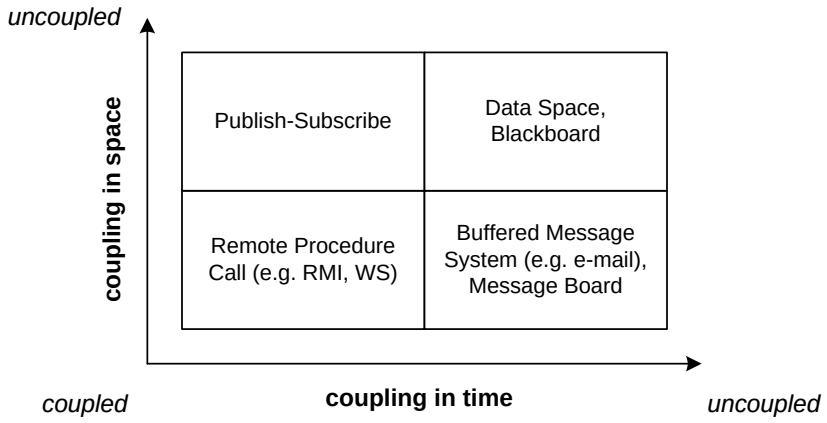

Figure 7: Classification of interaction models in terms of temporal and spatial coupling

Spatially coupled systems include message systems wherein sender and receivers are addressed directly in the message. Examples of such are once again the common e-mail system, or JMS. The most appropriate forms for the realization of spatially decoupled interaction models include publish-subscribe approaches, data spaces and blackboard systems.

Publish-subscribe approaches are anonymous messaging systems. They usually support two interaction directives, *publish* and *subscribe*. The *subscribe*-directive allows parties to sign in for the receipt of data from the interaction medium. While subscribing, the parties also define a filter criteria for messages they are interested in, and a callback address for notification in the occurrence of an appropriate message. Due to the undirected nature of message delivery, messages are rather referred to as 'events'. Emitting a new event is done with the *publish*-directive. The interaction medium is supposed to verify the filter criteria of the subscriptions on all new events, and to delegate events to the corresponding subscribers. Eugster et al. in [20] identify three major methods for event filtering, by topic, type and content. Type- and topic-based filter criteria depend on the events to be either typed in terms of programming languages or tagged with a key word, the topic. Both types and topics may be organized hierarchically. Hence, filter criteria are defined to match particular types or topics, or appropriate super-types and –topics. In contrast, content-based filter criteria refer to the data composing an event and may, for instance, be represented by conditional expressions or program code.

Publish-subscribe approaches can either be realized in a temporally coupled or temporally decoupled manner. Temporally coupled realizations require all subscribers to be available at the moment an event is published. Delegations of events to unavailable parties are discarded. Temporally loosely coupled approaches queue events for subscribers until they are once more available.

Data spaces in terms of Gelernter et al. [21] represent a spatially and temporally decoupled interaction model. They are similar to message boards, but the data they contain is not individually dedicated to certain parties. Rather parties interact while reading from the data space, adding new data, and manipulating existing data. Accordingly, the original interactions directives are *read*, *in*, and *out*, whereby *out* allows removing and reading data from the data space in a single step. However, data are retrieved in an associative manner like in publish-subscribe approaches, i.e., data units within the space are addressed by templates or queries. If there are multiple data units matching a template, one is chosen in a non-deterministic manner.

Recent implementations of data spaces like MARS [22], JavaSpaces [23] and EventHeap [24] include publish-subscribe functionality in addition to active interaction directives. Hence, a party is no longer required to continuously poll the data space to recognize changes within the data it contains. Instead, templates can be subscribed and newly inserted data is published immediately. Even so, one major difference between data spaces and publish-subscribe systems is the management of states. While temporal decoupled publish-subscribe systems queue data related to subscribers, in data spaces no distinction is made with regard to the possession of data. Thus, as long as data is kept within the data space, it is accessible to all parties.

An interaction model deliberately aligned to the creation of a common knowledge base is implemented by blackboards systems in terms of Erman et al. [25]. Blackboards are shared storages, similar to data spaces. Since blackboards are designed for collaborative problem-solving, the parties compile several solutions by alternately adding facts, i.e., data with a particular meaning to the blackboard. Blackboard systems are based on publish-subscribe approaches. Hence, parties can subscribe for changes within the knowledge base and may manipulate this knowledge base when notified. Concurrent manipulations are managed by a controller. The controller determines the next participant who is allowed to write to the blackboard on every change. During this process each party can view the whole knowledge base. Thus on the one hand the parties are required to interpret the blackboard data on each change they are notified about while on the other they need to understand the whole problem domain or at least a reasonable part of it. Unlike data spaces, blackboard systems represent data units that explicitly refer to each other rather than discrete data units without any relations.

3.1.3. Coupling in Representation

The contents of exchanged data are the significant drivers for interaction, especially in spatially decoupled systems. Although participants are not required to share a common namespace in terms of addresses, they usually need to agree on common data structures, vocabularies and vocabulary semantics to describe the data communicated among them. In an open, heterogeneous environment finding such agreement either requires the previous standardization of all these elements or loose coupling in representation. Loose coupling in representation calls for all parties involved in an interaction to only share minimal knowledge about an application domain and the terminology describing it. . Therefore, the language describing data during an interaction needs to be ad hoc interpretable, while the interacting parties should be able to understand it and conceive its semantics. Decoupling in representation thus addresses both representation of data and representation of meaning.

3.1.3.1. Representation of Data

A data structure defines a formal order for an atomic set of data units. Generally speaking, structuring of data benefits the exchange and processing of information through machines. While free text documents are regarded as unstructured, tuples, tables or trees are simple examples of structured data. More comprehensive representations of data structures and the dependencies between several segments of the structure are provided by relational and object-relational data models, originally introduced by Codd et al. [26]. These models are especially employed for large amounts of co-related data in databases. The appearance of such a comprehensive structure can be described by a schema.

However, while interacting in a distributed environment participants are required to either implicitly share a common schema or publish the one they will apply. This way other participants are enabled to process the corresponding data. Publishing means to refer to an open accessible schema or to embed the schema within the data submitted. Such approaches of self-describing data are referred to as ‘semi-structured’ as argued by Abiteboul et al. in [27]. Today’s most common format for semi-structured data is the Extensible Markup Language (XML) [28]. The XML standard defines how simple datasets can be structured and composed as trees. Vocabulary definitions and structural constraints on data, e.g., typing or the number of child-nodes per node, can be specified with XML Schema [29] for each document type. Additionally, the XML schema allows the combination of terminologies and structures from different document types in one single XML document. Thus any known scheme may be referenced and included, while namespaces support the differentiation of vocabularies from different schemes and the distinction of terms that are defined twice.

XML is designed to describe data structures in self-contained documents. Consequently, there is no common method of representing relations between several documents and the data they contain. If, for example, two parties in a temporal decoupled interaction exchange XML documents that need to refer to each other for representing a dialog, such reference needs to be represented in a proprietary manner. For this reason, in addition to XML, the Resource Description Framework (RDF) [30][122] was also introduced. Relations between resources in RDF are represented by triples that contain a subject, a predicate and optionally an object. Subject and object identify the related resources, while the predicate determines the kind of relation between them. The resources are either defined in place or referenced with Uniform Resource Identifiers (URI) [121][31]. Objects can also represent atomic data values (e.g. floating point numbers and character strings) or may even be left blank. Combining triples results in a directed graph that can span, if need be, the contents of several documents. A model graph is depicted in Figure 8, where resources and predicates are represented by simple names instead of complete URIs. However, the name Bob could also represent a URI like 'http://www.example.net/names#Bob'.

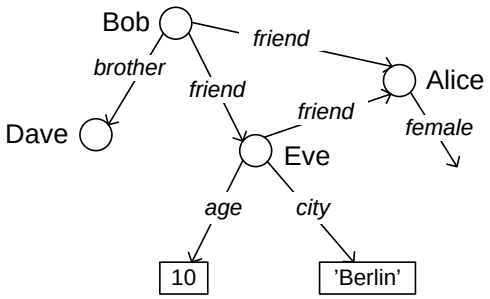

Figure 8: Simplified version of a graph in terms of RDF

3.1.3.2. Representation of Meaning

XML and RDF allow the representation of arbitrary structures and their relations, but do not include considerations about the meaning of either, as is argued by Berners-Lee et al. in [32]. There is, for instance, no common way to describe and consequently recognize the semantic equality of two entities if they have different names. The most powerful representation of meanings according to [33] are natural languages like English or German because they obviously allow us to describe anything. However, the structure of natural languages is too complex for their efficient utilization in today's computing environments. Instead the formal logic of statements is extracted to give a simple machine-interpretable representation. The domain of a problem actually defines which assertional elements are required for a reasonable description. For instance, representations of social networks between peoples require concepts like relationships, while workflows are based on a com-

mon understanding of time and concepts like before and after. The advantage of logical representation based on common concepts is that it offers a way to describe general correlations. Thus, from a given set of facts further ones can be inferred – for instance if cups are generally defined as having a handle, then a particular cup can also be expected to have one.

In terms of interaction models the semantics of exchanged documents are especially relevant. The interacting participants are required to describe the contents of these documents in relation to commonly accepted knowledge. Each one is then allowed to understand and verify the statements of others. One widely accepted form of logic aligned to the representation of static knowledge based on "is-a"-relationships is Description Logic (DL) as outlined by Baader et al. in [34]. DL is more of a category than a standard, and therefore always implemented somewhat differently. However, its common core elements are concepts, individuals, and role assertions. A concept is usually a superset of semantically equivalent things. Each concept is defined through combinations of other already existing concepts applying operators of set theory. For instance, a cup may be specified as equivalent to an intersection of tableware and fluid containers. An individual may be regarded as an instance of a concept. The relations between an individual and a concept are referred to as 'membership assertion'. To continue with the cup-example, a certain cup can be defined as an individual of the concept 'cup'. On the other hand, role assertions define the relations between several individuals. Hence, they allow us to state that a particular cup has a particular handle. Some DL-related logics additionally support definitions for types of role assertions on concepts.

Inferring in knowledge bases is a complex proceeding, so that initiating an inference process should be target-oriented. Hence, only when questioning for a certain fact should a search-process be triggered (e.g. simply breadth-first or depth-first), wherein all the known facts are successively combined until the question can be answered. A major benefit DL has unlike more expressive logics (e.g. FOL) is decidability. The elements of DL prevent an inference process from running ad infinitum, since there is only a limited set of possible combinations of facts.

Although inference processes are deterministic, the answer to a certain question depends on the assumption about the world reflected by the facts of the knowledge base. This world can either be closed or open [35]. A closed world is defined as being completely described by the known set of facts. Within an open world, however, the existence of more facts in addition to the known ones is implicitly assumed. Given, for instance, the facts that each handle belongs to one cup only and handle A belongs to cup B and to cup C. In a closed world this definition causes inconsistency in the knowledge base. In contrast, in an open world the inference process would need to assume that cup B and C are one and the same, even if this fact is not explicitly given. Another more abstract example is the distinction between good and evil. In a closed

world, everything that is not good can assumed to be evil. In an open world no conclusion can be drawn for things that are not good.

3.1.3.3. Loose Coupling in Representation

Loose coupling in representation is defined above as the need for composing self-describing data representation with a framework for the reasonable description of the required semantics. Based on RDF new standards have emerged to improve the interoperability of distributed systems through extended semantic descriptions. One standard that includes a definition set for Description Logics is the Web Ontology Language (OWL) [36]. OWL DL is intended to support interoperability in open environments and is thus based on an open world assumption. According to Berners-Lee et al. paradoxes and unanswerable questions are the price that must be paid to achieve versatility [32]. Some core elements have been given different names. For instance, role assertions are referred to as properties and concepts are called classes. However, the caption 'individual' has been kept from the original terminology of Description Logics. OWL DL also introduces restrictions on properties. These restrictions may either constrain the range of a property when applied on a certain class, or define the cardinality of the property. Classes are still defined in relation to other classes through concepts of set theory (i.e. union, intersection, complement, etc.). Nevertheless, the term 'class' was not arbitrarily chosen since there is also an option for sub-typing classes, similar to inheritance hierarchies in object-oriented programming languages.

3.1.4. Loose Coupling

Loosely coupled interaction models weaken temporal and spatial dependencies, while requiring only a minimal correspondence of data representations supported by the participants. Temporal decoupling can be achieved through buffering data between originator and receiver, i.e., by utilizing queues or shared storages. Either way there are differences in the conditions under which the state of the interaction medium is kept, i.e., data are buffered. On the one hand data may be buffered on behalf of the receiver so that the interaction medium needs to be aware of the receiver, as is realized in publish-subscribe approaches. On the other, data may be globally buffered, as for instance is realized by message boards. This way the interaction medium is not required to know potential receivers for newly originated data. Spatial decoupling can be achieved if the originator of data is not required to address a receiver of these data directly. The receiver is rather supposed to be addressed by the content of data. For that purpose, data may either be tagged with keywords to allow a fast classification, or the data contents themselves need to be interpretable. Hence, especially for content-based interactions, a consensus about the representation of data and the expression of knowledge about the application domain is required. For representing data in open systems, semi-structured descrip-

tions have proven suitable. A starting point for the description of data contents is logic. Forms of logic differ in the type of information they enable to describe, i.e., the axioms they provide. However, elementary concepts and their relations can be effectively described with Description Logic.

Spatial and temporal decoupling are reasonably combined in data spaces. They allow content-based retrieval of data and support real temporal decoupling, since they are not required to be aware of the receivers when new data is submitted. Moreover, the interacting participants do not need to interpret all data within the space to find relevant data as required by blackboard systems. Even so, adding considerations about loosely coupled representations through semi-structured semantic descriptions to the original concept of the data space does result in a more flexible interaction model.

This concept is realized in the semantic tuple space sTuples [37]. In sTuples each tuple is more an entire document, represented in RDF and DAML+OIL [38] (a predecessor of OWL) rather than a list of attributes. According to the original tuple spaces directives, sTuples supports insertion, retrieval, and withdrawal of documents. Retrieving and withdrawing documents is realized in an associative manner, i.e., through content templates. Although the represented data is based on RDF, the strict separation of data units hardly allows for the representation of relations among single documents. Thus relationships spanning multiple documents may only be described implicitly, and there is no way to retrieve a structure spanning several documents at once. Consider, for instance, the case of three documents within the space, the first defining the color blue, the second defining the color yellow, and the third describing the color green as a result of mixing the first two while referring their definitions. In sTuples there is no way to retrieve the entire conclusion as a closed set of data. This feature is especially relevant if the context of an interaction needs to be reproducible, i.e., if a third party needs to understand how green was composed. Another critical aspect of this scenario is the withdrawal of data. Although data withdrawal allows the synchronization of several participants by preventing tuples from being read twice, the removal of documents containing assertions referenced in other documents may result in inconsistencies.

Further approaches for semantic data spaces are TripleSpace [39] and Semantic Web Space [40]. Both are inspired by tuple spaces, but regard a single RDF triple as a basic data unit. Hence, the space itself constitutes the document whereby the document contains a coherent data representation instead of numerous independent data units. Consequently, manipulation of the spaces is also based on operations addressing single triples. Indeed, structures in higher level logic descriptions like OWL are usually composed of multiple RDF-triples. So, on the one hand, a single operation call may not suffice to reach a consistent representation of information in terms of the higher level description within the data space while on the other re-

trieval of a set of triples representing a reasonable document is impossible. Nor does either approach provide access methods for reactive retrieval, e.g., in terms of change notifications. Participants rather need to continuously poll the tuple spaces to recognize changes.

3.2. Service Models

The service model defines the actual service provisioning and therefore the subject of an interaction. Specified are the roles of the involved parties and their duties, the documents that need to be exchanged as well as the sequence and conditions for the document exchange. The service model defines a guideline for the participants of the service provisioning that guarantees successful service execution.

3.2.1. Traditional middleware

The interconnection of heterogeneous devices with common middleware platforms, such as CORBA[116], UPnP[63], or Jini[10], enables them to expose their services and properties to peers as objects, and to have them accessed by external entities. Let's consider a possible solution for the exemplary scenario described in Section 2.2 – Scenarios for an Ambient Intelligent World – realized with traditional middleware approaches such as CORBA. The task to switch on the light according to input values of specific sensors merely requires agreeing upon an appropriate interface for the light node and the sensor nodes, so the application logic can first query the sensors' state and, second, invoke the appropriate method on the light node when the sensor data indicates motion.

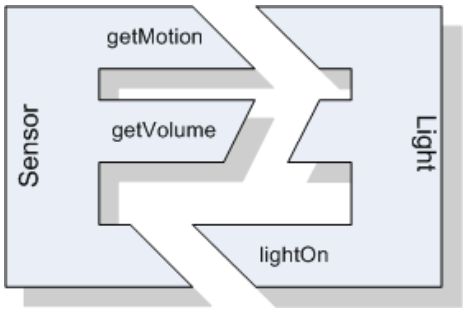

Figure 9: Sensor and actuator as CORBA objects

However, agreeing on an interface is no easy task considering the number of nodes that might be connected to a switch, not to mention the complexities of more advanced devices. In fact, to invoke the "lightOn" method, the same interface definition must be present on both sides, coupling the components unnecessarily and making them incompatible with independently developed components. Figure 9 illustrates

the interconnections of distributed objects with specific interfaces: both parts match each other, but independently developed components cannot be connected.

Alternatively dynamic invocation can be utilized, allowing one of the devices to discover the interface of remote components by their names and parameters. This, however, further burdens the nodes, already constrained with running a middleware, with a runtime analysis of their counterpart's interfaces. Furthermore, this approach will only work as long as the appropriate means for the discovery of the counterpart are known at compile time. Most traditional systems either rely on an externalized discovery service, such as UDDI[11] with UPnP, which requires an additional network load, or on a standardized discovery interface that all nodes must comply to, and which is hard to change once a critical number of nodes are deployed.

In addition to the invocation syntax, participating components also have to be configured at the control node. One possibility could be using one of the general discovery protocols, such as SLP, SSDP or a proprietary discovery service offered by the middleware. Even so, the user is still required to make the final decision as to which devices are to be connected. As devices such as light bulbs or sensors have little means to interact with the user directly, the configuration needs to be performed via an external client, translating user input to the protocol understood by the target device.

Finally, the interconnection of components is restricted to configurations foreseen by the vendor or application developer, making integration of arbitrary off-the-shelf components a game of chance. One could argue that this is a question of standardization; however implementations of compliant software components on constrained nodes are often impossible, given the complexity of the middleware and the level of transparency required by applications.

This trivial scenario illustrates the shortcomings of traditional middleware when dealing with resource-constrained, loosely coupled systems. The main issues are:

- Overly restrictive interfaces restrain interactions in pervasive systems to the extent the application programmer has foreseen.
- Complex invocation semantics depend on interface definition at compile time, and prevent implementations from being tailored to the capabilities of the application.
- Binary interaction protocols prevent the use of generic user clients and tying services to predefined libraries.
- Lack of semantic description of interactions and data other than implemented at the application level
- Fixed roles of participants defined at compile-time prevent an extension of the service due to fixed interfaces.

- Breakable identifiers which basically live and die with the referenced object and the publishing server, or require a naming service to be running.

3.2.2. Super Distributed Objects

The I-centric Communication Reference Model outlined above in Section 2.1.1 was specified, standardized, and implemented in detail as Super Distributed Objects by the Object Management Group (OMG). The above scenario involving sensors and lights together can be implemented using Super Distributed Objects (SDO) and thus solves some of the problems outlined in the previous section. The SDO specification defines generic interfaces for configuration, monitoring of status changes, and service invocation, atop of which concrete services can be implemented. Figure 10 illustrates concrete nodes and services thereon wrapped through SDO interfaces.

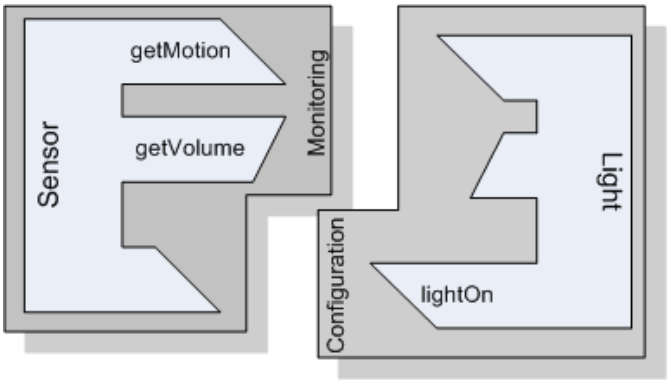

Figure 10: Sensor and actuator wrapped as SDOs

Applied to the exemplary scenario, the Monitoring interface reduces the problem of extensible interface definitions to a pair of subscribe and notify functions. Thereby, the node encapsulating the light subscribes at the switch, and acts as an input sensor to be notified of any status changes in its SwitchState property. Upon reception of the current status via the notify method, it maps the received value on its own LightState property.

The largely semantic-free invocation of the light's functionality brings us way ahead in interconnecting independently developed components. Even so, the subscription and notification methods are still defined in the IDL file and considerable effort is involved in invoking them via dynamic invocation as there is a lack of a common interface definition at compile time. In addition, dynamic invocation still requires the configuration of the invocation target with the application merely verifying user decisions by querying the interface repository.

Secondly, the interpretation of incoming data is hard-coded in the application. The light SDO subscribes to a particular parameter and maps the incoming data onto its own properties, making it compatible with just one kind of switch. Once again the

problem could be avoided by redesigning the application to make it more configurable and setting the mapping at run-time. Yet this wouldn't make the application any more extensible or flexible.

Finally, connecting the light to other sensors that could produce different kinds of data - such as a motion sensor providing Boolean values, and a light sensor producing integer numbers - has not been considered at all. With both of them invoking the notify method, the light service is forced to dispatch the input based on incoming values, making an agreement on keywords, such as 'dimValue' or 'switchState' necessary.

Summarizing we can say that a generic interface, independent of application defined syntax, is suitable for interconnecting independently developed components. However, the implementation as one function dispatching requests also introduces additional dependencies. Instead of dynamically identifying the appropriate methods to call, application logic has now to determine the appropriate property names to subscribe to or send data to. Since these are ordinary string constants in the SDO implementation, they can be easily exchanged. However, advanced transformation or mapping of different properties is still not possible given the lack of semantic annotations.

3.2.3. Service-oriented Architectures

Service-oriented Architectures are characterized by open interfaces described in a platform-independent manner. Actually there are only a handful of technologies that innately fulfill these characteristics. The most appropriate candidates are UPnP [63] and Web Services [05]. Interactions in these technologies are strictly separated by the two processes responsible for finding an appropriate service and invoking a found service - service discovery and service execution. While the structures of the documents and their exchange sequences are always the same in service discovery, for service execution both aspects are individually defined by a service description provided in the discovery phase. A service description is offered by the service provider and the instruments for describing services are specified by the corresponding standards. Hence, the standards also constrain the range of representable information in the documents and the appearance of document exchange patterns. In addition to a plain explanation of the service interface, the description may also define the semantics of a service, usually according to some special ontology. The service models of the UPnP and Web Service together with service semantics and their representation are discussed in detail below.

3.2.4. Universal Plug and Play

UPnP is a simple service architecture that mainly addresses service executions between hardware devices in ad hoc environments. The standard defines services to be owned by devices, a fact that is especially reflected in the composition of entities as utilized in service descriptions. The services themselves contain a set of actions and state variables. Each action addresses some logic executed on demand by the service provider wherein the action is intended to change a state variable of the service as a side-effect. Since the documents exchanged in UPnP are atomic and self-contained, they can rather be considered as messages.

Service discovery in UPnP is designed to match the requirements of dynamic ad hoc environments. The processes are aligned to manage a lack of knowledge about the names and addresses of devices involved in the same environment. The UPnP standards provide two ways to discover devices in the environment and consequently also their services. First, each device may continuously publish Advertising messages via group communications to all listening receivers. Second, a device may send a parameterized Discovery message to all other devices in the environment to search for candidates with special capabilities. Each device supposed to be addressed with the Discovery message is intended to respond with an appropriate response message. Discovery messages describe a certain subject to be searched for such as devices or services. To this end the message contains some kind of search pattern to address a particular device or service type, and the maximum time to wait for a response.

However, service discovery is neither able nor intended to address devices or services by advanced semantic descriptions. Identifiers used in advertisements and discovery messages are thus pre-defined by the UPnP forum. They are, however, extendable due to the XML encoding of messages.

Service execution in UPnP is realized as a simple request-response message exchange. Once a service consumer has recognized the address of an appropriate service provider for the required service, and obtained and interpreted the service description, it may send an Action Request message to the service provider - i.e. the corresponding device. The device is required to answer this request either with a respective response, if the action is finished successfully, or otherwise by reporting an error.

3.2.5. Web Services

Web Services (WS) are special resources in the World Wide Web providing any functionality to be executed on demand. In the WS architecture the actual service is an organizational construct that composes a set of operations. An operation represents a single, executable piece of the overall service functionality, similar to the actions in

UPnP. Thus each operation addresses a particular interaction between service provider and consumer.

Like any other resource on the web, each WS is given a global identifier, the Uniform Resource Identifier (URI) [31]. Since URIs are intended to be globally unique, they usually also serve as synonyms for addresses in a network. Hence, a URI allows the service consumer to address a WS independent of location and time, and always in the same manner. The Web Services Architecture specification [05] does not need to include specifications for distinctive discovery mechanisms like UPnP. The description of the service interface is self-contained and identified through the particular URI that also serves as locator for the actual document. The relation between service description and service interface is represented by a special part of the service description – the service grounding.

However, although not specifically needed during run-time, there are approaches for discovering Web Services, i.e. to discover the URIs of requested services. These serve as a first approach to loose coupling since these approaches merely store and return URIs and therefore act rather like directories or registries. The discovery systems are usually designed similarly to search engines in the web. A widely accepted system for finding Web Services is OASIS's Universal Description, Discovery and Integration standard (UDDI) [11]. UDDI is based on a registry, i.e., it has a central infrastructure for listing services. Unlike a directory, services are required to register themselves with the UDDI system. These registries basically manage sets of business-dependent meta-data about WSs that refer to service URIs. The meta-data structures describe such things as service families and addressed user roles (e.g. personnel administration). Each meta-description is usually given a unique identifier or some keywords for categorization. The actual discovery - i.e., locating a WS in the registry - is done by matching identifiers and keys introduced in the meta-data. Thus a potential service consumer is required to search for an appropriate service or service provider by naming a meta-data structure that describes them. UDDI defines a heavy-weight interface with numerous operations for doing so. The interface itself is again represented as WS. The strength of UDDI is the way it enables service descriptions to be browsed from an abstract point of view. However, UDDI does not directly help to find a service through its functionality; it rather depends on some annotated meta-data.

The documents exchanged between service providers and consumers in WS are referred to as messages. A finite set of messages, but at least two, are exchanged for performing a particular service operation. The supported message types are declared in the service description as well as the direction enabling a certain type of message to be sent either from the consumer to the provider or vice versa. As mentioned above, the message exchange is spatially coupled which means that the originator of a message addresses the receiver directly. However, messages basically

contain a name and some content. andthe message name uniquely identifies the message in the context of a particular exchange sequence. This way each participant involved in the interaction may verify whether the exchange sequence defined in the service description is upheld by the others or not. Message content can describe any form of structured data such as lists of typed values, graphs, or trees as long as the appearance of the data structure was previously defined or referenced in the service description. The meaning of a message in the context of a particular interaction is only addressed by its content which makes it application-dependent.

In general the flow of messages between service consumer and provider during the execution of each operation is defined as a static sequence in service descriptions. There is, for instance, no possibility to define that two messages of different types can be send alternatively at a certain point in the message flow. The only type of message that may disrupt the flow is a fault message. Fault messages may be sent when a party is unable to communicate an error condition inside the normal message flow or when a party wishes to terminate a message exchange and consequently the service execution. The actual error description is application-dependent. However, message flows may additionally be constrained by some properties and features describing consumer and provider behavior in detail. Features are usually optional attributes defining, for instance, reliability and security aspects supported by the service provider and that can be utilized by service consumers. Properties are non-functional attributes of a service and define the number of retries for a message transmission in the case of network failure. Unlike the features, the constraints declared by properties should be met by the service consumer.

3.2.6. Discussion

UPnP and Web Services fulfill the criteria of service-oriented architectures, although both technologies address very different application domains and seem to be able to support loose coupling as derived from the requirements analysis in Section 2.3. While UPnP is dedicated to numerically limited, dynamic service environments, Web Services are rather intended for fixed, large-scale settings. This difference is clearly visible in the service discovery process. During this process the service consumer tries to find an appropriate instance of a required service, i.e., some party providing the service so that the service consumer is required to interpret the service descriptions of all candidates itself or needs to resort to a third party lookup-service.

UPnP is designed for a highly dynamic environment setting wherein each party providing a service is required to continuously announce itself. This way potential service consumers are always kept informed about the capabilities of their neighbors. In the WS world each service has a globally unique identifier with which it can be addressed. Additional service registries help the service consumer to find a service with the required capabilities, similar to search engines on the web. However,

these registries usually require the indexed services to be annotated with categorizing key words.

In UPnP as well as in WS the service consumer needs to interpret a service description that specifies in detail how to interact with the service. A successful service execution depends on the addressability of the service instance cited in the service description and on compliance with the interaction scheme. However, the spatial coupling of service consumer and provider causes repetition of the service discovery if a service instance can no longer be found. As suggested above, the Web Service Architecture is not designed to cope with a dynamic service environment, but in UPnP as well continuously repeated lookups in consequence of disappearing service providers may place a high burden on the physical resources of the service-consuming system or even the entire environment.

UPnP and WS also differ in terms of the constraints placed on interactions during service execution. Whereas in WS neither the number nor the contents of the exchanged messages are limited, in UPnP each action is initiated by exactly one message and concluded with another one containing the results. Moreover, in UPnP the structure of each message is oriented to a list of parameters. In practice most WS operations are also based on simple two-way message exchanges, i.e., a request and a response, since they are mapped to functions or methods of the programming languages they are implemented in.

The service model of UPnP primarily addresses features and functionality of devices, and statements about the device states affected by a service execution are explicitly included in the service description. Although even WSs often change the state of the system running the service, the service description does not allow corresponding explanations. With pure WSDL description [12][124] this means there is now a way to express whether a service changes a state or only retrieves some information. Both technologies therefore lack the proper means to describe service semantics as needed for the layer reference model introduced in Section 2.3.

3.3. Semantics of Services

As outlined above, loose coupling in representation requires description of service semantics. Usually, in service architectures like the ones introduced so far the characteristics of the service models - i.e. the range of supported interactions - are solely defined by standards. Each service is specified by a description, explaining how to handle service execution appropriately. Thus the description includes which messages are to be exchanged between the parties involved in the service execution and how the data structures in the messages are to be composed. All entities communicated between the parties are given a specific handle identifying an entity either as globally unique or at least as unique within the scope of that particular

interaction. Thus it is guaranteed that service consumer and provider share a common understanding of the messages and the data they contain. However, the actual meaning of this data in the context of a certain application domain can only be represented implicitly. Service description languages must offer additional description elements to annotate service interfaces with such features as natural language comments for developers. Additionally, separate specification documents addressing service developers are required to provide a detailed understanding and a view of the possible side effects of what exactly this service is doing and in which situations it may used. Unfortunately, explaining services with closed, stand-alone descriptions conflicts with the intention of SOA which is the loose coupling of service consumers and service providers. If two services addressing the same problem are described with different terminologies, their semantic equivalence cannot be automatically recognized. An application that is designed to bind a specific service to solve a particular problem at run-time needs to know the exact name of the service type.

This shortcoming was the main reason behind the development of ontologies describing the semantics of service interfaces and functionality such as OWL-S [14], WSMO [15], and SWSO [13]. Ontologies for the representation of service semantics basically comprise of two types of descriptions. First, they explain the context of application and the service environment in which the utilization of the service is reasonable and permitted. Secondly, they describe the processes performed during the service execution and the flow of input and output data. For that purpose semantic description are either provided as separate documents referring to a particular service, such as OWL-S or as inline annotations within the technical service description as proposed in WSDL-S [41]. Today's semantic service descriptions are designed to achieve three major goals. First, they are intended to support the automatic discovery of services by explaining the service capabilities in detail. Secondly, they are supposed to benefit the automatic execution of services while dispensing with the need of applications to implement fixed services terminologies. Thirdly, they are intended to enable an automatic service composition to solve abstract tasks. The correct treatment of semantic description to meet these goals is usually realized by special match-making frameworks, like IRS II/III [42], METEOR-S [43], or the Semantic Web Service Architecture [16].

As argued above, service semantics can be regarded from two different points of view, a state-based perspective and a process-based one. The state-based view describes service perquisites and behavior with regard to the state of service environment and the changes made to this state during the service execution. The process-based view describes the activities that will be performed when executing the service and the data flow between service consumer and service provider.

3.3.1. State-based View

The definition of the state-based view comprises of concepts referred to as preconditions, assumptions, post-conditions, and effects. Preconditions and post-conditions address the direct information space of the service, i.e., all information the service accesses explicitly or creates during execution. Assumptions and effects, on the other hand, affect the informational state of the environment the service operates in, i.e., the context of its execution. However, it is important to realize that all these concepts abstract from data flows such as input and output. They only apply to plain information and its corresponding implications, both before and after service execution.

A precondition defines which information is actually needed to invoke the service and the criteria this information shall fulfill. Consider, for instance, a service that is bound to a mobile displaying device like a tablet-pc and allows for the viewing of video clips. A precondition for such a service could claim that there is initially a clip named to be shown. Assumptions describe the conditions that need be fulfilled in the informational state of the environment before a service will be executed. In the example above an assumption could define that the displaying device needs to have enough battery power to play at least a reasonable part of the clip. Post-conditions describe the state the information space will have after a successful service execution, e.g., through defining how the preconditions will change. With reference to this example, a post-condition could describe that the playback of the addressed clip has either started or is already finished, depending on the service logic. Effects again describe the expected changes in the context of the service environment independently of the states explicitly affected by the service. In our example an effect could describe how much battery power was consumed during the service execution. In OWL-S neither preconditions and assumptions nor effects and post-conditions are regarded separately. Instead, there are only preconditions and effects, whereby the effects can be constraints with conditions. These conditions serve as guards for the effects and basically address the purpose of assumptions.

Since descriptions of conditions and changes in the environment state require mathematical elements such as comparisons and functions, the preconditions, post-conditions, assumptions, and effects need to be represented in an expressive type of logic. However, as the complexity of reasoning processes can be considered as proportional to the complexity of the descriptive logic, in the context of strongly constrained physical resources, expressive logics are expensive as are the descriptive elements introduced.

3.3.2. Process-based View

The process-based view describes a service execution in terms of the internal processes and the data flow between service consumer and service provider, i.e., the

inputs and outputs of the service. The inputs and outputs of a service can be compared to the parameters and return values of functions in programming languages. Inputs are concrete information provided to a service instance for execution. On the other hand outputs are information produced by the service during execution. For instance, in WSs inputs and outputs are the messages respectively received and originated by the service. Semantic descriptions of these data structures enable an interpreting party to understand their meaning in the context of the application. If the semantics of all data structures composing a complex input for service are known by the service executing system, the meaning of the input can be inferred from the meaning of the single data structures.

The relations between input data, output data, and the execution flow of a service are explained with process descriptions. These descriptions specify the process that corresponds to a service execution, i.e., the single activities internally performed and the dependencies among them. There are mainly two approaches for the representation of the execution flow: workflow structures and state machines. Whereas in OWLS-S and SWSO workflow structures are favored, WSMO utilizes Abstract State Machines, in terms of Gurevich [44]. Both approaches starting from an initial situation allow a description of how to, achieve a particular aim under particular external conditions, inputs, and outputs. The expressiveness of both approaches is similar, but while state machines focus on modeling the states before and after execution of sub-processes, workflows rather model the sub-processes themselves and the conditions for their execution. In either case the description of the service internals enables an interested party like a potential service consumer to determine how a service works and how it will behave in a given situation.

Service execution processes can be described from two different points of view; one addresses the service choreography, the other the service orchestration, as argued in [45]. The service choreography is an abstract description only representing those activities that are required to understand the interactions with the service. Choreographies allow a service consumer to adapt its informational state while running the service, and to recognize at which point in the execution particular inputs are required or outputs produced. Thus the choreography describes an exchange pattern for inputs and outputs. Specific to this pattern is its conditional nature as the sequence of inputs and outputs may vary from service execution to service execution depending on the information exchanged. Let us return to the service that allows the playback of video clips on a mobile displaying device: if the application logic behind the service determines that the codec for processing the clip is not locally available, it may obtain permission to retrieve the codec from the internet before the playback is started. In this case additional outputs and inputs would be required, e.g., a request and some acceptance or denial.

While the service choreography only contains abstract activity descriptions that suffice to familiarize a potential reader with state changes and exchange patterns, the orchestration models the real sub-processes composing a service execution. Unlike the choreography the orchestration addresses less the service consumer than the planner of composite service executions that are needed for mapping complex business processes. From the orchestration can be inferred, for instance, which real life processes a service runs on execution, how these processes depend on each other, and if they refer to other services. This information may be utilized to plan and schedule service bindings and executions.

The description of service choreography and orchestration is neither new nor specific to semantic service descriptions. For Web Services there exist standards like the Web Service Choreography Interface [46] or the Business Process Execution Language for Web Service [47]. However, a semantically enhanced representation of processes and process-dependencies allows a party to reason the service functionality without exactly knowing the terminology and concepts employed in the description. Indeed, the definition of both vocabulary and concepts needs to be reducible to a common description about the application domain. If this is not possible, some kind of mediator is required. Moreover, semantic descriptions of processes again require expressive logics that support the representation of dynamic knowledge. To represent conditional branches or state transitions, comparisons and functions are needed. For this purpose OWL-S utilizes rule languages from the semantic web stack. SWSO and WSMO are rather based on First Order Logic derivatives as specified in [13] and [15].

3.3.3. Discussion

Service semantics can be described from a state-based perspective and a process-based point of view. Ontologies for semantic service description like WSMO, OWL-S, and SWSO combine both aspects to address a wide range of applications. However, both views are based on the assumption that a service changes states, either the state of the system executing the service, or the informational state of the consumer, or both. Hence, a service can be regarded as an abstract state transition.

A state transition function may be represented as a set of constant mappings, but any increase in the number of mappings required to represent such a function also raises the complexity of the description. Therefore the relation between source and target state is primarily described in a relative manner. The state transition is represented as a composition of multiple primitive functions and operators linked with variables that serve as placeholders. Consider for instance a service for booking a flight ticket that is related to the occupancy state of the corresponding flight. To represent that after execution of the service one less seat can be offered for sale, some kind of ‘decrease’ function is needed. However, to represent functions and

variables, expressive logics are required. For that purpose in WSML and SWSO derivations of First Order Logic [33] are utilized in OWL-S the Semantic Web Rule Language [48]. Expressive logics may be comfortable, but their correct use is difficult and their processing is expensive. As the search trees in the inference processes usually become very large and some problems are not decidable at all, semantic service descriptions are intended to be processed by comprehensive service integration and planning systems, like IRS III or METEOR-S. But running those systems, in turn, requires powerful hardware.

Planning and execution frameworks are also intended to process the descriptions of service choreography and orchestration. Description of service choreography is usually only reasonable if the interactions during execution are non-atomic. In UpnP, for instance, there is no conditional message exchange pattern; there are only action request, response, and fault messages. Service orchestration describes the concrete implementation of a service in terms of executable sub-processes so that orchestration is rather a useful aid for the party that actually runs the service instance. A service consumer could at best use it to interpret how the service works and which other third party services are included. However, the description of choreography and orchestration requires special ontologies defining the meaning of processes, workflows, and their relationships. These descriptions are rarely simple, and terms are needed to represent conditional branches. The decision to enter a particular branch depends on a guard that needs to be evaluated, and guards are again based on operators and functions.

Descriptions of service semantics are normally defined above common service descriptions and service architectures. Layering supports the reusability of one service ontology for multiple implementations of the same service in different service architectures. For instance OWL-S can be grounded to UPnP or WS, as realized in [49]. Furthermore, the parties actually performing the service execution are not forced to understand the service semantics. But limiting the use of expressive, descriptive instruments to an abstract view on services does not directly benefit the interactions between service consumer and provider. Both are still required to utilize the same representations for data they want to exchange. Solely a mediator translating the data based on the semantic descriptions of inputs and outputs could make good this shortcoming.

3.4. Context-awareness and User-Level Service Adaptation

The previous section introduced semantic descriptions for expressing the inner workings of services. According to our layer reference model for Ambient Intelligence introduced in Section 2.3, semantic data can also be utilized for user-level service adaptation, commonly described as context-awareness. As outlined above, the challenge of context-aware service adaptation is the ability to utilize services

agnostic to Ambient Intelligence, i.e. legacy services. This not only reduces the implementation complexity of these services through a separation of concerns, but also allows the Ambient Intelligent System to pool these service adaptations to accommodate not only single users but complete user groups.

This section discusses the respective fundamentals and related work and is organised as follow: section 3.4.1 outlines both the potential for adapting context-agnostic legacy services with model web search engines, and the possibilities of parameter adaptation while section 3.4.2 discusses the Semantic Web, a technology for user-centric semantic descriptions.

3.4.1. Adaptation of Context-agnostic Services

The Internet offers a huge number of services typically solely characterised by their input parameters and output results. One of the most used service categories is web search services such as Google, Amazon, or Yahoo! that are able to search on request in a large indexed dataset for the occurrence of user-entered input. Several smart search and rating algorithms are used to improve the precision and recall of the returned results. Precision and recall are measures rating the relevance and completeness of the search results. The higher the number of relevant documents returned is, the higher is the recall. The lower the number of returned non-relevant documents is, the higher is the precision. In most cases the desired web search result is a maximum in recall and precision. The deployed algorithms benefit from research in the fields of information retrieval (IR) and information filtering (IF). Leaving aside the differences between information retrieval and information filtering [87], both aim at the selection or elimination of documents from a database with the help of user profiles, queries, relevance feedback, or similarity measures.

Service adaptation refers to the influencing of service behaviour in a manner currently directly or indirectly desired by the user. There are several possibilities for adapting services, including respecting different user preferences or terminal capabilities. In general, services can be adapted in two ways: internally and externally. The first option, internal adaptation, refers to self-adaptable services where service functionality changes according to certain conditions. In this case the logic responsible for the adaptation process resides within the service, whereas the conditions usually describing the user context are implicit or explicit input parameters passed during service execution. The external adaptation refers to components located outside the service implementation that are responsible for processing the service input or output in an appropriate manner.

One advantage of utilising an integrated adaptation component is that no additional component is required for service adaptation. Furthermore, the respective provider is able to control the adaptation process which facilitates support and maintenance.

On the other hand, with internal service adaptation the respective adaptation logic is closely connected with the service itself. Therefore it is often difficult to transfer the integrated component to other services or applications. Similarly, it is hard to add additional and independent components regarding security issues and other services or application areas. Another disadvantage of an integrated approach concerns the lack of transparency that may increase the user's doubts about the usefulness of the service adaptation.

These drawbacks can be avoided with the help of an external adaptation component if designed in an appropriate manner. However, as external adaptation does not directly adapt the service, but rather its input parameters and output, it is possible to adapt a service without accessing its actual implementation. Furthermore, an external component is able to consider further aspects such as deciding for a service based on certain conditions before adapting the inputs or presenting the outputs in an appropriate manner - an aspect that enables service composition.

As mentioned above, service adaptation can be done with respect to various needs and aspects. On the one hand, the adaptation process is intended to directly influence the contents of the service results. On the other, a further purpose of service adaptation is to enable the choice of a proper presentation format in line with terminal capabilities. For example, HTML may be transformed to WML to present a web site on a mobile phone [114]. In this context, the kind of adaptation assumes that certain information is available, e.g. user preferences or terminal types.

3.4.2. Semantic Web Related Technologies

The Semantic Web [72] is meant as an extension of the traditional web, modelling explicitly the semantic data typically hidden from non-human interpreters within the content of web sites. In order to make the information machine processable and utilisable, the Uniform Resource Identifiers described in RFC 2396 [88] and the Resource Description Framework (RDF) [89] were specified as the base for the respective semantic annotation of documents.

The envisioned benefits of semantic descriptions of (thus-far) only humanly comprehensible information on web sites are that, as outlined above in the description of Ambient Intelligence, future computer systems should be able to understand the context of a web site. Given such understanding, the ratings of web sites in search engines could be vastly improved. Furthermore, since all information is understood by all the systems involved, the common representation of semantic annotations enables the set-up of semantic networks and the assembly of content currently belonging to different areas. In this manner, the Internet will evolve to become an increasingly collaborative instrument.

In this context, ontologies as well as the Web Ontology Language (OWL) are presented in Section 3.4.2.1 and Section 3.4.2.2. Classification and inference mechanisms are closely related to ontologies and are presented in Section 3.4.2.3 and Section 3.4.2.4.

3.4.2.1. Ontologies

Ontologies are one of the main building blocks of the Semantic Web. They are used to represent and structure the semantics of things in a machine processable way. An ontology, as far as information technology is concerned, is an explicit formal specification of objects or concepts and their relationships. It is typically in the form of hierarchically structured data whose key concepts are listed in the following.

Classes are general concepts that are physical or virtual objects of interest. As in object-oriented programming languages, it is possible to define *individuals* of classes with concrete names and values. Classes and individuals are related to each other with the help of *properties*. The hierarchical structure of ontologies is accomplished by defining subclasses which give a tree-like structure. The general attributes of the classes defined by properties are linked to specific values by individuals in terms of a domain and a range denoting the type of classes the relationship refers to. There are two types of properties: object property and data type property. Depending on the property type, an individual is linked to another individual or a data value. With *restrictions* it is possible to affect the affiliation of individuals to a class. These restrictions are used in the classification process discussed in detail further on.

3.4.2.2. The Web Ontology Language (OWL)

The Web Ontology Language (OWL) [70] is a standardised language written in XML for the purpose of describing ontologies. OWL offers some advantages over other technologies for ontology description:

- It is written in XML and therefore human readable text. Each person who understands the language is able to understand at least the structure of a given ontology without having to use a specialised editor.
- It is standardised and already supported by a number of tools. Because of standardisation assurance is given that all valid OWL files in the Internet can be processed by these tools.
- OWL ontologies can be distributed, merged, classified, and checked for integrity and validity.

In terms of the set of vocabulary and corresponding limitations, OWL can be divided into three groups: OWL Lite, OWL DL, and OWL Full. OWL Lite does not support all the OWL language constructs, but provides tool builders with some basic functionalities of OWL. OWL DL, which is labelled according to the research field of description logics [34], makes the maximum expressiveness of OWL available, requiring that all

conclusions are computable and ensuring decidability. Each valid model in OWL Lite is also a valid model in OWL DL. OWL Full has no restrictions as far as the OWL vocabulary is concerned and comprises of OWL DL.

Figure 11 shows the process of development for the Web Ontology Language. The Extensible Markup Language (XML) serves as a base for the ontology language and was recommended in 1998. The Resource Description Framework (RDF), which was recommended by the W3C in 1999, is an approach to represent resources of the World Wide Web (WWW) utilising XML syntax. With the help of RDF, metadata of web resources can be expressed in a subject-predicate-object notion. RDF Schema (RDFS) extends RDF and defines a basic type system as well as offering a way to specify descriptive elements for a class of resources or to express constraints. The DARPA Agent Markup Language (DAML) [118] uses XML and RDF to allow classification and inference. Another similar approach is the Ontology Inference Layer (OIL) that is compatible with RDFS. The efforts of these projects resulted in DAML+OIL and provided a machine-readable and understandable language to describe information in 2001. Further research resulted in the W3C recommendation of OWL in 2004 making a standardised ontology language available and triggering many initiatives to extend OWL, e.g. OWL-S [82] for web services.

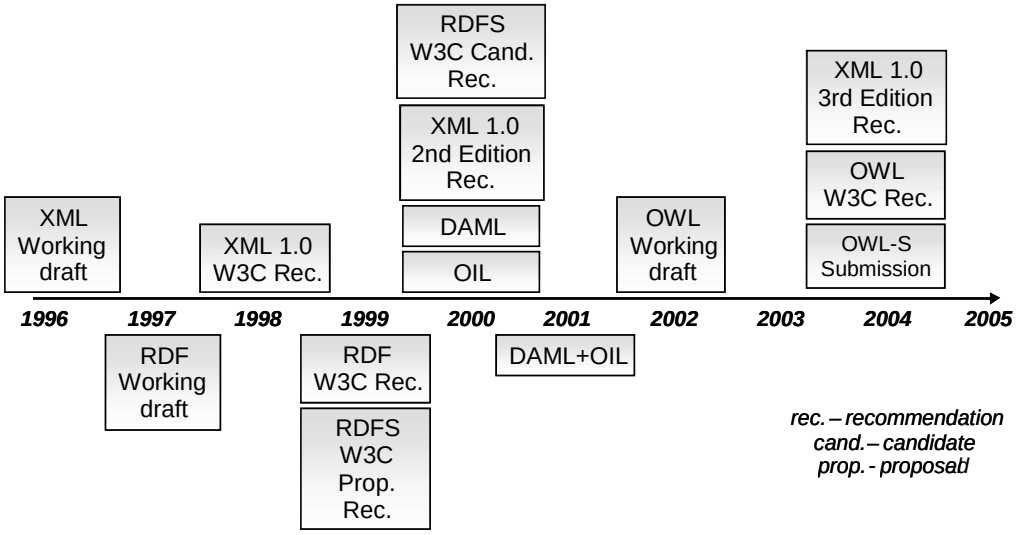

Figure 11: Timeline of the ontology language evolution proposal

Since OWL is written in XML it is possible to edit OWL files with a simple text editor. However, there are some more sophisticated ways of editing ontologies and rules comprising a lot of different approaches. One of the most famous graphical editors is Protégé [92][93]. This ontology editor is open source and can be extended with several plug-ins, e.g. with plug-ins for OWL, OWL-S, and SWRL, the Semantic Web Rule Language [95].

Another useful tool is the Semantic Web Development Environment (SWeDE) [94]. SWeDE is an Eclipse plug-in that provides syntax highlighting, ontology validation, Java interface generation, and other tools for ontology editing.

In addition to the vast variety of editors and environments for OWL, there are several frameworks that can be used to process it, including Jena [101], OWL API [51], and KAON2 [102]. Basically, each of these frameworks is able to load and process ontologies, i.e. to add, create, or remove classes, properties, and instances. Furthermore, most frameworks support classification and inference issues, although often in a restricted way. For example, KAON2 is able to cope with SWRL ontologies while Jena has its own rule syntax.

3.4.2.3. Classification

Classification refers to the determination of class hierarchies and the categorisation of objects according to the restrictions of a given ontology. It is typically done by a reasoner program such as RACER [71] or Pellet [54] for OWL. Besides the creation of classes and subclasses that give a class hierarchy, the definition of restrictions enables reasoners to detect implicit knowledge residing in the ontology and state new facts accordingly.

A restriction refers to a class of individuals fulfilling certain requirements. In this respect, OWL has two types of property restrictions. A “value constraint” restricts the type of a property value, whereas a “cardinality constraint” is a restriction on the property’s cardinality. An example with omitted namespaces for a value constraint is given in Figure 12.

```
<owl:Class rdf:ID="Situation"/>
<owl:ObjectProperty rdf:ID="hasParticipant">
        <rdfs:domain rdf:resource="#Situation"/>
</owl:ObjectProperty>
<owl:Restriction>
        <owl:onProperty rdf:resource="#hasParticipant"/>
        <owl:allValuesFrom rdf:resource="#Friend"/>
</owl:Restriction>
```

Figure 12: Example for a value constraint

First of all, the class Situation and the object property hasParticipant are defined. The restriction on the property hasParticipant requires that all property values be of the type #Friend. Figure 13 shows an example of a cardinality constraint on the property #hasParticipant where the cardinality of the property is restricted to two.

```
<owl:Restriction>
        <owl:maxCardinality rdf:datatype="#nonNegativeInteger">2</owl:maxCardinality>
        <owl:onProperty rdf:resource="#hasParticipant" />
</owl:Restriction>
```

Figure 13: Example for a cardinality constraint

There are some other property restrictions regarding these two types, which are detailed in OWL Web Ontology Language Reference [70]. OWL also provides some optional global property restrictions, such as the possibility of stating that there is only one unique property value.

The classification capabilities for OWL are limited in some ways. For example, OWL has restrictions on the comparison of data type property values [98]. Without extending the ontology with XML Schema Datatypes [100], it is not possible to check restrictions for whether a property value is above a certain value or not.

Another restriction that cannot easily be solved with the help of classification can be illustrated by a simple example: if a badge is in a certain room and the badge belongs to a specific person, the person is also located in the room.

Thus, another means is needed to be able to dynamically express coherences, namely inference rules that can be used in combination with OWL classification.

3.4.2.4. Inference

The inference process refers to the application of a set of rules to a knowledge base, e.g. an ontology, for facilitating the deduction of new facts. A rule typically consists of a body and a head. The head determines the conclusion on condition that the body is true. In this respect, a rule is comparable to an 'if-then' statement as it is accepted in traditional programming languages. However, a uniform rule language and a tool for its interpretation do enable the easy and flexible modification of the corresponding conditions and conclusions without having to program.

There are a lot of different approaches for rule languages, but none of them is currently standardised. Some languages are directly related to the ontology language, e.g. SWRL to OWL, others are specific to an API, e.g. the rule language used by the Jena API.

The Rule Markup Language (RuleML) [104] is a rule language written in XML and based on Datalog. The Semantic Web Rule Language (SWRL) [95] is closely related to RuleML while mixing it with aspects of OWL. In this manner it is possible to embed rules directly in an OWL ontology. However, the XML dialect requires a lot of text and constructs in order to define a rule. One framework that is able to cope with SWRL is KAON2 [102]. The Jena Semantic Web Framework [101][103] has its own representation for rules that are applied to the internal ontology representation. One important disadvantage of the specific rule syntax is the necessity of using Jena

API for processing. Furthermore, unlike SWRL, Jena rules can only be embedded in OWL with the help of a special property or by making appropriate extensions to OWL.

Even so, the vast variety of rule languages triggered the development of a framework that can be used to harmonise the different representations. In this respect, SweetRules [105] is able to cope with Jena rules and SWRL. Once installed SweetRules can be used to combine the capabilities of different rule languages. Thus, it is possible to understand rules from different sources without having to define a fixed rule language.

3.4.2.5. Processing Rules and Inferred Statements

In general, rule-based systems consist of a knowledge base and an inference engine as described by Hayes-Roth [106]. The knowledge base is a storage comprising of facts and rules, whereas the inference engine is responsible for interpreting the knowledge. Depending on the derived facts, further actions can be triggered. The basic components of a rule-based system and the data flow are depicted in Figure 14.

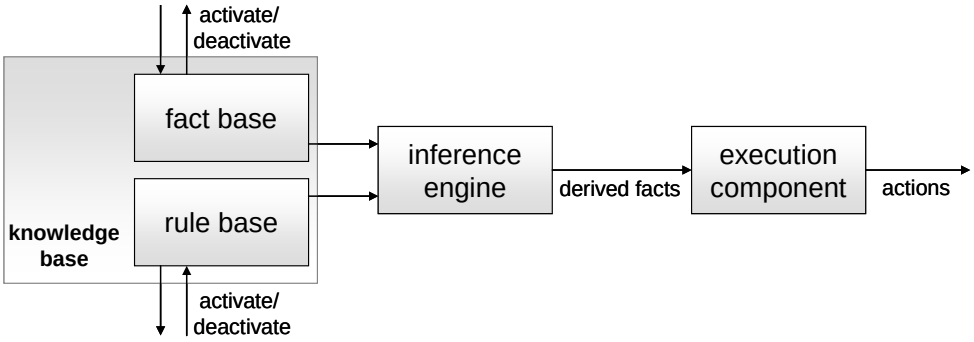

Figure 14: Basic components of a rule-based system

The outcome of the inference process can usually be affected by changing the rules, activating and deactivating certain facts, or altering the behaviour of the inference engine. In this respect, derived facts may either be added to the knowledge base or processed separately by an execution component which strongly depends on the purpose of the system.

3.4.3. Context Provisioning

This section deals with several approaches and applications related to the provision of context in services. It classifies existing ideas into three categories: extended services, context-aware systems, and information retrieval. Each domain has its specific key points, advantages, and disadvantages, which are mentioned in the respective sections.

3.4.3.1. Extended and Combined Services

This section depicts approaches that extend or change extant service logic to enable context-awareness. As this idea is closely related to the internal adaptation of services depicted in Section 3.4.1, most of the approaches have its respective benefits and drawbacks. An example for this approach is the new ranking mechanism proposed by Google [108] which includes a "PersonalisedScore" in addition to the well-known "PageRank". This score is calculated with the help of explicit or implicit user profile information, e.g. user entered preferences, previous queries, or selected results.

In terms of search services, there are also some approaches that combine existing services. For example, Yahoo! Maps [109] may be regarded as a map service extended with a search engine. To date, it is restricted to the United States and can be used to find restaurants or similar things on a selected map. It depends on user input, but can be useful to filter out search results based on the location.

However, extended services are application specific and often very restricted in terms of the use of context and its gathering, and representation.

3.4.3.2. Context-aware Systems

Context-aware systems aim at the provision of multiple services and often provide separate components intended for certain tasks, e.g. context modelling and service invocation. Unlike extended services, context-aware systems aim at supplying several external applications with different tasks for context-aware information. An overview of existing approaches to a context-aware system is given by Baldauf et al. in [97]. To give some examples, two approaches that use ontologies and inference are shortly discussed in the following.

The Service-oriented Context-Aware Middleware (SOCAM) aims at the "building and rapid prototyping of context-aware mobile services" [110]. However, this approach focuses on the acquisition, discovery, and dissemination of context using a simple ontology that covers some general aspects as well as concepts relevant to the current domain. Services interested in specific context information need to actively discover it with the help of the framework's service. Furthermore, service developers are able to define rules that trigger service methods should the conditions be met.

Another example for an architecture incorporating OWL ontologies and inference is the Context Broker Architecture (CoBrA) [111]. It aims at providing context-awareness in smart spaces such as meeting rooms. In CoBrA a central component, the *context broker*, is responsible for all the tasks related to the context including acquisition, representation, interpretation, and policy management. The context broker communicates with devices, services, and agents in the respective intelligent

space in order to update the knowledge base and invoke typical services such as dimming the lights or controlling devices.

However, as these systems do not aim at adapting existing services that expect sophisticated and multifaceted input in a semantic and pluggable manner, they do have drawbacks, when it comes to a flexible service description, the semantic representation of service input and output, and situation-specific user interaction.

3.4.3.3. Context-Aware Retrieval (CAR) and Query Expansion

Context-aware Retrieval (CAR) refers to the use of context information in the retrieval process. Jones and Brown [119] depict a number of aspects related to this topic, e.g. the interactive and proactive retrieval paradigm. The first paradigm denotes a request explicitly initiated by the user, whereas the second refers to annotated documents that are automatically delivered to the user if the context matches. They build an experimental system to verify the ideas of CAR based on their own retrieval engine. Basically, certain weighted fields, e.g. for location and temperature, are filled on the basis of the predicted context, and retrieved documents are matched with the user profile afterwards. The documents are then delivered to the user should the results still be needed.

Storey et al. analysed the usefulness of user profiles in the process of expanding user queries [86]. In addition to ontologies and lexicons that are used to disambiguate the query and to determine the user's domain knowledge, profiles divided into frames, slots, and values also enrich the original query. If a predefined frame, e.g. a restaurant, is identified with the help of the current query, the respective preferences, i.e. the values of the slots, are added, e.g. vegetarian. In a model restaurant setting they prove that the results of the extended queries are more relevant than those of the original queries.

In terms of the disambiguation of queries, much research has gone into considerations of concept relationships, domain knowledge, or the past behaviour of the user. These approaches are often referred to as query expansion or query enhancement [112][113] and consider the context of the query items entered by the user rather than the user context itself. For instance, if a word the user has entered bears two contradictory meanings, respective domain knowledge can be used to determine the user's intention. Hence, additional words are added to the query or the word itself is substituted by another.

3.4.3.4. Summary

As the description in the previous section shows, CAR needs to be extended in many ways to offer a context-aware framework which is able to adapt several existing services. Storey et al. have already proven the usefulness of this approach when applied to search services and user profiles. Thus, the strong points of context-aware

systems depicted in Section 3.4.3.2, i.e. context representation, inference, and the use of distributed context sources, need to be combined with the ideas of CAR. Some other aspects such as a more comprehensive profile representation, a reasonable knowledge base structure, a flexible service description for pluggable services, and context-dependent user feedback also have to be considered. In short, the benefits of these approaches have to be combined, so that the need for implementing domain specific context-aware services from scratch can be generally avoided.

3.5. Conclusion

In this chapter we have studied the technologies most suitable for the layer model for Ambient Intelligent Systems. Whereas each layer required an individual set of technologies, we observed that semantic descriptions can be utilized for describing and deriving the context situation the user is currently in, and the subsequent context-aware service adaptation as well as for a representation-independent description of services. We will describe both these uses in Chapter 4.

We also presented the theoretical background for decoupled service interaction models and their existing service models ranging from traditional middleware systems to newer service-oriented architectures such as web services. [06][07][08] However, even though already designed with abstraction from specific devices in mind, the latter is not able to fulfill the requirements outlined above since it still relies on specific interface descriptions which are neither replaceable nor adaptable during run-time. This means that this tightly coupled interaction model of web services is neither able to cope with the continuously changing appearance of the computing environment nor with the physical characteristics of the devices and the networks that constrain the reliability of interworking. Thus, the development of services in pervasive computing requires more consideration of the dynamism of the computing environment and context-aware service adaptability.

We therefore introduce a new paradigm called *Service Request Oriented Architecture* (SROA) that is better suitable for Ambient Intelligence Systems. SROA is a further development of distributing service components which adds adaptability and service requests as novel underlying paradigms. The Service Request Oriented Architecture is based on a message-like communication model in which the interworking of devices is realized through the exchange of *Service Requests*. In contrast to traditional approaches in which service interaction is realized through the direct invocation of methods according to a well-defined interface specification, the service request-oriented approach relies on the capability of the system to autonomously interpret, adapt, and deliver these requests to suitable service objects. This communication paradigm calls for a complex communication model which not only supports formal function calls but also supports high-level colloquial descriptions of user needs.

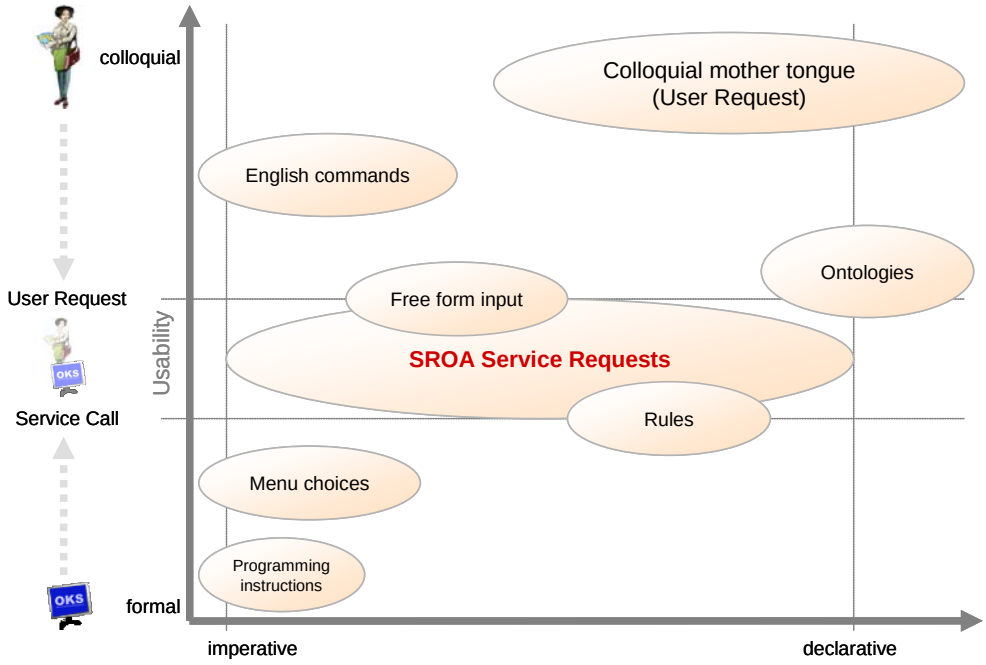

Figure 15: Communication dimensions

Whereas traditional approaches rely on specific formal interface descriptions, the Service Request Oriented Architecture tries to open up this limitation by supporting mapping from high-level user requests to specific service calls as depicted in Figure 15. The full potential - but also the challenges - of this approach depend on the quality and the possibilities of such mapping. However, allowing communicating computing systems to better understand the intention of the user will generally facilitate the design of more adaptable and user-friendly systems.

4. Service Request Oriented Architecture Specification

The Service-Request Oriented architecture enables the execution of service requests in a loosely coupled, distributed, and ad-hoc manner. Below we will describe the interaction model, and how the initial Service Request is analyzed and executed. Up to now, temporal coupling has generally required all participants involved in an interaction to be synchronized in time. Communication time is neglected and delays in data exchange are assumed to be minimal. As outlined above, one common example for a temporally coupled interaction is the Remote Procedure Call (RPC). Here one party performs a procedure that was previously requested by another. Until the procedure is finished, the caller is usually blocked while awaiting the results. Uncoupled interaction models can be realized by queued messaging systems or shared message storages (e.g. message boards, etc.). Likewise, spatial coupling implies that the participants involved in an interaction know about each other and can be individually addressed by name. A spatially decoupled interaction, on the other hand, enables participants to stay anonymous. In addition to temporal and spatial decoupling, we would also like to introduce representational decoupling, whereby the messages exchanged, i.e. the Service Requests, Generic Service Descriptions and Service Calls, are generically described in an extendable format like XML with no need for every participant to understand every part of the message.

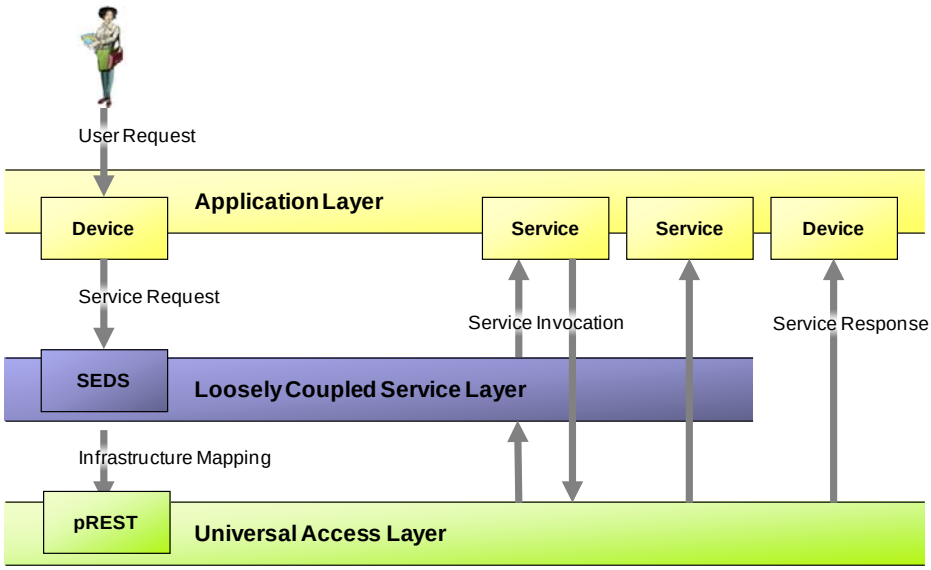

Figure 16: Components in the layer model

Figure 16 shows how the components of Service Request Oriented Architecture interact in the layer model introduced above. The application layer holds the application domain logic implemented in either device applications or back-end services. The basic tasks of Service Request routing and transformation are accom-

plished in the loosely coupled service layer which holds all requests and distributes them through the semantically enhanced data space (SEDS) which utilizes the universal access layer to request information. The following sections describe these layers and the tasks accomplished within them.

Apart from introducing the general pREST access layer architecture for universally accessing data from all devices, we specifically address two important aspects of the user-specific relevance of data that to date have received only insufficient coverage: the semantic meaning of the user request and the context of the respective user, e.g. the location, user specific preferences, or current weather. The allocation and use of context information pertains to the design of the service itself. The automatic adaptation of existing services is preferable to the development of a context-aware service for each domain.

Finally we present a solution that is able to provide the user context needed to adapt services with the help of profiles and rules. A simple semantic service description facilitates the easy and flexible use of existing services, while an intuitive approach is developed that enables the user to influence the adaptation process and respond to the outcome.

4.1. The Access Layer

This work investigates the application of Web like interactions and data description, commonly referred to as the Representational State Transfer (REST) architectural model for the interconnection of independently developed heterogeneous components. The goal is to provide the same simplicity and holistic view of services and data that the World Wide Web introduced to the Internet.

In the course of this work, we shall analyze the interconnection paradigms enforced by traditional middleware platforms, and illustrate their shortcomings when applied to loosely coupled, heterogeneous systems. We also present a number of desirable characteristics for middleware for resource-constrained sensor nodes and massively distributed systems based on current sensor network and pervasive computing research.

We take the concept of a resource as meaning a typed, uniquely addressable communication endpoint that can be used as an abstract addressing scheme for devices, data and services in pervasive systems, and we further define a common set of operation semantics as methods to access and manipulate resources.

Subsequently we evaluate the representation of data and description of components with domain-specific XML vocabularies to aid the user in the composition of nodes to provide higher level services. To support the validation of component assembly on the system side, we introduce MIME types to specify the transmission encoding of data, and as an extensible mechanism to describe application level types.

Extending the standard interaction pattern proposed by REST, i.e. synchronous document submission and retrieval, we introduce asynchronous delivery of events and data to support extensible and scalable assembly of components and realize in-network processing and data-centric routing schemes.

The reason for these shortcomings is seen in the strict coupling of interactions between the parties involved in service provisioning. Therefore, the coupling of interactions is analyzed with regard to temporal dependencies, spatial dependencies, and dependencies in the representation of exchanged data.

4.1.1. The Concept of Resources

Objects are the predominant abstraction of data and functionality in software engineering. An object is defined by its state, behavior and identity [69]. Implicitly the object model assumes that implementation details are hidden behind the object's interface, and that state can only be modified by using the object's provided methods. Middleware platforms such as CORBA or Java RMI try to preserve this abstraction for distributed applications as well, by hiding the remote invocations behind the interface of an object.

In service-oriented architectures [61], on the other hand, no interface is exposed to clients and no state is held within the server component. Rather the state is transmitted within the messages, and all computation is performed on the message content. Interoperability is ensured through message syntax which is common for the whole component rather than a single method.

REST takes a different approach. In it the primary abstraction of information and functionality are resources. Any information that can be named is a resource: a document, a function, a device configuration or a property. Fielding [09] emphasizes the equivalence of a resource's representation and its identifier in a view which is influenced by the synergistic relationship of URLs and documents on the World Wide Web. For component interconnection, the interactive aspect of resources is more important. A resource in the sense of component interconnection is a uniquely addressable entity, service or data whose content can be retrieved via a uniform mechanism. Its representation contains the information needed to access and manipulate the resource itself and the resources nested therein.

While a resource has state and identity just like an object, interaction is performed via a uniform interface for all components. An operation on a resource is performed by sending a representation (or an identifier) and possibly receiving another representation in return, as in service oriented architectures. In fact, resources have more in common with distributed objects than with services. Nevertheless, resources encompass the abstraction of information, like images, documents and structured data, as well as functionalities like services and methods.

In pervasive systems, the dynamic nature of resources needs to be considered along with uniform accessibility. Dynamic properties such as sensor values and their counterparts in actuators produce or consume constant streams of data. This interaction pattern allows an ad-hoc federation of components to form a dynamic, autonomous system, but in turn it also requires runtime negotiation of resource contents and transmission encoding, thus placing additional constraints on the description, addressing and typing of resources.

4.1.2. Resource Representation

We have defined a resource as a uniquely addressable, universally accessible, typed communication end point. Based on this characterization, a platform independent way to address and access resources and represent their content needs to be defined.

4.1.2.1. Resource Typing

The interconnection of arbitrary resources introduces a requirement to verify such compositions and to report meaningful errors should a user try to connect incompatible resources. As resources are abstractions of data, their type is defined by the encoding understood by the resource and the application level types it produces or consumes.

Interactions in heterogeneous systems require a common format for the transmission of values; however, the variety of data that can be encountered in pervasive environments ranges from logical values such as Boolean or numeric types to media types. As it is unlikely that an image producer will ever be communicating with a consumer of numeric values, it is reasonable to specify specific encodings for media types and logical values.

On the World Wide Web multipurpose internet mail extension (MIME) types define a common yet extensible set of standard encodings, and cover the whole spectrum of media types produced or consumed by electronic devices. The respective encoding should be enclosed in the resource meta-data. Limited by the scope of this work, the transactions treated here will be considered to involve only textually encoded values such as plain ASCII and XML.

The data types produced or consumed by a particular resource are defined at the application level. Barton et al. [57] propose the usage of Accept and Provide headers to agree upon a common data format. However, the values allowed by them only define the encoding type, whereas the interconnection of components producing and consuming logical values needs specification at the datatype level. The following section introduces a common set of data types to interconnect resources.

4.1.2.2. Primitive Values

The components treated here exchange logical values rather than multimedia types. For a platform-independent representation of values one has to decide whether to transmit data as binary data or in ASCII form. While the former is more efficient in terms of packet sizes and processing overheads, it also introduces issues of byte ordering and sizes of data values.

ASCII encoding has proven successful for data interchange on the web as it offers platform-independent representation without byte-ordering issues. What's more, it is human-readable and the default input is provided by off-the-shelf clients.

To ensure compatibility of interfaces, a common set of basic data types need to be defined. The XML schema specification [128] defines one such set consisting of about 19 primitive types "believed to be so common, that if they were not defined in this specification many schema designers would end up 'reinventing' them", and a lot more derived types. Such an abundance of types is overkill and impedes the interconnection of sources and sinks, as it reduces the number of compatible endpoints.

Accordingly, as a basis for the exchange of values, a subset of the primitive types defined by XML schema is taken as common to all components. The set of primitive types comprises:

Type	Allowed values
int	representing integer values
float	for floating point numbers
string	for textual data
boolean	contains binary values
enum	holding one of several predefined values, the actual values need to be retrieved from the interface descriptor.

Table 1: Primitive data types

Additionally constructed types may be exchanged by components in ASCII representation, only this is considered bad design since it undermines efforts to ensure interoperability among devices.

4.1.2.3. Complex Data

Whereas parameters and simple properties can be represented sufficiently well with primitive data types, complex values and user interfaces need richer forms of presentation. Atop of ASCII encoding of values, a representation is needed that contains semantic tags as well as actual values.

For platform-independent, self-describing representation of arbitrary data XML is a favorite choice. Although encoding with ASN.1 [125] offers platform independence

as well, and binary transmission according to the Basic Encoding Rules is more efficient in terms of network traffic and computational overhead, XML [126] is still preferable for a number of reasons.

- the same syntax is utilized for data description and transmission
- descriptors are extensible without invalidation
- descriptors are human readable
- generation of user interfaces via Web-forms or VoiceXML is made easier
- complementary standards for interlinking, transformation and querying are given
- additional semantics can be specified with RDF and OWL

Even choosing XML as the encoding for structured data, different schemes or conventions for tags to be used can be distinguished using shared, predefined tags - also known as meta-schema mapping - and using instance-specific tags - also known as schema mapping.

Meta-schema mapping categorizes elements by one of their aspects, while the elements' distinct characteristics are specified as attributes or tag contents. An exemplary meta-schema mapping for pervasive systems might define a set of tags consisting of component, data source, data sink and description and require participating components to describe their capabilities in such terms. The motion sensor in the exemplary scenario would be described as:

```
<component>
   <description>Sensor capturing motion and
volume</description>
   <source name=”motion” type=”boolean”/>
   <source name=”volume” type=”int”/>
</component>
```

Schema mapping defines custom tags for each data instance, while providing category information such as types or classifiers as attributes. Encoding the sensor in the exemplary scenario according to a schema mapping would yield a document containing motion, volume and light as child nodes of sensor, also specifying them as being capable of producing or consuming data via attributes. A descriptor for a sensor might then read:

```
<sensor description=”Sensor capturing motion and volume”>
   <motion type=”boolean”/>
   <volume type=”int”/>
</sensor>
```

The advantage of meta-schema mapping is easier validation and integration since specific data values are contained in parts which are not subject to validation. As tags are defined for a class of documents, documents can be syntactically verified against a common schema. Moreover, such a mapping scheme reflects the intention of XML which is to specify the meaning of data in the tags and its value in the tag's

content. The requirement for this kind of mapping is, however, that elements of a distinct type can be meaningfully subdivided into categories by a certain aspect.

The more universal a description scheme, the less specific is the information that can be derived from tags. A sensible trade-off between expressiveness and usefulness for integration is given by domain specific vocabularies or ontologies. An ontology would be specified as an XML schema, wrapping the representation of instance data in XML and values expressed in their ASCII representation.

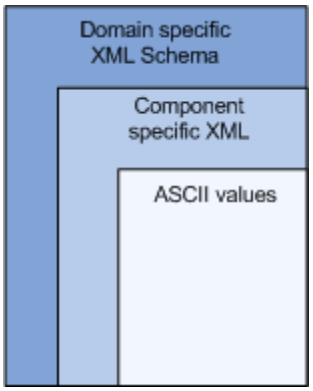

Figure 17: Representation of complex data

There are a number of initiatives that define ontologies for particular domains. The Dublin Core Metadata Initiative [127], for instance, is an endeavor to standardize a set of elements to describe web resources in such terms as title, author, audience etc. The Resource Description Framework provides a set of tags and conventions to address semantic information as structured data. The Web Ontology Language [123] extends the RDF by more specific relationships like inheritance, ownership and composition, allowing automated reasoning to be performed on data described in this way.

The Super Distributed Objects specification [68] defines a vocabulary for distributed, autonomous systems. Components are defined in terms of properties and interfaces for configuration, monitoring and service invocation. This ontology is defined in the Resource Data Model. Device specific data and services are provided as instance data via these interfaces. The mapping to XML elements used to describe resources in pervasive computing environments based on this ontology will be presented later.

4.1.3. Device Representation

A resource representation needs to contain all the necessary information to access and manipulate the resource. Component descriptors in pervasive systems aiming at both inter-object and human-object communication have therefore two purposes: interface definition and user interaction.

4.1.3.1. User Interaction

One purpose of interface descriptors is the generation of user interfaces or, more generally, to provide users with sufficient information to make use of a particular component. User interaction is not equivalent to the invocation of services or requests for data, as the user might just want to find out the correct communication endpoint to which to connect a data source. This is especially true in terms of the transparency requested by the users that allow them to comprehend the workings of the Ambient Intelligent System.

For this reason the component descriptors should contain the type information of nested resources as well as a human-readable description of their meaning or functionality. Transforming a service descriptor to a specific user interface allows the creation of graphical, voice or gesture controls depending on available input devices and the user's situation [58]. Hodes et al. [60] propose the use of service descriptors as a way to externalize a portion of the system state, allowing this state to be altered via document authoring, in addition to an API access. This approach contributes to the overall adaptability of component based systems since the system can be adjusted to application requirements without recompiling the involved components

Interaction with the user would then be performed via a suitable client after processing the descriptor to fit a concrete client. For interaction via a web browser, for instance, the tags contained in the descriptor need to be transformed to HTML with hyperlinks for each contained resource. Other presentation types such as speech via VoiceXML are also feasible.

Presentation-specific elements in descriptor documents such as the checkboxes and selection lists proposed by Roman et al. [65] introduce more disadvantages than benefits, as Nichols et al have pointed out. [64]:

Descriptors get longer by providing UI elements for each client type
Descriptors might lose forward compatibility with future clients
Over-specification of the interface impedes the generation of user interfaces with a common look and feel.

4.1.3.2. Interface Description

As outlined in Section 4.1.1, a resource descriptor contains the information needed to access the resources or communication endpoints nested within. As it is not possible to know the interfaces of all components in ad hoc environments at compile time, one main purpose of resource descriptors is to allow nodes capable of analyzing such descriptors to construct invocations based on this information.

The descriptor document contains a list of all nested resources as well as the information needed to interconnect them such as their data type, or URLs for requesting further information. The amount of meta-data about resources should be balanced,

so as not to overload the descriptors (after all, energy is still scarce). The user can request additional information via a separate request on a specific resource.

Since the interface description is static, it can be stored in a non-volatile memory of the device and be treated like a string constant. In this manner even constrained devices not capable of handling XML structures at runtime or parsing their content can publish their interfaces.

Dynamic invocation is also offered by middleware platforms such as CORBA. However, experience has shown that the runtime analysis of an interface is cumbersome and ends up translating configuration data to the internal invocation format. With REST the internal and external representation is equivalent, hence user input can be directly translated onto invocations. Even so, due to the additional logic involved in runtime interface analysis, user interaction can be expected to be derived from the descriptor documents.

4.1.4. Device and Service Properties

A resource's state is the sum of all (non-static) property values belonging to that resource. Since the capability to process XML documents cannot be assumed of all nodes in the network, the minimum functionality of server-side components or devices is to return primitive property values on request. Primitive type properties are those which cannot be further decomposed, as opposed to the complex types represented by XML descriptors. Instead of embedding the values of primitive properties directly in the descriptor, resource representations contain identifiers which can then be used to retrieve these property values. This concept of indirection is known from distributed hypermedia systems.

It offers two advantages. First, it keeps the descriptors compact. A recursive listing of all the elements contained potentially containing nested elements themselves would blow the descriptors in all but trivial cases and waste energy during transmission. Secondly, resource-constrained nodes might not be able to construct complex state representations at runtime, but should at least be capable of transmitting a single value in reply to a request.

Separation of the resource description and the actual (and dynamic) property values allows constrained nodes to provide an immutable descriptor first, while actual data values can be retrieved upon subsequent requests. What's more, links between components will be by property values sent to a data sink, either regularly or upon change. Such an interconnection would be difficult to realize with values embedded in XML documents.

In the exemplary scenario, a user could request the representation of a sensor, located at:

```
http://sensor/
```

and in return receive the descriptor constant descriptor.

```
<sensor desc=”A simple sensor”>
   <motion type=”boolean” href=’./motion’>
         Motion in proximity
   </motion>
   <volume type=”int” href=’/volume>
         Volume in dB
   </volume>
   <vibration type=”int”> Vibration in Hz </vibration>
</sensor>
```

which is rather unexciting to subscribe for. First issuing a request for

```
http://sensor/motion
```

would then render the current value of the property as

```
false
```

Subscription of a light for the property would connect the sensor to an actuator.

4.1.5. Services

Services are by definition consumers of data, since the data provided by them is dependent on the input provided. Services provide some higher level functions other than the retrieval and manipulation of properties such as subscribing to a notification of status changes. A service descriptor needs to specify the parameters the service expects as well as a human-readable description of the service’s function.

Unlike descriptors of complex resources, service descriptors do not contain communication end points, but rather specify the expected parameters to be submitted for invocation of that service. Whereas in traditional distributed software systems the remote invocation of services or methods is the most commonly used interaction paradigm, in pervasive systems it should rather be the exception.

The reason is that services tend to have rather complex signatures requiring specific knowledge on the client side. Defining services as communication endpoints that accept a specific combination of parameters drastically reduces the number of compatible data sources which can be connected to them. Thus services should rather be designed as consumers of events that contain the necessary parameters.

Services and service interaction models are described above in more detail in Section 3.2.

4.1.6. Resource Identifiers

We have identified the unique addressing of communication endpoints as a major advantage of REST-based systems as opposed to traditional middleware. As with

data representation and resource access, the World Wide Web provides a proven and scalable solution. Resources in REST are addressed with URLs.

The term “Uniform Resource Locator” (URL) refers to the subset of Uniform Resource Identifiers that describe via a representation of their primary access mechanism, e.g. their network location and protocol. With REST this would be “http://”, or “httpu://” depending on the transport protocol followed by the node address and resource path. The node address is in the simplest case of an IP address. This address may be retrieved via a broadcast protocol, a machine-readable tag attached to the device or by manual configuration [62]. Alternatively, symbolical names may be used that address nodes by their function, location or properties [67].

The use of URLs as resource identifiers further decouples interacting components, allowing late binding of a concept to its actual implementation [59]. Unlike CORBA, interoperable object references URLs are designed to be transcribed and stored externally in a memory, a file or on the back of a napkin. Thus their validity is not dependent on the existence of an object in a given server but rather on a local resolution.

Name resolution may be realized via an orthogonal protocol (a domain name service) which places a central component in the architecture. Alternatively, using REST interactions names can be resolved by storing references resource values, and making use of redirect messages.

Because of their hierarchical structure, references can be created by requesting a general descriptor from a top level node, and subsequently appending the identifiers of nested resources. Given the following descriptor:

```
<light>
   <state type=”boolean”>
         The current state accepts true and false
   </state>
</light>
```

The light state can be accessed by appending the element name to the address of the parent node i.e.

```
http://light/state
```

4.1.7. Summary

This section introduced a generic access layer that can be used to retrieve and update generic data on arbitrary devices. Unlike other technology, this pREST access layer is light weight, universal, scalable and easy to implement. The following section describes how this layer is used to implement a new loosely coupled service interaction model.

4.2. Service Interaction Model

Since the service environment each time in a pervasive setting is formed in accordance with the current context of the user, the number of appropriate services - especially those provided by the physical environment - is limited in a natural way. Therefore, the service environment can be regulated and controlled by the involved parties themselves so that no third parties are required to manage special aspects of service provisioning. Thus, the entire system becomes more resilient against changes in the environmental setting. In terms of the supposed characteristics of the service environment, this service model is based on loosely coupled interactions according to the three dimensions of space, time, and representation. This section examines the realization of these three aspects with regard to service provisioning.

As argued in Section 3.2, the service models of today's SOAs typically realize a strict separation of the two phases of service discovery and service execution. There are interactions first to find and interpret the capabilities of a service and second to execute the service. The separation ensures that during execution each service instance is only given the information which it can interpret and is able to create an appropriate answer for. Thus, a major intention of the service discovery is to enable spatially coupled interactions of service consumers and providers during service execution. One practical benefit of separating service discovery from service execution is that a service consumer can execute a service, once discovered, multiple times. This has a positive impact on the performance of service provisioning and saves physical resources. Even so, these advantages only hold true in static service environments. In a highly dynamic setting, such as is usually encountered in pervasive computing environments, service discovery needs to be repeated continuously to keep the consumer's view of the service environment up-to-date. Moreover, choice of available services is limited by the current environment of the user. Hence, there is no need for the service-consuming system to determine the service-executing system by name as long as the functional and non-functional requirements for the requested service are fulfilled.

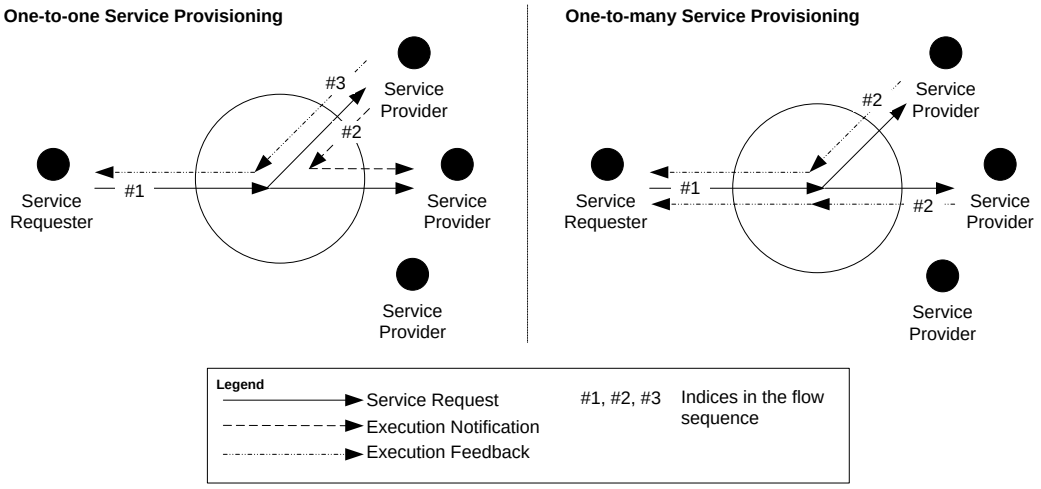

Figure 18: Abstract control flow in service provisioning.

The central element of this service model is the service request. The service request is originated by the so-called Service Requester and includes a self-contained specification of the task a service should perform. The parties providing services are supposed to analyze the service request in order to determine whether or not they are capable of processing the specified task. In the following a service providing party is referred to as a Service Provider.

Since the service model is required to be spatially uncoupled, all information published in the provisioning process can be obtained by all parties. A service request is published to all interested Service Providers in the environment of the Service Requester. The type of request defines whether a service should be executed by exactly one Service Provider or by multiple Service Providers. Figure 18 illustrates both methods. The first method of service provisioning is referred to as one-to-one service provisioning, the second method as one-to-many service provisioning. In the one-to-one service provisioning Service Providers initially need to agree on which one of them will execute the requested service, assuming that there are multiple Service Providers offering this service. Here a first-come-first-served approach is utilized. The first Service Provider that notifies the receipt of a service request is also supposed to execute the service. Thus all Service Providers capable of solving a requested task are required to listen to the notifications of the others before notifying the receipt themselves. Synchronization is handled by the interaction medium. However, in both methods of service provisioning the completion of a service execution is notified with an appropriate feedback for the Service Requester.

4.2.1. Control Flow in Service Provisioning

In the following three sections the procedure of service provisioning is explained in detail from an information-centric point of view. For simplicity's sake the term 'control entity' is used whenever some structured data with a concrete semantic is

meant. All control entities contain a reference to their originator, and some control entities have a limited life-span not necessarily defined by a date or a time-span. The expiration of a control entity is rather bound to the occurrence of a certain event in the context of the service environment such as the publication of new information. However, control entities generally may refer to each other but only in a directed, acyclic manner.

The control flow in the one-to-one service execution is illustrated in Figure 19 as a finite state machine. Since the involved parties determine the current state of the provisioning process with the obtained control entities, the state machine is only an abstraction. However, a service execution is initiated by the Service Requester by publishing a Service Request that contains the specification of the required service and some input data the service should be executed on. Before a Service Provider can start the execution, it first needs to acknowledge the request. The Service Provider that publishes the first Service Acknowledgement is allowed to execute the service. Thus, all providers receiving a Service Acknowledgement are forbidden to execute the corresponding requested service. While executing long-running services, the Service Provider is allowed to publish further Service Acknowledgements to indicate that execution is still in progress. For this purpose the Service Acknowledgement is one of the control entities with a limited span of life. If the execution process creates some provisional results, these may additionally be included in the Service Acknowledgements. Service Requester and third parties may use the extra Service Acknowledgements to determine whether a provider is still processing a requested service or has possibly disappeared from the service environment.

When the Service Provider successfully finishes the service execution, it publishes a Service Response. Otherwise, if problems occur during execution, the provider should publish these problems as Service Failure and cancel all processes related to the service. The failure of a service execution is implicitly assumed if the last valid Service Acknowledgement expires and no further ones are published. In both cases, no matter if the execution is explicitly canceled or the provider is gone, another Service Provider may accept the original Service Request. Possibly existing Service Acknowledgements containing provisional results may then be used to recover the service execution in the last known state.

A Service Requester can also cancel its own Service Request with a Service Request Invalidation. Unless the execution is already completed the executing system should cancel all related processes and return to the initial state. However, as a Service Request may also expire on the occurrence of a certain event, Service Requesters may publish Service Confirmations whose meaning is analogous to the meaning of the extra Service Acknowledgements. Service Confirmations can be used by Service Providers and third parties to determine if the Service Requester is still waiting for the response or has disappeared from the service environment.

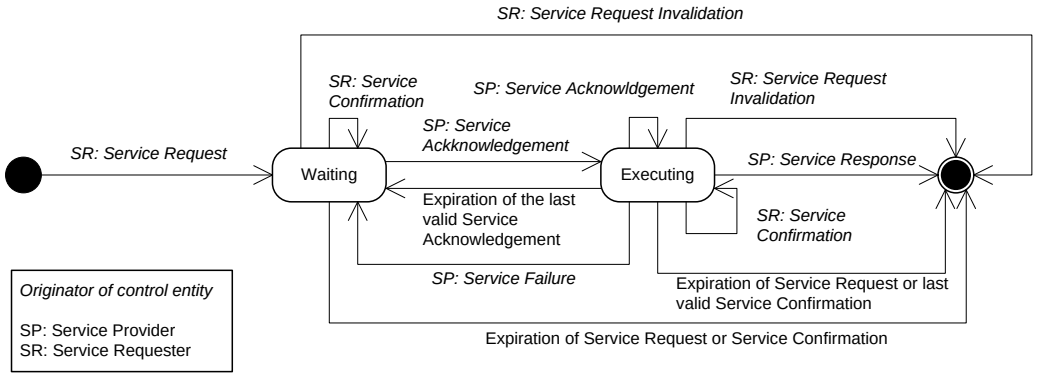

Figure 19: State chart of one-to-one service provisioning from external

The one-to-many service execution, as illustrated in Figure 20, is based on a best-effort approach. All Service Providers that understand a Service Request should process the corresponding task. The execution begins with the first Service Acknowledgement of a Service Provider. Unlike in the one-to-one service execution multiple, Service Acknowledgements from different originators are allowed. Once again Service Providers may publish the Service Acknowledgment to notify that execution is still in progress. Similarly, Service Requesters need to publish Service Confirmations for the same purpose. This way, the number of currently running, finished, and failed service executions can be determined at anytime. The first successful completion of an execution does not necessarily mean the end of all interactions. In fact, all processes are continued until the Service Requester publishes a Service Request Invalidation for the Service Request or the Service Request expires or the last valid Service Confirmation expires, allowing other Service Providers executing the service in parallel to complete execution. Even so, a one-to-many service execution is still regarded as successful if at least one Service Provider publishes an appropriate Service Response. Interrupted execution processes caused by failures or by the disappearance of Service Providers are not recovered.

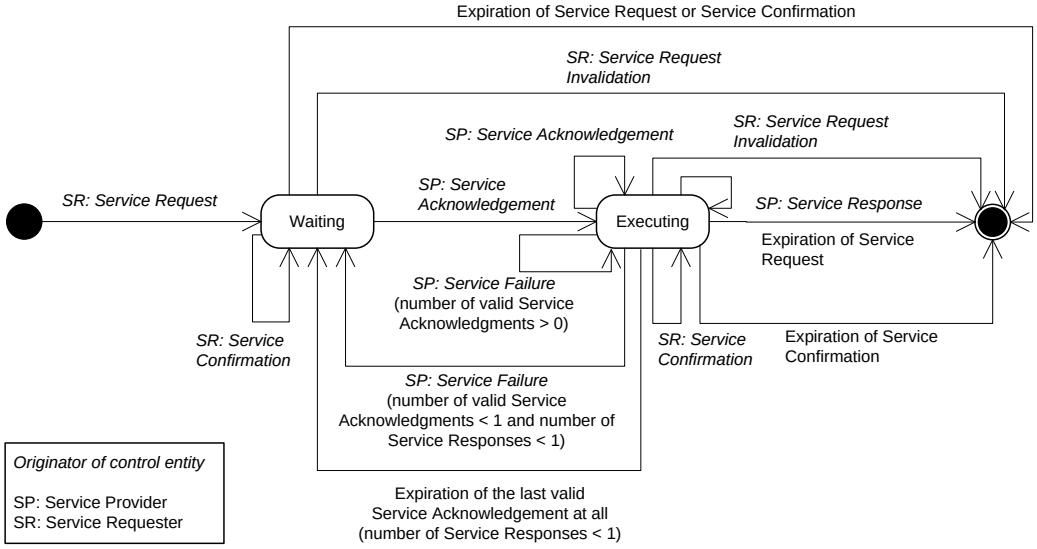

Figure 20: One-to-many service provisioning from an external point of view.

Summarizing, the essential control entities are:

Service Request

The Service Request is originated by the Service Requester and describes the problem to be solved by specifying service capabilities that should be matched by an appropriate service and some given data this service should be executed with. Moreover, the Service Request indicates whether the service should be executed by exactly one or rather by multiple Service Providers.

Service Confirmation

The Service Confirmation is originated by the Service Requester to revalidate a Service Request. Service Requests have a limited span of life. A Service Confirmation may be published to avoid expiry of a request if the original invalidation condition is fulfilled. At least one valid Service Confirmation is required to mark a request as valid although a Service Requester is allowed to publish as many Service Confirmations as needed.

Service Acknowledgement

The Service Acknowledgement is originated by the Service Provider to announce that it accepts a particular Service Request. In one-to-one service provisioning the Service Acknowledgement also indicates to all other potential Service Providers for the requested service that there is no need to proceed in processing the request for the moment. During service execution the Service Provider may use extra Service Acknowledgements to communicate that the execution is still in progress. Moreover, these extra Service Acknowledgements may contain provisional results in terms of a seamless service recovery. If the last valid Service

Acknowledgement of the Service Provider expires and there is no appropriate response for the requested service, the service execution is considered to have failed.

Service Response

The Service Response is originated by a Service Provider executing the requested service to signalize the successful completion of the execution.

Service Request Invalidation

The Service Request Invalidation can be published by the Service Requester to cancel currently running executions or to avoid that a Service Request being processed by any provider at all. Naturally a Service Request expires at a certain time as argued above, but if the needs of a Service Requester change before the request is accepted or the processing finished, the Service Request Invalidation can be used to publish this circumstance.

Service Failure

The Service Failure can be originated by the system executing the service to communicate that some exceptions, e.g., errors, caused the execution to be aborted. In the one-to-one service execution it may serve as starting point for the recovery of the requested service by another Service Provider.

4.2.2. Semantics of Service Provisioning

In general a service is executed on behalf of the Service Requester since it is the Service Requester that formulates the problem to be solved. Whereas in traditional service environments the service consumer may chose an appropriate service out of a list of descriptions, in this loosely coupled approach the Service Requester specifies the needed service itself. Accordingly, the Service Provider is required to determine whether or not it can interpret the specification and accept the service request.

As explained in section 3.1.2, data exchanged within interactions are required to contain enough reasonable information to be delegated to an appropriate destination. With reference to the service model this means that the content of a service specification needs to address a Service Provider. Given its task-oriented nature, the description of the problem a service should solve is more relevant to the Service Requester than the description of the service implementation. Complex process descriptions, as realized in SWSO and OWL-S, are intended to support the planning of a composite service execution. This model addresses services that represent atomic processes. As each service is intended to be executed in a single step, neither descriptions of service internal workflows in terms of orchestrations, nor descriptions

of overall data flows in terms of choreographies are included in the service specification. Instead, in the beginning of a service execution the Service Requester provides a set of input data to the executing system, while at its end the service executing system responds with some output data. However, this model does not exclude the representations for the entire composition of services. In fact, the specification of services is already designed to take account of higher-level models as the following sections suggest.

The service model refers to an environment based on cooperating objects around the user. In terms of pervasive computing, the services provided by these objects indirectly allow users to inform themselves about, and to manipulate, their physical environments. A lamp, for instance, may provide a service that allows switching it either on or off. Thus, calling the service changes the state of the executing system. A service that provides some information, such as about the weather in a certain location, can in reverse be seen to change the state of the consuming system or at least its informational state. Therefore, the service specification is oriented to the representation of state transitions on the service executing system as well as on the service requesting system.

In contrast to the simplified view of variable symbols and assignment rules, for the service specification a variable symbol is represented as tuple of the form (Function, $Term_1$, $Term_2$, ..., $Term_n$), which in the following will be called the Partial State. A Function is regarded as an identifier of a virtual object, while, in the context of a Partial State, Terms are arguments that determine a particular property of this virtual object. Each Term is itself either a Partial State or a Constant, where Constants are any atomic or complex data structures and do not need any further interpretation for the service specification. However, a Partial State is not required to contain any Terms at all, only its Function is mandatory. Representing variable symbols as Partial States naturally also affects the assignment rules which are henceforth referred to as Updates. The head of an Update is always a Partial State, while the body is a Term, i.e., either a Partial State or a Constant, formally written as tuple (Partial State, Term).

Based on the introduced terminology a service specification is defined through a finite set of Updates on Partial States. The relation of the service specification to Service Requester and Service Provider is realized through the Functions addressed by the Partial States in the Updates. As indicated above, each Function refers to a virtual object which may have a counterpart in the real world, and may thus be represented by the Service Requester or Service Provider. Therefore updating the Partial State for a Function means to change the corresponding virtual object and thus the current state of Service Requester or Service Provider. However, neither do all functions correspond to virtual objects represented by the Service Provider or Service Requester nor is the use of Functions in the Updates arbitrary.

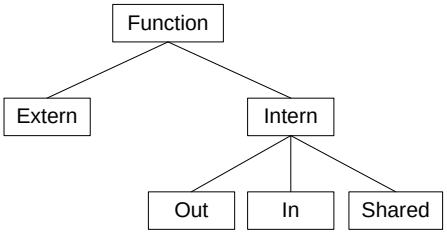

Figure 21: Hierarchy of function types for the Service Specification.

For clarification Functions are first separated into external Functions and internal Functions, as shown in Figure 21. An external function refers to a virtual object that is defined outside of the service specification. Thus, neither Service Requester nor Service Provider represents this object. The values for corresponding Partial States are rather considered to be generally known and are not allowed to be updated. On the other hand, internal functions address virtual objects represented by the Service Requester or Service Provider And the service specification also differentiates between three types of internal functions, henceforth referred to as in-Functions, out-Functions, and shared-Functions. The Function types suggest in which way the represented virtual object may be accessed. An in-Function represents a virtual object that is only accessible for reading data; an out-Function represents a virtual object that is only accessible for writing data; and a shared-Function represents a virtual object that may be accessed for reading and writing data. Moreover, in-Functions and out-Functions are defined to refer exclusively to virtual objects represented by Service Requesters, while shared-Functions exclusively refer to virtual objects represented by Service Providers.

The differentiation of Function types is especially relevant for the Updates, since the type of Function addressed by a Partial State defines whether the value represented by the Partial State may be updated or not. However, within one service specification one function of each type is allowed at most, i.e., in, out, and shared. However, unlike the shared-Function, the in-Function and out-Function are optional to service specification.

Service Specification

Functions
in -> input
out -> output
shared -> playing

Updates
(playing, song) := (input)
(playing, volume) := 50
(output) := (playing, length)

Precondition
(playing, song) = nil

Figure 22: Example of a service specification

Figure 22 shows an example of a service specification which addresses the playback of a song with a certain volume. To this end the in-Function *input*, the out-Function *output*, and the shared-Function *playing* are all defined. In the updates the Function *playing* is required to be changed in the Partial States *song* and *volume*, whereby the value for *volume* is a constant while the value for *song* is obtained from the Partial State of *input*. Moreover, the Partial State of output needs to be updated with the approximate length of the playback. The example also illustrates a construct called Precondition. The Precondition is introduced to enforce a certain state of the virtual object represented by the shared-Function before the Updates are performed. In the example the Precondition is used to indicate that the Partial State of *playing* for *song* should not be set. Therefore, proprietary to this example, the constant *nil* is introduced as placeholder for an undefined reference.

As explained, the service specification only describes the difference between two states without making any assumption about the states themselves, except for the Precondition. Consequently, Service Providers are intended to analyze the service specification with regard to Partial States referring to the shared-Function. If a Service Provider represents a virtual object that corresponds to the shared-Function and supports all required operations, i.e., updates and data retrieval, this Service Provider is allowed to accept the requested service. On the other hand, the Service Requester should provide representations of the in-Function and out-Function named in the service specification, if they are needed.

4.2.2.1. Data Values for Service Execution

The above section shows how services are specified in this service model. The service task is described with a set of Updates on Partial States wherein Partial States are supposed to represent values for certain properties of Functions. In the Updates, however, Partial States are only placeholders since the actual value represented by these states is not considered. Thus, in addition to the service specification, Service Provider and Service Requesters are also required to provide the actual values of the Partial States required for execution of the Updates. With reference to the example given in Figure 22, the Service Requester would be required to provide a constant value of the Partial State for the function *input*, while the Service Provider should finally provide a constant value for the Partial State of the Function *playing* and the argument *length*.

To this end a tuple of the form (Partial State, Constant), referred to as Evaluation, is introduced. During service execution Evaluations are used to communicate data values between the Service Provider and Service Requester. In general, the Service Requester is supposed to provide Evaluations for Partial States of the *in*-Function, while the Service Provider is supposed to provide Evaluations for Partial States of the *shared*-Function.

4.2.3. The Service Model Ontology

In terms of the three dimensions of loose coupling, spatially decoupling is explicitly achieved while delegating information by its content and temporally decoupling is implicitly achieved since no information exchange is strictly bound in time. The last requirement to be fulfilled is a loose coupling is representation. In Section 3.1.3.2 the utilization of proper logics is identified as a suitable approach with which to address this challenge. Thus the control entities and the formalisms for the representation of service tasks are composed to a Service Model Ontology (SMO). Since the control entities and the specification formalisms are static structures based on 'is-a'-relationships, they are defined with Description Logics (DL). Rather than words in a finite vocabulary, the types of information used in this model are defined as concepts. Although concepts are identified by names, they are not isolated but rather defined in relation to each other and to common base concepts. This way, a concept can be addressed without knowing its name, just by circumscribing it with its defining concepts. . A concrete datum can be seen as instance of a concept, i.e., the concept types the datum. Connections between these instances are represented as binary relations in this ontology, since each n-ary relation can be represented as a new concept whose instances have n binary relations.

The adoption of logic descriptions for the representation of concepts also allows the formalization of conditions and constraints on concepts as theorems. However, the scope of a theorem is not limited to the concepts it is proposed for because theorems are rather required to hold true even in instances of new concepts that inherit significant characteristics from those originally named in the theorem definition. Thus theorems can be used to enforce common, structural restrictions for the data exchanged. During service provisioning the validity of obtained concept instances can be verified by proving the theorems, i.e., by checking the conditions and constraints on those instances. The theorems are explicitly contained in the ontology. When extending the ontology or defining new ontologies based on the original one, new theorems can also be introduced. The parties involved in service provisioning may thus even check the constraints and conditions for concepts from new ontologies without any need for modification of the implementation.

The concepts of instances exchanged among the participants of the service provisioning are illustrated in Figure 23 and described in the following sections where the relations between concepts and possible constraints are explained in detail. In general all relations have the cardinality one. It should be noted that for greater clarity not all relations described below are depicted in the figure.

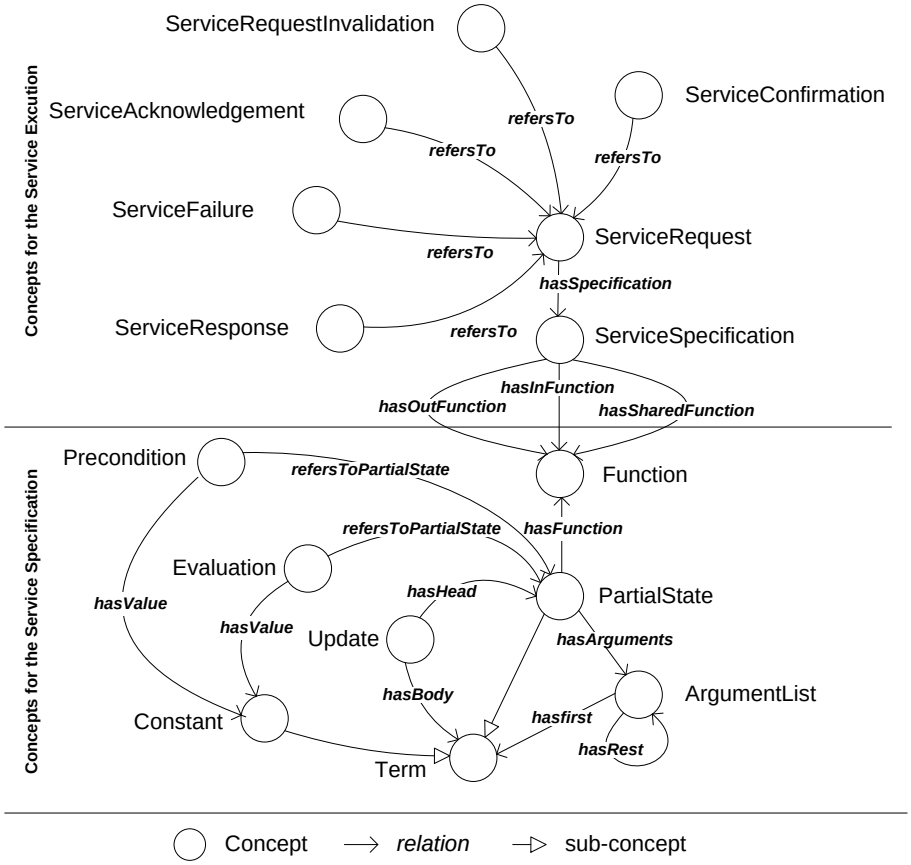

Figure 23: Overview of the Service Model Ontology and their relations

The following subsections describe the key entities of the Service Model Ontology in more detail along with their basic relations.

4.2.3.1. ServiceRequest

The concept *ServiceRequest* represents the control entity of the same name. Instances of this concept may be utilized by the Service Requester to indicate the need for the execution of a service. Therefore, the concept also has a particular relation to a concept for service specification.

Relations

hasSpecification	This relation references an instance of *ServiceSpecification*. This instance needs to be analyzed by Service Providers to determine whether they can answer the request or not.
isExclusive	The relation points to one of the two constants *True* or *False* and determines whether the request needs to be answered by exactly one Service Provider or multiple Service Providers. This is especially important for distinguishing between one-to-many and one-to-one service provisioning.
hasAddressee	This optional relation references the name of a particular Service Provider that is needed to execute the requested service. The relation may be utilized if a desired provider is already known as originator of other instances for concepts of control entities, ideally instances of the concept *ServiceResponse*. This relation is included for convenience and intended to help a Service Requester get responses to a sequence of service requests from the same Service Provider.
expiresOn	The *expiresOn*-relation references an instance of the concept *ExpirationCondition*. This instance defines under which conditions the instance of *ServiceRequest* should be regarded as having expired.
hasOriginator	This relation is a reference to the name of the party that originated the request, i.e., the name of the Service Requester.

4.2.3.2. ServiceConfirmation

The concept *ServiceConfirmation* represents the control entity of the same name. An instance of *ServiceConfirmation* revalidates an instance of *ServiceRequest* as long as the former has not itself expired.

Relations

refersTo	The refersTo-relation names the instance of *ServiceRequest* to be revalidated.
expiresOn	This relation references an instance of *ExpirationCondition* that defines under which conditions the instance of *ServiceConfirmation* should be regarded as having expired.
hasOriginator	The relation is a reference to the name of the party that originated the instance of *ServiceConfirmation*, i.e, the Service Requester that also originated the referred instance of Service Request.

4.2.3.3. ServiceAcknowledgement

The concept *ServiceAcknowledgement* represents the control entity of the same name. The Service Provider may use instances of this concept to accept a service request or to indicate that the execution of a requested service is still in progress. The expiration of all valid instances of *ServiceAcknowledgement* from the same originator implies that the execution has failed. However, the *ServiceAcknowledgement* may also be extended in ontologies for particular application domains to allow the representation of the application-dependent provisional results of an ongoing service execution.

Relations

refersTo	This relation names an instance of *ServiceRequest* that is supposed to be marked as accepted by the originator of the instance of *ServiceAcknowledgement*.
expiresOn	The expiresOn-relation refers to an instance of *ExpirationCondition* that defines under which condition the current instance of *ServiceAcknowledgement* becomes invalid.
hasOriginator	The relation is a reference to the name of the party that originated the instance of *ServiceAcknowledgement*, i.e., usually the Service Provider.

4.2.3.4. ServiceFailure

The concept *ServiceFailure* represents the control entity of the same name. An instance of this concept implies that the execution of the corresponding requested service has failed.

Relations

refersTo	This relation refers to the instance of *ServiceRequest*, for which a Service Provider executing the requested service needs to suggest a failure.
hasOriginator	The *hasOriginator*-relation is a reference to the name of the party that failed while executing the requested service, i.e., usually a Service Provider.

4.2.3.5. ServiceRequestInvalidation

The concept *ServiceRequestInvalidation* corresponds to the control entity of the same name. The Service Requester may use instances of this concept explicitly to invalidate an earlier originated instance of *ServiceRequest*.

Relations

refersTo	This relation points to the instance of *ServiceRequest* that should be marked as invalid.
hasOriginator	The hasOriginator-relation refers to the name of the party that originated the instance of *ServiceRequestInvalidation*. The originator should be the same as the one that requested the service to be invalidated.

4.2.3.6. ServiceResponse

The concept *ServiceResponse* represents the control entity of the same name. Instances of this concept are used by Service Providers to suggest the successful completion of a service execution.

Relations

refersTo	This relation refers to the instance of *ServiceRequest* for which a Service Provider wants to suggest a successful execution.
hasOriginator	The *hasOriginator*-relation is a reference to the name of the party that originated the instance of *ServiceResponse*, i.e., usually a Service Provider.

4.2.3.7. Function

The concept *Function* represents the construct of the service specification with the same name. Instances of this concept are identifiers for virtual objects. In the Service Model Ontology differentiation between in-Function, out-Function, and shared-Function is not modeled explicitly, but ratherthe type of an instance of the concept

Function is determined by the relation of this instance to an instance of *ServiceSpecification*.

4.2.3.8. ServiceSpecification

The concept *ServiceSpecification* represents a bridge between instance of *ServiceRequest* and a set of instances for *Precondition*, *Evaluation*, and *Update*. Instances of this concept are thus related to three instances of *Function* whereby the relation determines the type of each instance of *Function*.

Relations

hasInFunction	The *hasInFunction*-relation refers to an instance of *Function* that should be treated as in-Function. Based on this instance, corresponding instances of *PartialState* and *Update* may be found that also belong to the same service specification.
hasOutFunction	This relation refers to an instance of *Function* that should be treated as out-Function. The instance may be used to obtain instances of *PartialState* and *Update* that also belong to the same service specification.
hasSharedFunction	The *hasSharedFunction*-relation refers to an instance of *Function* that should be treated as shared-Function. The instance may be used to find instances of *Precondition*, *PartialState*, and *Update* that refer to the same service specification.

4.2.3.9. Term

The concept *Term* is a placeholder and only introduced to simplify the definition of constructs that alternatively require instances of *Constant* or *PartialState*.

4.2.3.10. Constant

The concept *Constant* refers to a constant term. Hence, instances of this concept can be seen as data values that do not require further interpretation within a certain service specification. These instances may, for example, be any numbers and characters like “1”, “2”, or “H”, as well as complex structured data like “Movie (‘Life of Brian’, 1979)”.

4.2.3.11. PartialState

The concept *PartialState* represents the construct of the service specification with the same name. Consequently, instances of this concept are supposed to refer to precisely one instance of *Function* and an arbitrary number of instances for *Constant* or *PartialState*.

Relations

hasFunction	This relation refers to the instance of *Function* addressed by the instance of *PartialState*.
hasArguments	The optional *hasArguments* relation points to an instance of the concept *ArgumentList* that contains a list of instances for *PartialState* or *Constant,* and parameterizes the referred instance of *Function* in detail.

4.2.3.12. ArgumentList

Since relations between concepts are unordered, there is no direct way to define a certain sequence for a set of instances. The concept *ArgumentList* thus represents a recursive list of instances for the concepts *Constant* and *PartialState*.

4.2.3.13. Update

The concept *Update* represents the constructs of the service specification with the same name. Consequently, instances of this concept refer to an instance of *PartialState* to be updated and the instance of *Constant* or *PartialState* that is supposed to provide the new value. Instances of *Update* may also appear in combination with instances of ServiceSpecification.

4.2.3.14. Evaluation

The concept *Evaluation* represents the construct with the same name introduced to communicate data values during service execution. Instances of *Evaluation* are may appear in combination with instances of *ServiceRequest* and *ServiceResponse*.

Relations

refersToPartialState	This relation refers to the instance of *PartialState* the value is provided for.
hasValue	The *hasValue*-relation is a reference to an instance of *Constant* which represents the actual data value.

4.2.3.15. Precondition

The concept *Precondition* represents the construct of the service specification with the same name. Instances of this concept may appear in combination with instances of *ServiceRequest* and *ServiceSpecification*.

Relations

refersToPartialState	The relation references an instance of *PartialState* that is required to have a particular value.
hasValue	This relation names an instance of *Constant* that is represented by the required value for the *PartialState*.

4.2.4. Bringing the Interaction Model and Service Model Together

Since the interaction and service model are defined to fulfill the same requirements with respect to loosely coupled interactions, the combination of both models does not require complex mappings of features. The major challenges are the representation of data and semantics. While the interaction model deals with a graph-based data representation, the service model utilizes DL-based ontologies for the description of concepts, relations, and individuals. One approved and widely accepted representation that combines both levels of description is the Web Ontology Language (OWL) of W3C, specified in [36]. As shown in section 3.1.3.2 above, OWL also includes a subset for Description Logics so that the service model ontology may be realized in this language without the need for any modifications.

However, the interaction subject in a service provisioning process is the actual execution of a service. This means that the Service Requester is required to open the Data Space when interactions with the Service Request should be transacted. Consequently, the Service Request is required to be originated immediately afterwards. The Service Requester is also supposed to close this Data Space if execution was successfully completed. During the service provisioning process control entities, as represented in the Service Model Ontology, reflect the intentions of the involved parties. Since the descriptions of these intentions are self-contained, all control entities, i.e., individuals of the corresponding concepts and related data, are added to separate Data Planes in a Data Space. On writing data to a plane, the resulting graph is not only a structural check but also supposed to be proven for semantic consistency and completeness as enforced by the service ontologies.

4.3. Ontology for Modelling User Context

Modelling of context to facilitate a context-aware system is not an easy task as it deals with the representation of context information such as sensor values in a form that can be processed by a machine and understood by a human being. One of the major problems context modelling has to deal with is the variety of sources providing snippets of context as well as the high distribution of such sources. In this respect, the most important sources for context information are:

- Sensor information: information that is gathered from sensors is usually transient. Possible sensor systems are badge systems, Radio Frequency Identification (RFID), or MICAz nodes [117].
- User input: examples for user input are preferences or other data the user has entered. Naturally this type of information does not change as often as sensor information.
- Abstract and general knowledge: this type of information normally comprises of general statements about the world or the context such as facts about the character of things. This kind of knowledge is often established and changes rarely.

To make use of this information what is needed is a mechanism to gather, save, and allocate context sources. Furthermore, the facts gathered from different sources need to be combined in a meaningful way in order to draw useful conclusions. There are a wide variety of different approaches to the modelling of context. Strang and Linnhoff-Popien [107] evaluate key-value, mark-up scheme, graphical, object-oriented, logic-based, and ontology-based models in terms of various criteria such as distributed composition and the level of formality. They conclude that the ontology-based approach is the one that best meets the requirements and offers the most promising way of modelling context.

However, it is important to decide not just how context is to be modelled, but also what aspects are related to the current context. In the majority of cases it is neither possible nor reasonable to apply all the facts that may influence the current context. Therefore, either a decision regarding the possible context sources has to be made or an appropriate interface has to be provided enabling the flexible addition or subtraction of such sources. There are indeed other properties of sources that may be relevant at modelling time. For instance, information that changes often may be separated from information that remains the same for a long time in order to facilitate easy and flexible adaptation to the current context.

4.3.1. Context Information

The modelling of context information is only the first step towards a context adaptation layer. In order to make use of the modelled information other components that are able to understand the context are needed. Tasks and components can be developed that are able to cope with the inputs and produce the desired outputs.

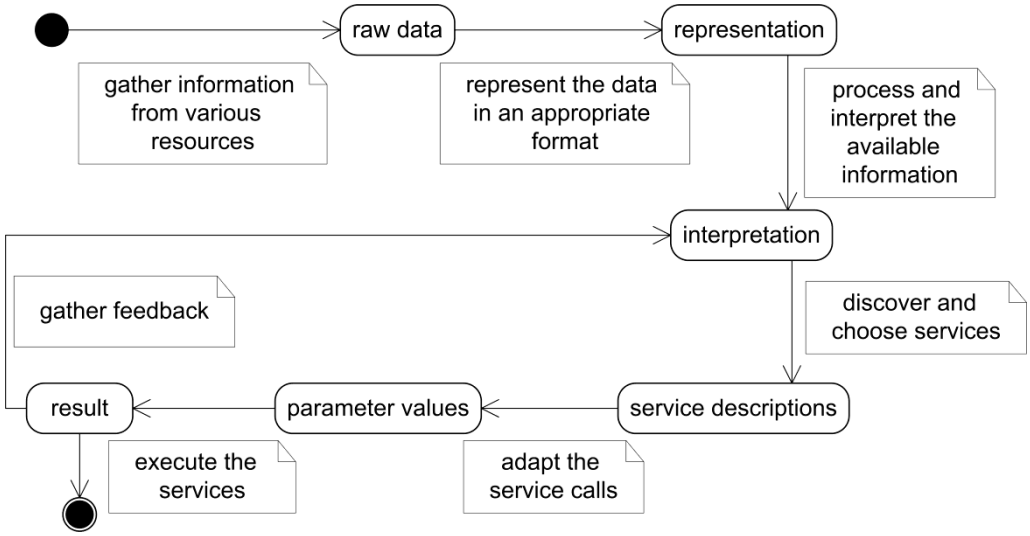

Figure 24: Tasks and actions of the context adaptation layer

Context information being sent over the network is gathered from several resources and accessed in order to represent it in an appropriate format. The representation comprises at least of the *Input Parameters* and *Context Data*. Depending on the service discovery mechanism utilised by the overall framework, the *Service Descriptions* may also be represented here, or are externally located, or need to be discovered later on. The data residing in the representation format is processed with the help of a suitable API enabling the interpretation of the information and giving the *Derived Parameters* depicted in the previous section. If the *Service Descriptions* are not yet on hand, they need to be discovered. Depending on the *Derived Parameters,* suitable *Service Descriptions* and *Service Parameters* are determined, the results are adapted and the service calls are finally executed. Feedback information gathered from the user may require the data to be interpreted once again.

With respect to ontologies, the proper means to interpret the contained knowledge are classification and inference. The whole process as depicted in Figure 25 comprises of the modification of property restrictions, the classification of a given ontology based on specified rules, and the application of those rules. The resulting facts are then added to the original ontology and can easily be retrieved using an API or a query language. It is important to note that the process may be repeated several times since derived facts may influence the classification of the knowledge base and vice versa.

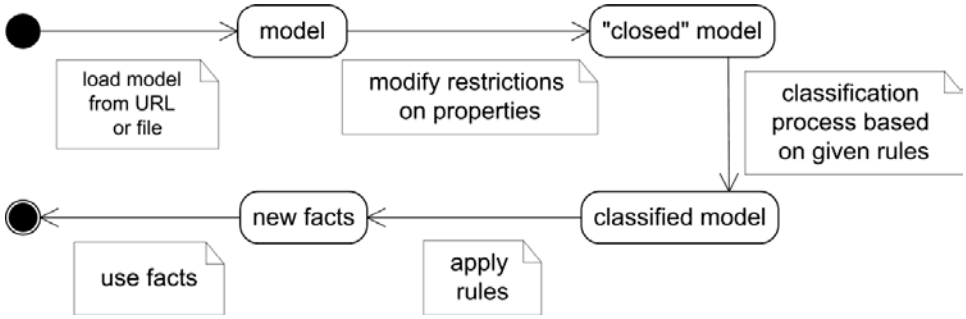

Figure 25: An exemplary inference process using rule-based classification

In terms of this model, there is no specification either of where the context information and rules come from or of what is done with the deduced facts. However, it is assumed that an application is able to process resulting information in a meaningful way and is therefore context-aware.

Data stemming from external context providers is harmonised by *Context Brokers* which make this information available to other components in the context adaptation layer. Ontologies are used by the *Context Broker* as a knowledge-base representation of context data. However, ontologies are only one means that can be used for context modelling and bring their own specific advantages and disadvantages. The most important aspects are outlined below..

One of the main benefits of ontologies is the interoperability they offer when using a uniform way of representation. As the common understanding of a certain domain enables the combination of distributed knowledge as well as the reuse of existing information, it is easy to design an upgradeable, machine-readable, and flexible knowledge representation that is adaptable to state-of-the-art web technologies.

Another key feature of ontologies is the implicit knowledge that resides within this kind of representation and that enables reasoning techniques. Apart from the stated information, with ontologies it is possible to add significance to the plain facts allowing their verification and the inference of additional statements, while the hierarchical structure of ontologies and the given properties of objects enable virtual navigation along links as well as basic functionalities such as generalisation and specialisation.

On the other hand, one drawback is their need for a suitable means of representation, i.e. an expressive language. Since ontologies should be designed to be flexible and open, the structure of this language is often complicated. Hence, huge ontologies are often exceedingly complex, making the reasoning process expensive and slow. Work with ontologies is often more difficult than with alternative technologies such as databases.

The context of an entity comprises of heterogeneous information gathered from many different resources. Context-awareness presumes comprehension and inter-

pretation of the represented context. Ontologies meet these requirements perfectly and are an adequate means of modelling context as well as having enough flexibility to adapt to the ongoing evolution of information technology.

Since there are many specific ontologies, dealing with a certain domain or intended for a specific purpose, the following subsections depict a selected variety and evaluate them in terms of their application to the context adaptation layer.

4.3.2. Ontology Domains

Depending on the type of the application that uses ontologies for context modelling or other purposes, different requirements need to be met by the ontology. As there are a variety of ontologies for different domains and use cases, it is vital to evaluate existing approaches and solutions in order to find the most suitable one. This section specifies the most important criteria in terms of this present thesis while the following sections evaluate some of the most common popular ontologies in terms of these criteria. Finally, a conclusion is drawn as to which of these ontologies are the most suitable for utilisation within this layer.

Type	The ontology type is determined by its application area. Ontologies may vary in the domain they try to describe, but also in their focus or degree of abstraction. Depending on the ontology's type the requirements differ, i.e. a global ontology only needs to define general concepts whereas a specific ontology has to provide certain concepts if it is to be beneficial.
Support	The support criterion comprises of aspects such as numbers of users, updating levels, types of license, and chances of being standardised. Preferable results in this area are: regular updates, free availability, visible standardisation efforts, and widespread application use of the ontology.
Languages & Mappings	This aspect refers to the representation of the ontology. As shown in the previous chapter, an ontology written in OWL or at least a mapping to OWL concepts is preferable. Extant translations and concept mappings for other ontologies are further features. The better an ontology scores in this criterion, the more flexible it is.
Complexity	The complexity of an ontology is determined by the number of concepts, relations, and instances it comprises as well as its hierarchical structure. In this regard it is also important to know how long it takes to process an ontology using a classification

	tool or an editor. The complexity level strongly depends on the type and domain of the ontology. In short, low complexity is preferred in order to facilitate fast and easy processing.
Structure & Expandability	This criterion relates to ontology design which should be intuitive, clear, correct in the common understanding of things, and easy to expand. The embedding of new concepts has to be simple without the need for extensive tests. Meta-concepts and abstract objects are less important than concrete items serving as service input values. An ontology should also be distributable and extendible, a criteria which is normally fulfilled by OWL ontologies.
Concepts	In terms of the framework, some concepts occupy key roles like the concepts *Situation*, *Location*, and *Person*. The ontology needs to include these facets in an adequate way or has to be extendible according to them.. It is necessary to evaluate which concepts are ignored, incompletely covered, or even overrated.

4.3.2.1. Upper Ontologies

Upper ontologies try to model virtual and physical objects and constructs that can be found in the real world so that particular objects can be hierarchically arranged in an easy way. These ontologies are normally used as a base and can be extended by more specific ones. However, most of them are very large and comprise of a many different concepts and relationships that are hardly ever used. Even so, use of an upper ontology is advisable to merge information stemming from different sources and to avoid semantic heterogeneity [74].

Some common upper ontologies are briefly depicted in the following subsections with regard to the criteria mentioned in the previous section. Important requirements for an upper ontology include support for OWL, free availability, reasonable complexity, high expandability, and adequate completeness.

4.3.2.1.1. SUMO

The Suggested Upper Merged Ontology (SUMO) [74] is a free upper ontology written in the SUO-KIF language [76]. The IEEE owns SUMO and maintains its web site with the help of the Standard Upper Ontology Working Group (SUO WG). Besides the original English KIF version, there are also language templates, some provided domain ontologies as well as a mid-level ontology in SUO-KIF referring to SUMO, a mapping to WordNet concepts [77], and a translation to OWL. The current OWL (Full) representation of SUMO is incomplete as compared to the original KIF version but sufficient for use as an upper ontology. The OWL model comprises of over 600

classes, 200 properties, and 400 individuals. The averaged level in the tree view of the model and the branching factor is fairly high. However, the OWL representation makes an ontology extension relatively simple and an online mapping to WordNet concepts eases embedding of new concepts in the hierarchy. Since SUMO is an upper ontology, some concepts are not completely covered such as situations or private relationships. On the other hand, the concepts time and location and their aspects are sufficiently well represented. In short, although SUMO is fairly complex and still needs to be extended in terms of this thesis, it is supported by the IEEE, provides many mappings, and contains most of the necessary concepts.

4.3.2.1.2. DOLCE

The Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) [77] is an upper ontology in English that isavailable in the Knowledge Interchange Format (KIF) and OWL where it has the version name *DOLCE-2.1-Lite-Plus*. The OntoWordNet project provides a beta version of a top-level alignment from DOLCE to WordNet. DOLCE consists of a core ontology with upper level concepts that are less complex than SUMO. It is extendible with other provided ontologies that model specific aspects. However, the core ontology and even its extensions comprise of many abstract concepts that need to be expanded if they are to be used as part of the system ontology. In short, the DOLCE ontology is similar to SUMO, but has drawbacks in terms of certain criteria, namely support, available mappings, and provided concepts.

4.3.2.1.3. OpenCyC

OpenCyc [79] is an upper ontology and the open source version of Cyc, a knowledge base and reasoning engine. OpenCyc is currently available in the version 0.9 and written in CycL, the Cyc representation language. According to the web site, OpenCyc contains 47,000 concepts and 306,000 assertions. OWL mappings are available but are extremely large and cannot be processed by the ontology editor Protégé [92] in reasonable time. Due to its dimensions, the ontology is relatively complete. Similar to SUMO, OpenCyc is supported by the Standard Upper Ontology Working Group (SUO WG). However, as the OWL mappings are unusable for framework implementation, the OpenCyc ontology is not utilised as an upper ontology.

4.3.2.1.4. WordNet

WordNet [80], which was evaluated in its version 2.1, is rather a lexical reference system than an upper ontology. The main building blocks of WordNet are lexical concepts represented by synonym sets comprising of nouns, verbs, adjectives, and adverbs. Synonym sets can also be related to each other, e.g. by defining generalisations or specializations. WordNet is freely available and can be used online or offline. The offline version of the database can be accessed with the help of an API.

As the language of WordNet is English, there have been some attempts to map the English concepts to concepts in other languages. However, project results with European languages are restricted and require payment of a license fee. According to the web site, the database of WordNet includes approximately 150,000 unique strings comprising of all of the concepts needed by the service framework. Even so, there are some drawbacks of WordNet that prevent use of the application as an upper ontology. Firstly, it is not intended to be anontology, but a lexical database with a main focus on words and their relations. This means that there is a lack of the useful aspects of ontologies as depicted above. Secondly, the expandability of WordNet is very restricted making it less flexible than alternative forms of representation such as OWL. In short, WordNet should rather be used to complement the service framework than to serve as a base ontology. Possible applications include the identification of synonyms and translation purposes.

4.3.2.2. Domain Ontologies

In contrast to upper ontologies, domain ontologies are intended for use within the scope of a certain application or domain. These ontologies are normally smaller than the huge upper ontologies and are only applicable for a specific purpose. The following ontologies have been selected for the evaluation process since they cover the relevant aspects of context modelling. Important requirements with respect to domain ontologies include sufficient simplicity, high usefulness of the described concepts, and observable support for the ontology.

4.3.2.2.1. SOUPA

The Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA) [73] is described with the Web Ontology Language (OWL) and includes modular components for representing key aspects in pervasive computing. SOUPA consists of a core (nine concepts) and its extensions (nine concepts) and is very small. The concepts needed for the modelling of situations are present, but have to be mapped to the concepts of an upper ontology in order to represent the context. The latest version of SOUPA was released in 2004 and there have been a few attempts to utilize it. Concepts such as location, person, and time are covered by SOUPA, but others are not like relationships. This means that the SOUPA ontology can only be used as an extension for situation modelling.

4.3.2.2.2. CONON

The CONtext ONtology (CONON) [81] is another OWL approach for modelling context in pervasive computing environments. In their paper Wang et al. classify the proposed model into an upper ontology and a specific ontology. The current version of CONON contains 197 OWL classes. The authors of the paper merge CONON with the Cyc ontology and evaluate the performance in terms of the reasoning process.

Depending on the ontology size, process run-time can take up to several seconds. The main application of CONON is the classification of the situation for meetings and home scenarios. Concepts such as location, activity, and person are included, but time and relationship are insufficiently covered. However, as Wang et al. do not provide an OWL file, an OWL mapping has to be constructed manually. Furthermore, CONON is far less supported than upper ontologies such as SUMO or DOLCE.

4.3.2.2.3. OWL-S

OWL-S [82] is an ontology for Web Services based on OWL DL. According to the documentation, OWL-S has four main purposes:

Automatic Web Service Discovery
Automatic Web Service Invocation
Automatic Web Service Composition and Interpretation
Automatic Web Service Execution Monitoring.

The ontology comprises of four building blocks. The *Service* ontology includes four classes and eleven properties and serves as a base for the other three blocks - *Process*, *Profile*, and *Grounding*. Utilising the four ontologies, it is possible to describe a service and its effects, to compose service chains, and to invoke Web Services automatically. The OWL-S proposal, which uses OWL-Time [83], has already been submitted to the W3C. The version 1.1 has been available since 2004 and has been utilised by some projects like the OWL-S API of Mindswap [49]. Even so, there are still some approaches that extend the abilities of OWL-S in terms of the semantic description of Web Services, such as the First-order Logic Ontology for Web Services (FLOWS) which is part of the Semantic Web Services Ontology (SWSO) belonging to the Semantic Web Services Framework (SWSF) [90]. Another option in terms of Semantic Web Services specifications is the Web Service Modeling Ontology (WSMO) [91] based on the Web Service Modeling Framework (WSMF) [96].

In short, the semantic description of services is indeed a necessary and important aspect of a service framework based on ontologies. OWL-S comprises of some interesting features in this respect, but does not cover all possible aspects for the development of alternatives or extensions such as SWSF or WSMO.

4.3.2.2.4. FOAF

The Friend of a Friend (FOAF) ontology [84] models persons and their personal information, groups of people, and related documents or accounts. FOAF can be represented in RDF and is already used for the annotation of web sites and search engines. One available OWL mapping includes 23 classes, 20 data type properties, and 37 object properties. However, some of the properties are very specific, such as certain chat IDs. Persons, groups, and organisations are sufficiently well considered,

but relationships between persons are hardly covered at all. Without extending the FOAF ontology, it can only be used for constructing profiles.

4.3.2.2.5. OWL-Time

OWL-Time [85] is an ontology that models concepts related to time. The university project resulted in several papers in 2004 and 2005. The concepts of OWL-Time are used by OWL-S and are mapped to concepts of SUMO. The complexity is comparable to that of the respective component of SUMO and includes most of the necessary preconditions for modelling time in terms of the current context. However, if SUMO is used as an upper ontology, OWL-Time is not essential.

4.3.2.3. Summary

Table 2 shows the evaluation results outlined in the previous sections. The depicted rating of the ontologies refers to the criteria mentioned in Section 4.3.2; the minimum score per ontology and criterion is zero and the maximum score is three.

Ontology	Type	Support	Languages & Mappings	Complexity	Structure & Expandability	Concepts
SUMO	Upper Ontology	+++	++	+	++	++
DOLCE	Upper Ontology	++	+	++	++	+
OpenCyC	Upper Ontology	++	O	O	+	-
WordNet	Lexical Reference	+++	+	+	+	+
SOUPA	Domain Ontology	+	++	+++	++	+
CONON	Domain Ontology	O	+	++	+++	++
OWL-S	Domain Ontology	++	++	++	++	++
FOAF	Domain Ontology	++	++	++	+	+
OWL-Time	Domain Ontology	+	++	++	++	++

Does the ontology meet the requirements? O Not at all. + Poorly. ++ Almost. +++ Yes. - Not rated.

Table 2: Ontology evaluation results

SUMO, DOLCE, and OpenCyc are classified as upper ontologies. SUMO meets the requirements for the criterion *Support* and scores well in terms of *Languages&Mappings*. Most of the needed concepts are also covered by this upper ontology. However, it is extremely sophisticated. OpenCyc on the other hand, seems not to be the best solution for the planned framework since it is too complex. DOLCE is comparable to SUMO and slightly less complicated, but has drawbacks with respect to languages, support, and concepts.

As WordNet is a lexical database, it scores well in the *Support* criterion. However, it is not applicable as an upper ontology and meets the other requirements only poorly.

CONON is classified as a domain ontology as its focus is on situation reasoning. Hence, in contrast to common upper ontologies, many abstract categories are not designated. The structure of CONON fulfils the requirements, but the lack of support is a severe drawback. SOUPA - which is also ranked as domain-specific due to its use in pervasive applications - is a very simple ontology though hardly containing necessary concepts. Support also seems to be unsatisfactory.

OWL-S, FOAF, and OWL-Time are all domain ontologies with different application areas. FOAF and OWL-Time do not play an important role as their concepts are not needed or already contained in the upper ontologies. The application of OWL-S depends on the need for Web Services and the required support for alternative technologies such as REST.

In short, for an upper ontology SUMO is to be preferred over DOLCE and OpenCyC, and is also used by the context-aware framework.

WordNet can be used in two ways. Firstly, it can be deployed as a service for finding synonyms or, if extended, for translation purposes. Secondly, WordNet can be used to interpret the input and output of services by finding the meaning of things. For example, the mapping of WordNet concepts to SUMO classes and properties enables the processing and interpretation of textual information by the framework.

With the exception of OWL-S, the evaluated domain ontologies are not used since their concepts are already covered by SUMO or not needed. In terms of OWL-S, two aspects are particularly useful when realising the context-aware framework: the semantic description of services and its automatic invocation. However, the sophisticated and manifold input of legacy services in the Internet, e.g. search services, requires a more subtle and accurate semantic description than is currently provided, i.e. a versatile and detailed parameter composition characterisation. Other aspects of OWL-S such as service composition facilitating service chains and service discovery are also very beneficial, but not within the scope of this thesis. The same goes for the alternatives mentioned that support further features for Semantic Web Services, namely SWSF and WSMO.

4.4. Conclusion

This chapter specified the Service Request Oriented Architecture by defining a universal access layer, a novel service model for loosely coupled interaction, and by outlining the general components as well as appropriate ontologies for adapting context-agnostic services to user needs.

5. Implementation

5.1. pREST Access Layer

In the following section two approaches are presented to realize a REST interface for distributed components. The first one presented below is designed as a general purpose middleware that makes few assumptions about the components to be published (most of the assumptions concern naming conventions, such as a “get” prefix for access methods). It is intended to publish arbitrary objects as REST resources.

The second approach targets resource constrained systems, and integrates communication handling, message parsing, and service provision. For those limited devices a simple yet fully functional REST implementation has been realized.. Section 5.1.4 summarizes the implementation of a REST compliant interface on embedded hardware.

The pREST access layer provides portability across platforms and programming languages by specifying a system-independent data format and common set of operations for all resources. Whereas resources published this way are universally accessible across system boundaries, invocations from outside need to be translated to platform specific calls.

The common approach to binding an implementation to an externally visible, platform-independent interface is to place stub and skeleton objects on each side, communicating natively with the server and client respectively, while transmitting data according to the middleware protocol between each other.

5.1.1. Generic middleware interface

To provide for portability to future en-vogue technologies without subsequent code changes, the application code should be kept clear of middleware specific types and concepts. To realize such abstraction, an additional software layer is needed which maps application level behavior such as event publication or remote access onto the mechanisms of the underlying middleware.

This abstraction layer is specified as a set of Java interfaces to be implemented by a custom middleware such as pervasive REST or an adaptation component mapping application level behavior onto a middleware specific API such as CORBA or Axis. The interfaces are kept simple in terms of both contracts and method signatures.

Unlike the CORBA based implementation where objects derive from Portable Object Adapters pass Any-type parameters around parse Structured Events half of the time and throw up low-level errors such as ServantNotActive or InvalidPolicy, the abstrac-

tion layer's interface to the application is defined purely in java.lang types, predominantly Object and String. The stub objects, implementing the application level interfaces of the respective remote objects are created in runtime and thus can be treated without regard to their nature as placeholders.

The abstraction layer wraps the specific mechanisms of the middleware into general methods such as 'lookup' or 'publish'. The functionality it provides is defined in terms of interfaces with contracts specified as effects on the application layer.

Upon invocation of 'bind', for instance, subsequent calls to lookup will yield an object representing the previously published one. If the published object happens to be in the same address space as the caller, lookup might return the object itself; if not, a proxy object would be returned. The invocation of publish with a group identifier and a message object causes all parties that have previously subscribed for this group to be notified.

5.1.2. pREST middleware specification

The pREST middleware is an implementation of the general naming and event service interfaces specified in the previous section. It consists of two parts, the first one the RestServer which is responsible for publishing arbitrary objects as resources and translating REST invocations to method calls. In addition the server also provides publish-subscribe communication.

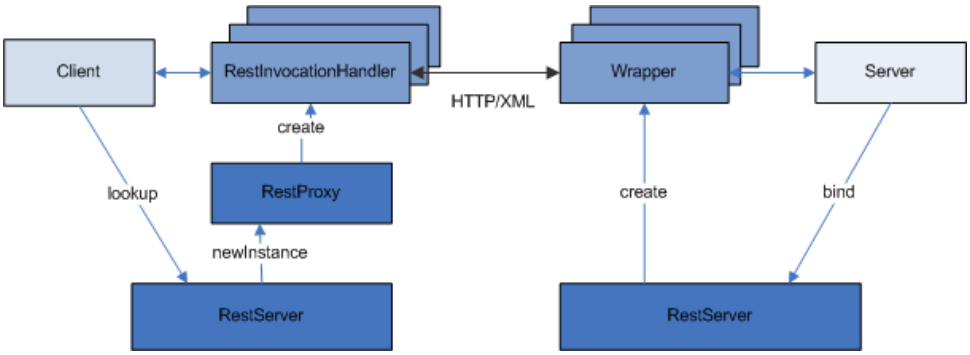

Figure 26: pREST middleware architecture

The second part is the RestProxy which generates InvocationHandler objects which act as stubs allowing invocation of methods on remote resources the same way as local objects. Each addressable entity of a server object such as a method, field, array element etc. is associated with a Wrapper object mapping resource access onto the respective entity.

Wrapper classes realize the abstraction of Java elements as resources. As addressable entities vary by representation and their retrieval in Java, separate wrappers exist for objects, fields, and methods. REST communication via HTTP and XML takes place between the InvocationHandler and the Wrapper classes.

5.1.3. Interaction examples

5.1.3.1. Object publication

To make an object remotely accessible, the server invokes the bind method with an identifier and a set of interfaces. The RestServer constructs a valid URL out of the given identifier and context, replacing illegal characters with an appropriate representation and making sure the path starts with a slash.

Then the server creates a new object wrapper for the object and stores it in the map of bound objects using the path as the key. If the server has any peers it is aware of, it creates a HTTP redirect entry at the peers so the object can be looked up on remote servers.

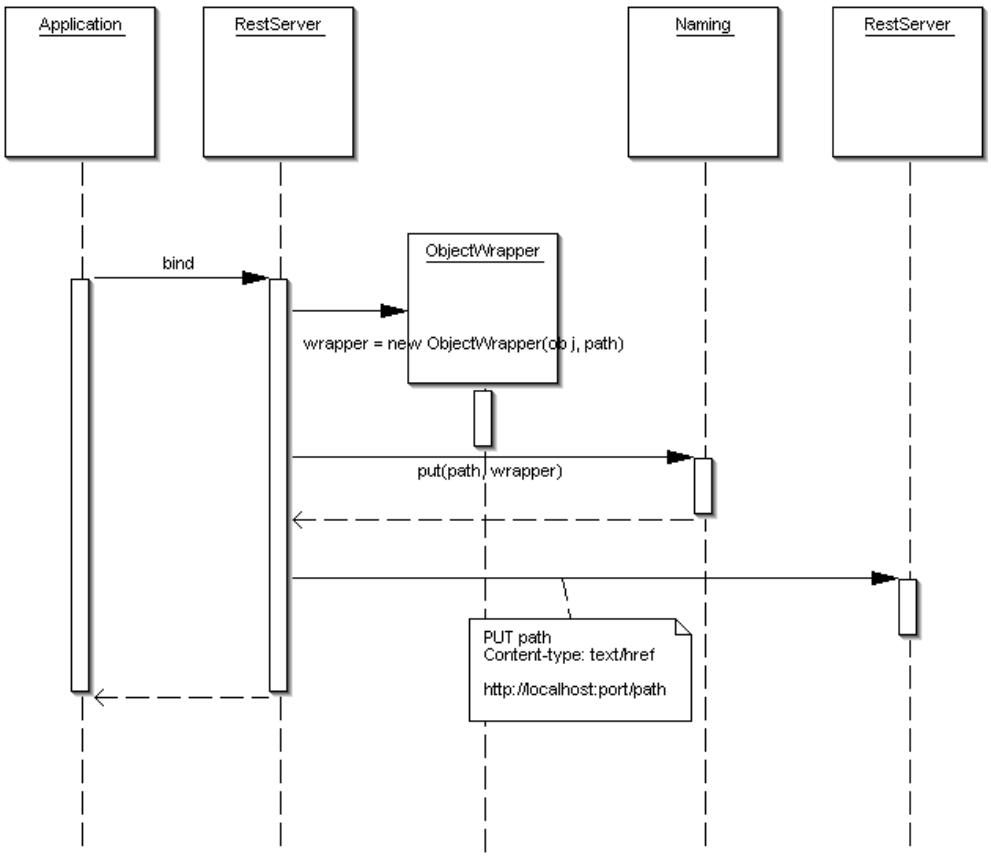

Figure 27: Object creation

The creation of redirects realizes a simple decentralized naming service. Components unaware of the actual URL of a remote object query the list of known peers for a symbolic name (the local path). Since any object deposits an absolute URL at its peers, the request will yield a redirect message with the actual URL in its Location header.

For a real peer-to-peer lookup, the naming tables ought to be partitioned and distributed redundantly among nodes to ensure scalability and efficiency of lookup.

5.1.3.2. Object Lookup and Remote Invocation

To look up a remote object, the client invokes the lookup method of its local naming service, i.e. the RestServer. The server consults the map of local objects and retrieves the wrapper bound to this path. The getObject method of the wrapper returns either the original object or an invocation handler associated with the URL of the remote object.

The way the wrapper retrieves the wrapped object depends on the kind of resource it wraps. Object wrappers return the wrapped object without additional processing. Field and accessor wrappers invoke the respective get method or read the particular field and return the result as an object. Redirect wrappers request the RipProxy to create a new InvocationHandler instance and establish a connection to the original resource.

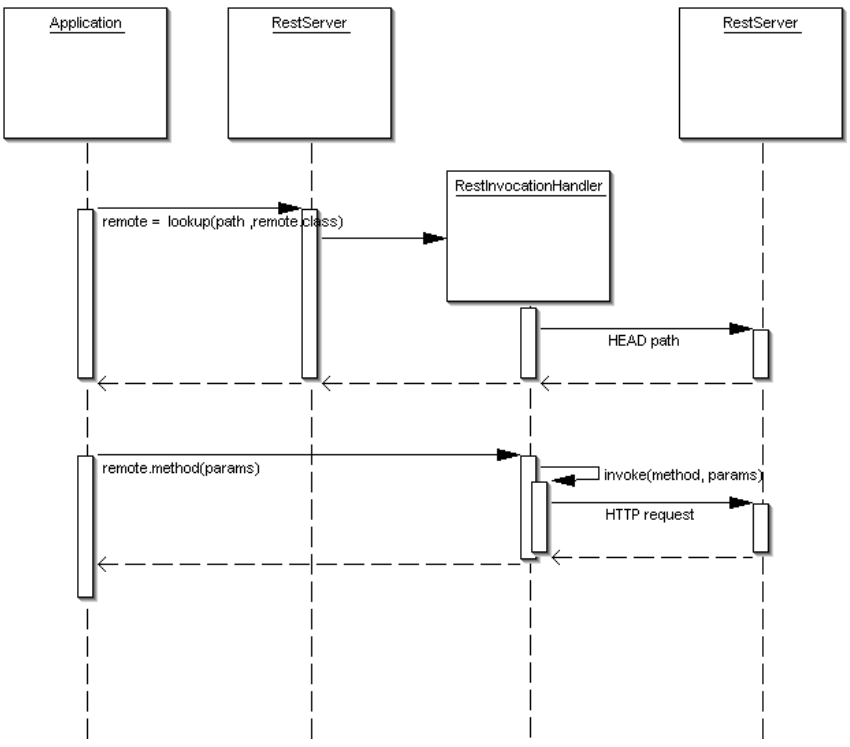

Figure 28: Object lookup and remote invocation

Lookup requests for remote objects will always result in the return of an invocation handler. When created, the invocation handler sends a HEAD request to the remote resource to verify the given URL. If the request yields a 200 OK response, the handler will direct any subsequent invocations to that URL; if it results in a redirect message, the URL is retrieved from the message and the handler establishes the connection with the resource at its actual location.

All invocations to the handler instance, are internally directed (by Java reflection mechanisms) to the invoke method with the name of the method passed as a parameter. The handler transforms the invocation target (consisting of an object and the method name) to a URL, analogous to the wrapper classes. The 'get' prefix of access methods is stripped. Likewise the id parameter of selectors is appended to the request path.

The HTTP method to request a resource is determined according to the method invoked and the return value. Access and selector methods are transmitted as a GET or HEAD request depending on whether the returned object is a primitive property or a complex resource. Methods that do not start with get and expect parameters are invoked via a POST message with the parameters passed in the message body.

5.1.3.3. Request Processing

Connection requests via TCP are accepted by the ConnectionHandler and processed by a separate thread. The multithreading approach allows the processing of several requests at a time, but also prevents deadlocks if a resource requests data back from the invoking server. The DatagramHandler and MulticastHandler classes do not start separate threads since the caller does not wait for the callee to finish, and deadlocks shouldn't occur.

The handler thread creates a new RipRequest object from the data received and passes it on to the wrapper associated with the requested path. The wrapper retrieves the requested data (if the HTTP method is GET or HEAD) or invokes the called method based (for POST requests) and returns the result in a RipResponse object.

If the resulting object is simple enough to be returned as a string the RipParser is requested to marshal the value which is then enclosed in the response object. If the requested object is complex or a service an appropriate resource descriptor is returned.

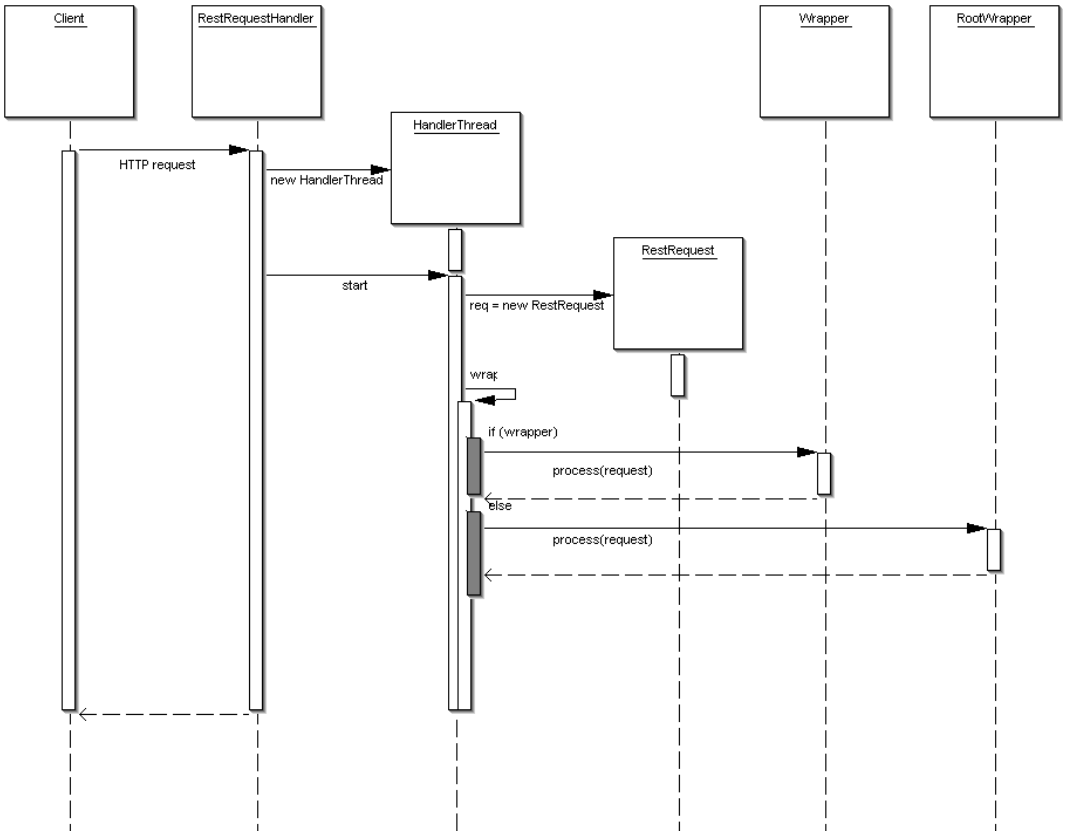

Figure 29: Request processing

If no wrapper is associated with the requested path, the root wrapper is consulted. The request for an unknown path may have three reasons.

- the resource really does not exist. In this case a 404 Not Found response is sent back.
- the resource is to be created via a PUT request equivalent to redirect creation. In this case a new RedirectWrapper is instantiated and a 201 Created returned.
- the resource is a child of an existing object (such as a method, field or array element) for which a wrapper doesn't yet exist.

In all cases the root wrapper needs to iterate through the map of known resources and check whether they have a child element that matches the requested path. This is done by stripping off path components one by one until the path matches an existing wrapper. Once a matching (grand) parent-wrapper is found, a search starts for the child elements named after the stripped off path components.

For each found child-element a new wrapper is created until either the whole path is reconstructed or no matching child found. In the former case the request is processed by the newly created wrapper, in the latter a not-found error is sent back.

5.1.3.4. User interaction

Representation of resources as XML documents allows the generation of web interfaces for user interaction. The actual generation is performed by the browser via an XSL style sheet stored externally and referenced in the descriptor document.

Separate style sheets exist for services and resources. The service style sheet transforms the whole document into an HTML form, and nested elements into input elements according to their type. Enumeration type parameters are presented as selection lists, Boolean values as checkboxes, and the remaining types as text fields. Figure 30 shows a service interface for the subscribe method defined by the SDO monitoring interface.

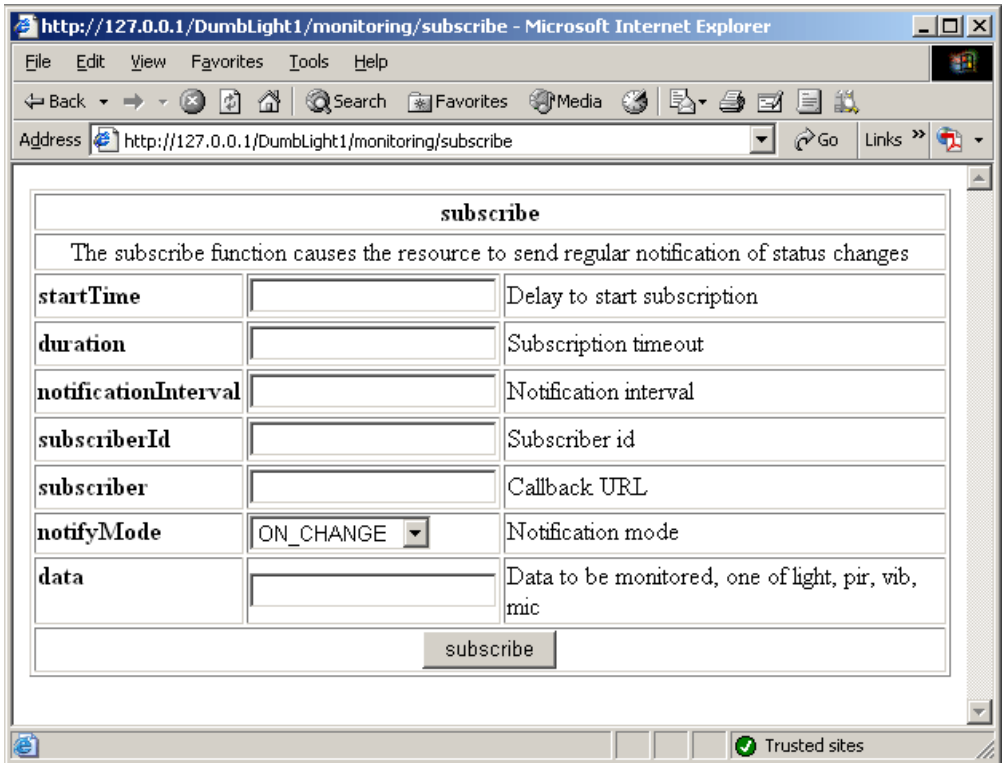

Figure 30: Generated user interface

Complex resources are transformed into a plain HTML document listing the nested elements as hyperlinks. Figure 31 shows a dynamically generated descriptor of a sensor node.

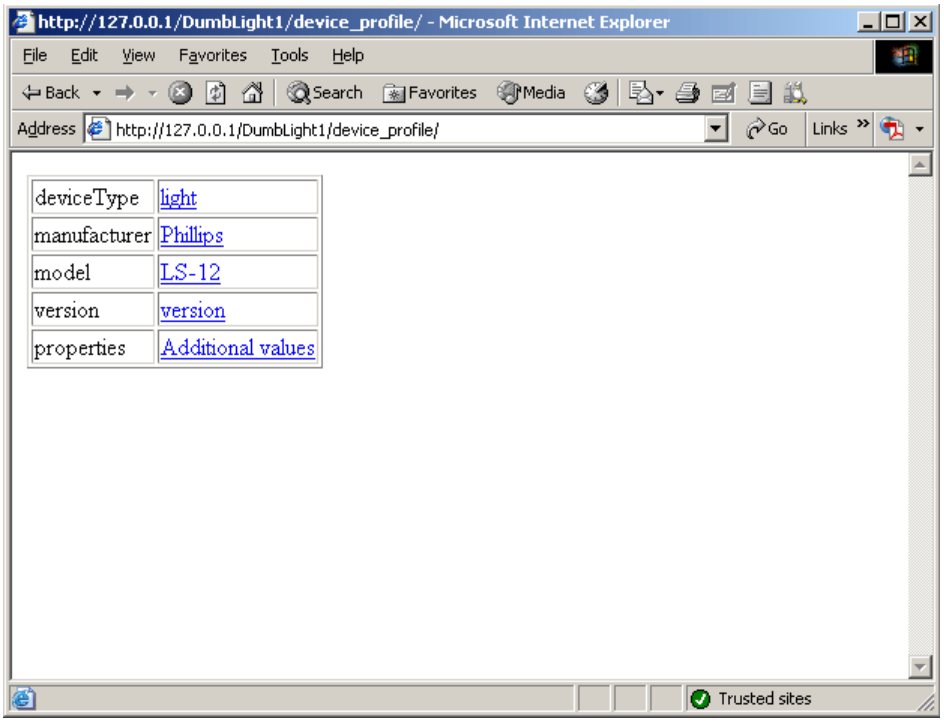

Figure 31: Resource descriptor transformed in the Web browser

5.1.4. Implementation on Embedded Hardware

While the existing applications are designed in an object-oriented manner, and are not constrained by resource limitations, pREST is also intended to interconnect very simple devices such as sensor nodes. The sensor hardware is capable of receiving data via a serial port and through radio communication. Sensing capabilities include a light sensor, a microphone, infrared signal reception and motion. The implementation utilizes the freely available µIP stack as a starting point.

The sensor node has a total of 64k flash ROM for firmware and applications, and 2k of RAM for application data. The memory footprint of the various components of the implementation is presented in Table 3.

	Firmware only	Firmware + µIP	Firmware + µIP + REST Application
Memory	500 Byte	1.818 Byte	2.008 Byte
Code	17.096 Byte	23.504 Byte	37.096 Byte

Table 3: Memory footprint of embedded pREST

The basic firmware provides for communication via serial line and radio, sensor operation and support for application processes. The µIP stack uses about 1,300 byte of memory which includes two buffers of 552 bytes (512 bytes application data + 40 bytes TCP/IP header) and the data structures to hold the application state.

The combined pREST and application layer does not add overly to memory usage. Only application buffers to store partial requests and subscriptions for data notification can make a substantial difference to code size - a consequence of the design decisions outlined in section 5.1.4.4. New functions can still be added to existing applications even though the available 2k of memory is almost exhausted.

5.1.4.1. Architecture

The pREST implementation on sensor nodes is implemented in C and is compiled with a GNU compiler for the MSP430 embedded processor. The firmware receives data packets through the serial port or the radio, and calls the µIP layer both upon data reception and at regular intervals to allow application-initiated data transfer.

The interface to the application layer is realized via two callback methods called by µIP, one for TCP communication and one for UDP data. Connection-oriented TCP communication is used for user interaction and querying and sends data only upon request. UDP is used to deliver data such as sensor readings asynchronously and is initiated on the client side.

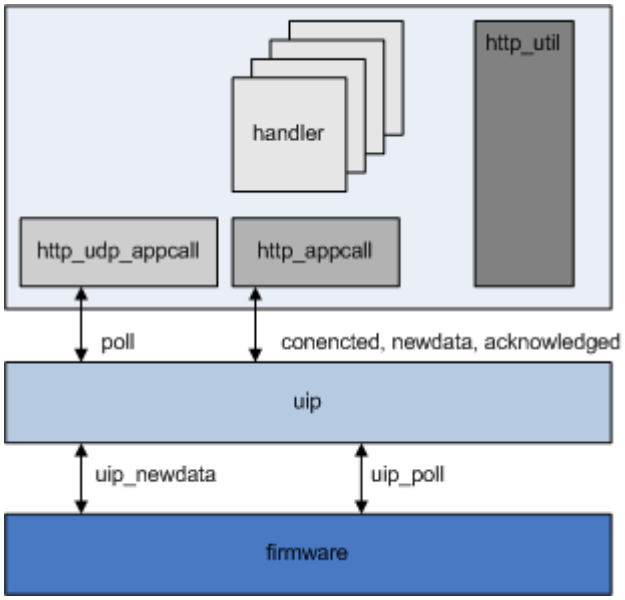

Figure 32: Embedded pREST server architecture

The UDP callback function http_udp_appcall is relatively simple, responding to poll events by the µIP layer, and invoking just one sub-function for message construction. The processing of client requests via TCP has a more complex flow of control. µIP notifies the TCP function http_appcall of events such as connection establishment, new data and acknowledgements. Upon reception of a request message the application parses the request for the requested path, method and parameters and dispatches the request to one of the handler methods. In addition the 'http_appcall'

function keeps track of the transmitted bytes and notifies the handler when Content-Length bytes have been received.

Besides the actual application logic contained in the handler functions, there is also a utility module e that provides functions for URL comparison, save string concatenation (without exceeding the buffer) and creation of HTTP response messages, which is used both by the callback and the handler functions.

5.1.4.2. Synchronous Interactions

When a complete IP packet is received by the firmware it is placed in the IP buffer which performs the necessary processing to determine port, protocol and sender address and invokes one of the application methods.

The application for TCP connections is capable of processing valid HTTP requests and sending back static descriptor documents or dynamic values in their string representation. The server is implemented as a state machine with a separate state for processing the request line, headers, body and sending the response message back. An outline of the state machine is presented in Figure 33.

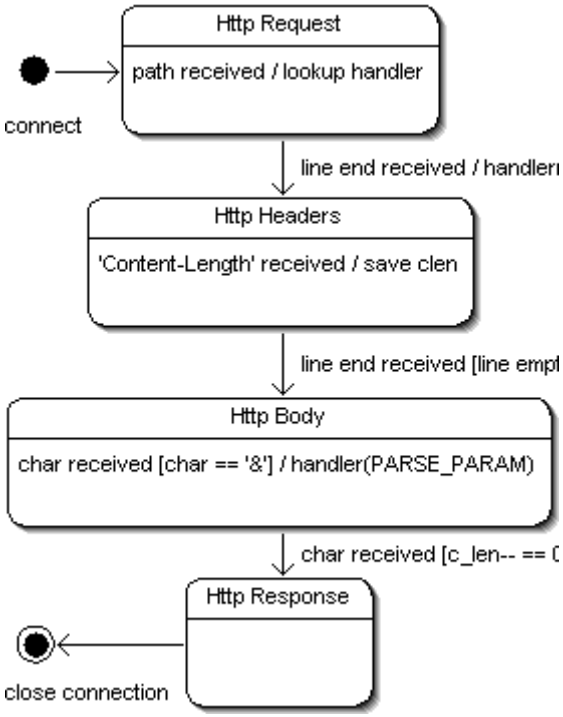

Figure 33: pREST server state diagram

The handler function for a URL is invoked several times during message processing with a flag parameter indicating the state of the processing. Immediately after the correct handler for the requested path is determined, the handler function is invoked with the flag set to INIT, to allow the handler to set default parameter values

etc. Upon reception of each parameter (detected by one of the separator characters '&', ';' or newline) in the message body, the received part is passed to the handler to parse the plain text and the parameter in a memory-efficient format. The parameter is kept in the application level buffer bound to the TCP session.

When the end of the request message has been detected the handler is invoked with the flag set to EXCUTE, and the application expected to place a response message in the IP buffer.

5.1.4.3. Asynchronous data delivery

The delivery of data to subscribers is initiated by a regular poll by the firmware. Depending on the type of subscription, data is sent in regular intervals or upon change of a sensor value.

For each subscription a separate UDP connection is kept. The data-producing function is polled regularly every 100 ms, and checks whether a subscribed-for parameter has changed or if an interval timer has run out. In both cases a message is sent containing the current status of the relevant parameters, otherwise the function returns with the ip_datalen variable set to zero and no IP packet is sent out. The subscription times out after a time specified in the request.

5.1.4.4. Implementation experience

During the implementation operational memory was the scarcest resource, particularly given the buffer space inherently required by network applications (see Table 3).

The requirement to allow configuration of the sensor via a telnet client makes the implementation more complex and requires additional buffer space. This is because data cannot be stored in the IP buffer until the whole request is received as each new character arriving overwrites the previous IP packet.

To process partial request reception, such as with telnet access, the application layer keeps a buffer of its own to store incoming data. The structure of HTTP messages allows the incoming request to be processed partially, one line or one parameter at a time. At the end of the line the input can be transformed into a more memory-efficient form to be stored until the end of the connection or simply discarded.

The mapping of URLs to content, for instance, is realized via a lookup table associating a URL with a function pointer. This way each URL has an own handler method associated with it in a design that trades code size for memory usage. After the handler function has been determined, the URL can be safely discarded since the 4-byte pointer uniquely identifies the data to be returned. On the other hand, code reuse is largely prevented.

5.2. Semantically Enhanced Data Space Layer

The section on the realization of loosely coupled service provisioning comprises of descriptions for appropriate implementations of the Semantically Enhanced Data Space (SEDS) and the Service Model Ontology (SMO) as proposed in the last chapter. Thus the SEDS is deployed in a peer-to-peer framework based on Representational State Transfer while the SMO is modeled as OWL DL ontology.

In the preceding chapter an interaction model and a service model of a Service-oriented Architecture in a pervasive computing environment were explained. The interaction model, i.e., the SEDS, describes a virtual space wherein, bound to interaction subjects, data can be placed that need to be exchanged between several interacting parties. Such data are represented as logics descriptions based on graph structures and do not name particular receivers. Hence, interactions are loosely coupled with respect to time, space and representation. The service model is based on the same considerations on loosely coupled interaction. The center of this model represents the specification of a desired service by the service consumer. This specification is embedded in the service request while the request can be seen as the actual interaction subject in terms of the interaction model. After the publication of the request in a spatially uncoupled manner each party providing the specified service may react.

The environment in which these models are deployed is composed ad hoc by the set of devices useful for the current context of the user. The resources and services of a device are represented through a uniform software abstraction and accessible through the universal access layer pREST. With regard to the processing of tasks on behalf of the user, the devices are also autonomous and may provide and consume their capabilities among each other in order to fulfill a demanded task. From a network-centric point of view devices can be seen as interacting peers in a peer-to-peer network and are therefore also realized in that way.

5.2.1. Realization of the Semantically Enhanced Data Space

As suggested above, the SEDS is realized as a self-contained software component with a well-defined interface derived from the operational interface proposed in section 4.2. However, as the SEDS is embedded in a particular peer of a peer-to-peer environment, the realization also includes an appropriate adaptation of the interface of the SEDS to the communication interface of this environment and vice versa. For the peer-to-peer environment a framework referred to as pREST is utilized. This framework is based on Hypertext Transfer Protocol (HTTP) communication in accordance to Representational State Transfer (REST) as described by Fielding et al. in [09]. This means that pREST includes server and client implementations as well as several tools and abstractions for the representation of served and requested re-

sources. Moreover, an integrated naming service allows peers to look up each other in a feature that especially supports discovery of the peer actually serving the SEDS. Further details are given in section 5.2.4 where the integration of the SEDS to the peer-to-peer framework is discussed.

Since the pREST-Framework is written in Java and there are numerous Java-based semantic web tools, SEDS is also implemented with Java 5. However, realizations related to the processing of RDF and OWL ontologies are based on the semantic web framework Jena 2.3 [50]. Jena provides a graph abstraction for RDF documents and various utilities to read data from these graphs and edit them with regard to higher-level logics like OWL DL. Moreover, Jena also serves as platform for the integration of third-party components like reasoning systems and query analyzers. Comparable open tools like the OWL-API [51] proved insufficient due to their lack of functionality and bugs in the implementation.

Implementation of the Semantically Enhanced Data Space is separated into three major parts. The actual data storage, i.e. the Data Space Environment, is addressed by the SEDS Core System implementation. This implementation is accessed through the SEDS Interface, i.e. a set of Java interface classes that realizes the operational interface introduced in section 4.2.3. The SEDS Server implementation realizes the connection of the Core System to the pREST-Framework and thus to the HTTP interface. Accordingly, it maps all methods provided by the SEDS Interface to a set of resources addressable with HTTP requests to certain URLs. Finally, the SEDS Client implementation represents the counterpart on the client side and provides stubs of the SEDS Interface based on the pREST client. As the SEDS Client is more intended for convenience than to achieve location or access transparency, a client may also directly utilize HTTP requests. In the following sections the Data Space Interface and its implementation are described in detail.

5.2.2. The SEDS Interface

The SEDS Interface realizes the operational interface explained in section 4.2.3. which means that the introduced operations are integrated in a set of interface classes which serve as template for the semantic of the Data Space Environment. Thus, all implementations of the interface classes are considered to exactly fulfill the claimed behavior. However, in the following sections the descriptions of interface classes are separated into management interfaces, querying interfaces, and feedback interfaces. The management interfaces deal with data manipulation, the querying interfaces the control of data retrieval, and the feedback interfaces the representation of retrieved data.

5.2.2.1. Management Interfaces

The management interfaces are derived from the organizational elements of the SEDS, i.e. Data Space Environment, Data Space, and Data Plane. Accordingly the methods of the interfaces implement the claimed functionality of the corresponding operations as defined in section 4.2.3 in an object-oriented fashion. The data of a Data Plane are supposed to be set only once since data manipulation within the Data Spaces is not allowed to prevent inconsistencies. Consequently, assigning data to a Data Plane twice causes an exception.

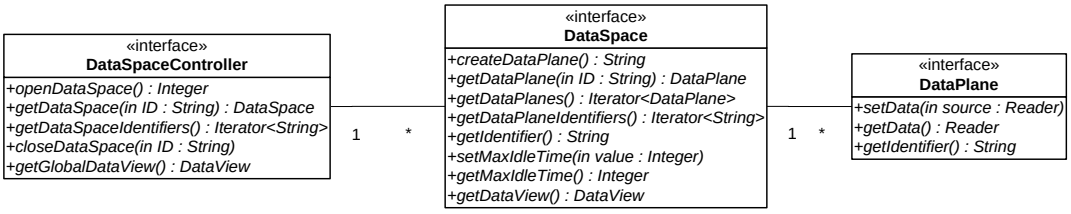

Figure 34: Interface classes for data organization and manipulation in the SEDS Interface

5.2.2.1.1. DataSpaceController

The DataSpaceController serves as an entry point to the Data Space Environment. The interface allows the creation and removal of Data Spaces driven by the party that originates an interaction subject.

Method(s)

openDataSpace	This method creates a new Data Space and returns the identifier for this Data Space. The identifier is supposed to be unique and never reusable even if the corresponding Data Space is removed.
getDataSpace	This method retrieves an interface to a Data Space for a particular identifier. If there is no Data Space assigned to the given identifier a null reference is returned.
getDataSpaceIdentifiers	From this method an interface to iterate over the identifiers of all Data Spaces known to the controller can be obtained.
closeDataSpace	Calling this method removes a Data Space. If there is no Data Space with the passed identifier this method has no effect.

getGlobalDataView	This method returns an interface to a composite view on all Data Spaces in the Data Space Environment. The global data view is intended as entry point for parties that are rather reacting to the actions of others than originating interaction subjects themselves.

5.2.2.1.2. DataSpace

The interface DataSpace represents a Data Space and thus a collection of Data Planes. It provides the access to functionality needed for adding and retrieving Data Planes and also offers a composite view on the graphs contained in all Data Planes.

Method(s)

getIdentifier	This method returns the identifier of the corresponding Data Space.
createDataPlane	Calling this method creates of a new Data Plane. The unique identifier of the new Data Plane is returned.
getDataPlane	This method returns an interface to the Data Plane with the passed identifier. If this Data Plane does not exist a null reference is returned.
getDataPlanes	This method returns an interface to iterate over the collection of Data Planes contained in the Data Space.
getDataPlaneIdentifiers	This method returns an interface to iterate over the identifiers of the collection of Data Planes contained in the Data Space.
getMaxIdleTime	By this method the maximum idle time of the Data Space is obtained. The idle time should be represented in seconds.
setMaxIdleTime	This method allows for setting the maximum idle time of a Data Space in seconds.

5.2.2.1.3. DataPlane

The interface DataPlane represents a Data Plane and allows the insertion of new data to a Data Space. As long as no content is assigned to the corresponding Data Plane it should be disregarded when matching patterns.

Method(s)

getIdentifier	This method returns the unique identifier of the Data Plane.
setData	This method allows the assignment of an RDF graph to the Data Plane. Before the graph is set to the Data Plane the consistency is verified with regard to the entire graph spanned by all Data Planes of the Data Space. If any problems are recognized the operation is cancelled and the new graph rejected. However, since changing the data in a Data Space is not supported, the graph of a Data Plane can only be successfully assigned once.
getData	This method returns the model contained in the data plane or a null reference if no model is yet assigned.

5.2.2.2. Querying Interfaces

The Data Space Environment supports active and reactive retrieval of data. Data Views are introduced as a common abstraction for all operations related to the data retrieval. Data Views are provided by the interfaces DataSpace and DataSpaceController. Consequently, a Data View allows either accessing all the data from one Data Space or from the entire Data Space Environment. However, the Data View of the Data Space Environment does not consider the data from all Data Spaces as a composite graph; but rather represents a delegate interface to all Data Views of the single Data Spaces.

While the active retrieval of data is realized with a simple method call, reactive retrieval is based on a listener software pattern. The Data Space client is required to implement a callback interface and to subscribe this implementation combined with the Pattern to be evaluated. On the appearance of data matching the Pattern, the assigned callback implementation should to be executed by the implementation of the Data Space Environment. The corresponding interfaces are illustrated in Figure 35 and explained below.

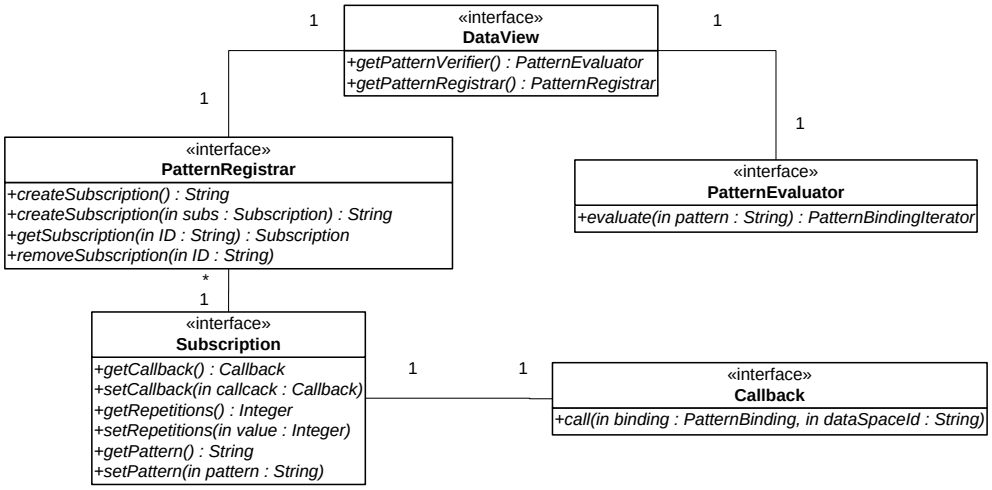

Figure 35: Interface classes for data retrieval in the SEDS Interface

5.2.2.2.1. DataView

The interface DataView represents a Data View either for all the Data Planes of one Data Space or for all Data Spaces in the entire Data Space Environment. The latter case is addressed by the interface DataSpaceController. Thus, the actual scope of the implementation depends on the object that provides this interface.

Method(s)

getPatternEvaluator	This method returns the interface of a component that allows the immediate evaluation of a Pattern.
getPatternRegistrar	Calling this model returns the interface of a component that supports the subscription for the appearance of data in the data view.

5.2.2.2.2. PatternEvaluator

The interface PatternEvaluator supports the direct evaluation of patterns. The graph actually addressed by the evaluator depends on the corresponding data view.

Method(s)

evaluate	This method retrieves all sub-graphs that match a given graph pattern. The order of these so called Pattern Bindings is non-deterministic. Thus, the order may vary with each call of this method with the same pattern.

5.2.2.2.3. PatternRegistrar

The interface PatternRegistrar allows subscribing for the appearance of a particular graph. Subscriptions should be evaluated for the first time when being registered

and subsequently only on the appearance of new data in the corresponding Data View.

Method(s)

createSubscription	This method creates a new Subscription and returns an identifier for this Subscription. If a Subscription is also passed as a parameter to this method, the fields of the new Subscription are also initialized with the values of the passed one.
getSubscription	This method returns the Subscription for the given identifier. If the identifier is not assigned to any Subscription, a null reference is returned.
removeSubscription	Calling this method removes the Subscription with the given identifier. If no Subscription can be found for the identifier, this method has no effect.

5.2.2.2.4. Subscription

The Subscription represents all information relevant for the reactive delivery of data. A Subscription is intended to be first activated when all fields are set, i.e., each set-method was called once to initialize the corresponding field of the implementation.

Method(s)

setCallback	This method assigns a callback function to the subscription.
getCallback	The invocation of this method returns the Callback referenced by the Subscription.
setRepetitions	This method returns the number of repetitions for the notification of Pattern matches via the Callback referenced by the subscription. With each successful execution of the Callback, i.e. each notification of a Pattern Binding, the number of repetitions is decreased by one. If the value reaches zero, the subscription should be deactivated. The corresponding Pattern is not evaluated any more against the Data View and no further notifications are published.
getRepetitions	This method returns the current number of repetitions for a Subscription. If the value reaches zero, the subscription should be deactivated.

setPattern	With this method the pattern to be evaluated is assigned to the Subscription.
getPattern	Calling this method returns the pattern currently assigned to the Subscription.

5.2.2.2.5. Callback

The Callback is an interface of the subscriber intended to serve as the endpoint for publishing the occurrence of new data matching the pattern of a Subscription.

Method(s)

call	This method is called if the pattern of the Subscription the Callback belongs to matches any data of the corresponding Data View. The passed parameters identify the Data Space wherein the match was found and represent an appropriate Pattern Binding. The actual implementation of this method depends on the object providing the Callback.

5.2.2.3. Feedback Interfaces

Retrieval of data from a Data Space is realized through evaluating patterns on a composite view of all graphs represented in the appropriate Data Planes. Although the size of the retrieved sub-graphs is limited by the structural dimensions of the pattern, processing RDF graphs and any OWL DL semantics they might contain is very complex. Thus, clients of the Data Space Environment should not process these graphs, and the representation of retrieved data, i.e. the Pattern Binding, is kept simple. Instead of returning the entire sub-graph matching a pattern only the nodes and relations indicated by the free variables in the pattern should be returned. A Pattern Binding is thus a list of name-value pairs. The corresponding Java interfaces are illustrated in Figure 36 and explained below.

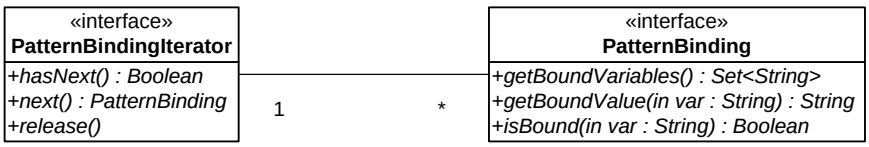

Figure 36: Class diagram of the interfaces used for data retrieval

5.2.2.3.1. PatternBinding

The interface PatternBinding represents a Pattern Binding, i.e. precisely one possible answer to a pattern including free variables. The values bound to the variables may either be some URIs representing node and relation names or atomic values like integers, string or floats.

Method(s)

getBoundVariables	This method returns the names of all variables that were bound while matching the corresponding pattern.
isBound	This method checks whether a variable of the given name is bound or not.
getBoundValue	This method returns the value assigned to the given variable in the current Pattern Binding.

5.2.2.3.2. PatternBindingIterator

The PatternBindingIterator represents an interface to access the collection of all Pattern Bindings for a particular pattern in a Data Space. This collection should be in a non-deterministic order. Hence, two instances of the PatternBindingIterator representing the set of Pattern Bindings for the same pattern matched twice against a certain Data Space are not guaranteed to return the single Pattern Bindings in the same order.

Method(s)

hasNext	This method checks whether or not there are any more bindings of a particular pattern in the underlying list of results.
next	This method returns the next Pattern Binding for the underlying collection or nothing if there are no more Pattern Bindings.
release	Calling this method indicates that all remaining bindings for the corresponding Pattern can be discarded.

5.2.3. The SEDS Core System Implementation

The implementation of the SEDS Core System addresses two views, the view of a client of the Data Space Environment and the view of the developer configuring the behavior of the Core System. The view of the Data Space client is implemented according to the SEDS Interface introduced in the previous section. Thus, all interface classes are realized with normal Java classes that implement the behavior claimed. For the representation of graphs inside the Data Planes Jena models are utilized so that each Data Plane can be seen as the envelope of exactly one model. These models are abstractions of RDF graphs and provide actual access to a collection of RDF triples, i.e. subject, predicate and object. The generality of the models allows them to serve as containers for all RDF-based data representations so they also support the representation of OWL ontologies. The classes of the SEDS Core System and the implemented interface classes are illustrated in Table 4.

The view of the developer is realized with a configuration system that allows, for instance, the definition of Hooks for significant actions in the Data Space Environment. The classes of the SEDS Core System are not supposed to be accessed directly as a special factory is applied that only provides a view on the SEDS Interface and hides the details of the implementation. In the following sections the representation and evaluation of patterns is explained and the configuration system described.

Class	Nested Class	Interface
ModelStorageManager		DataSpaceController
ModelStorage		DataSpace
ModelEnvelope		DataPlane
QueryManager		Dataview PatternRegistrar PatternVerifier
	Binding	PatternBinding
	BindingIterator	PatternBindingIterator
GlobalQueryManager		Dataview PatternRegistrar PatternVerifier
	BindingIterator	PatternBindingIterator
SimpleSubscription		Subscription

Table 4: SEDS Core System classes with the implemented interface classes

5.2.3.1. Pattern Representation and Evaluation

The patterns utilized to retrieve data from a Data Space are usually graph structures containing variables as placeholders. If any sub-graph in the entire Data Space matches the pattern, i.e. a set of data can be found as substitution for the variables, these data are considered to be a match. Powerful representations for patterns are provided by query languages. Since the data within the space are described in OWL/RDF, the RDF query language SPARQL [52] is used for the representation of patterns in the SEDS Core System. The syntax of SPARQL is similar to that of SQL. A SELECT-clause defines the variables to be bound by the query and a WHERE-clause the actual graph pattern to be matched. Optional bindings and filter criteria may also be defined for the graph pattern to enforce such as certain types of literals or to exclude resulting sub-graphs with undesirable characteristics.

```
<rdf:RDF
    xmlns:dialog="http://host/dialog-terms.owl#"
    ...
    xml:base="http://host/doc1">
  ...
  <dialog:Question rdf:ID="RequestForCurrentTime"/>
</rdf:RDF>
```

Figure 37: OWL fragment describing an abstract question

```
<rdf:RDF
    xmlns:dialog="http://host/dialog-terms.owl#"
    xmlns:doc1="http://host/doc1#"
     ...
    xml:base="http://host/doc2">
  ...
  <dialog:Answer rdf:ID="ResponseCurrentTime"/>
  <rdf:Description rdf:about="doc1#RequestForCurrentTime">
    <dialog:hasAnswer rdf:resource="#ResponseCurrentTime"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 38: OWL fragment describing an answer to the question of Figure 37

```
PREFIX dialog: <http://host/dialog-terms.owl>

SELECT ?question ?answer
WHERE { ?question dialog:hasAnswer ?answer }
```

Figure 39: Example of a SPARLQL query

```
question=<http://host/doc1#RequestForCurrentTime>,
answer=<http://host/doc2#ResponseCurrentTime>
```

Figure 40: A binding for the query illustrated in Figure 39

Patterns are matched against a composite view of all data in a Data Space so the models from all Data Planes are merged for the evaluation. Consider, for instance, a Data Space that contains two data planes with the RDF graphs illustrated in Figure 37 and Figure 38 as content. Both graphs are defined with reference to an ontology termed 'http://host/dialog-terms.owl', which defines the concepts 'Question' and 'Answer' as well as the relation 'hasAnswer'. Performing the query illustrated in Figure 39 on this Data Space gives the binding visualized in Figure 40. However, for the interpretation of SPARQL queries in Jena models the ARQ [53] toolbox is utilized. ARQ uses an object abstraction for the queries so it preprocesses, i.e. parses and verifies queries first to make them usable. To avoid repeated preprocessing of a particular query, all queries are cached centrally for the entire Data Space Environment.

A Data Space supports on the one hand the direct evaluation of queries on demand and on the other registration of queries to be evaluated when any changes in the data occur. The query registration includes a Callback that allows notifying the originator about a variable binding, i.e. a Pattern Binding, if evaluation of the corresponding query is positive. However, if this originator is temporarily unavailable the corresponding Callback is not executable. This means that all Callbacks that cannot be executed immediately when a query is evaluated as positive are added to a Callback Controller. This Callback Controller tries continuously to execute the Callbacks in short intervals. With each try the duration of these intervals is increased. The total number of retries is limited by a configurable threshold. If this threshold is exceeded, the Callback is canceled. Callbacks are also canceled if the variable binding

they are supposed to notify about becomes invalid, e.g., due to changes in the corresponding Data Space.

5.2.3.2. Data Space Configuration

Developers are allowed to configure details of the SEDS Core System behavior while utilizing a special configuration object when creating an instance for DataSpaceController with the corresponding factory. This object allows definition of the utilized reasoning system, some Hooks, and Model Injectors. The specified reasoning system is used whenever data need to be inferred from the raw data in the Data Planes. Thus, the logic actually supported by the Core System implementation and consequently the comprehension of meta-knowledge may be changed at any time according to needs. The SEDS Core System implementation is not necessarily limited to the representation of OWL DL.

The Hooks included in the Data Space Configuration are similar to event handlers. There are some predefined events raised by the SEDS Core System that signal, for instance, the appearance of new data or the creation of a new Data Space. These events are delegated to the Hooks for processing. However, Hooks are not executed in parallel threads but rather evaluated in the flow of the actions that originate the events, and the actions are suspended until the processing of all Hooks is completed. Hooks may even throw up critical exceptions to explicitly interrupt the program flow they are executed in. This way, for instance, the insertion of data to a Data Plane can be rejected if such data fail to meet the required criteria.

The third instrument that influences the SEDS Core System is the Model Injector. As the name suggests, Model Injectors are used to add particular information when reasoning the graph of a Data Space. An injected model may represent dynamically provided knowledge like the current environment context such as the current time or even an ontology bridging two terminologies. Such type of ontology may serve as mediators in interactions among multiple parties using different ontologies for the description of data added to Data Spaces.

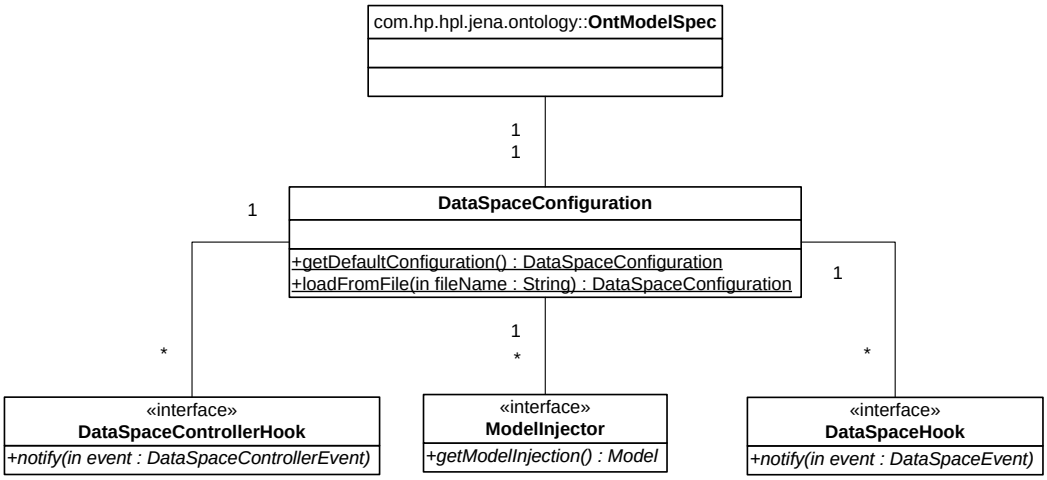

Figure 41: Overview of the classes for Data Space Configuration

5.2.3.2.1. DataSpaceConfiguration

The DataSpaceConfiguration aggregates the instruments for Data Space Configuration introduced above. An instance of this class is required when creating an instance for the interface DataSpaceController.

Method(s)

getDefaultConfiguration	This method provides the minimum configuration required to run the SEDS Core System. The returned configuration does not include Hooks nor Model Injectors.
loadFormFile	This method loads a Data Space Configuration from the file of the given name. This file needs to be a simple text file containing a list of name-values pairs separated with a carriage return-character and line-feed-character. Each name-value pair is also required to be separated by an equal sign. The supported names for these pairs are listed in the table below whereby the index 'k' in the names is a placeholder for a counter that starts with zero. Thus, each name with an index may appear many times in the file.

5.2.3.2.2. ModelInjector

The ModelInjector represents a container for a single model. This model is included in the set of models provided by the Data Planes when retrieving data from a Data Space.

Method(s)

getModelInjection	This method returns the model contained in the corresponding Model Injector.

5.2.3.2.3. DataSpaceHook

The DataSpaceHook represents a listener for events that notify about changes in Data Spaces. The interface serves as a callback for the SEDS Core System.

Method(s)

notify	This method is called by the SEDS Core System to process events that notify about changes in any Data Space in the entire Data Space Environment. The event types that may be passed to this method are specified in Table 5.

Event	Description
BeforeDataAssignmentEvent	Is raised before data is assigned to a Data Plane.
AfterDataAssigmentEvent	Is raised after data is assigned to a Data Plane.

Table 5: List of all events that notify about changes in a Data Space

5.2.3.2.4. DataSpaceControllerHook

The DataSpaceControllerHook represents a listener interface for events that notify about changes in the Data Space Environment. The interface serves as a callback for the SEDS Core System.

Method(s)

notify	This method is called by the SEDS Core System to process events that notify about changes in the Data Space Environment. The event types that may be passed to this method are specified in Table 6.

Event	Description
DataSpaceOpenedEvent	Is raised on opening a new Data Space.
DataSpaceClosedEvent	Is raised on closing an existing Data Space.

Table 6: List of all events that notify about changes in the Data Space Environment

5.2.4. Integration into the pREST Access Layer

The SEDS Core System is integrated into the pREST access layer to make it available in a distributed setting. Implementation of the data space adapters is based on the software abstraction for resources included in the pREST access layer. These resources represent mappings to particular entities and in the software model are distinguished according to the types of operations supported by the represented entities. The types are realized through the marker interfaces GET, HEAD, POST, PUT, and DELETE that correspond to the HTTP methods with the same name. Delegate methods of these interfaces allow access to the attached entities through writing or requesting raw data along with metadata represented by a marker interface named Content. The resource abstractions may be arranged in trees so that resources may be bound to Node elements wherein, the root node is attached to a special Servlet, i.e. some kind of HTTP request handler that maps the path of a URL to a path in the tree of nodes.

5.2.4.1. SEDS Server Implementation

The SEDS Server implementation provides a set of adapter classes that are intended to wrap the interface classes of the SEDS Interface. The adapter classes implement a combination of the interface classes Head, Get, Put, Post, and Delete from the resource model according to the features of the wrapped object. Hence, each adapter represents a pREST resource. The adapter classes also implement the Node interface introduced above so the entire adapter setting is arranged as a tree. In practice the adapter model is not supposed to be utilized directly but a special server peer should be instantiated which binds the adapter model in the background to the pREST server. This peer also registers the name of the data space controller at the naming service of pREST to enable collocated peers to discover the SEDS Server.

A coarse-grained view of the adapter model is provided by Figure 42. All classes implement the interfaces Resource and Node from the pREST resource model but for simplicity's sake only the most important methods are visualized and explained below. In addition to the illustrated adapter classes there are also adapters for the atomic data fields like the maximum idle time of a data space or the pattern of a subscription. These adapters support HTTP requests of the types HEAD, PUT, and GET whereby HEAD acknowledges the existence of a resource, PUT sets the value of the entity corresponding to this resource, and GET returns the value. The transferred data need to be represented as plain text. However, adapters for the Callback of a Subscription are realized in a special way. In the HTTP interface of the Subscription the Callback is only referenced, while the actual implementation of logic that is supposed to handle the Patten Binding is realized on the client side. An adapter for a Subscription thus only stores a callback-URL that points to the corresponding client

implementation. When notifying a Pattern Binding with a Callback, the bound variables and their values are represented as a name-value list.

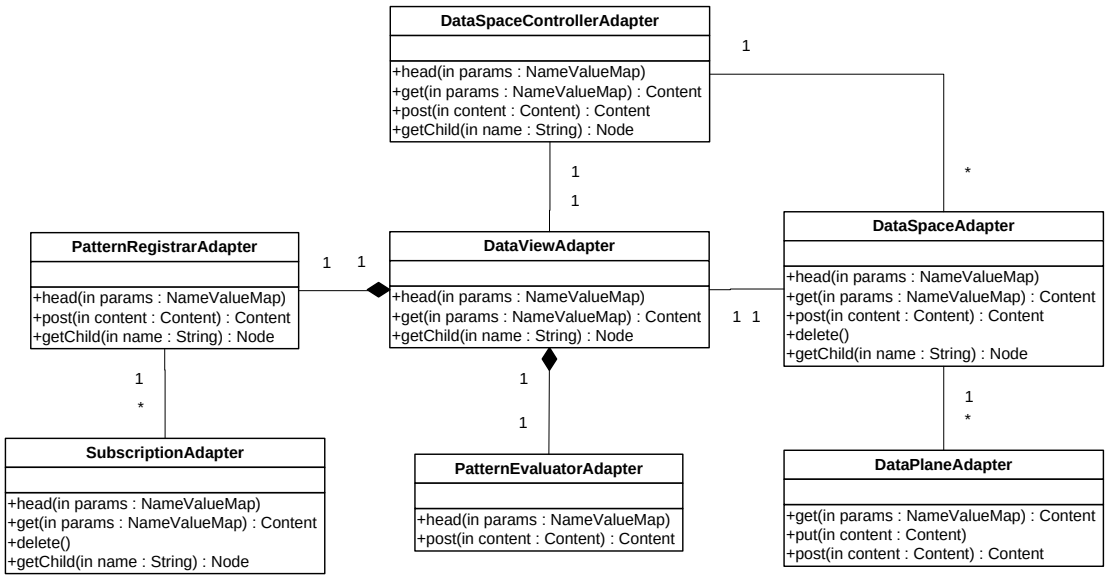

Figure 42: Coarse-grained class diagram of the adapters for the SEDS Server

The following subsections describe the entity-specific mapping to the common pREST methods, HEAD, GET, POST, etc.

5.2.4.1.1. DataSpaceControllerAdapter

The DataSpaceControllerAdapter wraps any instance of the interface DataSpaceController and maps HTTP requests to the methods of this interface.

Method(s)

head	This method allows verifying whether or not the adapter is available.
get	A call of this method returns a simple text entity that contains the name of the data view and the identifiers of all contained data spaces as a list. Passed parameters are ignored.
post	This method initiates the creation of a new Data Space. The identifier of the new Data Space is returned as a text entity. Passed parameters are ignored.
getChild	This method returns either an adapter for the global Data View or an adapter for a Data Space according to the passed name.

5.2.4.1.2. DataSpaceAdapter

The DataSpaceAdapter wraps any instance of the interface DataSpace and maps HTTP requests to the methods of this interface.

Method(s)

head	This method allows verifying whether or not the adapter, i.e. the corresponding resource is available.
get	The method returns the name of the maximum idle time adapter and the identifiers of all contained Data Planes as a list. This list is encoded as plain text. Passed parameters are ignored.
post	This method initiates the creation of a new Data Plane. The identifier of the new Data Plane is returned as plain text. Passed parameters are ignored.
delete	The method removes the wrapped data space and consequently all contained data planes.
getChild	This method returns either an adapter for the maximum idle of the Data Space or an adapter for a Data Plane according to the passed name.

5.2.4.1.3. DataPlaneAdapter

The DataPlaneAdapter wraps any instance of the interface DataPlane and maps HTTP requests to the methods of this interface.

Method(s)

head	This method allows verifying whether or not the adapter, i.e. the corresponding resource is available.
get	A call of this method returns the data contained in the Data Plane as a RDF graph. Passed parameters are ignored.
put	This method sets the data of a Data Plane. Therefore the given data need to be represented as a RDF graph.
post	The method has exactly the same behavior as the put-method above. It does not return any data.

5.2.4.1.4. DataViewAdapter

The DataViewAdapter wraps any instance of the interface DataView and maps HTTP requests to the methods of this interface.

Method(s)

head	This method allows verifying whether or not the adapter, i.e. the corresponding resource is available.
get	The method returns the name of the adapter for the Pattern Evaluator and the name of the adapter for the Pattern Registrar. The names are encoded as plain text.
getChild	This method returns either an adapter for the Pattern Evaluator or an adapter for the Pattern Registrar according to the passed name.

5.2.4.1.5. PatternEvaluatorAdapter

The PatternEvaluatorAdapter wraps any instance of the interface PatternEvaluator and maps HTTP requests to the methods of this interface.

Method(s)

head	This method allows verifying whether or not the adapter, i.e. the corresponding resource is available.
post	This method executes the evaluation of a pattern passed as a parameter to this method. The pattern is required to be contained in a text entity. The results of this method, i.e., potential Pattern Bindings are encoded as a list of name-value pairs. Therefore, the variable names in the Pattern Bindings are given separate name prefixes according to Pattern Binding.

5.2.4.1.6. PatternRegistrarAdapter

The PatternRegistrarAdapter wraps any instance of the interface PatternRegistrar and maps HTTP requests to the methods of this interface.

Method(s)

head	This method allows verifying whether or not the adapter, i.e. the corresponding resource is available.
post	This method creates a new subscription. Optionally, a list of name-value pairs may be passed to this method which contains the number of repetitions, callback-URL and pattern. The values are used to set the fields of the Subscription directly. If no parameters are passed to this method, the fields of the Subscription need to be initialized by separately putting data to the corresponding adapters. The method returns the identifier of the newly created subscription as plain text.

getChild	This method returns an adapter for a subscription according to the passed name.

5.2.4.1.7. SubscriptionAdapter

The SubscriptionAdapter wraps any instance of the interface Subscription and maps HTTP requests to the methods of this interface.

Method(s)

head	This method allows verifying whether or not the adapter, i.e. the corresponding resource is available.
get	The method returns the names of the adapter for the number of repetitions, the adapter for the callback-URL, and the adapter for the pattern. The adapter names are represented as plain text.
delete	The method removes the wrapped Subscription.
getChild	This method returns according to the passed name either an adapter for the number of repetitions or an adapter for the callback-URL or an adapter for the pattern.

5.2.4.1.8. The Resulting HTTP Interface

As proposed in the preceding sections each adapter is given a name with respect to the Node abstraction of the pREST resource model. While adapters for Data Spaces, Data Planes, and Subscriptions in each case are named in line with the identifier of the wrapped entities, the names of all other adapters are fixed. However, paths in the trees resulting from the concatenation of node names should be represented by the path segments of URLs so that each adapter is addressable with a particular URL. Figure 43 illustrates the navigation scheme in this tree as a graph since the Data View exists on a Data Space controller as well as on a Data Space. The HTTP methods supported by the corresponding adapters are once again annotated in a compact form. All fixed names and name prefixes are underlined.

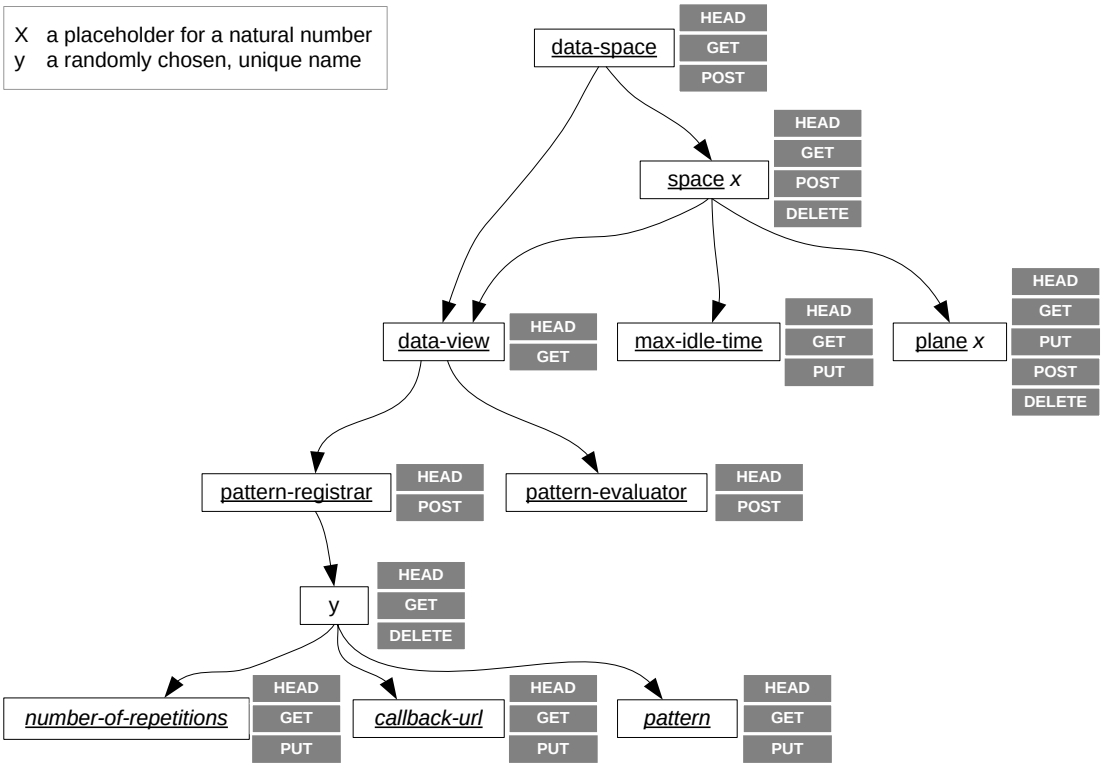

Figure 43: Navigation schema in the HTTP interface of the SEDS Server

Based on this schema, a Data Plane with the identifier '21' in a Data Space with the identifier '1' may be addressed with the relative URL '/data-space/space1/plane21'. Setting the pattern for a subscription '7abcd5d9' on the global Data View may be realized with a PUT-request to '/data-space/data-view/7abcd5d9/pattern'.

5.2.4.2. SEDS Client Implementation

Integration of the SEDS to the pREST access layer also requires a client implementation. This client implementation is composed by a set of stubs re-implementing the interface classes of the SEDS Interface. The stubs create appropriate HTTP requests for each method call on the interface classes and send them to the SEDS Server, i.e., the adapter implementations. Consequently, the URLs receiving these requests, and the encoding utilized to represent the transmitted contents are aligned to the implementation of the adapter classes for the corresponding data space elements. Table 7 thus only summarizes the stub classes and the interface classes they implement. However, the stub implementations are not accessed directly. Like the classes of the Core System they are rather hidden behind a factory which only provides a view on the SEDS Interface.

Class	Nested Class	Interface
DataSpaceControllerStub		DataSpaceController
DataSpaceStub		DataSpace

DataPlaneStub		DataPlane
DataViewStub		DataView
	PatternRegistrarStub	PatternRegistrar
	PatternEvaluatorStub	PatternEvaluator
	PatternBindingIteratorStub	PatternBindingIterator
SubscriptionStub		Subscription

Table 7: Stub classes and the implemented interface classes of the SEDS Interface

In section 5.2.4.1 a special treatment of Callbacks for the reactive retrieval of data from the HTTP interface is suggested. In fact the Callback logic is supposed to be kept at the client while the server is given a URL to invoke this callback logic on demand. The SEDS client thus also includes an adapter for Callbacks. Since the stubs themselves are stateless, a Callback Manager stores and controls the associations between Callbacks, adapters and the corresponding Subscriptions on the server side.

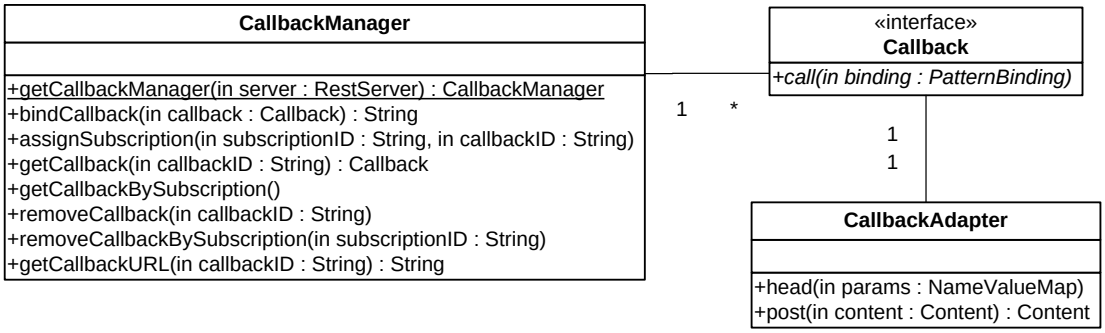

Figure 44: Classes for callback management on the client side

5.2.4.2.1. CallbackManager

The CallbackManager is a singleton class that manages the binding of Callbacks to URLs. An instance of the pREST server is utilized to make Callbacks available on an HTTP interface. In addition to the mapping of Callbacks to URLs the CallbackManager also stores the association of each Callback to the corresponding Subscription on the server side.

Method(s)

getCallbackManager	This method returns an instance of the CallbackManager for a given pREST server. Since the CallbackManager is a singleton, this method always retrieves the same instance for each server.
bindCallback	The method binds the given Callback on a randomly chosen, unique identifier to the pREST server of the CallbackManager. An identifier for the binding is also returned.
assignSubscription	This method allows assigning the identifier of a Subscription from the server side to the corresponding implementation of the contained Callback on the client side.
getCallback	Calling this method returns the Callback for the given identifier or a null reference if such a Callback cannot be found.
getCallbackBySubscription	This method returns a Callback by the identifier of the corresponding Subscription from the server side. This method is necessary since the SEDS Interface - and especially the Query Registrar - only provides the identifier of a Subscription to the client system but not the identifier for the association between Callback and callback-URL.
removeCallback	This method removes the Callback with the given identifier.
removeCallbackBySubscription	This method removes a Callback by the identifier of the corresponding Subscription at the server side.
getCallbackURL	Calling this method returns the complete URL of the CallbackAdapter for the given Callback identifier.

5.2.4.2.2. CallbackAdapter

The CallbackAdapter is similar to the adapters of the data space server based on the pREST resource model - its intention is to map HTTP requests to the methods of the callback interface.

Method(s)

head	This method allows verifying whether or not the adapter, i.e. the corresponding resource, is available.
post	This method executes the call-method of the Callback corresponding to this adapter. The Pattern Binding as well as the Data Space identifier should be contained in a list of name-value pairs given as parameters to this method. All variables are supposed to have the same name prefix.

5.2.5. Realization of the Service Model Ontology

Section 4.2.3 introduces the service model ontology. All data structures to be exchanged between the Service Requester and the Service Provider are represented as concepts in terms of Description Logic. This semantic approach for the representation of data is intended to support loosely coupled interactions. For instance, the interacting parties are allowed to understand each other although using slightly different terminologies to express themselves. Indeed, the terminologies need to be declared on common base concepts. As argued in 4.2.4, OWL DL is a suitable language for the realization of the Service Model Ontology. In OWL concepts are referred to as classes while a property is the correspondent to a relation or a role assertion. Apart from classes and properties there are also atomic data types like integers or strings derived from XML Schema [29]. Accordingly, a separate type of property is dedicated to these simple data types, the so called 'data type property'. One extension special to OWL is restrictions. Restrictions allow definition of the cardinality of properties, and can thus be utilized to enforce a special structure for a data set.

The concepts of the Service Model Ontology are divided into concepts for the service specification and concepts for the service execution. This means that the realization of the Service Model Ontology is also a composition of two separate ontologies: the Service Specification Ontology and the Service Execution Ontology. These ontologies basically define OWL representations for the base concepts and their relations as introduced in 4.2.3. Additionally, common characteristics of these concepts and relations are modeled with special classes and properties. This refinement of the structural design for the realization is specifically addressed in the following sections.

5.2.5.1. Service Specification Ontology

The Service Specification Ontology is closely oriented to the concepts explained in 4.2.3. All concepts are realized as OWL classes as illustrated in Figure 45. The relations between the concepts are mapped to OWL object properties. Properties

intended to be used either once for each individual or never are also tagged as functional.

Since all relations are supposed to have the cardinality one, cardinality restrictions are introduced for the remaining object properties and their domains OWL, as highlighted with a hatched background in Figure 45. These restrictions require individuals of the corresponding classes to bind all restricted properties exactly once. For instance, an individual of the class *Update* needs to be related with exactly one individual of *PartialState* through the property *hasHead.*

The list of arguments utilized in the representation of Partial States is thus realized as a recursive list in the Service Specification Ontology. The end of an instance of this list may either be represented by a missing *hasRest*-property or by a bottom element referred to as *nil*, i.e. the corresponding individual that serves as a constant here.

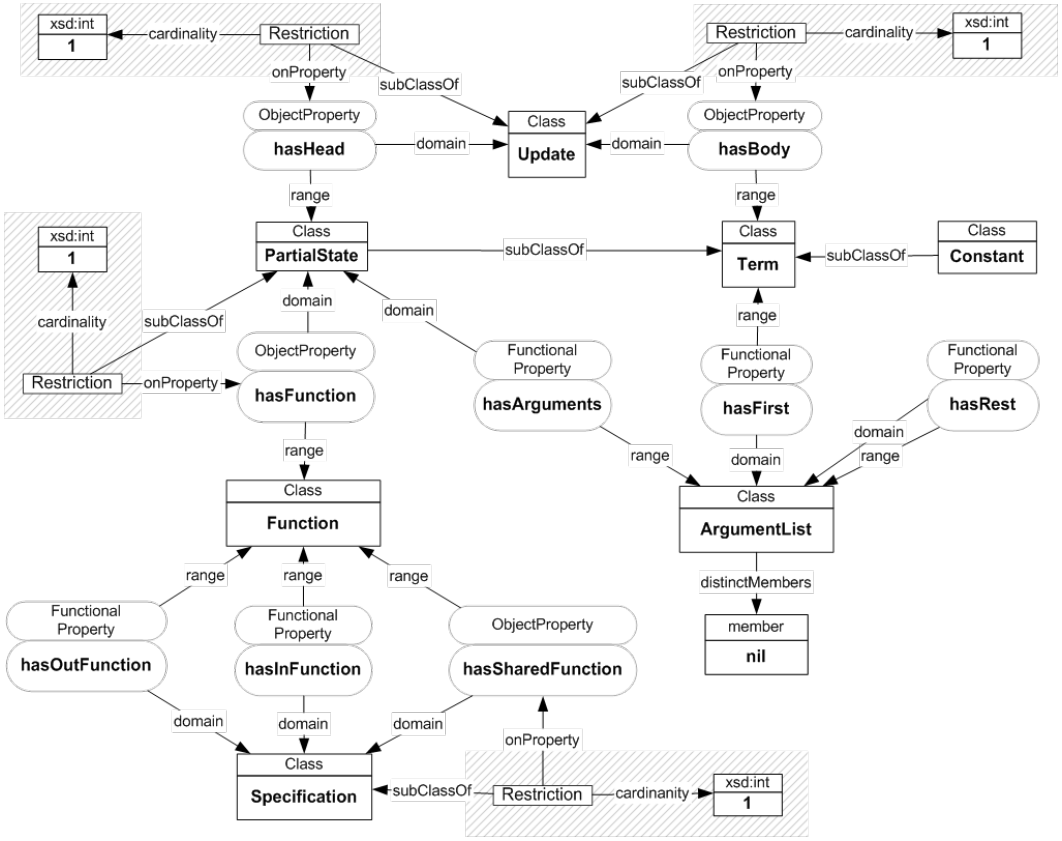

Figure 45: Core classes of the Service Specification Ontolgy and their properties.

Communication of explicit data values in service provisioning is realized with Evaluations, i.e. tuples relating a Constant to a Partial State. The same structural appearance is required for the Precondition which expresses the need for a certain initial value of a Partial State. Accordingly, the class *PartialValue* is introduced to the Service Specification Ontology. This class enforces the required structural constraints and serves as a base class for the classes *Precondition* and *Evaluation* as

highlighted with a hatched background in Figure 46. The properties are again constrained with cardinality restrictions which enforce each individual of *PartialValue* to reference precisely one individual of the class *PartialState* and precisely one individual of the class *Constant*.

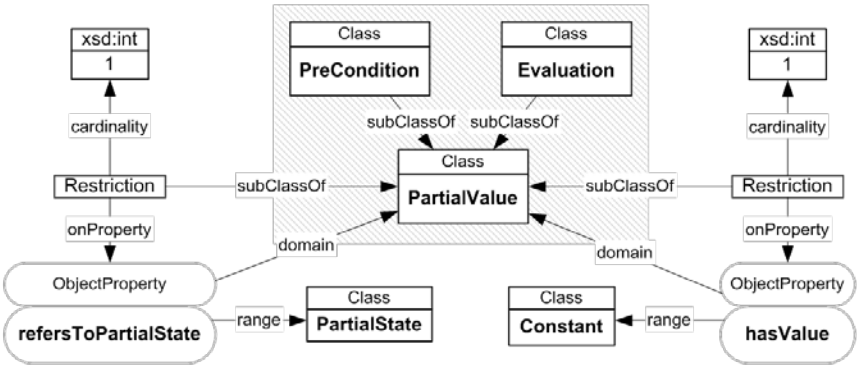

Figure 46: Data assignment constructs defined in the Service Specification Ontology.

5.2.5.2. Service Execution Ontology

The Service Execution Ontology realizes the concepts for the control flow introduced above. Thus, it mainly contains appropriate classes for the concepts representing control entities. Individuals of these classes and their properties enable control of the service execution state. However, since all control entities are supposed to point to their originator, the Service Execution Ontology includes a special class with a corresponding property *hasOriginator* for this structural constraint. This class serves as a base class for all other classes representing control entities and is referred to as the *ControlEntity*.

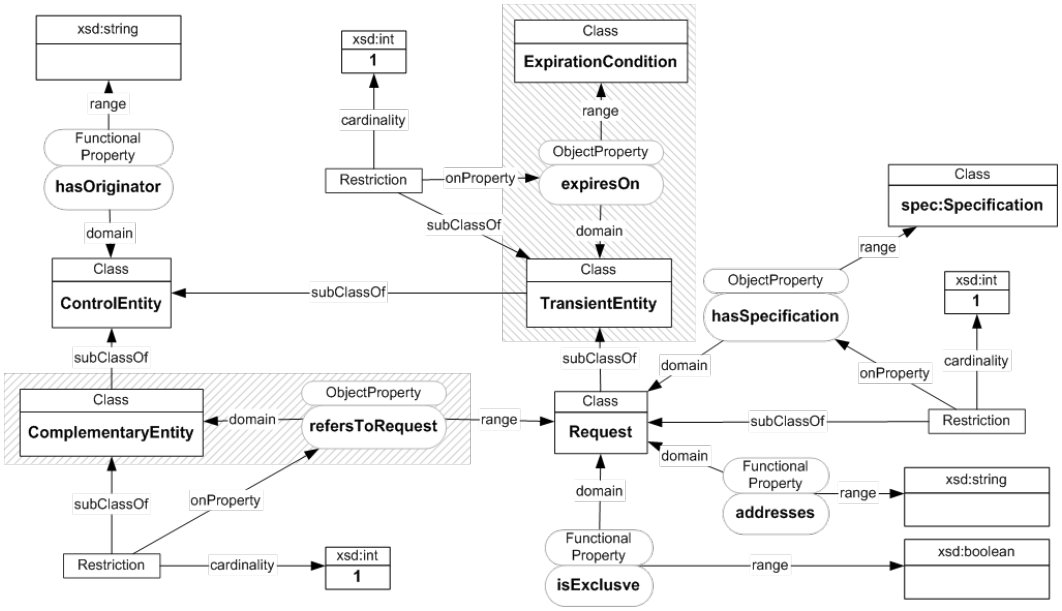

Figure 47: Base classes for control entities defined in the Service Execution Ontology

Furthermore, the classes *TransientEntity* and *ComplementaryEntity* are introduced to represent special characteristics of control entities. They are highlighted in Figure 47. *TransientEntity* is a classifier for control entities that should expire on the occurrence of a particular event, and thus serves as domain for the property *expiresOn*. The Service Execution Ontology also defines a class for a default expiration condition which should be fulfilled when a particular point in time is exceeded. This class is called *TimeExceeded.* Individuals of this class are required to reference a time-stamp, as illustrated in Figure 48. While exchanging individuals of *TransientEntity* the time-stamps referenced by the expiration condition need to be compared to the current time of the environment. An individual of *TransientEntity* is considered as expired if the environment time exceeds the time-stamp referenced by its expiration conditions.

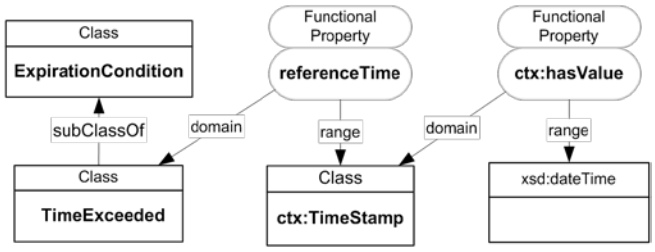

Figure 48: Constructs related to the expiration condition

ComplementaryEntity is a classifier for all entities that refer to a particular request. The class serves as domain for the property *refersToRequest* originally defined as a refers-to relation for the concepts Service Acknowledgement, Service Confirmation,

Failure, Invalidation, and Response. Consequently, the corresponding classes are sub-classes of *ComplementaryEntity* as illustrated in Figure 49. Since Service Acknowledgement and Service Confirmation are also defined to expire after a certain time-span, the corresponding classes are also sub-classes of *TransientEntity*.

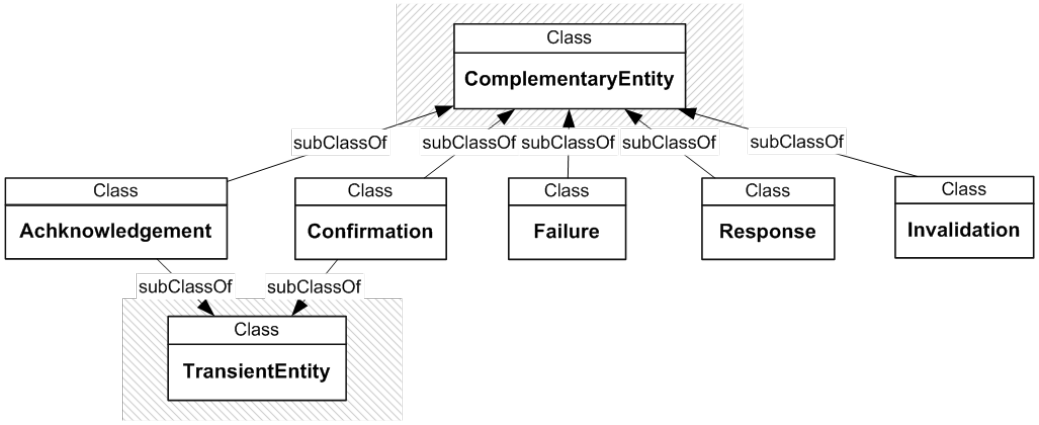

Figure 49: Inheritance hierarchy of control entities in the Service Execution Ontology

5.2.5.3. Data Space Configuration for the Service Model Ontology

Service provisioning based on the Semantically Enhanced Data Space and the Service Model Ontology is realized through the exchange of ontologies. For each new set of individuals representing a control entity and assigned data, Service Requester and Service Provider are required to create a new Data Plane and add this data set. The Data Space in return, enables temporal decoupling and verifies the validity of each new data set as it is added. Hence, the Data Space always provides a valid and consistent view of all contained data to the interacting parties. However, to check the correctness of data to be added, the SEDS obviously needs to partially understand their meaning. In terms of the Service Specification Ontology and the Service Execution Ontology this understanding addresses on the one hand the structural constraints enforced by the OWL DL representation and on the other the sequence and conditions for the origination of individuals for particular control entities. Both aspects are realized with a special data space configuration as introduced in Section 5.2.3.2.

The verification of structural consistency and correctness for newly added data in terms of OWL DL is realized with the Description Logic reasoner Pellet [54]. Pellet is based on the tableaux algorithms developed for expressive Description Logic ontologies. In fact, Pellet is also the only freely available reasoner that supports the complete expressiveness of OWL DL. Kaon2 [55], Bossam [56], and the Jena internal DL reasoner have proven insufficient or non-performing in this work.

Observation of constraints addressing the sequence of control entities is realized with a special Model Injector for the current time and two implementations of Da-

taSpaceHook. The first Hook ensures that time-stamps related to individuals of *TimeExceeded* are not outdated when added to the Data Space. The same Hook also searches for individuals of the class *Acknowledgement* in new data sets. If such an individual is found its regularity is checked against the data already contained in the Data Space. Thus, the Hook proves whether or not there is already a valid individual of *Acknowledgement* for the same service request. Indeed, this check is only performed for exclusive service requests and irregular service acknowledgements, i.e. the corresponding individuals and assigned data, are rejected. The second Hook only scans newly added data sets for individuals of *TimeExceeded*. Since these expiration conditions may influence the execution state of a service, updates of the data view are scheduled. For the appropriate time-stamps An update of the Data View of a Data Space causes all subscribed queries of the corresponding Pattern Registrar to be evaluated. In this way the expiration of the last valid service acknowledgement for an exclusive service request can be notified to potentially interested Service Providers.

5.3. Service Adaptation Layer

The aspects mentioned above together with some demo scenarios were implemented and tested in order to prove the functionality and usefulness of the approach. Section 3.4 gives the technical means employed in the development process and depicts the general interaction of the implemented components, constituting an overview and serving as a guideline for the other sections. The implementation of the components related to the context is depicted in Section 5.3.1. Section 5.3.2 deals with input of the framework gathered from the user through the sensor node implementation of the pREST access layer. The implemented functionality is detailed in Section 5.3.3 in terms of inference and classification. Section 5.3.4 covers aspects related to the description and invocation of services. Implementation of components related to the processing of user feedback is covered by Section 5.3.5.

The framework is implemented using Java programming language in the *Java 2 Platform, Standard Edition (J2SE) 5.0*. The ontology is represented in the *Web Ontology Language (OWL)* and processed with the help of the *Jena Semantic Web Framework 2.2* [103]. The *Protégé Ontology Editor* in the versions 2.1 and 3.1 are used for editing the ontology. The *Suggested Upper Merged Ontology (SUMO)* is used as the upper ontology of the system ontology and *NanoHTTP*, a small HTTP server in Java, is utilised for serving the ontology. Thus it can be classified by the *Renamed Abox and Concept Expression Reasoner (RACER)* used in version 1.7. As used in the demo scenarios, the graphical user interface is implemented utilising *HTML* as well as the *Java Server Pages (JSP)* that come with the *Java 2 Platform, Enterprise Edition 1.4*. The web application is deployed in the *Jakarta Tomcat Web Server* version 5.5.

An external application can use the functionality of the framework with the help of the *Context Hotspot* object serving as an entry point. The *Context Hotspot* manages the whole framework process and returns a service that can be executed. It provides methods to derive a service call based on the current context as well as for obtaining a *Feedback Manager* object which can be used by the client to provide user feedback.

First of all, the *Context Hotspot* retrieves the user and the respective profiles based on the provided request information. The next step is to call the *Query Manager* with the provided query. Then the *Context Broker* is asked for the current context model providing the user and profiles. The resulting model is then published to a user specific URL so that the external reasoner is able to access it. The *Inference Module* is now consulted in order to derive parameters that are then passed to the *Service Manager*. As service composition features are not implemented, the *Service Manager* only returns one *Service* object with respective parameters. The *Service Manager*, the *Service*, and the information of the request are then utilised to construct a *Feedback Manager* object which can be used to retrieve the available *Feedback Options* as well as the service results.

5.3.1. Context Broker and Context Snippets

The *Context Broker* is responsible for constructing the actual context ontology given the current user and a list of profiles. The ontology layers forming the resulting system ontology are gathered from different sources and do not have to reside on the local machine. Since some information changes from situation to situation it has to be altered dynamically as depicted above. *Context Snippets* are statements stemming from sensors and other devices that contain a piece of information vital for the context such as the current temperature. In order to alter such information in a dynamic way, *Context Snippets* can be added to and removed from the context model making it possible to include transient information if need be.

Figure 50 shows the interfaces that are related to the *Context Broker*. ContextBroker represents the provider for the context model. It has methods to add and remove ContextSnippet objects each wrapping a *Context Snippet*, and to get the current model and the ConceptMapper. ContextSnippet offers a method to add statements to the context model which is invoked by the respective ContextBroker if the model is constructed. To achieve this, ContextSnippet objects communicate with the respective data sources such as real sensors via a defined interface, i.e. the pREST access layer. The method getConceptMapper of the ContextBroker object is vital for the *Query Manager* and the *Service Manager* to translate strings into concepts and vice versa. ConceptMapper has a description and provides the method parseQuery that takes a string and returns an instance of ParsedMapping. This ParsedMapping can be used to get the concept representations and the unmatched inputs, i.e. the terms that

could not be mapped to a concept. The method getMappingTerm of ConceptMapper can be used to get a string given a reference to a concept, e.g. an URI.

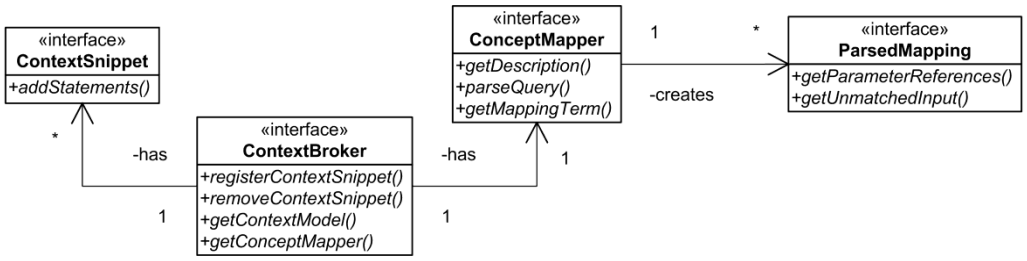

Figure 50: Interfaces belonging to the context package

The context model, i.e. the ontology, provided by the *Context Broker* is discussed in the remainder of this section. The following Section 5.3.1.1 depicts the data that make up the different ontology layers and mentions the links between them. Editing and the provision of ontology layers is detailed in Section 5.3.1.2 which also discusses how current context data is integrated into the ontology and how information residing within the ontology layers is processed by the framework.

5.3.1.1. Adaptation of the Ontology Layers

The system ontology, which consists of several layers, is used by the framework as the context model and has the same structure as depicted above. The following sections discuss the specific layers and mention the classes, instances, and properties specified in each case.

5.3.1.1.1. The Upper Ontology and its Limitations

We use the OWL representation of the Suggested Upper Merged Ontology as an upper ontology in line with the results of our evaluation. SUMO comprises of many basic and abstract concepts that can be referred to by the underlying layers. It is important to note that the current representation of SUMO utilises some OWL Full constructs, e.g. some classes are addressed as individuals. Hence, reasoners designed for OWL DL or OWL Lite either ignore these facts or cannot utilize them properly. For instance, RACER [71], which is used as an external reasoner, is able to load OWL Full ontologies even though as an OWL DL reasoner it is incapable of interpreting certain essential facts.

Many concepts that may be helpful when modelling context are not covered by SUMO and have to be specified in the *Specific Ontology* layer which imports SUMO and adds concrete concepts to the system ontology of importance for the current application. For example, the recommendation of activities depending on the current context assumes the definition of concepts such as *Bar*, *Friend*, or *Temperature*. However, when realising an intelligent jukebox, other concepts such as *Song* and

Artist are important too. The *Specific Ontology* may comprise of concepts suitable for a wide variety of different scenarios but the more concepts are added the more complex the ontology becomes which affects the duration of the inference process. In terms of the recommendation scenario, some examples of missing classes and instances as well as potential super classes of SUMO marked by the prefix "sumo" are:

- The classes *Friend*, *Colleague*, *Stranger*, and similar concepts defined as subclasses of the SUMO class *sumo:SocialRole*. These concepts are needed to express the social relationships of persons.
- Concepts referring to establishments such as *Bar* and *Opera* are inserted as subclasses of *sumo:Building*.
- Activities introduced extending the respective class, e.g. *Volleyball* is a subclass of *sumo:Sport*.
- Classes, instances, and properties representing the base for sensor information and other context data are added to the upper ontology. For example, *Temperature* is an instance of *sumo:CelsiusDegree* and has an additional *hasTemperature* property.
- Further freely definable concepts may be added as needed including *Evening* and *Morning* as subclasses of *sumo:TimeInterval.*

It is important to note that concepts defined in the *Specific Ontology* need to be employed carefully when it comes to expressing subjective and user-specific aspects. For example, the user Bob states in his profile that the user Eve is a friend of his using the concept *Friend* defined in the *Specific Ontology*. The user Alice does not know Eve who is therefore defined as an instance of class *Stranger* in her ontology. So if Bob, Alice, and Eve are involved in the same context, the user Eve is an instance of both, *Friend* and *Stranger*. Restrictions and rules referring to one of the classes also apply to Eve. Therefore, if Bob wants to write rules only matching his friends, he has to define a separate class in his profile ontology.

5.3.1.1.2. Framework Ontology

The *Framework Ontology* imports the *Specific Ontology* and comprises of the concepts needed by the framework to dynamically process input, infer facts, and call services. The facts contained in this ontology are directly interpreted and used to store values or parameters. The particular concepts given definition in the *Framework Ontology* are:

- The class *Situation* defined to cope with situations and its restrictions. Each situation has parameters. These parameters are dynamically added to the respective situation depending on the current context.

The class *Restriction* and *ParticipantRestriction* as a subclass added to the ontology. *ParticipantRestriction* has the property *hasParticipant* to cope with the Open World Assumption (OWA).

The instance *ssf_situation* of type *Situation* and *ParticipantRestriction* is added. This is necessary since the current situation shall be classified in terms of the context and class restrictions. Rules can then refer to the current type of the *ssf_situation* and set service parameter values accordingly.

The class *Input* is defined in the *Framework Ontology.* It has parameters representing the user input. This definition is necessary to specify rules that set parameter values depending on the user input. An individual of the classes *Input* and *Situation* may also have the property *hasNoneConceptParameter.* This property - which is discussed in detail in Section 5.3.2 - can be used to consider user input that cannot be matched to concepts by the respective *Concept Mapper*, but is also relevant when constructing the service call.

The definition of the class *Rule* and the property *hasRepresentation* enables the embedding of rules in an alternative syntax such as Jena rules in profiles.

The property *hasPriority* of a parameter and the class *Priority* with the individuals *Ignore*, *Low*, *Normal*, and *High* make it possible to link priorities with parameters.

All users important to the framework are defined as being instances of *Man* or *Woman*, e.g. *Alice*, *Bob*, *Dave*, and *Eve*. In this manner, all underlying layers as well as different user profiles refer to the same respective entity.

Some values of properties defined in the *Framework Ontology* are set in the *Transient Ontology* as depicted in Section 5.3.1.1.4.

5.3.1.1.3. Profile Ontology

Domain profiles, mood profiles, and user profiles add preferences to the system ontology. The profiles are stored in files residing on the local machine or can be accessed remotely by providing a URL. The domain profile is application-specific and chosen in line with the current context, e.g. the intelligent juke box scenario requires a domain profile comprising of music-related preferences and settings. The user specifies the mood profile to use when querying the system by providing an identifier. A *Profile Manager* tries to match this identifier to a profile. In the same manner the user profile of the requesting user is also identified. Unlike the mood profile there may be multiple user profiles added to the system ontology since several users may be important in terms of context. However, the profile types are only distinguishable as far as their purpose and contained facts are concerned. Once they are added to the *Profile Ontology* layer it is not possible to determine which statement derived from which profile.

5.3.1.1.4. Transient Ontology

The *Transient Ontology* sets properties defined in the other ontologies depending on the current values of sensors and other resources. It is constructed with the help of the Jena API and imports the user profiles. The ontology is then passed to an ontology server, the NanoHTTP. With the help of an OWL writer the server publishes the ontology in OWL to a specified URL. The URL is passed to RACER which loads the ontology and is able to classify it. Since the ontology imports all the other layers directly or indirectly, RACER is aware of the whole system ontology.

During the construction process the registered *Context Snippets* are added to the ontology model using the API. For example, the concrete restriction on the property *hasParticipant* is set in line with the participants in the context. The number of participants in the context is provided by a *Context Snippet* or alternatively by a component that is aware of the locations of all users. *Context Snippets* representing sensor information such as temperature and location are also added to the ontology.

5.3.1.2. Editing and Processing the Ontology Layers

The OWL version of the upper ontology SUMO was not altered at all and was used as provided on the web site. With the exception of *Transient Ontology,* the other ontology layers , were edited with the help of Protégé and a simple text editor. Since all ontologies may be distributed, the small NanoHTTP server only consisting of one Java class was chosen to publish them on specified URLs. The Tomcat server could also publish them but requires many more resources and is therefore only used for the scenario web application.

Whenever the *Context Broker* is asked for the current context ontology with the requesting user and an optional mood profile as input, an internal ontology representation is constructed with the help of the Jena Semantic Framework. In detail, the *Context Broker* executes the following sequence:

1. A new empty ontology model is constructed which can be used to add ontology facts.
2. If a mood profile ID was provided by the user, a *Profile Manager* is consulted to retrieve the ontology model which is added to the model created in step one.
3. All registered *Context Snippets* stating current sensor values or other data are added to the ontology. A configuration file is used to register all available snippets to the framework and to set initial values. It is important to note that the *Query Manager* also provides a snippet with the concepts of the current user query which is added in this step.
4. The current model is used to figure out the participants of the current context as reported by the respective sensors or a combination of these and a

database. The *Profile Manager* is used to determine the profiles of the users which are also added to the ontology model.

5. The available domain profiles are added to the model.
6. The other ontology layers, namely the *Framework Ontology*, the *Specific Ontology*, and the *Upper Ontology*, are imported in the current model.
7. The model is returned.

5.3.2. Mapping Query Concepts

The *Query Manager* uses a *Concept Mapper* to create a *Context Snippet* comprising of the *Input Parameters.* A *Concept Mapper* is responsible for mapping input terms stemming from the user interface to concepts included by the current ontology base layers, i.e. either the *Upper Ontology* or the *Specific Ontology* layer. Therefore, whenever the respective ontologies are adapted, the appropriate *Concept Mapper* should be updated otherwise the provided *Input Parameter* would be a concept not included in the ontology but still referable by rules unless the ontology is checked for consistency.

Figure 51 shows the interfaces QueryManager and QuerySnippet which are important for processing the user input. QueryManager provides methods to register and remove a new ContextBroker. The method updateQuery currently has one parameter: the user input string. If other kinds of user input need to be supported, this method has to be changed accordingly. If executed, a suitable ContextBroker is chosen and the respective ConceptMapper is used to obtain a ParsedMapping as depicted in Figure 50. With the help of the ParsedMapping object, the QuerySnippet is created that extends ContextSnippet and is added to the model of the respective *Context Broker*.

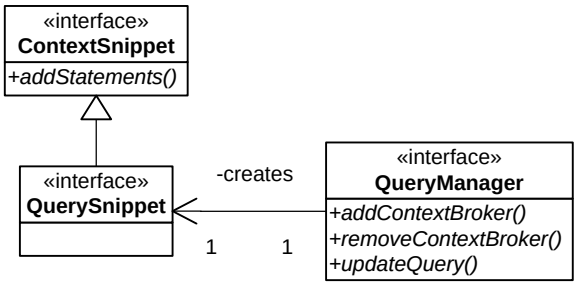

Figure 51: QueryManager and QuerySnippet

The following box illustrates a simple *Context Snippet* generated with the help of a "Text-to-SUMO" *Concept Mapper*, which simply parses the string input and maps it to as many concepts defined in a text file as possible, using the input query "recreation".

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

```
      xmlns:j.0="http://ontServer/context/ssf_ontology.owl#">
    <rdf:Description
        rdf:about=
          "http://ontServer/context/ssf_ontology.owl#ssf_input">
      <j.0:hasParameter
        rdf:resource="http://ontServer/context/
           upper_ontology.owl#RecreationOrExercise"/>
    </rdf:Description>
  </rdf:RDF>
```

It is also possible for the user to provide terms that are not contained in the ontology. To include input which cannot be mapped by the *Concept Mapper* to ontology concepts the property *hasNoneConceptParameter* is defined in the *Framework Ontology*. An individual of type *Input* or *Situation* can provide such a property value. However, the *Query Manager* only sets the property value of the respective individual of type *Input*. To apply this parameter to the situation as well, a rule can be defined similar to the one depicted in the following box where "ns" determines the namespace of the concepts.

```
[noneConcept: (ns:casaf_input ns:hasNoneConceptParameter ?x) ->
(ns:ssf_situation ns:hasNoneConceptParameter ?x)]
```

It should be noted that the rule may also contain additional conditions in the body, making it possible to restrict the number of cases in which it is possible to provide concepts not included in the ontology. For example, if a bar is recommended it may be desirable to use features provided by a search service, such as excluding a certain type of bar.

5.3.3. Inference and Classification

The *Inference Module* is responsible for deriving parameters depending on the current context model. The resulting list of *Derived Parameters* is then passed to the *Service Manager*. The *Inference Module* executes the following tasks in each inference process:

1. The Jena API is used to load the model provided by the *Context Broker* and the Jena rules which are either embedded in the context model, e.g. the profiles, or which reside in a separate file are extracted.
2. The rules are parsed and each fact in the rule body that refers to the class hierarchy of the model or the types of individuals causes a request sent to the reasoner, i.e. RACER. If the fact in question is true, it is added to the model.
3. The model containing the facts added in step 2 is now used to derive new facts using the extracted rules. The Jena Framework automatically applies the rules and infers a new model accordingly.
4. A table with priorities based on the inferred model is created, i.e. all facts that refer to the priority of concepts are put in a table.

5. All statements that are set by the rules and that refer to potential parameters for current situation are listed. For each parameter the following procedure is triggered:
6. If the parameter refers to a class, the reasoner is asked for all super classes. If the parameter refers to an individual, the reasoner is asked for the class of the individual and all super classes. The resulting information is linked to the respective parameter.
7. If one of the classes retrieved or the parameter itself has a priority value linked with it in the priority table, the respective value is set. If the priority value is set to *Ignore*, the parameter is discarded.
8. A *Derived Parameter* is created setting the reference to the respective ontology concept, the priority value, and all super classes which facilitates subsequent processing by the *Service Manager*.
9. The connection to the reasoner is closed and the parameter list is returned.

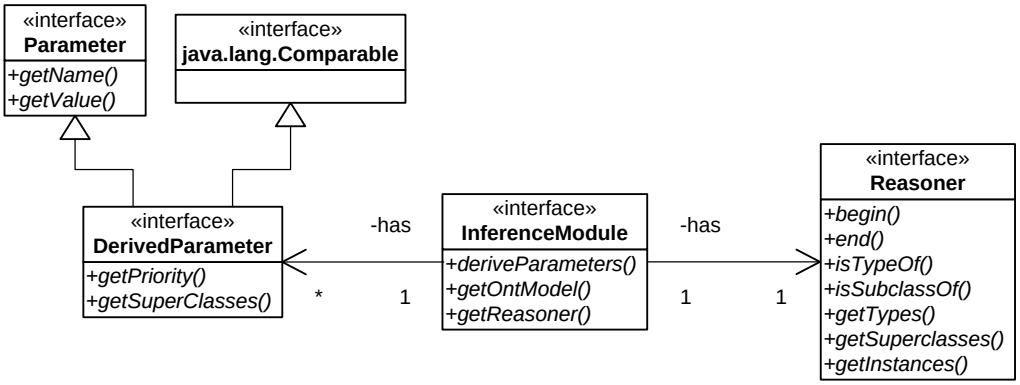

Figure 52: Important inference interfaces

Figure 52 depicts the main interfaces of the inference package. The InferenceModule provides the method deriveParameters that returns a list of DerivedParameters based on the current context model. For this purpose it utilises a reasoner. Since the Jena API is used for the processing of rules, no additional interface is needed as far as the rule reasoner is concerned. Rules within profiles are identified with the help of the ontology class *Rule* as defined in the *Framework Ontology*. The extracted rules are passed to the rule reasoner after the classification reasoner is applied. The Reasoner interface is used to communicate with the classification reasoner, the RACER. The interface provides methods to start and shut down the reasoner. Given two parameters referring to ontology concepts, the methods isTypeOf and isSubclassOf return a Boolean value. The methods getTypes, getSuperclasses, and getInstances return a list of URIs of concepts with respect to the concept provided. Since DerivedParameter extends Parameter and java.lang.Comparable, parameters of this kind can be sorted

according to priority. DerivedParameter also has methods to determine the parameter's priority and its super classes.

5.3.4. Executing Services

The *Service Manager* is the central component responsible for service administration and the inference of service calls. *Service Templates* announced to the *Service Manager* are used to decide which suitable service to invoke. The structure of a *Service Template* is described in Section 5.3.4.1. Section 5.3.4.2 depicts the process of deriving *Service Parameters* and choosing suitable services based on available *Service Templates*. Finally, an implementation overview is given in Section 5.3.4.3.

5.3.4.1. Service Templates

A service is made available to the framework with the help of one or more *Service Templates*. The current implementation utilises a fixed number of XML tags to describe the *Service Templates* which are parsed by the *Service Manager*. It is also possible to group multiple templates in one file. The supported tags and its attributes are listed in Table 8.

Name	Description	
Services	May encapsulate multiple *Service Templates* and is the root tag of the XML file.	
Service	This tag starts a new *Service Template* description and has three attributes. A 'Service' tag may enclose multiple 'InputParameter' and 'OutputParameter' tags.	
	Attribute name	Description
	name	The unique name of the *Service Template*.
	describes	The reference to the actual service that is described. The value points to a class or file used to execute the service.
	priority	The optional priority of the *Service Template*. The default priority is 'normal'.
InputParameter	An 'InputParameter' tag refers to a parameter of a service such as the string of a search service. It can comprise of multiple 'Component' tags and has two attributes.	
	Attribute name	Description
	describes	The name of the parameter that should correspond to the name used in the class or file that is utilised to execute the service.
	adapter	The optional adapter class that is used to compose the input parameter before executing the service. The default adapter class simply strings all components together separated by a blank.
OutputParameter	Describes an output parameter of a service and may have the same features as an input parameter. However, in the majority of cases the	

	components are determined dynamically at runtime and cannot be described a priori in a static template. The detailed description of output parameters is only needed if service composition is desired.	
Component	Describes a component of an input or output parameter. The value of a component is the URI of a semantic concept contained by the system ontology. It has two attributes.	
	Attribute name	Description
	minCard	The minimum number of terms matching the component concept. Allowed values are numbers >= 0.
	maxCard	The maximum number of terms matching the component concept. Allowed values are number >= 0 and * (unlimited).

Table 8: Service Template tags

In terms of the current implementation, the attribute "describes" of the "Service" tag points to a Java class that extends AbstractService. This abstract class implements the Service interface and is used to set and retrieve service parameters. The method to execute the service is implemented by the extending class, i.e. the respective Web Service is called and the results are returned. However, if the framework is not able to find a proper Java class by reflection, other possibilities may be tried such as automatically executing the service based on its OWL-S description.

Another attribute that requires further explanation is the attribute adapter of the tag InputParameter. This attribute points to a Java class that implements the interface ParameterAdapter and is executed with the help of Java reflection. This interface provides a method responsible for composing the final input parameter with the help of the given components. The simplest way of doing this is to string the components together and separate them by a blank. However, other possibilities are also feasible. For example, in some cases it may not be desirable to allow input that cannot be mapped to an ontology concept by the *Concept Mapper* as discussed in Section 5.3.2.

5.3.4.2. Derive Service Parameters

The *Service Manager* is responsible for finding a matching *Service Template* and respective parameters based on the context model, the list of *Derived Parameters* stemming from the *Inference Module*, and the *Concept Mapper* of the respective *Context Broker*. The following sequence is triggered to achieve this:

1. If available, the unbound string is extracted from the model. This string does not match any concept included in the model and may be set by a rule as discussed in Section 5.3.2.
2. The *Derived Parameters* are sorted in the descending order of their priorities.

3. Each *Service Template* currently available is checked to see whether or not it can be fulfilled.
4. The *Service Manager* iterates through the components of the template and checks if there is a matching parameter, considering both the concept itself and the super classes of the *Derived Parameter*set by the *Inference Module.* The rules that are applied for this process are:
5. Parameters with a high priority are checked before those with a lower priority. That is why the parameters were sorted in step 2.
6. If the minimum number for a component is larger than zero, but has no matching *Derived Parameter*, the whole template is dropped.
7. If there is more than one parameter possible for a component, as many *Derived Parameters* as possible are utilised.
8. If the maximum number of parameters of a component is reached, no further *Derived Parameter* is checked.
9. A *Service Template* is fulfilled if all components are satisfied and no restrictions are infringed.
10. The *Service Manager* compares a fulfilled template to the best current one since only the best template is chosen at present. It applies the following rules:
11. A template with components of higher than average priority is preferred.
12. If the average priority of the components is equal to the best average, the template that has a higher priority is chosen.
13. If the template priority and average component priority are equal, the template that fulfils more components is preferred.
14. If a template is equal to the current best template in terms of the rules mentioned above the former best template is preferred.
15. If the *Service Manager* has found the best matching template it creates the respective service call with the *Service Parameters.*
16. The *Concept Mapper* is used to find a proper string representation for a matching concept. If no mapping is found, the namespace of the concept is simply neglected giving the concept's local name.
17. The group of components that belong to an 'InputParameter' are composed with the help of the adapter class. The unbound string may be also considered by the adapter.
18. Finally, the *Service Manager* creates the service call using the created inputs and pointing to the respective Java class or alternative description.

The following box comprises of two exemplary *Service Templates* referring to one service class, the YahooWebService class which uses the Yahoo! Web Services [115] implementation. Both *Service Templates* describe the same InputParameter that consists of three components, whereby "ns" denotes the namespace of the respective ontology concept.

The first template requires at least one parameter of the type *ns:GeographicArea* and an optional parameter belonging to the classes *ns:IntentionalProcess* and *ns:TimeMeasure.* The concept *ns:IntentionalProcess* defines a kind of process that is performed, e.g. by a human, with a certain purpose. The second template is similar to the first but requires a parameter of the type *ns:Building* instead of *ns:IntentionalProcess*. The OutputParameter of the service is not described precisely in either template.

```
<?xml version="1.0"?>
<Services>

    <Service name="YahooWebService1"
describes=
  "de.fhg.fokus.casaf.extern.service.web.YahooWebService"
priority="normal">
        <InputParameter describes="Query" adapter=
      "de.fhg.fokus.casaf.service.adapter.StringAdapter">
         <Component minCard="1" maxCard="*">
         ns:GeographicArea
      </Component>
            <Component minCard="0" maxCard="1">
         ns:IntentionalProcess
      </Component>
            <Component minCard="0" maxCard="1">
         ns:TimeMeasure
      </Component>
        </InputParameter>
        <OutputParameter describes="Result">
        </OutputParameter>
</Service>
<Service name="YahooWebService2"
describes=
  "de.fhg.fokus.casaf.extern.service.web.YahooWebService"
priority="normal">
  <InputParameter describes="Query" adapter=
      "de.fhg.fokus.casaf.service.adapter.StringAdapter">
            <Component minCard="1" maxCard="*">
         ns:GeographicArea
      </Component>
         <Component minCard="0" maxCard="1">
         ns:Building
      </Component>
            <Component minCard="0" maxCard="1">
         ns:TimeMeasure
      </Component>
        </InputParameter>
  <OutputParameter describes="Result">
        </OutputParameter>
    </Service>
</Services>
```

In terms of the exemplary *Service Templates*, Table 9 shows some examples of *Derived Parameters*, the selected *Service Parameters*, and the *Service Template* chosen by the *Service Manager* according to the selection rules depicted above. Here, *Berlin*, *Bar*, *Opera*, *Volleyball*, and *Germany* denote the respective concepts. *Berlin* and *Ger-*

many belong to the class *ns:GeographicArea*, *Volleyball* to the class *ns:IntentionalProcess*, and *Bar* and *Opera* to the class *ns:Building*.

Derived Parameters	Service Template	Service Parameters
Berlin, Bar	YahooWebService2	Berlin, Bar
Berlin, Volleyball	YahooWebService1	Berlin, Volleyball
Berlin	YahooWebService1	Berlin
Berlin, Bar, Opera	YahooWebService2	Berlin, (Bar or Opera)
Berlin, Bar, Volleyball	YahooWebService1	Berlin, Volleyball
Volleyball	none	-
Berlin, Germany, Bar	YahooWebService2	Berlin, Germany, Bar

Table 9: Derived Parameters and corresponding Service Templates and Parameters

5.3.4.3. Implementation Overview

Figure 53 depicts the static structure of the interfaces connected to the service selection and invocation process.

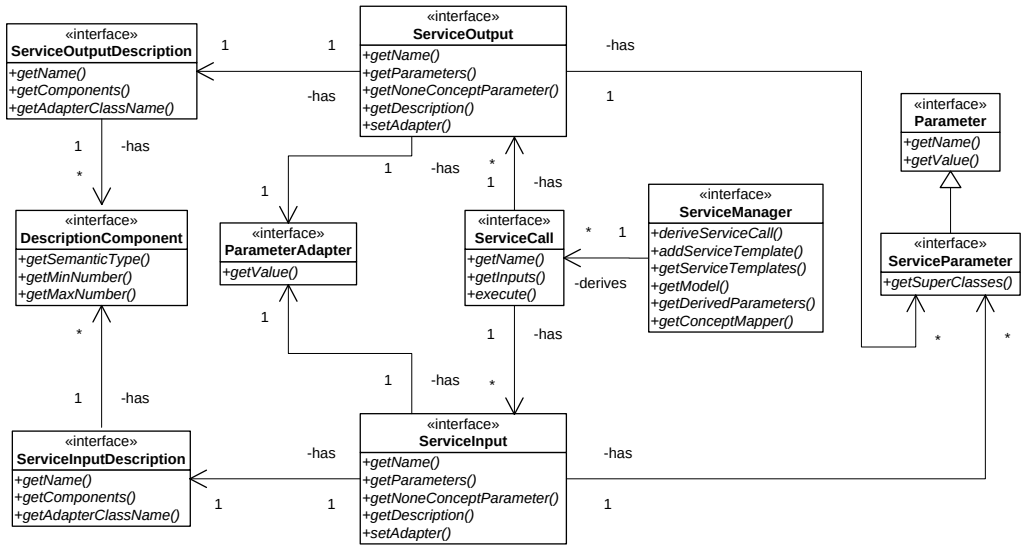

Figure 53: The service interfaces

The ServiceManager is the entry point for all tasks related to the selection and invocation of services. A new instance can be created using a constructor that includes the context model, a list of *Derived Parameters*, the current *Concept Mapper*, and a list of available *Service Templates* as parameters. The deriveServiceCall method can then be used to determine a ServiceCall representing the service to invoke. *Service Templates* can be added or the current settings can be found out. A derived ServiceCall has a name and ServiceInputs set by the ServiceManager instance. If the respective service is executed using the execute method, ServiceOutputs are re-

turned. ServiceInputs and ServiceOutputs are described by the ServiceInputDescriptions and ServiceOutputDescriptions. Respective instances of the description classes are created during the process of parsing the available *Service Templates.* For each semantic component of an input or an output parameter, a DescriptionComponent is created. A DescriptionComponent has a semantic type, a maximum, and a minimum. The ServiceOutputDescription and the ServiceInputDescription can be used to retrieve the actual component descriptions as well as the name of the adapter class and the name of the parameter. A ServiceInput has a name, a short description, and a ParameterAdapter as described by the respective parameter description. It may also have a value that could not be classified by the *Concept Mapper* and which can be received with the help of the getNoneConceptParameter method. The getParameters method returns the current ServiceParameters as set by the *Service Manager*. The method getValue inherited from ParameterAdapter, returns an object that serves as the actual service input such as a string received by concatenating the parameter values. ServiceParameter extends Parameter and therefore has a name and a value referring to the semantic concept. Furthermore, the super classes of a parameter can be determined which enables the *Service Manager* to find matching templates given the *Derived Parameters*. The methods getParameters, setAdapter and getNoneConceptParameter of ServiceOutput are only important if service composition features are required and is therefore not implemented in the current context.

5.3.5. Considering User Feedback

As illustrated above the inference process of the *Context Hotspot* returns a *Feedback Manager* that can be used by the user interface to obtain service invocation results and available *Feedback Options.* The abstract classes FeedbackManager and FeedbackObject represent the conceptual components and are depicted in Figure 54.

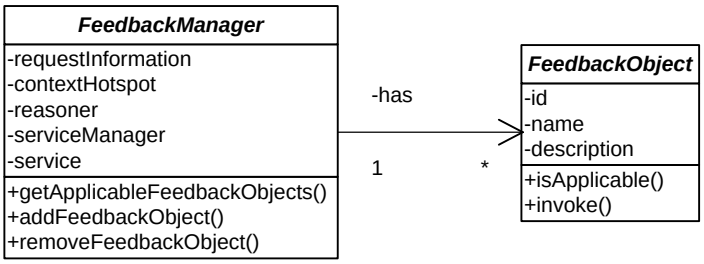

Figure 54: Abstract classes Feedback Manager and Feedback Object

The FeedbackManager class has several attributes that can be retrieved using the respective getter methods which are the current ContextHotspot, the reasoner used, the ServiceManager, the Service, and request information comprising the user ID, the user query, and the ID of an activated mood profile. FeedbackObjects can be added

and removed from the FeedbackManager. A FeedbackObject has an ID, a name, and a description. An instance of this class has to implement the abstract methods isApplicable and invoke. The method isApplicable returns true if the creation of a new *Feedback Option* is meaningful given the current settings of the FeedbackManager. The invoke method is executed if the user chooses the respective *Feedback Option* and returns a new FeedbackManager object. The method getApplicableFeedbackObjects of FeedbackManager returns a list of FeedbackObjects whose method isApplicable returned true.

Implemented FeedbackObjects affect the user query in the desired manner, substitute or drop *Derived Parameters*, or neglect certain service results. For example, Figure 55 shows the relations of spatial concepts used by a *Feedback Object* that expands the current location. The property *partlyLocated* connects different concepts and can be used to substitute one concept instance with another.

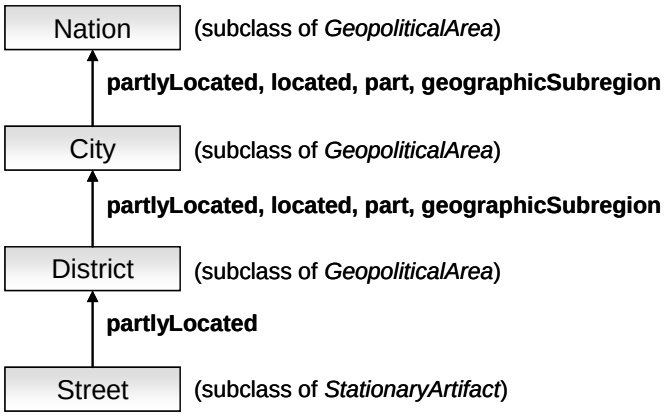

Figure 55: Scenario ontology structure of important location concepts

As the purpose of the *Feedback Options* is to provide a simple means for the user to influence framework results, they need to be sufficiently significant. For example, the ontology structure shown in Figure 55 allows for more than one *Feedback Option*, e.g. one for substituting a district with the city and a city with the nation. However, this is not desired since the user risks losing track of the available options. It is preferable to use only one option for expanding the location. Table 10 shows the possible outcomes of such a spatial *Feedback Option*.

Available Service Parameter types	Possible impact of the Feedback Option
Street and Nation	Replaces the street with the respective district.
Street, District, City, and Nation	Neglects the street.
District and City	Neglects the district.
City	Replaces the city with the respective nation.

Table 10: Possible impacts of a Feedback Option for locations

For instance, if all four location concepts *Street*, *District*, *City*, and *Nation* are already contained by the current *Service Parameters*, it is reasonable to neglect the parameter of the type *Street*. On the other hand, if only the city is provided it should be replaced with the respective nation.

6. Summary

This chapter summarizes the thesis, and reiterates the major steps and the novel contributions to the research field of Ambient Intelligent Systems. The outlook describes the next steps to be taken and hurdles to be tackled for on-going enhancement of the communication life of the user.

6.1. Conclusion

This thesis has presented a novel layer model for Ambient Intelligent Systems based on the analysis of various single- and multi-user scenarios situated in heterogeneous communication environments. The new layer model derived from this requirement analysis then led to a new paradigm of service request orientation and the specification of the corresponding servicerequest-oriented architecture. Service requests decouple the interpretation and processing of user needs from service calls by being formal enough to allow grounding to specific service calls while still encapsulating semantic annotations describing the current user goal to allow for context-aware service adaptation. Loosely coupled service interaction in time, space, and representation is also utilized to accommodate varying user environments. Given the underlying generic information access layer – pREST – the proposed architecture is highly distributable from small sensor nodes to high-end computers, as well as being scalable, and non-disruptive which gives it the capability to include existing context-agnostic services such as traditional web search engines.

6.2. Outlook

Given the modular design of the Service Request Oriented Architecture presented in this thesis, further research can easily extend the proposed system. .The author considers the following issues to be paramount in future research in this field:

Support for service request routing. While the underlying information layer is already able to span several networks due to underlying pREST information access layer and its URI based addressing, service requests are currently distributed only through the semantically enhanced data space. Although the data space itself can be distributed over several nodes, all service requests within that data space belong to the same domain which is not always a desirable feature as when different users have parts of separated data spaces. If service requests need to cross several of these separate data spaces they must be routed which involves having the means for service request routing.

Dynamic and distributed service composition and evaluation. Due to the loosely coupled service interaction pattern employed in the Service Request Oriented Archi-

tecture, the semantic descriptions found in service requests cannot be used to enable on-the-fly composition of new services if no appropriate service consumer can be located in the semantically enhanced data space. The same holds true for abstract user requests that can guide dynamic service composition. Once services can be composed dynamically, the appearance of new devices with new capabilities can be used for dynamic performance evaluation and possible reconfiguration if other devices are able to better execute the service request.

Extended user evaluation of context-aware service adaptation. Although designed with the user in mind, the implementation of the Service Request Oriented Architecture has not yet been scientifically tested with a large user group. Large user group testing would enable fine-tuning of the context-awareness component as well as providing further insights into user needs and sparking the development of further adaptation components.

7. References

[01] M. Satyanarayanan, “Pervasive Computing: Vision and Challenges,” IEEE Personal Communications, vol. 8, no. 4, pp. 10-17, August 2001

[02] M. Weiser, “The computer for the 21st century,” Scientific American, vol. 265, no. 3, pp. 94-104, September 1991

[03] A. Chen, R.R. Muntz, and M. Srivastava, "Smart Rooms," in Smart Environments: Technology, Protocols and Applications. D. Cook and S. Das, Ed. Hoboken: Wiley-Interscience, November 2004

[04] A. K. Dey, "Providing Architectural Support for Building Context-Aware Applications," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, USA, 2000.

[05] D. Booth et al., "Web Services Architecture,“ April 2004. [Online]. Available: http://www.w3c.org/TR/ws-arch/

[06] A. Arsanjani, “Service-oriented modeling and architecture”, November 2004. [Online]. Available: http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/. [Accessed: Mar. 24, 2006]

[07] D. Sprott and L. Wilkes, "Understanding Service-Oriented Architecture," January 2004. [Online]. Available: http://msdn.microsoft.com/architecture/soa/default.aspx?pull=/library/en-us/dnmaj/html/aj1soa.asp. [Accessed: Mar. 24, 2006]

[08] E. Ort, "Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools," April 2005. [Online]. Available: http://java.sun.com/developer/technicalArticles/WebServices/soa2/. [Accessed: Mar. 24, 2006]

[09] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D.dissertation, University of California, Irvine, USA, 2000

[10] Sun Microsystems, “Jini Architecture Specification,” December 2001. [Online]. Available: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html. [Accessed: Mar. 24, 2006]

[11] OASIS Committee, "OASIS UDDI Specifications TC - Committee Specifications, UDDI v2, UDDI v3,“ March 2006. [Online]. Available: http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm. [Accessed: Mar. 24, 2006]

[12] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, World Wide Web Consortium (W3C) recommendation, January 2006. [Online]. Available: http://www.w3c.org/TR/wsdl20/. [Accessed: Mar. 24, 2006]

[13] S. Battle et al., "Semantic Web Services Ontology (SWSO)," April 2005. [Online]. Available: http://www.daml.org/services/swsf/1.0/swso/. [Accessed: Mar. 24, 2006]

[14] The OWL Services Coalition, "OWL-S: Semantic Markup for Web Services," November 2004. [Online]. Available: http://www.daml.org/services/owls/1.1/. [Accessed: Mar. 24, 2006]

[15] D. Roman et al., "The Web Service Modeling Ontology," IOS Press, vol.1, no.1, pp.77-106, 2005.

[16] M. Burstein et al., "A Semantic Web Services Architecture,“ IEEE Internet Computing, vol. 9, pp. 72-81, October 2005

[17] G. Cabri, L. Leonardi, and F. Zambonelli, "Mobile-Agent Coordination Models for Internet Applications,“ IEEE Computer, vol. 33, no. 2, pp. 82-89, February 2000

[18] Sun Microsystems, "Java Remote Method Invocation," 1997. [Online]. Available: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html. [Accessed: Mar. 26, 2006]

[19] Sun Microsystems, "Java Message Service", April 2002. [Online]. Available: http://java.sun.com/products/jms/docs.html. [Accessed: Mar. 26, 2006]

[20] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," ACM Computing Surveys, vol. 35, no. 2, pp. 114-131, June 2003

[21] D. Gelernter, "Generative Communication in Linda," ACM Transactions on Programming Languages and Systems, vol. 7, no. 1, pp. 80-112, January 1985

[22] G. Cabri, L. Leonardi, and F. Zambonelli, "Engineering Mobile-agent Applications via Context-dependent Coordination," IEEE Transactions on Software Engineering, vol. 28, no. 11, pp. 1034-1051, November 2002

[23] Sun Microsystems, "JavaSpaces(TM) Service Specification," March 2006. [Online]. Available: http://java.sun.com/products/jini/2.0/doc/specs/html/js-title.html. [Accessed: Mar. 24, 2006]

[24] B. Johanson and A. Fox, "The Event Heap: A Coordination Infrastructure for Interactive Workspaces," in Proceedings of the 4th IEEE Workshop on Mobile Computer Systems and Applications, 2002, p. 83

[25] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D.R. Reddy, "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," ACM Computing Surveys, vol. 12, no. 2, pp. 213-253, June 1980

[26] E. F. Codd, "A relational model of data for large shared data banks," Communications of the ACM, vol. 13, no. 6, pp. 377-387, June 1970.

[27] S. Abiteboul, P. Buneman, and D. Suciu, Data on the Web: From Relations to Semistructured Data and XML. San Francisco: Morgan-Kaufmann, 1999

[28] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Extensible Markup Language (XML) 1.0 (Third Edition), World Wide Web Consortium (W3C) recommendation, February 2004. [Online]. Available: http://www.w3c.org/TR/2004/Rec-xml-20040204/. [Accessed: Mar. 24, 2006]

[29] C. Fallside and P. Walmsley, XML Schema Part 0: Primer Second Edition, World Wide Web Consortium (W3C) recommendation, October 2004. [Online]. Available: http://www.w3c.org/TR/xmlschema-0/. [Accessed: Mar. 24, 2006]

[30] G. Klyne, J. J. Carroll, B. McBride, Resource Description Framework (RDF): Concepts and Abstract Syntax, World Wide Web Consortium (W3C) recommendation, February 2004. [Online]. Available: http://www.w3.org/TR/rdf-concepts/. [Accessed: Mar. 24, 2006]

[31] T. Berners-Lee, R. Fielding, and L. Masinter, Editors, Uniform Resource Identifier (URI): Generic Syntax, Network Working Group RFC 3986, January 2005. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3986.txt. [Accessed: Mar. 24, 2006]

[32] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," Scientific American, pp. 28-37, May 2001

[33] S. J. Russell and P. Norvig, Artificial Intelligence. A Modern Approach. 2nd Edition, Englewood Cliffs: Prentice Hall, 2003

[34] F. Baader, D. Calvanese, D.L. McGuiness, D. Nardi, and P.F. Patel-Schneider, The Description Logic Handbook: Theory, Implementation and Applications. Cambridge: Cambridge University Press, 2003

[35] "Closed World vs. Open World: the First Semantic Web Battle," June 2005. [Online]. Available: http://www.betaversion.org/~stefano/linotype/news/91/. [Accessed: Mar. 24, 2006]

[36] L. McGuinness and F. van Harmelen, OWL Web Ontology Language Overview, World Wide Web Consortium (W3C) recommendation, February 2004. [Online]. Available: http://www.w3.org/TR/owl-features/. [Accessed: Mar. 24, 2006]

[37] D. Khushraj, O. Lassila, and T. Finin, "sTuples: Semantic Tuple Spaces," in Proceedings of 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, August 2004, pp. 268-277.

[38] The DARPA Agent Markup Language Program (DAML), "Annotated DAML+OIL (March 2001) Ontology Markup," March 2006. [Online]. Available: http://www.daml.org/2001/03/daml+oil-walkthru.html. [Accessed: Mar. 24, 2006]

[39] C. Bussler, "A minimal Triple Space Computing Architecture," Digital Enterprise Research Institute, Ireland, DERI Technical Report 2005-04-22, 2005.

[40] P. Bontas, L. Nixon, and R. Tolksdorf, "A Conceptual Model for Semantic Web Spaces," AG Netzbasierte Informationssysteme, Freie Universität Berlin, Germany, Report B 05-14, 2005

[41] R. Akkiraju et al., Web Service Semantics - WSDL-S, World Wide Web Consortium (W3C) member submission, November 2005. [Online]. Available: http://www.w3c.org/Submission/WSDL-S/. [Accessed: Mar. 24, 2006]

[42] F. Hakimpour, J. Domingue, E. Motta, L. Cabral, and Y. Lei, "Integration of OWL-S into IRS-III: The Structured Approach," in Proceedings of the 1st AKT Workshop on Semantic Web Services, 2004, vol. 122

[43] A. Patil, S. Oundhakar, A. Sheth, and K. Verma, "METEOR-S Web Service Annotation Framework," in Proceedings of the 13th International Conference on World Wide Web, 2004, pp. 553-562

[44] Y. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics," in European Assoc. for Theor. Computer Science Bulletin, vol. 43, pp.264--284, February 1991

[45] C. Peltz, "Web Services Orchestration and Choreography," Computer, vol.36, no.10, pp. 46- 52, October 2003

[46] A. Arkin et al., Web Service Choreography Interface (WSCI) 1.0, World Wide Web Consortium (W3C) note, August 2002. [Online]. Available: http://www.w3.org/TR/wsci/. [Accessed: Mar. 24, 2006]

[47] T. Andrews et al., "Business Process Execution Language for Web Services version 1.1," May 2003. [Online]. Available: http://www-128.ibm.com/developerworks/library/specification/ws-bpel/. [Accessed: Mar. 24, 2006]

[48] I. Horrocks, P. F. Patel-Schneider, H.Boley, S. Tabet, B. Grosof, and M. Dean, SWRL: A Sematic Web Rule Language Combining OWL and Rule ML, World Wide Web Consortium (W3C) member submission, May 2004, [Online]. Available: http://www.w3c.org/Submission/2004/SUBM-SWRL-20040521/. [Accessed: Mar. 24, 2006]

[49] Mindswap - Maryland Information and Network Dynamics Lab Semantic Web Agents Project, "OWL-S API," March 2006. [Online]. Available: http://www.mindswap.org/2004/owl-s/api/. [Accessed: Mar. 24, 2006]

[50] Jena - Semantic Web Framework. Hewlett-Packard Development Company, 2005. [Online]. Available: http://sourceforge.net/project/showfiles.php?group_id=40417. [Accessed: Mar. 24, 2006]

[51] S. Bechhofer, Departement of Computer Science, University of Manchester, "OWL API," March 2006. [Online]. Available: http://owl.man.ac.uk/api.shtml. [Accessed: Mar. 24, 2006]

[52] Prud'hommeaux and A. Seaborne, SPARQL Query Language for RDF, World Wide Web Consortium (W3C) working draft, February 2006. [Online]. Available:

http://www.w3.org/TR/2006/WD-rdf-sparql-query-20060220/. [Accessed: Mar. 24, 2006]

[53] ARQ. Hewlett-Packard Development Company, 2006. [Online]. Available: http://sourceforge.net/project/showfiles.php?group_id=40417. [Accessed: Mar. 24, 2006]

[54] Pellet - OWL DL Reasoner. Mindswap - Maryland Information and Network Dynamics Lab Semantic Web Agents Project, 2005. [Online]. Available: http://www.mindswap.org/2003/pellet/download.shtml. [Accessed: Mar. 24, 2006]

[55] Kaon2. FZI - Forschungszentrum Informatik Karlsruhe, 2006. [Online]. Available: http://kaon2.semanticweb.org/. [Accessed: Mar. 25, 2006]

[56] Bossam Rule/OWL Reasoner, 2006. [Online]. Available: http://machine-knows.etri.re.kr/bossam/. [Accessed: Mar. 25, 2006]

[57] J. Barton, T. Kindberg, H. Dai, N. B. Priyantha, and F. Al-bin-ali, “Sensor-enhanced mobile web clients: an XForms approach”, Proceedings of the twelfth international conference on World Wide Web, Budapest, Hungary, ACM Press 2003, pp. 80--89

[58] A. Dey, G. Abowd, and D. Salber, "A Context-based Infrastructure for Smart Environments", Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE '99), pp. 114-128, 1999

[59] R. T. Fielding and R. N. Taylor, “Principled design of the modern Web architecture” ACM Transactions on Internet Technology vol. 2/2 pp. 115--150, May 2002

[60] T. D. Hodes and R. H. Katz, "Enabling 'Smart Spaces': Entity Description and User Interface Generation for a Heterogeneous Component-based Distributed System” University of California, Berkeley, CSD-98-1008, pp. 8ff, Jun. 1998

[61] W. Vogels , “Web Services are not Distributed Objects: Common Misconceptions about Service Oriented Architectures”, IEEE Internet Computing, Aug. 2003

[62] T. Kindberg, J. Barton, J. Morgan, G. Becker, I. Bedner, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, C. Pering, J. Schettino, B. Serra, and M. Spasojevic, “People, Places,Things: Web Presence for the Real World”, MONET, vol. 7/5, Oct. 2002

[63] Microsoft Corporation, “Understanding UPnP - a white paper”

[64] J. Nichols, B. Myers, T. Harris, R. Rosenfeld, S. Shriver, M. Higgins, and J. Hughes, "Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances", Proceedings of IEEE Fourth International Conference on Multimodal Interfaces Pittsburgh, PA, pp. 377-382, Oct. 2002

[65] M. Roman and R. H. Campbell, "A Middleware-Based Application Framework for Active Space Applications", 2003

[66] S. Steglich, R. N. Vaidya, O. Gimpeliovskaja, S. Arbanowski, R. Popescu-Zeletin, S. Sameshima, and K. Kawano, "I-Centric Services based on Super Distributed Objects", Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems, Apr. 2003, pp. 232

[67] D. Ueno, T. Nakajima, I. Satoh, and K. Soejima, “Web-Based Middleware for Home Entertainment”, Lecture Notes in Computer Science, vol. 2550 pp. 206 - 219, Springer-Verlag Heidelberg, Jan. 2002

[68] Super Distributed Objects DSIG, "Platform Independent Model and Platform Specific Model for Super Distributed Objects", 2003

[69] G. Booch, "Object Oriented Analysis and Design with Applications", Addison-Wesley Professional , 1994

[70] OWL Web Ontology Language Reference. 2004. Available at: http://www.w3.org/TR/owl-ref/. Accessed April 10, 2006.

[71] V. Haarslev and R. Möller. RACER system description. In Proceedings of the First International Joint Conference on Automated Reasoning, pp. 701-706, 2001.

[72] T. Berners-Lee. 1998. Semantic Web Roadmap. Available at: http://www.w3.org/DesignIssues/Semantic.html. Accessed April 10, 2006.

[73] H. Chen, F. Perich, T. Finin, and A. Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In International Conference on Mobile and Ubiquitous Systems: Networking and Services, Boston, MA, August 2004.

[74] A. Halevy. Why Your Data Won't Mix. Semi Structured Data, Volume 3 (8), October, 2005.

[75] Suggested Upper Merged Ontology (SUMO). Available at: http://www.ontologyportal.org/. Accessed April 10, 2006.

[76] Standard Upper Ontology Working Group – SUO-KIF. Available at: http://suo.ieee.org/SUO/KIF/index.html. Accessed April 10, 2006.

[77] I. Niles and A. Pease. Linking Lexicons and Ontologies: Mapping WordNet to the Suggested Upper Merged Ontology. In Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE '03), Las Vegas, Nevada, June 23-26, 2003.

[78] DOLCE: a Descriptive Ontology for Linguistic and Cognitive Engineering. Available at: http://dolce.semanticweb.org/. Accessed April 10, 2006.

[79] OpenCyc. Available at: http://www.opencyc.org/. Accessed April 10, 2006.

[80] WordNet – a lexical database for the English language. Available at: http://wordnet.princeton.edu/. Accessed April 10, 2006.

[81] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung. Ontology Based Context Modeling and Reasoning using OWL. In Proceedings of the 2004 Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS2004), San Diego, CA, USA, January 2004.

[82] OWL-S: Semantic Markup for Web Services. Available at: http://www.daml.org/services/owl-s/. Accessed April 10, 2006.

[83] F. Pan and J. R. Hobbs. Time in OWL-S. In Proceedings of the AAAI Spring Symposium on Semantic Web Services, Stanford University, CA, pp. 29-36, 2004.

[84] FOAF Vocabulary Specification. Available at: http://xmlns.com/foaf/0.1/. Accessed April 10, 2006.

[85] J.R. Hobbs, F. Pan. An Ontology of Time for the Semantic Web. ACM Transactions on Asian Language Processing (TALIP), Volume 3 (1), pp. 66-85, 2004.

[86] V. C. Storey, V. Sugumaran, and A. Burton-Jones. The Role of User Profiles in Context-Aware Query Processing for the Semantic Web. In Proceedings of the Ninth International Conference on Applications of Natural Language to Information Systems (NLDB), F. Meziane and E.Métais (eds.), Salford, UK, pp. 51-63, June 23-25, 2004.

[87] N. J. Belkin and W. B. Croft. Information Filtering and Information Retrieval: Two Sides of the Same Coin? Communications of the ACM, 35 (12): 29-38, December, 1992.

[88] RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax. Available at: http://www.faqs.org/rfcs/rfc2396.html. Accessed April 10, 2006.

[89] Resource Description Framework (RDF). Available at: http://www.w3.org/RDF/. Accessed April 10, 2006.

[90] Semantic Web Services Framework (SWSF) Specification. Available at: http://www.daml.org/services/swsf/. Accessed April 10, 2006.

[91] Web Service Modeling Ontology (WSMO). Available at: http://www.wsmo.org/. Accessed April 10, 2006.

[92] The Protégé Ontology Editor and Knowledge Acquisition System. Available at: http://protege.stanford.edu/. Accessed April 10, 2006.

[93] Protégé OWL Plugin. Available at: http://protege.stanford.edu/plugins/owl/. Accessed April 10, 2006.

[94] Semantic Web Development Environment (SWeDE). Available at: http://owl-eclipse.projects.semwebcentral.org/. Accessed April 10, 2006.

[95] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Available at: http://www.w3.org/Submission/SWRL/. Accessed April 10, 2006.

[96] D. Fensel and C. Bussler. The *Web Service Modeling Framework* WSMF. Electronic Commerce: Research and Applications, Volume 1 (2), pp. 113-137, 2002.

[97] M. Baldauf, S. Dustdar, and F. Rosenberg. A Survey on Context Aware Systems. In International Journal of Ad Hoc and Ubiquitous Computing, forthcoming.

[98] G. Dobson, R. Lock, and I. Sommerville. Quality of Service Requirements Specification Using an Ontology. SOCCER Workshop, Requirements Engineering '05, 2005.

[99] L. Barkhuus and A. Dey. Is Context-Aware Computing Taking Control Away from the User? Three Levels of Interactivity Examined. In Proceedings of UBICOMP 2003, 5th International Symposium on Ubiquitous Computing, pp. 149-156. October 12-15, 2003.

[100] J. J. Panel and J. Z. Pan. XML Schema Datatypes in RDF and OWL. Available at http://www.w3.org/TR/swbp-xsch-datatypes/. Accessed April 10, 2006.

[101] B. Bride. Jena: Implementing the RDF Model and Syntax Specification. Hewlett Packard Laboratories, 2001. Available at: http://www.hpl.hp.com/personal/bwm/papers/20001221-paper/. Accessed April 05, 2006.

[102] KAON2 – Ontology Management for the Semantic Web. Available at http://kaon2.semanticweb.org. Accessed April 10, 2006.

[103] Jena – A Semantic Web Framework for Java. Available at: http://jena.sourceforge.net/. Accessed April 10, 2006.

[104] RuleML Homepage. Available at: http://www.ruleml.org/. Accessed April 10, 2006.

[105] SweetRules Project Homepage. Available at: http://sweetrules.projects.semwebcentral.org/. Accessed April 10, 2006.

[106] F. Hayes-Roth. Rule-Based Systems. Communications of the ACM. Volume 28 (9), pp. 921-932, September, 1985.

[107] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004, September, 2004.

[108] Personalization of placed content ordering in search results. 2004. Available at: the United States Patent and Trademark Office (http://www.uspto.gov/), Published Application No. 20050240580. Accessed April 10, 2006.

[109] Yahoo! Maps. Available at: http://maps.yahoo.com/beta/. Accessed April 10, 2006.

[110] T. Gu, H. K. Pung, and D. Q. Zhang. A Middleware for Building Context-Aware Mobile Services. In Proceedings of IEEE Vehicular Technology Conference (VTC2004), Milan, Italy, 2004.

[111] H. Chen, T. Finin, and A. Joshi. Semantic Web in the Context Broker Architecture. In Proceedings of PerCom 2004, Orlando FL., March, 2004.

[112] A. Sieg, B. Mobasher, R. Burke, G. Prabu, and S. Lytinen. Representing User Information Context with Ontologies. In Proceedings of HCI International 2005 Conference, Las Vegas, Nevada, July, 2005.

[113] G. Akrivas, M. Wallace, G. Andreou, G. Stamou, and S. Kollias. Context-sensitive semantic query expansion. In Proceedings of the IEEE International Conference on Artificial Intelligence Systems (ICAIS), Divnomorskoe, Russia, September, 2002.

[114] J. Korva, J. Plomp, P. Määttä, and M. Metso. On-line service adaptation for mobile and fixed terminal devices. In Proceedings of the Second International Conference on Mobile Data Management, Hong Kong, pp. 252 – 259, 2001.

[115] Yahoo! Developer Network. Available at: http://developer.yahoo.net/. Accessed April 10, 2006.

[116] CORBA/IIOP Specifications at OMG. Available at: http://www.omg.org/technology/documents/corba_spec_catalog.htm. Accessed April 10, 2006.

[117] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim. TinyREST – a Protocol for Integrating Sensor Networks into the Internet. In Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05), 2005.

[118] The DARPA Agent Markup Language Homepage. Available at: http://www.daml.org/. Accessed April 10, 2006.

[119] G. J. F. Jones and P. J. Brown. Context-Aware Retrieval for Ubiquitous Computing Environments. In *Mobile and ubiquitous information access*, Springer Lecture Notes in Computer Science, Volume 2954, pp. 227-243, 2004.

Web References

[120] Hypertext Transfer Protocol -- HTTP/1.1
http://www.ietf.org/rfc/rfc2616.txt

[121] Uniform Resource Identifiers
http://www.ietf.org/rfc/rfc2396.txt

[122] W3 Consortium, “Resource Description Framework”
http://www.w3.org/RDF/

[123] W3 Consortium, “Web Ontology Language Overview",
http://www.w3.org/2001/sw/WebOnt/

[124] Web Services Description Language (WSDL) 1.1, W3 Consortium, 2001
http://www.w3.org/TR/wsdl

[125] ASN.1 specification, John Larmouth
http://www.oss.com/asn1/larmouth.html

[126] Extensible Markup Language, W3 Consortium
http://www.w3.org/XML

[127] DCMI Metadata Terms, Dublin Core Metadata Initiative
http://dublincore.org/documents/dcmi-terms

[128] XML Schema, W3 Consortium
http://www.w3c.org/TR/xmlschema-0

8. Appendix

8.1. List of Figures

8.2. List of Tables