universität
wien

MASTERARBEIT

Titel der Masterarbeit

Towards a Context-Sensitive Infrastructure for a Mobile Semantic Desktop Integration

verfasst von

Carina Christ, BSc

angestrebter akademischer Grad

Diplom-Ingenieurin (Dipl.-Ing.)

Wien, 2014

Studienkennzahl lt. Studienblatt:	A 066 935
Studienrichtung lt. Studienblatt:	Masterstudium Medieninformatik
Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Declaration of Authorship

I, Carina Christ, declare that this thesis titled, 'Towards a Context-Sensitive Infrastructure for a Mobile Semantic Desktop Integration' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

As mobile phones are nowadays more and more used as organizers and are a fixed part of our everyday life, the demand for *intelligent* applications that help to save time in particular situations is increasing. The adoption of Semantic Web technologies for mobile desktops enables the development of *intelligent* mobile phones. The goal of this work is the development of a system that combines personal data items and contextual data in one data structure and provides an interface that enables applications to use information from this system. Therefore, we introduce a new type of data structure, which enables the semantic representation of data items stored on a mobile phone as well as the representation of sensed contextual data. Furthermore, an infrastructure that facilitates the development of context-aware smartphone applications is developed. We propose an infrastructure that considers the inherent characteristics of mobile devices and does not rely on external servers by adopting Semantic Web technologies respectively Semantic Desktop approaches for mobile desktops. Already existing approaches meet only individual aspects of the before mentioned but do not imply all of these requirements. We demonstrate the practicability of the presented approach through a prototypical implementation of the infrastructure and an application using it. The initial study of the prototypes conducted by external test persons showed that the system works as expected and performance is acceptable.

Zusammenfassung

Da Mobiltelefone heutzutage immer mehr zum Organisieren von Terminen, Schreiben von E-Mails und Verwalten von Daten verwendet werden und deshalb ein ständiger Begleiter und fixer Bestandteil unseres täglichen Lebens sind, wächst auch der Bedarf für *intelligente* Anwendungen, die uns in bestimmten Situationen helfen sollen Zeit zu sparen. Die Einführung von Semantic Web Technologien für mobile Desktops ermöglicht es *intelligente* Mobiltelefone zu entwickeln. Das Ziel dieser Arbeit ist die Entwicklung eines Systems, das persönliche Daten, wie zum Beispiel E-Mails, Dateien, Kalendereinträge und dergleichen, sowie kontextuelle Daten in einer Datenstruktur vereint und eine Schnittstelle zur Verfügung stellt, die es Anwendungen ermöglicht die gesammelten Informationen des Systems zu nutzen. Dafür, führen wir eine neue Art von Datenstruktur ein, die die semantische Repräsentation von Daten, welche auf einem Mobiltelefon gespeichert sind sowie die Repräsentation von kontextspezifischen Daten, die von Sensoren erfasst werden können, ermöglicht. Weiters wird eine Infrastruktur entworfen, die die Entwicklung von kontextsensitiven Smartphone Applikationen erleichtert. Diese Infrastruktur berücksichtigt die inhärenten Eigenschaften von mobilen Endgeräten und ist nicht von externen Servern abhängig. Um dieses Ziel zu erreichen, wenden wir Semantic Web Technologien beziehungsweise Semantic Desktop Ansätze für mobile Desktops an. Es existieren bereits Ansätze, die einzelne der vorher erwähnten Aspekte, jedoch nicht alle diese Anforderungen, erfüllen. Um die Umsetzbarkeit des präsentierten Ansatzes zu demonstrieren, wird ein Prototyp der Infrastruktur sowie eine Beispielanwendung, die diese Infrastruktur nutzt, implementiert. Eine erste Evaluierung mit einer kleinen Gruppe von Testpersonen hat gezeigt, dass das System wie erwartet funktioniert und die Performance ausreichend ist.

Acknowledgements

First, I would like to thank my first advisor Dipl.-Inf. Dr. Stefan Zander, MSc, M.I.T. - who accompanied me through the bigger part of this work - for his guidance and valuable feedback as well as input during the preparation of this work. I also like to thank my subsequent two advisors, Dipl.-Ing. Peter Kalchgruber and Dipl.-Ing. Elaheh Momeni Roochi, BSc - who took over this function after Mr. Zander left the University of Vienna - for the guidance through the last part of this work and their likewise valuable feedback and input.

Furthermore, I would like to thank those who spent time on discussing different aspects of this work with me and gave valuable inputs, those who participated in the evaluation process of the prototype implementation and those who spent time on proofreading my work.

I would also like to thank my employer and colleagues for their understanding and support that facilitated the finalization of my thesis.

Special thanks are due to my family and friends for their support, encouragement, and sacrifices made throughout this work and the whole duration of study.

Contents

List of Figures

List of Tables

Abbreviations

AIDL	**A**ndroid **I**nterface **D**efinition **L**anguage
API	**A**pplication **P**rogramming **I**nterface
BBC	**B**ritish **B**roadcasting **C**orporation
CASS	**C**ontext-**a**wareness **s**ub-**s**tructure
CML	**C**ontext **M**odelling **L**anguage
CoDAMoS	**Co**ntext-**D**riven **A**daptation of **Mo**bile **S**ervices
CONON	**Con**text **On**tology
DAML-OIL	**D**ARPA **A**gent **M**arkup **L**anguage - **O**ntology **I**nterchange **L**anguage
DFKI	**D**eutsches **F**orschungszentrum für **K**ünstliche **I**ntelligenz
GPS	**G**lobal **P**ositioning **S**ystem
GSM	**G**lobal **S**ystem for **M**obile Communications
GUI	**G**raphical **U**ser **I**nterface
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
NEPOMUK	**N**etworked **E**nvironment for **P**ersonalized, **O**ntology-based **M**anagement of **U**nified **K**nowledge
ORM	**O**bject **R**ole **M**odel
OWL	**W**eb **O**ntology **L**anguage
PDA	**P**ersonal **D**igital **A**ssistant
PIM	**P**ersonal **I**nformation **M**anagement
PIMO	**P**ersonal **I**nformation **Mo**del
RDF	**R**esource **D**escription **F**ramework
RDFS	**R**esource **D**escription **F**ramework **S**chema
SeMoDesk	**Se**mantic **Mo**bile **Desk**top
SMS	**S**hort **M**essage **S**ervice
SOCAM	**S**ervice-**O**riented **C**ontext-**A**ware **M**iddleware

SPARQL	**S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
SQL	**S**tructured **Q**uery **L**anguage
UID	**U**nique **Id**entifier
URI	**U**niform **R**esource **I**dentifier
URIQA	**URI Q**uery **A**gent Protocol
W3C	**W**orld **W**ide **W**eb **C**onsortium
XML	E**x**tensible **M**arkup **L**anguage

Chapter 1

Introduction

Mobile phones are an important part of our private and business life. Technical advances and the omnipresence of them make it possible to achieve more computationally expensive tasks. Because of constantly enhancements, in terms of storage capacity and processing power, it is nowadays possible to accomplish more than just telephone calls and SMS with mobile phones. They are more and more used for organizing appointments, writing e-mails and surfing the Web, as well as for navigating to destinations. Furthermore, social features like Facebook and Twitter, are widely-used and there are mobile applications for all circumstances. That is the reason why for most people their mobile phone is a permanent part of their everyday life.

Therefore, it is important to organize one's data on a mobile device in such a way that the user can retrieve them always and anywhere respectively the mobile phone is able to show adequate information in different situations automatically. For this purpose, a new kind of data storage and data organization has to be introduced. On personal computers such an approach, called Semantic Desktop [Sau05b], is already used and can also be adopted for mobile devices. By means of this technology, one is relieved of searching data that is needed (at the moment) by special algorithms the computer is processing. An advantage of Semantic Desktop is most notably the saving of time when organizing data. In contrast to conventional file systems, where different file types are stored in variable ways without relations to each other, with the Semantic Desktop it is possible to interlink heterogeneously stored personal data semantically and hence data retrieval gets more facilitated (cf. Section 2.7). It enables, for instance, to relate e-mails, files, and contact data with a specific project and pro-actively get all these relevant data proposed in a situation, when such entities are needed. Because of the increasing storage capacity and processing power, such scenarios are also possible on mobile devices nowadays. By means of GPS data, it is additionally feasible that a mobile application reminds the user

of, for example, doing the shopping for the party at the weekend when they are going past a supermarket, because this task is stored as a to-do entry on the mobile phone and a supermarket is recognized nearby. However, this is only one of many scenarios that can be handled by interlinking data semantically. With this context-dependent linkage of Semantic Desktop and mobile device a big field of new applications is getting feasible.

Mobile applications that cover such scenarios need to acquire contextual data from different sources, represent them in a uniform machine-readable way and define rules in order to recognize which information could be important for the user in which situation. Contextual data is data on the mobile phone that defines the user's current situation/environment (cf. Section 2.4 and Section 2.5). The more relevant data about a particular situation is available the more specifically a "context-aware semantic personal assistance" application[1] is able to work.

As the basic requirements for most of these applications are the same, the goal of this work is to develop a common, generic infrastructure for them to accelerate and simplify the development process of "context-aware semantic personal assistance" applications. This infrastructure goes without an external server, i.e. all data is processed and stored on the mobile phone. It acquires available context-specific data, exposes them as RDF statements, interlinks and aggregates them, stores them on the mobile phone, and provides an interface for applications to use it. Hence, application developers of "context-aware semantic personal assistance" applications do not need to acquire and aggregate context-specific data nor link them. Another subgoal was to create a context model schema that includes all data that is needed for the infrastructure and reuse as many existing vocabulary as possible. With this infrastructure and the corresponding data structure for storing information the mobile phone will not be a conventional data storage any more, but gets a good organizer, which can save the user a lot of time and helps to achieve organizational tasks faster.

There has already be done a lot of research on this topic. Gnowsis [Sau05a], for example, is a Semantic Desktop application developed for personal computers, which uses a central server for storing and managing data. However, also mobile approaches like [HSP$^+$03], [dFRRR04], and [ZS12] - which are similar to this work - already exist. However, none of them implements all requirements we have defined for our approach (cf. Section 3.1). There are, for example, some that use external servers for storing the data or processing data. Some do not provide context history, which is very important for situation identification, and some do not use RDF based context models, which is important regarding reusability, reasoning, and exchange of data.

[1] These are applications for mobile phones that act like a "personal assistant" for the user and therefore use context-specific and semantically linked data.

The prototype implementation and its subsequent light evaluation show that the most important technical requirements for the realization of a mobile Semantic Desktop infrastructure are given and performance is acceptable. However, there are some technical issues that may be improved in future work.

The remaining work is structured as follows. First, in Chapter 2 all relevant concepts and technologies that our work relies on are introduced. In Chapter 3 existing context-aware middleware approaches, infrastructures, and applications are reviewed and compared, according to some selected qualities. Chapter 4 describes all components and aspects of our approach, including the specification of the used context model schema. All aspects are additionally exemplified by means of two concrete scenarios. The prototype implementation of a context-aware mobile Semantic Desktop infrastructure integration is discussed in Chapter 5. Therefore, a subset of the system's components has been implemented. Furthermore, an application using the mobile Semantic Desktop infrastructure has been developed as a proof of concept and described in Chapter 5. Chapter 6 describes the evaluation process of the prototype implementation and discusses its results. Finally, in Chapter 7 we draw conclusions of our findings and discuss future work.

Chapter 2

Background

In this chapter, all relevant background knowledge for this work is described. First, we introduce the concept behind the Semantic Web (Section 2.1) including its enabling technologies. These are the Resource Description Framework (RDF) that is used for representing information in a machine-readable way (Section 2.1.1), the Uniform Resource Identifier (URI) which serves as an identifier for all kinds of resources described with RDF (Section 2.1.2), and the RDF query language SPARQL, which enables the retrieval and manipulation of RDF data (Section 2.1.3). Moreover RDF-Schema (RDFS) is counted among the enabling technologies for the Semantic Web and is introduced in Section 2.1.4. In Section 2.2, we provide information about ontologies and OWL (Section 2.2.1) which are tools for describing the properties of resources and relationships between different types of resources. Afterwards, reasoning techniques are introduced in Section 2.3. The notion of context and various context model schemas (Section 2.4.1) that are used in similar works are discussed in Section 2.4. Then, context awareness is introduced (Section 2.5) and we describe the practice of personal information management in Section 2.6. In the end, the concept of the Semantic Desktop, which will be adopted for mobile phones in this work, is described (Section 2.7).

2.1 Semantic Web

To understand the Semantic Web, we first explain the word "semantic". Semantic can be defined as the meaning of something. We can write a sentence, for example, "*I like strawberries*", which has a certain meaning. As a human, we understand this sentence, i.e., we are able to comprehend the inherently given meaning of the sentence. On the other hand, if there is a document with the same sentence to be processed by a computer,

the computer knows the syntax, but does not know anything about its meaning. It needs semantic annotations and axioms in order to deduce the meaning of resources.

The goal of the Semantic Web is to explicitly express meaning of resources in order to get the computer recognize the semantics of them. A resource in the Semantic Web is everything that can be clearly identified [Mir08]. This can be both *information* and *non-information resources*. Information resources are all kinds of documents, like web pages, images, or video files, which "*can be transmitted electronically*" [Tau08]. Non-information resources are all other resources, like people, events, animals, or any abstract concepts. This enables the computer to learn what we are interested in and, as a result, actively supports us. Furthermore, this knowledge can enhance search engines and makes the information retrieval process more accurate [Spo]. For example, a conventional search engine is not able to deduce what somebody is exactly searching for, when the word "plane" is typed in. It may be a means of transport or a tool. However, a semantic search engine may know the difference - providing that there are rules, which enable it to reason about the actual meaning. According to [BLHH+06], the difference between a conventional search engine and a semantic search engine is that the former practices "information retrieval". Its goal is to provide documents that include the keywords of the query, without considering semantics. The aim of a semantic search engine is to "produce the correct answer to the query" [BLHH+06].

The Semantic Web has little to do with documents, but rather with things, respectively data about things. Each thing (in the Semantic Web context, a thing is called *resource*) on the Web has a unique identifier and everyone can make statements about these things. In contrast to the traditional Web, in the Semantic Web, information is linked by its meaning. This enables the computer to deduce relationships between things and infer new information [Spo]. An important step to achieve this is to define a uniform data format [Alt]. Therefore, a variety of technologies such as RDF (Section 2.1.1), RDFS (Section 2.1.4), and OWL (Section 2.2.1) have evolved and are described in the next sections. With these technologies, ontologies (Section 2.2) are developed to form a common understanding of data.

The Semantic Web Stack in Figure 2.1 shows the hierarchical structure of the Semantic Web. On the lowest level, the Uniform Resource Identifier is located, which builds the basis of the Semantic Web. RDF is located above XML and is used to exchange data. On the next higher level of the stack ontology-, query-, and rule-languages, which are described in the following sections, are situated [BLHH+06]. The aim of the proof layer is to make semantic activities and inference steps, processed by the machine, transparent and comprehensible for human beings. By means of digital signatures and cryptography, it is possible to proof the originality of information [Mir08].

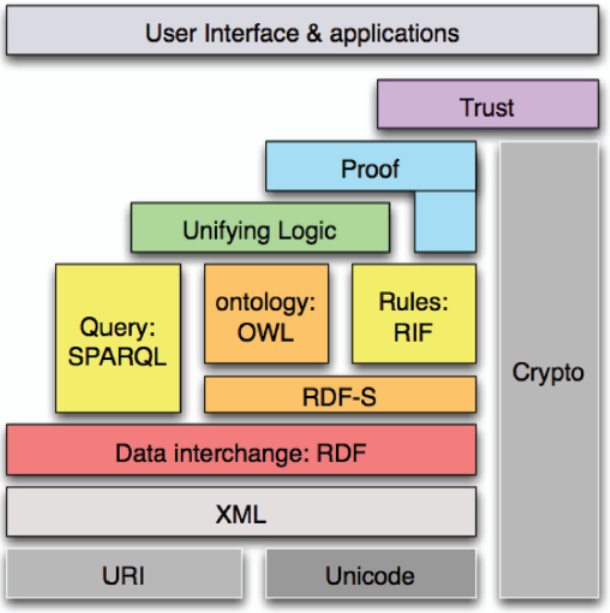

FIGURE 2.1: Semantic Web Stack. From [BLHH+06]

Some well-known companies already use Semantic Web technologies in order to improve, for example, their web presence. The British Broadcasting Corporation[1] (BBC) uses Semantic Web technologies since 2007 for a good portion of their websites. They introduced, amongst others, *BBC Programmes* and *BBC Music*. By means of the *BBC Programmes Ontology*[2], every information concerning programmes can be represented and interlinked with each other using RDF. This enables the presentation of all available programmes on their own web page that is composed automatically instead of handcrafted. BBC Music, furthermore, takes information from *Musicbrainz* and *Wikipedia* and presents it on their website [RSS+10].

Another example for the successful usage of Semantic Web technologies in a well-known company is BestBuy.com[3]. They use RDF in order to represent data about their products and prices using the GoodRelations vocabulary[4] for e-commerce. With this vocabulary, it is possible to describe products, the location of them, as well as opening hours of the location. BestBuy.com experienced a significant improvement in search engine optimization, since the introduction of RDF for the representation of product information [Sie09].

For this work the Semantic Web serves as a basic principle that provides important technologies enabling the development of a mobile Semantic Desktop infrastructure.

[1] `http://www.bbc.co.uk/`
[2] cf. `http://www.bbc.co.uk/ontologies/programmes/2009-09-07.shtml`
[3] `http://www.bestbuy.com/`
[4] cf. `http://www.heppnetz.de/ontologies/goodrelations/v1`

2.1.1 Resource Description Framework (RDF)

RDF is commonly defined as "*a language for representing information about resources in the World Wide Web*" [MM04] in order to be able to use it within applications that process such data. Information, which is represented by RDF, can furthermore be exchanged between applications. One aspect of RDF is to describe resources by means of property-value pairs. Each resource is identified by a Uniform Resource Identifier (URI) (cf. Section 2.1.2). Using these URIs, so-called *statements* can be expressed. A statement consists of a subject, a predicate, and an object, whereas subjects and predicates are represented by URIs and objects can either be represented by URIs too or be literals [MM04]. If the object is a literal, it can have an associated data type [Bru04]. An RDF statement in Turtle[5] syntax looks as follows:

```
<http://www.myns.at/emails/12> <http://www.semanticdesktop.org/ontologies/
        #sentDate> "February 25, 2012" .
```

Code 2.1: Example RDF statement

Code 2.1 represents the following meaning: "The resource identified by the URI `http://www.myns.at/emails/12` has a *date of sending* whose value is *February 25, 2012*." In this case, *http://www.myns.at/emails/12* is the subject, *date of creation* the predicate, and *February 25, 2012* the object. There exists, amongst others, an XML-based syntax (see Code 2.2), as well as a graphical representation (see Figure 2.2) for RDF statements. The graphical representation is a named, directed graph consisting of nodes and arcs, whereas the nodes represent resources and the arcs represent properties/predicates. Each arc with its associated nodes constitutes a so-called *RDF-triple*. Triples that end with a resource are called *resource-triples*, whereas triples whose values are literals are called *literal-triples* [Bru04].

In the graph in Figure 2.2 a resource `http://univie.ac.at/christ/contacts#pid37`, which is from type `http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#Contact`, has a *given name* "Carina", and a *family name* "Christ" can be seen. Furthermore, there is a resource `http://univie.ac.at/christ/emails#item3`, which is from type `http://ont.semanticdesktop.org/ontologies/2007/03/22/nmo#Email` and has a property `http://semanticdesktop.org/ontologies/2007/03/22/nmo#from` with the value `http://univie.ac.at/christ/contacts#pid37`.

URIs in RDF do not necessarily have to be dereference-able; they are primarily used for identification. Identical URIs in different statements have to refer to the same resource [BL03].

[5] cf. `http://www.w3.org/TeamSubmission/turtle/`

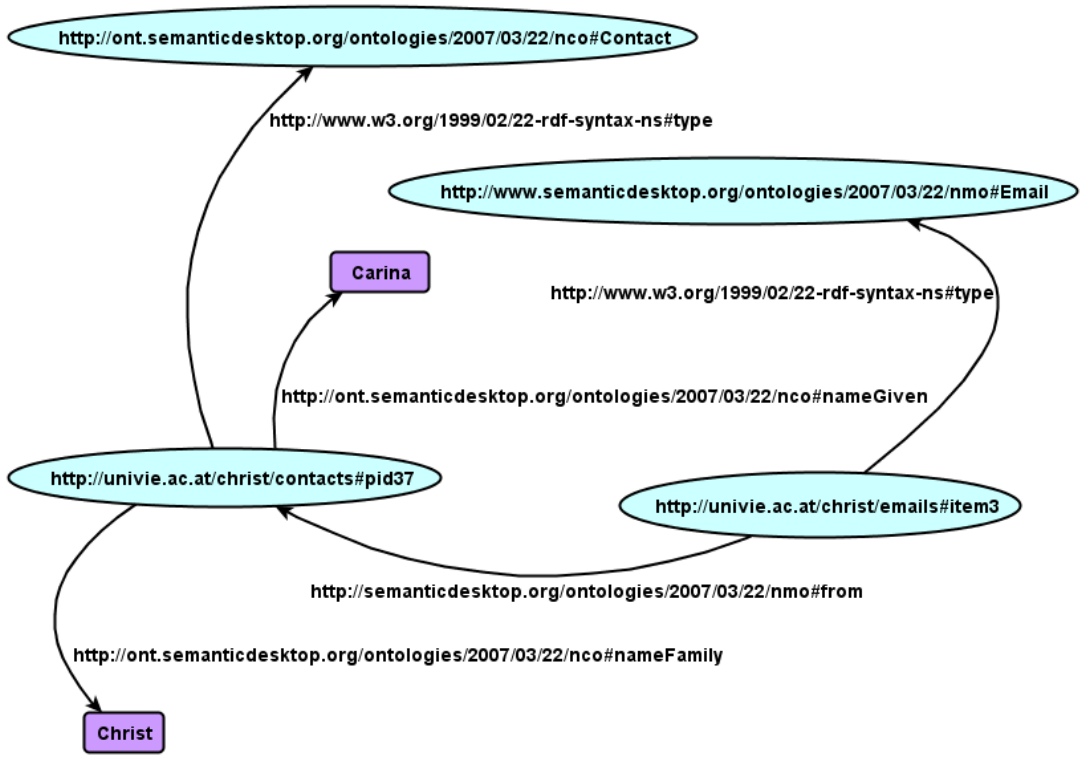

FIGURE 2.2: Example RDF graph

In the following example, the above graph is transformed into the RDF/XML serialization syntax:

```
1  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2           xmlns:nco="http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#"
3           xmlns:nmo="http://www.semanticdesktop.org/ontologies/2007/03/22/nmo#">
4    <rdf:Description rdf:about="http://univie.ac.at/christ/emails#item3">
5      <nmo:from rdf:resource="http://univie.ac.at/christ/contacts#pid37" />
6      <rdf:type
7         rdf:resource="http://www.semanticdesktop.org/ontologies/2007/03/22/
8          nmo#Email" />
9    </rdf:Description>
10   <rdf:Description rdf:about="http://univie.ac.at/christ/contacts#pid37">
11     <rdf:type
12        rdf:resource="http://ont.semanticdesktop.org/ontologies/2007/03/22/
13         nco#Contact" />
14     <nco:nameFamily>Christ</nco:nameFamily>
15     <nco:nameGiven>Carina</nco:nameGiven>
16   </rdf:Description>
17 </rdf:RDF>
```

CODE 2.2: RDF graph in XML syntax

RDF is used within our approach in order to represent all used resources in a uniform and machine-readable way. Furthermore, it facilitates reasoning and enables querying by means of SPARQL.

While RDF specifies possible syntaxes for the description of resources, we need RDF-Schema (Section 2.1.4) or OWL (Section 2.2.1) to define the semantics.

2.1.2 Uniform Resource Identifier (URI)

URIs are used to identify resources in the context of the Semantic Web. They have a global scope. A URI can either be used as a name or location for something and can also be both. In the case of a name, it is called Uniform Resource Name (URN) and in the case of a location, it is a Uniform Resource Locator (URL) as it is used in the conventional Web [BLHH+06]. A URI has a generic syntax, which is organized hierarchically. The building parts of a URI are: scheme, authority, path, query, and fragment [BLFM05], as shown in Figure 2.3.

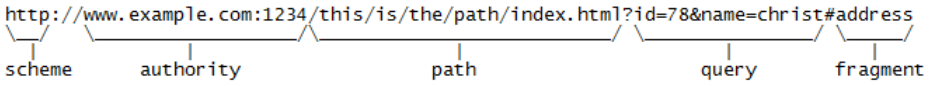

FIGURE 2.3: Building parts of a URI

A good introduction for the usage of URIs as identifiers in the context of the Semantic Web, is given in [SCV07].

URIs are amongst the most important parts of RDF and the Semantic Web. They represent resources and at the same time uniquely identify them.

2.1.3 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL is a query language for RDF. With its syntax, it is possible to express queries for data that exist in an RDF format [PS08]. It is not only possible to search in one data source, but also in different data sources at the same time. Furthermore, optional parts can be defined within queries. As a result, an RDF graph or a result set is provided. Result sets are like tables, where each row refers to one result. Each column header represents a variable and the cell contents contain the values of the variables. [PS08] describes SPARQL queries as follows:

> "*Most forms of SPARQL query contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples except that each of the*

> *subject, predicate and object may be a variable. A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables and the result is RDF graph equivalent to the subgraph.*" [PS08]

A SPARQL query is similar to an SQL query. The most simple one consists of a `SELECT` part and a `WHERE` clause. For example:

```
1  PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2
3  SELECT ?name
4  WHERE
5  {
6    <http://univie.ac.at/christ/contacts#pid37> nco:nameFamily ?name
7  }
```

Code 2.3: Simple SPARQL query

This query returns a result set with one column and usually one row because normally one contact has only one family name. However, SPARQL queries can also deliver more than one or no results. The result set of the query shown in Code 2.3 may look like the following:

name
Christ

Table 2.1: Result set of the query shown in Code 2.3

The `SELECT` clause in Code 2.3 line 3 defines the parts of the data that should appear in the result set and the `WHERE` clause in Code 2.3 lines 4 - 7 provides the basic graph pattern that has to match against the given data graph. The `PREFIX` part in the first line defines an abbreviation for the used namespace. Variables always begin with a "?" or a "$". In the above example, the family name of a given contact is queried.

Furthermore, it is possible to query for entries with specific literal objects. For example:

```
1  PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2
3  SELECT ?p
4  WHERE
5  {
6    ?p nco:nameGiven "Carina"
7  }
```

Code 2.4: SPARQL query using literal

Simple `SELECT` statements result in a result set, where variables are bound to values. However, there is the `CONSTRUCT` query form that returns an RDF graph. The representation form of the graph is defined in the `CONSTRUCT` clause (Code 2.5 lines 3 - 6). For example:

```
1  PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2
3  CONSTRUCT
4  {
5    ?p nco:nameGiven ?name
6  }
7  WHERE
8  {
9    ?p nco:nameFamily ?name
10 }
```

Code 2.5: SPARQL CONSTRUCT query

This query may return the following graph representation:

```
1  PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2
3  _:p nco:nameGiven "Cook" .
4  _:s nco:nameGiven "Murphy" .
```

Code 2.6: Resulting graph from query shown in Code 2.5

Another possibility to refine a query is the usage of `FILTER` expressions. They can be applied for constraining numeric and string values. An example of a numeric constraint is given in Code 2.7. This code snippet specifies that all people, which are allowed to be added to the result set, have to be older than 20 years:

```
1  PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2  PREFIX ns: <http://www.carinachrist.com/ns#>
3
4  SELECT ?name ?age
5  WHERE
6  {
7    ?p nco:nameFamily ?name .
8    ?p ns:age ?age .
9    FILTER (?age > 20)
10 }
```

Code 2.7: SPARQL query with FILTER expression

As mentioned before, it is also possible to include optional parts in the query. These are parts that do not have to match in order to be added to the result set. To achieve this, the keyword `OPTIONAL` has to be written in front of the basic graph pattern. There may be multiple optional parts in one query. For example:

```
1  PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2  PREFIX  ns: <http://www.carinachrist.com/ns#>
3
4  SELECT ?name ?age
5  WHERE
6  {
7    ?p nco:nameFamily ?name .
8    OPTIONAL { ?p ns:age ?age }
9  }
```

Code 2.8: SPARQL query with OPTIONAL part

This query adds all resources that have a property `nco:nameFamily` to the result set, no matter whether they also have a property `ns:age` or not. If a resource does not have this property, the cell in the result set remains empty. Table 2.2 shows a possible result set for the query shown in Code 2.8.

name	age
Christ	25
Jones	32
White	
Jackson	17

Table 2.2: Result set of the query shown in Code 2.8

Furthermore, there is a way to query for alternatives, which means only one basic graph pattern must match. This can be achieved by using the keyword `UNION`. If more than one alternatives match, all of them are added to the result list. A simple example is given here:

```
1  PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3
4  SELECT ?name
5  WHERE
6  {
7    { ?p nco:nameFamily ?name } UNION { ?p foaf:name ?name }
8  }
```

Code 2.9: SPARQL query with UNION expression

The query in Code 2.9 will provide a result set with one column in which all values of the properties `nameFamily` and `foaf:name` are listed. For this example, we use the following RDF data:

```
1 PREFIX nco: <http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#>
2 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3
4 <http://univie.ac.at/contacts#pid37> nco:nameFamily "Christ";
5         nco:nameGiven "Carina" .
6 <http://univie.ac.at/contacts#pid68> nco:nameFamily "Jones" .
7 <http://univie.ac.at/contacts#pid9> nco:nameFamily "Williams";
8         foaf:name "Willi";
9         foaf:age "39" .
```

Code 2.10: Data set used for the query shown in Code 2.9

Given this RDF data set, the result from the query shown in Code 2.9 will be the following:

Christ
Jones
Williams
Willi

Table 2.3: Result set of the query shown in Code 2.9

Like in SQL, there is also a `FROM` and an `ORDER BY` clause in SPARQL, as well as a few other constructs. A more detailed description of the SPARQL syntax is given in [PS08].

Within this work SPARQL is used to retrieve data from an RDF graph and to provide parts of the graph for applications that use our infrastructure.

2.1.4 RDF - Schema (RDFS)

As mentioned before, with RDF it is not possible to define semantics. For this purpose, RDFS can be used, which is a so-called "RDF vocabulary description language" [BGM04]. With this language, it is possible to define and describe different types of classes (`rdfs:Class`) and properties (`rdf:Property`), as well as relations between them, using RDF statements. RDFS' classes and properties can be compared to the concept of classes in object-oriented programming languages. RDFS provides mechanisms to specify the type of a resource or property; which type can have what properties; and the type of the property values. To show that a particular resource is of a specific type, the property `rdf:type` is used. With `rdfs:range`, the type of a property-value can be defined. To define the type of a resource that has a certain property, `rdfs:domain` can be used [BGM04]. For an example, see Figure 2.4.

The graph in Figure 2.4 defines that `http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#nameGiven` is from type `http://www.w3.org/1999/02/22-rdf-syntax-ns#Property`, has a domain `http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#Contact`, and a range `http://www.w3.org/`

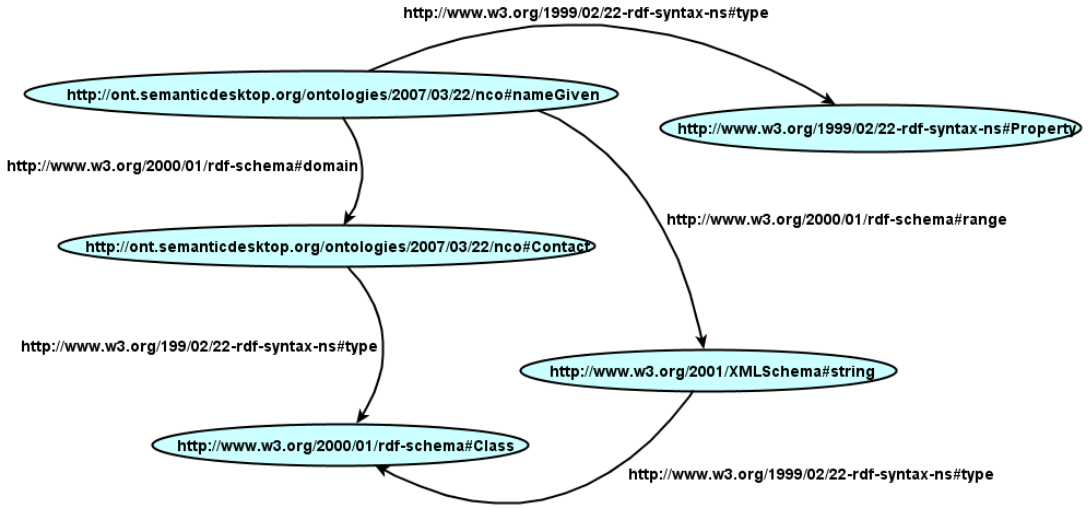

FIGURE 2.4: RDF-Schema: classes and properties

2001/XML-Schema#string, i.e., a resource that is from type `nco:Contact` can have the property `nco:nameGiven` and the value of the property has to be from type `xsd:string`. Furthermore, it is defined that `http://ont.semanticdesktop.org/ontologies/2007/03/22/nco#Contact` and `http://www.w3.org/2001/XML-Schema#string` are from type `http://www.w3.org/2000/01/rdf-schema#Class`.

It is distinguished between classes and instances of classes. With `rdfs:subClassOf` and `rdfs:subPropertyOf` it is possible to define subclasses and subproperties. If B is a subclass of A, all instances of B are instances of A as well. The URI-prefix of RDFS resources is `http://www.w3.org/2000/01/rdf-schema#`.

An Example of a defined class hierarchy is shown in Figure 2.5.

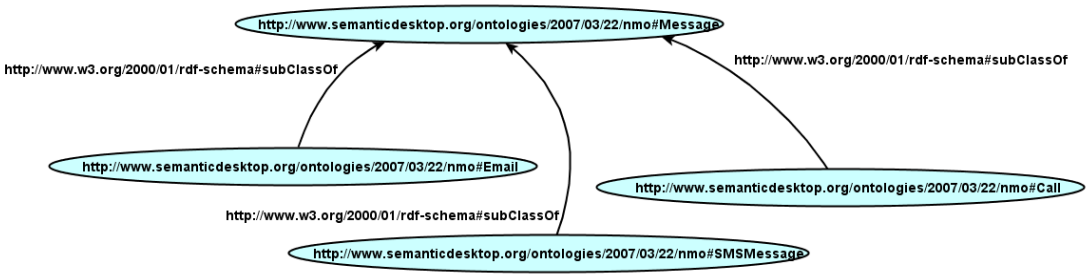

FIGURE 2.5: RDF-Schema: class hierarchy

In Figure 2.5 it is defined that `http://www.semanticdesktop.org/ontologies/2007/03/22/nmo#Email`, `http://www.semanticdesktop.org/ontologies/2007/03/22/nmo#Call`, and `http://www.semanticdesktop.org/ontologies/2007/03/22/nmo#SMSMessage` are subclasses of `http://www.semanticdesktop.org/ontologies/2007/03/22/nmo#Message`.

With RDFS, moreover, literals (`rdfs:Literal`), datatypes (`rdfs:Datatype`), and XML literals (`rdf:XMLLiteral`) can be defined. Literal values, like dates and integers, are instances of the class `rdfs:Literal`. Instances of `rdfs:Datatype` are typed literals and instances of `rdf:XMLLiteral` may be XML content. Besides `rdfs:domain` and `rdfs:range`, there are the properties `rdfs:label` and `rdfs:comment`. With `rdfs:label` it is possible to define a human-readable label for a resource. `rdfs:comment` is used in order to add a human-readable description of a resource.

We use RDFS for our approach to describe the structure of the used context model that is introduced in Section 4.4.

2.2 Ontologies

The word "ontology" has its origins in the Greek language. The first part "onto-" means "being" and the second "-logia" means "study" or "theory". With ontologies, one can specify concepts that are important in a specific knowledge domain by means of a predefined terminology as well as relationships between the defined things. A knowledge domain can be defined as a *specific area of expertise*. Examples would be sports, weather, or music. For instance, in the sports domain, we can have the classes *ball game*, *athletic sport*, and *martial arts*, which are all subclasses of the concept *sport*. All these subclasses can have individual properties, like *isTeamsport* or *usesRacket*. Instances of the classes might be the particular sport disciplines, like *basketball*, *tennis*, or *judo*.

In computer science, ontologies are foremost used to facilitate the sharing of knowledge between different actors, which can be both humans and machines. This is typically achieved by using machine-readable representation languages.

One of the most cited definitions for an ontology in the domain of computer science is the following:

> "*An ontology is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an Ontology is a systematic account of Existence. For AI systems, what 'exists' is that which can be represented.*" [Gru95]

This is a rather general definition of an ontology. A more precise definition is given in [UG96]:

> "'Ontology' *is the term used to refer to the shared understanding of some domain of interest [...].*

> *An ontology necessarily entails or embodies some sort of world view with respect to a given domain. The world view is often conceived as a set of concepts (e.g. entities, attributes, processes), their definitions and their interrelationships; this is referred to as a conceptualisation.*" [UG96]

This definition includes the main purpose of ontologies, namely the *shared understanding of some domain of interest*, as well as the constituents of them. In the following, these are further described [MS01]:

- *concepts* Concepts are abstract classes, which are hierarchically organized. They are interrelated by subclass relations and can have properties. In our approach, for example, "e-mail" or "contact" are concepts.
- *instances* Instances are concrete objects that are related to a concept. An instance of a contact may be "Peter James".
- *attributes* Attributes describe concepts and are of a particular data type. For example, the name and the telephone number of a contact are attributes.
- *relations* A relation is a connection between (non-hierarchical) concepts. In the context of our approach a relation may be "e-mail sent by contact".
- *axioms* An axiom is a rule, which is always true, without having to be proofed. For example: "If a person A is in a meeting with person B, then person B is also in a meeting with person A."

An advantage of ontologies is the common vocabulary for all agents that are involved in the knowledge sharing process. Furthermore, a common understanding about the structure of information between all agents can be achieved and the knowledge can be reused within the domain. With ontologies hypotheses according to a particular domain are made explicitly [NM01]. In this work, we especially use the aspect of reusability by composing and extending already existing ontologies in order to achieve a context model schema that meets our requirements (cf. Section 4.4).

Ontologies are already used in many different domains, like medicine, for example the Gene Ontology[6] or in the telecommunication sector, for example the BBC Ontologies[7] (cf. Section 2.1).

To some extend, ontologies can be compared to database schemas. With a database schema, one can define a structure of entities and relationships between them for efficient storage and querying. With ontologies, the same can be achieved, but database

[6] cf. `http://www.geneontology.org/`
[7] cf. `http://www.bbc.co.uk/ontologies/`

schemas focus more on efficiently storing data, whereas ontologies are meant to represent meaning and achieve common understanding of data. Furthermore, ontologies can often be reused, whereas database schemas are not. Ontologies make it possible to represent sensor data, context, and situations. Furthermore, rules for reasoning (see Section 2.3) are supported, which make it possible to check the consistency of the model - for example, it can be checked if it is legal that class A is a subclass of class B - and infer new information [BBH+10].

According to the survey in [SLP04], ontologies are very well suited for context modelling in ubiquitous computing scenarios. One often used language to define the structure of an ontology is OWL, which is described in the next section.

2.2.1 Web Ontology Language (OWL)

OWL is a language for describing the constituents of ontologies in a formal way. It is a W3C Recommendation [BvHH+04] and an enhancement of DAML+OIL[8], which is another ontology description language. It uses an RDF-based syntax (cf. Section 2.1.1). In OWL, each class is a subclass of `owl:Thing`. Like with RDF/RDF-Schema classes, relations between classes, and instances of classes can be defined. However, it is more powerful than RDF/RDF-Schema and is able to specify ontologies more precisely. OWL allows to differentiate between several kinds of properties; values can be restricted; cardinalities can be defined; equivalent and disjoint classes can be defined; and same-as relationships can be represented. Properties may be further divided into more specific ones, like transitive (`owl:TransitiveProperty`), symmetric (`owl:SymmetricProperty`), functional (`owl:FunctionalProperty`), and inverse functional (`owl:InverseFunctionalProperty`) properties. An example for a transitive property would be the property *isPartOf*. If instance A isPartOf instance B, and B isPartOf instance C, then A is also part of C. A symmetric property is *staysWith*. That means, if person A staysWith person B, then B also staysWith A. Functional properties are properties for which only one (unique) value is possible per instance. For example, a person can only have one unique Social Security Number. An example for an InverseFunctionalProperty would be a person who owns a car. In this example, it is possible to uniquely determine the subject of the statement, by means of the object. Furthermore, the cardinality of the objects associated to a predicate can be restricted. For example, it is possible to specify that a person can have only one mother.

There are three different subsets of OWL. *OWL Lite* provides only vocabulary for class hierarchies and only a subset of class restrictions, as well as cardinalities of 1 and 0 can

[8] cf. `http://www.w3.org/TR/daml+oil-reference`

be used. The other two, namely *OWL DL* and *OWL Full*, use the same vocabulary, but in OWL DL there exist a few restrictions. However, it guarantees decidability and computability in a finite period of time. In OWL Full, there are no restrictions. More detailed information about the syntax of OWL, is given in [BvHH+04].

As mentioned before, we reuse already existing ontologies, which are described by means of OWL. Moreover, we use OWL for describing additional concepts that are important for our approach.

2.3 Inferencing/Reasoning

The study of knowledge has its origin in philosophy and was already discussed by the early Greek philosophers. However, probably von Wright dealt first with reasoning about knowledge in his work from the early 1950's (cf. [VW51]). In order to reason about knowledge, *common knowledge* and *distributed knowledge* is used. Common knowledge refers to knowledge that everybody knows in a particular knowledge domain. For example, every car driver knows that red means "stop" and green means "go". Distributed knowledge, on the other hand, can be considered as "*what a 'wise man' - one who has complete knowledge of what each member of the group knows - would know*"[FHMV95]. In the context of our approach, common knowledge is, for example, the fact that every sensor has to deliver its data in RDF format. Distributed knowledge is every piece of knowledge a particular sensor has. This knowledge is delivered to the reasoner, which deduces new information from it.

In context-aware applications, it is important to adapt to changes in the user's context. In order to identify different user context, reasoning techniques are used. Reasoning in computer science is concerned with the process of deducing higher level context from raw/preprocessed sensor data, which is also called lower level context. Higher level context is inferred by combining lower level context data. For example, lower level context are coordinates gained from a GPS sensor, whereas higher level context would be "at school" or "at a restaurant" [NF].

Reasoning is performed in different applications, using different techniques. The approaches that we discuss in Section 3.2 use Bayesian networks, which is a machine learning technique (cf. Section 2.3.1), rule-based reasoning (cf. Section 2.3.2), or case-based reasoning (cf. Section 2.3.3). In the following, these three different reasoning techniques are described.

2.3.1 Machine Learning

Machine learning is an artificial intelligence approach and an umbrella term for different reasoning techniques. It is concerned with algorithms, which are used to teach a computer/program certain circumstances. Usually a so-called training phase is used to teach the system the basic situations by means of a training data set. This training data set is mostly created by domain experts. With every new piece of information, the system improves its capability recognizing actual situations on its own. It can learn correlations between sensor data and human behaviour. There exist many different algorithms and techniques for machine learning. The most common ones are: Bayesian networks [BG07], Hidden Markov models [Ram07], conditional random fields [Wal04], decision trees [dec], neural networks [SS], and suffix trees[YDM11]. In the following, we further describe Bayesian networks because they are used in an approach [KMK+03] discussed in Section 3.2.

Bayesian Networks

Bayesian networks, which are also called "belief networks" are representatives of "probabilistic graphical models". In particular, they can be described as directed acyclic graphs, whereas nodes represent random variables and edges represent probabilistic dependencies among variables. This directed acyclic graph representation is called the "qualitative" part of the network. However, there is also a "quantitative" part, which is the conditional probability distribution that is often provided in tabular form [BG07].

In Figure 2.6, an example of a Bayesian network graph is depicted. In this example, the event *burglary* may be dependent on whether the windows of the house are open (event *windows open*) or not, respectively whether the residents of the house are on holiday or not (event *on holiday*). If there is a burglar in the house, an alarm may be triggered with a certain probability (event *alarm*). However, an alarm may also be triggered by the neighbours' cat (event *neighbours' cat*). If an alarm was triggered, usually the police gets informed and arrives a few minutes later (event *police*), except somebody realizes that it was a false alarm and quits it (event *quit alarm*). Every variable in a Bayesian network is binary, i.e. it is either true or false. Next to each node, the conditional probabilities are listed in tabular form.

Bayesian networks are, for example, used for classifying spam e-mails or other classification tasks. In the same way it may be used to classify user situations or associate data items with particular projects.

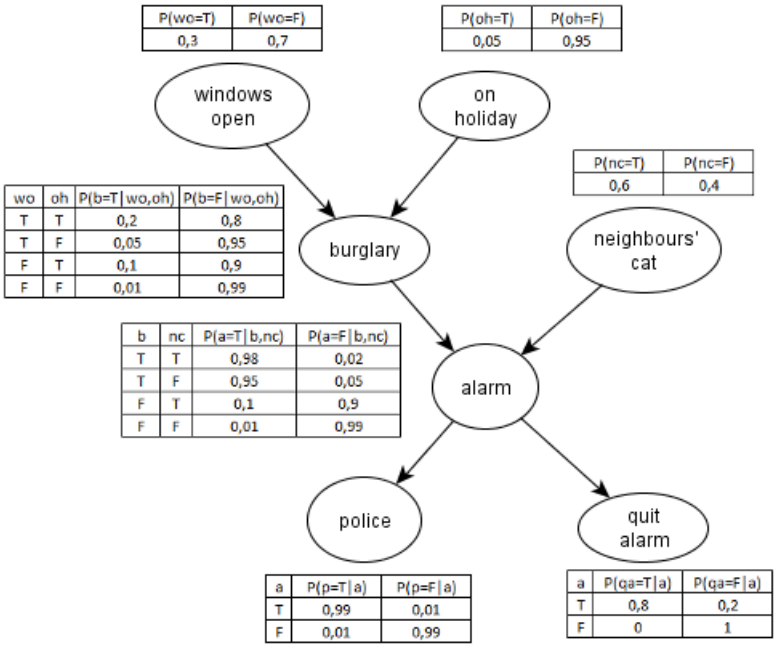

FIGURE 2.6: Bayesian Network example

2.3.2 Rule-based Reasoning

Another possible reasoning technique is rule-based reasoning. Besides the set of data we know about a domain, there is a rule base, which holds rules to be fired. Using these rules, an inference engine is able to derive new knowledge that may help to identify user situations. A rule generally consists of an antecedent part and a consequence part, which can be compared to an if-clause. The *antecedent* part may consist of a conjunction of conditions. The *consequence* part contains actions that are executed if the antecedent evaluates to true. In most cases, the consequence of a rule makes changes to the data set. It can add new information, delete information, or update certain pieces of data. However, consequences can moreover cause actions like printing messages, accessing files, or other things that do not affect the collected data [SRMR+07]. An example for a rule would be: **If** user is at work **and** there is a current appointment registered in the calendar **then** user is in a meeting. If we define that *user is at work* means their location is N 48 ° 25.08907 E 16 ° 38.08902 and assume that the current date and time is March 15, 2012 11 a.m., the mentioned rule can be formalized in Jena syntax as follows:

```
1  rule: (?location rdf:type pimo:Location)
2  (?location geo:long "48.418151")
3  (?location geo:lat "16.634817")
4  (?event rdf:type pimo:SocialEvent)
5  (?event pimo:dtstart ?startTime)
6  (?event pimo:dtend ?endTime)
7  le(?startTime, "2012-03-15T11:00:00")
8  ge(?endTime, "2012-03-15T11:00:00")
9  ->
10 (?user myns:inSituation myns:Meeting)
11
```

CODE 2.11: Example rule in Jena syntax

Code 4.1 shows an example rule, which can be used for rule-based reasoning. The rule evaluates to true, if there is a resource from type `pimo:Location`, which has a property `geo:long` with the value "48.418151" and a property `geo:lat` with the value "16.634817". Furthermore, there is a resource from type `pimo:SocialEvent`, which has the properties `pimo:dtstart` and `pimo:dtend`, whereas the start time must be less than or equal to (`le(...)`)) the current time (which is "2012-03-15T11:00:00" in this example) and the end time must be greater than or equal to (`ge(...)`)) the current time. If these conditions evaluate to true, we can infer that the user is in a situation *meeting* that is represented by `myns:Meeting`. `myns:Meeting` is a subclass of type `myns:Situation` and can be characterized by a time interval as well as a location.

Backward Chaining

Backward chaining is a technique, in order to satisfy a list of goals. The algorithm searches for rules, which match a desired goal. If the antecedent of the found rule is not true, it is added to the list of goals and the search is going on. It is continued until all goals are satisfied or a goal cannot be satisfied at all [CF]. Backward chaining is suitable for systems where the inferred amount of data would be too large to add them to the knowledge base and the inferred data is not used very often by the application. In order to overcome these issues, it is possible to either pre-compute and store the results, in case they are used frequently, but are not too large to store them efficiently; or—in case the resulting amount of data would be too large—store only a minimal amount, from which more complex results can be computed on demand [Hog11].

Forward Chaining

Forward chaining is a special technique of rule-based reasoning. The algorithm starts with the existing data and loops through all inference rules as long as it can find one

where the antecedent is true. Every time such a rule is found, the new information is inferred and the process is going on until no rule can infer new information any more [CF]. With this technique, implicit knowledge is made explicit and can be added to the knowledge base. The advantage of forward chaining is that it may be executed independently from the application [Hog11].

Forward chaining is more suitable for our approach because we are trying to identify situations by analysing existing context information, i.e., our initial point are low-level context data or already inferred high-level context data. Furthermore, it is less resource-intensive - so it can be executed every time a new context data is added to the knowledge base - and the inferred information can automatically be added to the knowledge base.

2.3.3 Case-based Reasoning

Case-based reasoning uses knowledge from previous situations in order to solve a current problem. The inference engine remembers situations, which happened in the past, and compares them to the current situation. It can use the experiences from the former ones to adapt to the current situation – like a human would do when remembering similar past situations. For example, Sarah is baking a cake from a recipe of a cookbook, which she has baked once before. The last time it burned because it was too long in the oven. Sarah remembers the situation and sets the alarm 10 minutes earlier. Mostly it is easier to solve a problem a second time because we remember the mistakes we made the first time and try to avoid them. A case-based inference engine can improve by getting feedback, which is used to notice mistakes, remember them and trying to avoid them next time. For example, the user is in a situation *meeting* and mutes their mobile phone. Next time the situation meeting is recognized the system automatically mutes the mobile phone. Additionally a notification is sent that asks them whether the action was adequate for the situation or not. There are two styles of case-based reasoning: problem solving and interpretive. The problem solving style uses already known situations as a guide in order to resolve new problems. They can provide possible solutions or warnings of possible mistakes for new situations. "*In the interpretive style, new situations are evaluated in the context of old situations.*" [Kol92] It can moreover be used to classify or evaluate solutions. This style is often used by lawyers, who "use a series of old cases to justify an argument in a new case". However, the two styles can also be mixed up and used both for the same situation [Kol92].

2.4 Context

There exist numerous definitions of the term *context* in many different domains. Probably one of the most cited is the following one proposed by Dey and Abowd [AD99]:

> "*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*" [AD99]

This definition is very well suited for the domain of computer science, as a generic one is not easy to give [ACG10]. However, the following characteristics[9] of context suit for every domain:

- context always relates to an entity
- is used to solve a problem
- depends on the domain of use and time and
- is evolutionary because it relates to a dynamic process in a dynamic environment

Furthermore, most context items are dynamic, i.e. they are changing frequently, like the location of the user in a mobile environment. Since context is often acquired by sensors, it may be imperfect [HIR02], e.g. when a sensor is not sending data because it is turned off or no connection can be established. A sensor may be disabled/unavailable or not accurate enough.

Context can be categorized according to its characteristics. One possibility is to differentiate between static and dynamic context. Static context is information that does not change at all, like a person's date of birth, whereas dynamic context changes over time. Another common classification is the differentiation between low-level and high-level context information. Usually low-level context is acquired by sensors and high-level context is derived from low-level context information. A third classification that is often applied uses three categories: direct/sensed, defined, and indirect context. Direct context refers to the context information that is sensed, defined context is explicitly gained by user input and indirect context is derived from direct context [SDCD09]. In this work, context is relevant for characterizing the current situation of users. Therefore, the low-level and high-level context categorization is used.

[9] these characteristics are directly quoted from [ACG10]

To achieve a common view of context in a specific domain, context models are created and formalized (cf. Section 2.4.1).

2.4.1 Context Models

A context model is the formalization of the acquired context in an environment. It is used to provide a common understanding of the context within a domain and should be designed to be reusable and extensible.

There exist many different approaches how to develop a context model, in literature. Three kinds of context model schemas that are used in related work (see Section 3.2) are described in the following. These are key-value models, object-role based models, and ontology-based models. An overview is given in [BBH+10] and [BCQ+07].

2.4.1.1 Key-Value Models

Key-value models are the most simple kind of storing context information, which was already used in early context modelling approaches. Context information is represented by a collection of key-value tuples, whereas the key is a unique identifier and values are usually simple data types. Though they are very simple to manage, there are some disadvantages. With key-value models it is not possible to relate context information to each other or to describe dependencies. There is no way to add timestamps or quality attributes to context information. Furthermore, reasoning on context is not possible [BBH+10]. These are only a few reasons why we do not use them for our approach.

2.4.1.2 Object-role based Models

This kind of context models "have their early roots in database modelling techniques" [BBH+10] and use the Context Modelling Language (CML), which is described in [HI06]. The name object-role-based models originates from the fact that CML is based on Object-Role Modelling (ORM) [HM08]. One characteristic of these models is that they not only support context representation and querying, but with its graphical notation also the design and analyses phase. Furthermore, they provide support for the management of imperfect and historical data. However, object-role based models are "flat" models, with which it is not possible to define hierarchical structures [BBH+10].

In [HIR03], a context model that is developed as an extension to the Object Role Model (ORM) approach is described. The core concepts of this model are *Entity Types* and *Fact Types*, whereas each fact type is attached to an entity type. They differentiate between

static and dynamic fact types, which are further divided into profiled, sensed, and derived types depending on whether they are supplied by users, sensors, or a derivation function. Besides the modelling of object properties, with this model it is also possible to capture histories, fact dependencies, and quality.

Another object-role based context model approach is described in [HIR02]. Its main concepts are entities and properties of entities, whereas properties can be associated with entities and entities can be associated with other entities. There are different types of associations to differentiate static and dynamic relationships between concepts. Dynamic associations are - like the fact types in [HIR03] - further divided into sensed, derived, and profiled associations. There are also notations for modelling dependencies and context quality. A dependency may exist between two associations and occurs whenever a change to one association may cause a change in another association. As mentioned in Section 2.4, context data may be imperfect, therefore a parameter of certainty/accuracy may be modelled [HIR02].

2.4.1.3 Ontology-based Models

As mentioned in Section 2.2, ontologies are used to formally describe knowledge domains, by defining class hierarchies and properties of entities, as well as instances of classes and the relations between them. In contrast to other context model schemas, with ontology description languages very complex domains can be described. Another strength of ontology-based models is the high degree of formalism, which facilitates the sharing of knowledge. This is particularly important for mobile applications because it is very likely that they exchange context information with servers or other mobile devices. Due to its axiomatically defined semantic, ontology-based description frameworks enable automatic reasoning, for example, checking for inconsistencies and the recognition of user activities [BBH+10].

Example ontology-based model schemas are CONON [WZGP04], CoDAMoS [PdBW+04], PIMO [SvED07], the NEPOMUK Ontologies [Con], and CC/PP [ccp07].

One of the most advanced ontology-based context models is described in [WZGP04]. It was developed for pervasive computing environments and used within a Service-Oriented Context-Aware Middleware (SOCAM) [WZGP04]. They provide an upper context ontology for general concepts and extensibility for adding domain specific ontologies. With the upper context ontology, information about location, user, activity, and computational entities can be represented. CONON is very flexible and can be used in different application domains. However, it is more suitable for the field of pervasive computing

than for applications that only run on mobile phones because in a standalone mobile application there are not that much different contexts that will pay off modelling different domains of context as a standalone model.

An often used ontology-based model for representing the personal mental model of a person is PIMO - a Personal Information Model Ontology. It is meant to formalize the personal mental model of a knowledge worker by using pre-defined concepts and relations. Every concept is a subclass of `pimo:Thing`. There are classes for "*all things and native resources that are in the attention of the user when doing knowledge work*" [SvED07]. It is a domain- and application-independent notation and may be reused for different applications [SvED07]. PIMO is one part of the NEPOMUK Ontologies and partially reused for our approach.

The team around NEPOMUK (Networked Environment for Personalized, Ontology-based Management of Unified Knowledge) developed the NEPOMUK Ontologies [Con] to formalize the information and data of knowledge workers. These consist of many sub-ontologies (cf. [Con]). The NEPOMUK Information Element set of ontologies contains the most relevant ontologies for our mobile Semantic Desktop approach.

CoDAMoS [PdBW+04] provides an ontology for representing many different parameters of mobile devices/users. Amongst others, there are concepts for environmental parameters like temperature, lighting, or noise level, as well as for representing aspects of location and time. This information may help to identify user situations like "sleeping" or "being in the cinema" and therefore, are reused in this work. Another aspect that is covered in [PdBW+04] and is useful for the objective of this work, is the concept that represents the user. Important properties are the user's profile, preferences, mood, and current activity.

Composite Capability/Preference Profiles (CC/PP) [ccp07] is an ontology based on RDF that was designed for representing user agent capabilities and user preferences. It supports servers in recognizing the most appropriate form of a resource to deliver to a client. A general CC/PP profile consists of components and attributes with its corresponding values. We reuse this structure and vocabulary for representing users' preferences in our context model (cf. Section 4.4).

Context models are important for the domain of context-aware and semantic applications in order to achieve a common model for all participating agents. The above examples show that context model schemas evolved and there are different ones for different purposes. A context model is mostly built for only one domain or application area. Within this domain, almost every information that is relevant may be modelled. For this work,

an ontology-based model is used. It is most suitable for a mobile Semantic Desktop because we can re-use existing ontologies, context-information can be easily shared across different agents and platforms, and it best supports reasoning, which is an integral part of our approach.

2.5 Context Awareness

Context awareness in the domain of computer science, was first introduced by Schilit and Theimer [ST94]. They define a context-aware system as a system that is able to acquire information about its environment (by sensors) and react to changes in it. However, there are numerous other definitions, which may be differentiated in two categories, those who refer to the usage of context and those who refer to the adaptation to context. In [AD99] they try to find a definition that includes both categories. They define context awareness as follows: "*A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.*" We think that this is a good combination of both. Furthermore, they define three different features, which may be supported by a context-aware system. These are:

- "presentation of information and services to a user"
- "automatic execution of a service"; and
- "tagging of context to information for later retrieval" [AD99].

In our approach all three of them are supported. Depending on the user's current context, different information and/or services may be presented to the user. For example, the user has arranged a meeting in half an hour and it takes 20 minutes to go there. Then an application extending the system may suggest the user to set off and additionally displays the route information. A service may be automatically executed, whenever an extending application detects that the user is driving a car. Then the hands-free speakerphone is automatically turned on. Often, information is only added to the context model and may be used in another situation. For example, an e-mail according to the current project arrives and is added to the context model as well as linked to the corresponding project. If the user is working on this project later, an application implementing this functionality realizes this, and the information may be displayed.

2.6 Personal Information Management

Personal information on a mobile device are contacts, files like music, photos, and videos as well as e-mails, SMS, and calendar entries. The main tasks of *Personal Information Management* (PIM) are acquiring, maintaining, organizing, storing, retrieving, and utilizing information of our everyday life. There are various tools that facilitate these tasks and hence, there is more time for using the information instead of managing it [Jon07]. "*In an ideal world we have the right information at the right time, in the right place, in the right form, and of sufficient completeness and quality to perform the current activity.*" [Jon07] Information is not always there when we need it. Personal information management is concerned with organizing information in such a way that we are able to recover it, when we need it because information is only useful when we are able to retrieve it in particular situations. Furthermore, information has to be protected against people that are not authorized to have access to it [Jon07].

One strategy of personal information management uses mind maps. They consist of nodes and edges, whereas nodes represent pieces of information and edges connect them with each other. This is similar to how the human mind works as well. Humans usually classify things. There are different categories and new information is classified into existing categories. Hierarchical file systems use classification as well. People create folders, which represent categories and files can be associated with them. The same can be done within an e-mail software. However, file system and e-mail software do not know anything about each other [Sau03]. For example, let us assume somebody is organizing a prom. In the file system on their computer, they have stored files in a folder, like the invitation or photos from the preparatory work. Within the e-mail software, the correspondence related to the event and the corresponding contacts are organized. Furthermore, there is an event homepage with a bulletin board that has to be maintained. The link to the homepage is stored as a bookmark within the browser. The organizer of the event can classify its information items in all of the three used applications, but nowadays it is not possible to relate items of different applications with each other. Moreover, the applications do not understand the concepts behind the classification. In this work, we try to find a remedy for these issues.

One approach of personal information management that uses metadata is *tagging*. Tagging is for example used by photo sharing websites. Users can add keywords to photos, which they think describe the photo best. These keywords are called tags and are stored related to the photo. When a user is searching for a certain keyword, the search engine finds all photos that are tagged with this word.

Apple has implemented Spotlight[10] for Mac OS X[11] and iOS[12] in order to support the user's personal information management process. Spotlight uses metadata of data items to improve its search results. With this search engine you can search your hard disk for simple text files, e-mails, contacts, audio and video files, as well as application files or system preferences at the same time.

At the moment the trend is going towards cloud storages. A cloud storage is a virtual storage pool that can be distributed over different servers, where different items of data can be stored. That means, it is possible to store contacts and e-mails in the same folder as conventional files. The storage is hosted by third-party companies and users can buy certain amounts of storage. This idea enables users to access their data from everywhere in the world over a web interface. There are already approaches, which try to enhance cloud storages with semantics, like the one described in [HMS+10]. Using Semantic Web technologies in order to improve cloud services, may lead to "intelligent personal information management" [HMS+10].

In this work we will try to facilitate personal information management on mobile devices by using Semantic Web technologies (cf. Section 2.1). Some approaches that already deal with this topic, are discussed in Section 3.2.

2.7 Semantic Desktop

Semantic Desktop approaches apply Semantic Web technologies to the desktop computer, which make it possible to manage different kinds of user's data in a uniform way. The computer is then able to assist the user with information management. Therefore, every piece of data on the computer must be identified using a URI. The most important technologies in order to achieve this are RDF (cf. Section 2.1.1) and the ontology (cf. Section 2.2). The ontology can be compared to the user's personal mental model of a project or context [Sau06]. For example, if a user is working on a homepage for their enterprise, there are a few people that help them with their task; there are documents, which they have already created; photos that will be on the homepage; and e-mail conversations with the team members. In the mental model of the user, all these things are somehow linked to each other, but on the computer all these things are nowadays managed separately. The e-mails are organized in an e-mail client, files and photos in the file system, and contacts maybe also in the e-mail client or in a standalone address book. However, there is no possibility to associate the different data items with

[10] cf. `https://developer.apple.com/library/mac/#documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html`

[11] cf. `http://www.apple.com/osx/`

[12] cf. `http://www.apple.com/ios/`

each other on conventional computers. Semantic Desktop approaches use the concepts of metadata and annotations in order to enhance/automate personal information management. An overall ontology is used from all applications and (new) data items are integrated automatically and related to existing ones [Sau06]. "*The Semantic Desktop is an enlarged supplement to the user's memory.*" [SBD05] According to [Sau06], an ideal Semantic Desktop is not a "conventional" computer where semantic technologies are installed to achieve a semantic linkage, but a computer, where these technologies are already integrated in the operating system and all applications are based on it.

There are two different approaches to achieve a Semantic Desktop. The goal of the *monolithic approach* is to implement as much functionality as possible into one application. The whole system uses the same ontology. An example for this approach is Haystack [QHK03], where all relevant tasks of a knowledge worker are realized within one graphical user interface. When the user clicks on a piece of data with the right mouse button, they can see every available method that can be applied to this object. On the other hand, the *integrative approach* tries to extend existing desktop applications and implements their functionality. An example is Gnowsis [Sau05a], which is described in Section 3.2. Its aim is to link information from different applications in one overall RDF representation and make minimal changes to existing user interfaces in order to integrate the two core functionalities, which are used to link resources with each other and display related resources. The advantage of this approach is that the user can work with its familiar programs and the RDF representation is provided additionally [Sau06].

Working on a Semantic Desktop means that the computer is able to identify the user's situation and reacts appropriately. For example, the system notices that the user is preparing for an upcoming meeting, hence it suggests files, e-mail conversations, and contacts in order to facilitate the preparing process and relieve the user from searching for these data items.

Independent of the used approach, every Semantic Desktop system consists of the following elements: The bottom layer is built up of the data sources, which have to be converted into an RDF format by an adaptor and can afterwards be stored, for example, in an RDF database. On the next level there is the formal representation of things with which the user is concerned by means of ontology, including the relations between them. This is the formalization of the personal mental model of the user. The last component is the Personal Semantic Web Server for the computer [Sau06]. For the query of data, SPARQL is used within many projects. Semantic applications should always interact with the user's personal ontology and data sources.

In this work, we like to create a mobile Semantic Desktop infrastructure. Therefore, the concepts of a conventional Semantic Desktop will be transferred to the mobile phone.

However, on mobile phones we are faced with different preconditions. In a mobile environment the user's context is way more important than on desktop computers because it is changing faster and is more differentiated. Furthermore, the importance of different data items is not the same. On desktop computers we may have much more files to manage, whereas on mobile phones contacts, e-mails, and calendar entries might be in the foreground. Obviously, there are also differences in the technical details. Mobile devices usually have less computing power and storage. Moreover, they do not always have a connection to the internet, but nevertheless it is important that users have access to its data from everywhere.

2.8 Summary

In this chapter, technologies and concepts that influence this work, have been introduced. For our approach, the notion of the Semantic Web serves as a basic principle and its transfer to the desktop as Semantic Desktop is used and adopted for mobile phones. RDF, RDFS, and OWL provide syntaxes for a standardized formalization of the designed ontology and the representation and description of data. They have already proved to be useful for these purposes in the Semantic Web and in other related works. In order to uniquely identify resources, we use URIs. SPARQL provides a syntax for querying RDF data and retrieving subgraphs that can be shared with applications which use our infrastructure. Furthermore, the analysis of different reasoning types depicted their advantages and disadvantages, as well as their possible application areas. For this work, we decided to use rule-based reasoning and forward chaining because we deal with facts rather than so-called cases, like they are used in case-based reasoning. Furthermore, rule-based reasoning does not need any training phase or domain experts compared to machine learning. As context is relevant for our approach in order to characterize user's situations, we described its notion. Existing types of context model schemas have been discussed and the ontology-based context model schema turned out to be the most suitable for our approach because existing ontology models can be reused, context-information can easily be shared across different agents and platforms and it best supports reasoning. Moreover, context awareness as an important feature of this work has been introduced. One part of our goal is to facilitate personal information management on mobile devices by using Semantic Web technologies and context information. Finally, we introduced the idea of Semantic Desktops and depicted inherent peculiarities and limitations of mobile devices.

Chapter 3

State of the Art

In this section, existing context-aware middleware approaches, infrastructures, and applications are reviewed and compared, according to some selected qualities. In Section 3.1, the qualities are described and afterwards their implementation in a subset of similar approaches is discussed (Section 3.2). Table 3.1 and 3.2 show how the requirements are implemented in the different approaches. In Section 3.2.1 and Section 3.3 the outcome of the analysis is discussed and summarized.

3.1 Requirements

The following aspects have been selected for the analysis because they characterize a mobile Semantic Desktop. There are some that concern the technical implementation, like *storage type*, *context model*, and *application type* and others that are important with regards to content, such as *provision of context history*, *reasoning mechanism*, and *support for personal information management*.

- **External context sources** Data acquired from external context sources are an important requirement for a context-aware mobile Semantic Desktop. It should be possible to acquire contextual data not only from built-in sensors of the device, but also from remote (web) services or sensors installed for example in buildings. They augment the available data either with further information to improve reasoning/inference or serve the application to maybe have a second information source of the same piece of data. It can furthermore be helpful if built-in sensors fail.
- **Provision of context history** As context history we name contextual data that is out of date, i.e. it was up to date in the past. These data can be used to enhance

reasoning/inference mechanisms. It helps to adapt to certain situations adequately by comparing the current situation with similar situations that happened in the past. For example, the system recognized that the user muted their mobile phone, when they were in a meeting, so next time the user is in a meeting the system mutes their mobile device automatically. One disadvantage of using context history is the increased storage demand. There has to be a strategy to classify information according to the time it should be kept. For example, timestamps may be used to evaluate the validity of certain pieces of context. Another possibility is to define a time frame that indicates the validity of data items.

- **Subscription/poll mechanism** The used mechanism for applications to get data respectively information about context changes is a relevant design decision for context-aware middlewares/infrastructures. Using the subscription method enables applications to subscribe for context change events and to get informed from the system every time this specific context has changed, e.g. the application wants to get informed every time the location of the user changes, hence it subscribes to the location change event. These can either be predefined change events or user defined SPARQL queries. On the other hand, the poll mechanism provides an interface for applications to poll the system, for example, in a fixed interval of time, whether a specific context has changed or not. While a poll mechanism is the synchronous way of getting informed about changes, the subscription mechanism is mostly implemented asynchronously. In mobile context-aware environments, an asynchronous mechanism may be best suited because of the permanently changing user context. It would be too resource intensive using the polling mechanism [BDR07].

- **Application type** The reviewed approaches are implemented as different kinds of applications. We differentiate between middleware, mobile middleware, context server approaches, mobile applications, Semantic Desktops, and mobile Semantic Desktop applications. According to [BDR07], we classify an approach as a *middleware* when it infers an abstraction layer between sensors and context management, it encapsulates the process of data acquisition. A *context server* approach [BDR07] extends the middleware approach with a central remote server that acts as the only access point for clients. Some architectures do not use any remote components, i.e. the whole system is implemented on the mobile phone, and have a predefined and completed scope of operation; they are called *mobile applications*. Systems, which are entirely located on a mobile device and act as a middleware, are hence classified as *mobile middlewares*. *Semantic Desktop* approaches are the ones, which integrate personal information management within their system. Furthermore, there may be a Semantic Desktop application that is entirely located on a mobile device and is therefore classified as a *mobile Semantic Desktop application*.

- **Existence of servers** Our goal is to develop a mobile context-aware infrastructure that does not need any servers at all, neither a context server nor any other server. A server has the advantage that it may relieve the mobile device from storing and/or processing data, but on the other hand, a connection to it has to be established in order to exchange and synchronize data. However, we believe that nowadays mobile phones should be able to manage these tasks on their own.

- **Storage type** Most analysed systems use a persistent storage for saving and retrieving data whenever it is needed. For mobile devices it is best to use an efficient and lightweight storage. Therefore, we also compare the different storages that are used for the selected approaches. Most of them prefer database systems as a storage type. However, some use the N-triple serialization of RDF triples or do not use any storage at all. We think that a lightweight database system, like SQLite, would be best for mobile environments.

- **Context model** In existing approaches different kinds of context models are used. Key-value tuples, object-oriented, and ontology-based are only some of the possibilities [BBH+10]. However, most of them use an ontology-based context model. According to [SLP04], an ontology-based context model is best suited for modelling context in ubiquitous computing scenarios. In order to arrive at this conclusion, they analysed requirements such as *richness and quality of information*, *level of formality*, and *applicability to existing environments*. Ontologies enable context reasoning [BBH+10] and are also used by Semantic Web data sources. So, if an ontology-based context model is used, Semantic Web data sources can be integrated applying an appropriate mapping into a context-aware mobile Semantic Desktop. Furthermore, ontologies use a defined schema for representing data, hence we consider them the most suitable choice for a context-aware mobile Semantic Desktop.

- **Reasoning/inference mechanism** Reasoning is used to derive higher level context from low-level context information collected from sensors (cf. Section 2.3). It is very useful for context-aware systems in order to augment the data by gaining additional information that cannot be sensed and to facilitate user situation identification. For example, a reasoner may deduce whether the user is at home or at work by analysing the current geographic coordinates of the mobile device. The choice of an appropriate reasoning/inference mechanism is an important decision in the design process of a context-aware system. Therefore, we also compare the existing systems according to the used reasoning algorithm and if reasoning on context data level is supported at all.

- **Support for personal information management (PIM)** Semantic Desktop applications focus on personal information management, hence another criteria for

the comparison of existing systems is whether they facilitate personal information management or not. The support of PIM helps the user with organizing their data and hence is saving time (cf. Section 2.6). Data units associated with PIM are, amongst others, contacts, files, e-mails, and calendar entries. Systems supporting PIM, annotate personal data items from users semantically and associate them with each other and with contextual information, respectively group them for example according to different projects.

- **Context types** As a context type we understand a specific type of information that is relevant to describe the user's context, like temperature, location, or time, and can be collected by a sensor. For a context-aware mobile Semantic Desktop, we believe it is important to integrate preferably many context types that help to identify a user's situation and support the user in organizing their data. To make the system generic it should be possible to easily extend the number of context types that are integrated. In the following, we analyse existing systems according to the used context types and extensibility.

3.2 Related Work

In [dFRRR04] "*a modular service infrastructure to support context-awareness in nomadic computing*" [dFRRR04] is described. The system was designed to enable context awareness on mobile devices. The architecture relies on the factory design pattern and it consists of extensible and reusable components. There are different context providers that provide different pieces of context, which can either be low-level or high-level context features. Such context sources can be sensors located on the device, user-dependent features, external sensors, or external services. The different data collected from sensors can be aggregated in order to get higher level context data. Each application that is built upon this middleware only instantiates the context providers it requires. The number and type of context providers is not restricted and because of its modular design, a high degree of extensibility is achieved. An event-based notification mechanism informs the application about context changes. The whole middleware is implemented on the mobile device itself, except external sensors. However, there does not exist a formalized context model, a persistent storage, and reasoning techniques, so far. Hence, no historical data can be provided.

The Contextor Infrastructure [RC04] implements the ontology presented in [CCRR02]. It consists of the following levels of abstraction. The first layer—the *Sensing layer*—encapsulates a collection of physical sensors, which generate numeric values for so-called *observables*. The types of context sources are not outlined for this approach. The

next layer, which is the *Transformation layer*, performs transformations on the numeric values and provides *symbolic observables*, which are enriched with meaning, in contrast to numeric observables. The Transformation layer is independent of the Sensing layer. On the third layer—the *Situation & Context Identification layer*—reasoning and inference (cf. Section 2.3) is performed in order to deduce the current situation of the user. The so-called *Exploitation layer* is the topmost layer and communicates with applications built on top of the middleware. Orthogonal to these layers, functions that support privacy, trust, and security as well as history management, and discovery/recovery of services are implemented for each layer. It is not mentioned how or whether a storage is used for this approach. The Contextor Infrastructure provides a subscribe-notify method, as well as a so-called query-answer method for applications in order to get informed about context changes. It is no external server used for this approach.

The Context Toolkit [SDA99] is a middleware for developing reusable context-aware applications and was inspired by the concept of GUI widgets. It uses so-called context widgets that provide context data to applications, while hiding details about the context acquisition process and the actual sensors from them. Context widgets notify registered applications about changes in context. Additionally, a polling mechanism is implemented, which allows applications to query context. Each widget provides only one particular piece of context data. However, they can be composed to provide higher level context. Composed widgets rely on interpreters, which deduce higher level context from the different context widgets. Besides widgets and interpreters, there is a third component that is important for the system. These are the so-called *generators*, which are responsible for acquiring context information. There is no restriction on the type of context a widget is able to provide. Two differences between context widgets and GUI widgets are that context widgets "*are active all the time*" and that they "*live in a distributed architecture*" [SDA99]. Context is stored as attribute-value tuples in a database, where also historical data is saved.

[MKP05] describes an approach for a domain-independent context middleware with the aim to be generic, user-friendly, and to facilitate context-aware computing in mobile environments. The presented context model includes data about the following context types: task, social, personal, spatio-temporal, and environmental context. The introduced framework consists of a mobile device, a *Context Tag*, which is a small Bluetooth-enabled communication unit, and net-based information services. A Context Tag provides a mobile user with information from a net-based information service. On the mobile device reasoning is performed. Every particular piece of context is represented by an attribute-value tuple and stored in a so-called context space, where not only the current context, but also context history is stored. The middleware itself provides context data to applications. They can subscribe for notifications that inform about

different kinds of changes in context. As a reasoning technique, case-based reasoning (Section 2.3.3) is used as a lightweight technique that is suitable for mobile devices. Because of the limited storage capacity on mobile devices, a two-tier reasoning mechanism is used, which consists of an on-line and an off-line part. The cases are stored "off-line" on the user's home network and the data is synchronized whenever the user is connected with the network.

In [KMK+03] a context management software framework—designed for the Symbian platform—that should facilitate the development of context-aware applications is described. They use a blackboard-based approach [CD91] and an expandable context ontology. The framework is composed of four components, which are context manager, resource server, context recognition service, and application, whereas all components can be located on the mobile device itself. However, the services may be distributed. Context data is stored in a database on the context manager building block, which is situated in the centre of the architecture. The resource servers collect context data from various sensors/sources and send it to the context manager, which forwards it to the applications that have subscribed for notification. Additionally to the subscription mechanism, there is also a polling mechanism implemented for context updates. They are able to use data about the sound level, acceleration, light, temperature, humidity, touch, running applications, internet, GPS, internal device processes, and time. As a context model an ontology described by means of RDF is used. The approach, furthermore, provides context history data. A naïve Bayes classifier is employed for reasoning and can be used as a context recognition service.

SOCAM [GKPZHW04] is an abbreviation for Service-Oriented Context-Aware Middleware and facilitates the development of context-aware mobile services. The architecture uses a two-level ontology-based context model that enables reasoning and consists of an upper-level ontology and a low-level ontology for specific domains. The upper-level ontology includes data about computational entities, personal data, as well as information about the user's current location, and activity. The low-level ontology is dependent on the type of application. The middleware's components are all designed as independent services. There are internal/external *Context Providers* that acquire context from various (virtual) sensors and represent them by means of OWL (cf. Section 2.2.1). *Context Interpreters* provide higher level context data after interpreting low-level data, by using built-in RDFS (Section 2.1.4), OWL, and rule-based reasoners (Section 2.3.2). Furthermore, a *Service Locating Service* is used for registering and locating context providers. For the persistent storage of context data a relational database is used. *Context-aware Mobile Services*, which are applications or services that make use of the middleware, can either poll for context information or subscribe for context changes to context providers. All components can be distributed over different networks and interact with each other.

Context-awareness sub-structure (CASS) [FC06] is a server-based middleware, which is designed in order to support the development of context-aware applications for mobile devices. The middleware consists of so-called sensor nodes with one or more sensors attached. They may be mobile or static. The type of the used sensors and other context sources is not restricted. On the server there are a database, which is used to persistently store data, and artificial intelligence components. To avoid having to be always connected with the server, applications can buffer data received from the server. Applications built on the middleware are not connected with sensor nodes, but only with the middleware. The main classes that build the middleware are the *RuleEngine* class, the *SensorListener* class, the *ContextRetriever* class, and the *ChangeListener* class, as well as the *Sensor* and *LocationFinder* classes. However, only the last three mentioned classes have communication capabilities. The ChangeListener is used by mobile devices in order to listen for context change events. Furthermore, CASS provides context history, which is stored in a database and employs rule-based reasoning (cf. Section 2.3.2) using the forward chaining technique (cf. Section 2.3.2). As a context model a relational data model is used.

The Hydrogen Approach [HSP+03] describes an extensible, three-layered architecture for supporting context awareness on mobile devices. Sensors and adaptors are reusable and exchangeable within this approach. Currently, time, location, device, user, and network are implemented context types. Pieces of context are organized as object classes. On the lowest level of the architecture, there is the *Adaptor Layer*, which is in charge of acquiring data from sensors and delivering the gained information to the *Management Layer*. On this layer, the *ContextServer* is located, which stores all contextual data and provides it to applications. Furthermore, mechanisms for retrieving and subscribing for context are implemented on this layer. The ContextServer is able to communicate with other devices in range and share contexts with them. There are two different mechanisms for applications to get information from the ContextServer: they can either query or subscribe for a specific context. It is possible that multiple applications have access to one specific context information at the same time. All layers are implemented on one device, which makes a network connection not necessary. One aspect that might be a drawback for some kinds of applications is that no context history is stored in this approach. Pieces of context are organized as object classes.

Gnowsis [Sau03] is an open-source implementation of the so-called Semantic Desktop (cf. Section 2.7) accomplished by the DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz GmbH[1]) - where techniques of the Semantic Web are applied to desktop scenarios. It should integrate all desktop applications and represents semantic by means of ontologies. In [Sau03] the architecture is compared to a tree, where so-called *Adapters*

[1] cf. `http://www.dfki.de`

represent the roots, the central server is the trunk, and the branches are several interfaces, including the user interface, called *GnoGno*. An *Adapter* fetches data from different sources and generates an RDF representation of them. All resulting graphs are centralized in the *Adapter Framework*. The types of context that are acquired are not defined in [Sau03]. The gnowsis server is the central part of the system, where all users' data is managed. It is responsible for the initialization of the other components, the configuration, and has access to the file-based RDF repository through the *RequestHandler*. The GnoGno user interface can be used by other applications and uses all data from the underlying components. For deducing what the user is currently working on, they use a technique, which analyses the resources the user is currently accessing, in order to derive what is on the user's mind [Sau03]. Applications can notify the *UserContext*, whenever a resource is accessed. Gnowsis has been continued as a commercial project called Refinder[2].

SeMoDesk [WW08] is a mobile Semantic Desktop implementation running on Microsoft Windows Mobile 5 operating system. It is a system for personal information management that comes with a graphical user interface, which is optimized for PDAs. Users have the opportunity to organize their personal data and define relationships between them using the graphical user interface. Additionally, the application is able to create resources and relations between them such as linking incoming phone calls and contacts on its own. The used data sources are contacts, phone classes, tasks, appointments, messages, web links, and files. Additionally, information about date, time, and length of time-based entities are represented in the context model. The graphical user interface provides possibilities to see the hierarchical structure of the resources in various display formats. As a data model, an ontology based on PIMO [SvEM09] is used and serialized in an SQL database. Although they are following the RDF data model (cf. Section 2.1.1), they do not use URIs (cf. Section 2.1.2) as identifiers for resources but UIDs, which are generated numbers implying the current time, as well as a counter in order to differentiate between IDs at the same point in time. For this approach, no external server and sensors are used. It is realized as a standalone mobile application and therefore no subscription/polling mechanism for context updates is required.

MobiSem [ZS12] describes a *context-sensitive Semantic Web framework for mobile devices*" [ZS12]. It enables mobile devices to process contextual data without using an external server, as well as performing personal information management by means of context. Context-relevant data can be acquired from different sources, like (external) hardware and software sensors, and Web 2.0 APIs. There is no restriction on the type of context. Data from external Semantic Web resources can be used in order to augment the local data with useful additional information. The central component of the

[2] cf. `http://www.gnowsis.com`

architecture is the *Context Dispatcher*, which aggregates context models provided from *Context Providers* and fulfills additional tasks, like reasoning and consolidation. In the *Context Description Queue* not only the current context, but also context information from the past, is buffered. Clients can either subscribe for context change events or poll for context changes in a regular time interval. As a context model representation they use a scalable and lightweight ontology. Furthermore, a rule-based forward chaining reasoner is used in order to get a coherent context model. As a persistent storage a local SQLite database is used. An Android content provider realizes the provision of local data for other mobile applications.

3.2.1 Discussion

Although these approaches are similar to this work, there are still a few differences. Many of them use central servers for the realization of their systems. However, we aim at getting along without servers because mobile devices do not always have the ability to be connected to a remote server. Approaches that do not rely on external servers are the modular service infrastructure [dFRRR04], the Contextor Infrastructure [RC04], the Hydrogen approach [HSP+03], SeMoDesk [WW08], and MobiSem [ZS12]. In [dFRRR04], they do not use external servers, but they do not have a formalized ontology model nor use context history. Most of the approaches, furthermore, do not allow for personal information management, which is a very important part of this work. Only Gnowsis [Sau05a], SeMoDesk [WW08], and MobiSem [ZS12] support personal information management within their systems. [WW08] is one of the systems, which fulfills both, no external servers and support for personal information management, but it uses a GUI and does not include external sensors for considering environmental conditions. A very similar approach to the one described in this work is discussed in [HSP+03]. Though, there are slight differences in the architecture of the infrastructure. Hydrogen [HSP+03] uses an object-oriented context model and includes peer-to-peer communication with mobile devices from other users. Furthermore, it does not use context history data and personal information management concepts. Gnowsis [Sau05a] is, besides SeMoDesk [WW08], the only real Semantic Desktop application. However, it is developed for personal computers and not for mobile devices and again uses a central server. Another very similar approach is described in [ZS12]. However, they focus more on the replication of structured data from the Semantic Web to mobile devices in order to adapt to the current user's information need, than on managing personal information units in consideration of the user's context. Reasoning techniques are used in most of the approaches. Only the modular service infrastructure and SeMoDesk do not use any. In the work about Hydrogen, there is nothing mentioned concerning reasoning. In [Sau05a],

	modular service infrastructure [dFRRR04]	**Contextor** [RC04]	**Context Toolkit** [SDA99]	**AmbiSense** [MKP05]	**Context Mgmt SW Framework** [KMK+03]
external sensors	yes	yes	yes	yes	yes
context history	no	yes	yes	yes	yes
subscription/poll mechanism	subscription	both	both	subscription	both
application type	middleware	middleware	middleware	middleware	middleware/mobile middleware
existence of external servers	no	no	yes	yes	not necessary
storage	none	?	database	context space	database
context model	not formalized	ontology	attribute-value tuples	attribute-value tuples	ontology
context types/data sources	depends on configuration, extensible	not outlined	not outlined	environmental, personal, social, task, and spatio-temporal context	sound, acceleration, light, temperature, humidity, touch, running applications, internet, GPS, internal device processes, time
reasoning technique	none	yes, technique not mentioned	yes, technique not explicitly mentioned	case-based reasoning	Bayesian
personal information management	no	no	no	no	no

Table 3.1: Comparison of existing approaches

	SOCAM [GWPZ04]	**CASS** [FC06]	**Hydrogen** [HSP+03]	**Gnowsis** [Sau05a]	**SeMoDesk** [WW08]	**MobiSem** [ZS12]
external sensors	yes	yes	yes	no	no	yes
context history	?	yes	no	yes	yes	yes
subscription/poll mechanism	both	subscription	both	?	no	both
application type	middleware	context server	mobile middleware	semantic desktop/context server	mobile semantic desktop application	mobile middleware
existence of external servers	yes	yes	no	yes	no	no
storage	database	database	yes	RDF/XML serialization stored in text files	database	database
context model	ontology	relational data model	object-oriented	ontology	ontology	ontology
context types/ data sources	computational entity, person, location, activity; extensible	not outlined	time, location, device, user, network; extensible	not outlined	contacts, phone class, tasks, appointments, messages, web links, files	extensible; depends on application scenario
reasoning technique	RDFS + OWL reasoned, rule-based reasoner	rule-based, forward chaining	?	yes, technique not mentioned	none	rule-based, forward chaining
personal information management	no	no	no	yes	yes	yes

Table 3.2: Comparison of existing approaches

[RC04], and [SDA99] reasoning is used, but the technique is not outlined. The others apply case-based reasoning, e.g. in [MKP05], rule-based reasoning, like in [GWPZ04], [FC06] and [ZS12], as well as Bayesian techniques in [KMK+03]. As a storage type most approaches use databases. Exceptions are Gnowsis [Sau05a], which uses RDF/XML serialized context data stored in text files and AmbiSense [MKP05] that uses a context space. The work about the Contextor Infrastructure is the only one that does not mention whether they use a persistent storage or not. Altogether, we have classified five approaches as middlewares, these are the modular service infrastructure [dFRRR04], the Contextor Infrastructure [RC04], the Context Toolkit [SDA99], AmbiSense [MKP05], and SOCAM [GWPZ04]. The Context Management Software Framework [KMK+03] can be considered as either middleware or mobile middleware. Other mobile middlewares are Hydrogen [HSP+03] and MobiSem [ZS12]. Furthermore, there are two approaches classified as context server approaches, which are CASS [FC06] and Gnowsis [Sau05a], whereas Gnowsis may also be considered as a Semantic Desktop application. The only mobile Semantic Desktop application is SeMoDesk [WW08] for Windows mobile phones.

3.3 Summary

In this chapter, we first discussed different analysis and classification criteria for related work (Section 3.1). We analysed the selected approaches according to these criteria and provide an overview in Table 3.1 and Table 3.2. Furthermore, a discussion regarding the differences between this work and related works is given in Section 3.2.1.

Most of the analysed approaches use external context sources. Since mobile phones have a limited number of built-in sensors, the integration of external context sources augments the contextual data with valuable additional information. The provision of context history may facilitate the situation identification process by using knowledge from the past. It is supported by most of the reviewed approaches. Applications extending a context-aware system can be informed about changes in context by either a subscription or polling mechanism. Most of the analysed works implement both of them. Only few approaches are standalone mobile systems that are not reliant on external servers/components. However, nowadays mobile phones have the required capacities to run a standalone situation-aware Semantic Desktop application. Databases are the most used storage type in related works because they are lightweight and can store every type of data. Ontology-based context models turned out to be best suited for a mobile Semantic Desktop application because they provide a high degree of formality and support reasoning best. This kind of model is foremost used by approaches that are most similar to our approach, like Gnowsis [Sau05a], SeMoDesk [WW08], and MobiSem

[ZS12]. Reasoning as a mechanism that makes implicit knowledge explicit and hence adds additional information to the knowledge base is used in most of the analysed approaches. Personal information management is only supported by three of the related works and is very important for Semantic Desktop applications. Most of the approaches do not limit the number of context types, but made their approaches extensible. Only SeMoDesk uses a limited and predefined number of context types.

The analysis of existing approaches showed that there does not exist an approach that meets all of our requirements. However, there are some systems, which meet most of them, like MobiSem [ZS12] or the Hydrogen approach [HSP+03].

Chapter 4

Approach

In this chapter, first the requirements and design considerations affecting our approach are outlined (cf. Section 4.1). In Section 4.2 two scenarios that have to be able to accomplish with the proposed system are described. On the basis of these two examples the whole system will be exemplified at the end of each section. In Section 4.3 the approach for the proposed system is described by conceptually dividing it into layers. Section 4.3.1 provides a more detailed description of the key functionalities acquisition, aggregation and consolidation as well as interlinkage, provision of context history, subscription and notification, and privacy. We use an ontology-based context model for representing contextual information and specify its schema in Section 4.4. In Section 4.5, the architecture of the proposed system is depicted and its individual components are described. Section 4.6 provides a graphical overview of the interactions between system and application for the use cases introduced in Section 4.2. A summary of this chapter is given in Section 4.7.

4.1 Requirements and Design Considerations

The requirements outlined in Chapter 3 serve as a basis for our approach, since they also affect our approach. Each bullet of this section outlines a specific requirement for this approach together with a corresponding design consideration.

- **Provision of context history** Although it causes an increased demand of storage, we decided to use context history for our system in order to facilitate situation identification. Therefore, every piece of contextual data is stored together with a timestamp for a certain period of time until it gets out-dated. A more detailed description about

the used method is given in Section 4.3.1. These data facilitates the process of reasoning and situation identification by using information from the past in order to come to an adequate decision for current situations. By means of historical data, third party applications can analyze such data to derive new reasoning rules according to previous behaviour of the user. For example, when the user is *at work*, they always want to have their mobile phone muted. Details regarding considered methods are given in Section 4.3.1.

- **Subscription mechanism** Applications that extend the mobile Semantic Desktop infrastructure need to be informed about changes in context in order to adapt their behaviour. Therefore, a mechanism that proactively notifies external applications about changes on data level is implemented. This functionality, for example, enables applications to change their behaviour according to the user's situation or synchronize their data with data in a Semantic Desktop. Applications should have the ability to individually specify when they would like to be notified.

- **Application type** Our system is, according to our classification scheme in Section 3.1, classified as a *mobile Semantic Desktop* application. This means, it is designed to be runnable on mobile phones without any dependency to remote components or remote infrastructures. Furthermore, this classification has another inherent requirement, which is the aspect of *Support for personal information management*, described later in this Section. In order to meet these specifications, the system must be capable of storing all relevant data on the mobile phone, and the execution of functionalities has to be optimized for the computing power of a mobile device.

- **Context and context representation** For the proposed system it is required to explicitly represent contextual information using ontologies (cf. Section 2.2). For each type of context that can be processed by the system, a specific context model schema has to be defined in order to ensure the proper operation of the system. The contextual data is represented independently of the function logic. Ontologies can be reused within one domain. Hence, it is not necessary to build up new ontologies for each type of information that should be represented. We are reusing parts of already existing ontologies for our context model. By adopting for instance PIMO, we can reuse schemas for concepts such as people, messages, events, and location (cf. Section 4.4).

- **Situation identification and reasoning mechanism** Since the proposed system should facilitate the identification of situations, a reasoning mechanism that is able to deduce higher level context information from low-level context information is needed. This can be achieved by applying rules, heuristics or logics and facilitates the accurateness of user situation identification. Using our system, an extending application

is capable of building up their own situation model and/or using the six "basic" situations (*at work*, *in a business meeting*, *available at work*, *spare time*, *at home*, and *en route*) that the system itself is able to identify providing that the user specifies their recognition features (cf. Section 4.4.12).

- **Support for personal information management** In order to support (situation-aware) personal information management, local personal data items like recorded data of phone calls, e-mails, contacts, calendar entries, and files have to be exposed as RDF and interlinked with each other. Furthermore, these data is combined with contextual data. Hence, we have to consider personal data items, as well as contextual data items, like the location of the user, when designing our context model schema (cf. Section 4.4).

- **Privacy** In order to enable users to decide on which data they would like to provide to which applications, privacy preferences have to be implemented. By means of these preferences, users have the possibility to classify data types into different levels of privacy. This classification can be additionally represented in the overall context model. Detailed information about privacy preferences are given in Section 4.3.1.

In this Section, we discussed the main characteristics of our approach. We decided to build a *mobile Semantic Desktop* application, according to the classification scheme in Chapter 3. This classification type inherits the requirement of supporting personal information management for our system. Furthermore, the proposed system provides context history in order to facilitate reasoning. As a communication channel between system and applications a subscribe-notify mechanism is implemented which informs applications about new data in which they are interested. We decided to represent contextual information by reusing terms of already existing ontologies. Moreover, a reasoning mechanism that facilitates situation identification is implemented. In order to protect the users' privacy, we decided to provide mechanisms that enable users to individually define privacy principles on their own.

4.2 Introduction of Possible Scenarios

In this Section, two scenarios that have to be able to accomplish with the proposed system are described. In the following, we exemplify our approach in each section on the basis of these two scenarios.

4.2.1 Annotate/Group data items

The first scenario provides the possibility to explicitly save different data items, such as files, emails, or calendar entries as a *subject* in the context model, i.e. provide a grouping functionality for items with similar characteristics. A project is one possible concept for categorizing personal information items and can include every type of personal data items stored on the mobile phone. When calling this function, the user is able to either choose an already existing subject and add the items to it or create a new one. The new information is added to the context model as additional RDF statements. At another point in time, maybe the user wants to retrieve data stored as a specific subject. Hence, they browse all stored subjects of the application and select the desired one. Then, the system shows all data items that are related to it. This function is useful to, for example, easily access project-related e-mail correspondence or generally all data items for the project one is currently working on.

4.2.2 Remember locations

There may be an application using our system which is able to remind the user to buy, for example, foodstuffs when they are around a supermarket, for instance, one day before guests are coming to their place. Therefore, this application enables the user to annotate locations which they would like to remember. This means, for example, the user is in a supermarket and likes to store its geographical coordinates in the context model, because they want the application to recognize when they are near this location at another point in time. Therefore, the user opens the application, tells it that its current location is from type supermarket, and stores it in the system. Furthermore, the user defines that one can buy foodstuffs and beverages at this location and gives it a name. Now they are able to define a calendar entry that says, for example, "buy foodstuffs" with a due date of April, 28th and sets a reminder two days before. Hence, the application has to remember the user of going shopping from April, 26th till April, 28th whenever they are around a location of type supermarket. Furthermore, with such an application a user can, for example, define a location as a doctor's address. Later, they add a calendar entry stating that they have an appointment at the doctor's. Then the application calculates how long it is taking the user to go to the doctor from their current location, every time the position of them changes. If the current time plus the required time plus, for instance, fifteen minutes is equal to the time of the appointment, the application notifies the user to set off.

4.3 Mobile Semantic Desktop Infrastructure

In the following, the mobile Semantic Desktop infrastructure is described. Therefore, the different steps of processing are functionally divided into layers (cf. Figure 4.1), whereas the four bottom layers correspond to our system and the topmost layer corresponds to applications extending the system. At the end of this section we exemplify the functionality of our system by means of two scenarios introduced in Section 4.2.

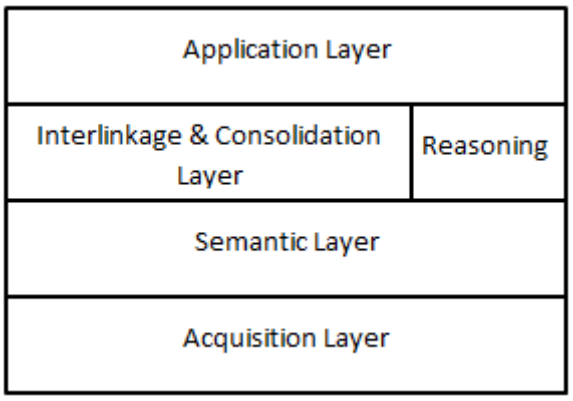

FIGURE 4.1: Conceptual design of the Mobile Semantic Desktop Infrastructure

On the bottom layer there are sensors and data sources. In this work, we define a sensor as an internal or external physical device or a web service. All of them acquire contextual information describing the environment of a mobile phone and its user. Data sources on the other hand, extract and provide personal data items stored on the mobile phone, like e-mails, files, or contacts. There are some sources that directly provide data to the system and others only notify the system that a data update is available. In the second case, the system has to fetch the data and find out what is new, before processing it. All raw data acquired from sensors and data sources are processed and as a result represented as RDF models, using controlled vocabulary and ontologies (cf. *Semantic Layer* in Figure 4.1). Furthermore, there are optional components that are able to generate additional information from data that are acquired by sensors respectively data sources. For example, a service that returns textual address information, like name of city, name of street, and house number, from given longitude and latitude values. These components start processing every time there is new data that they are able to process and as well return RDF models. For each type of data, an individual model, according to a predefined schema, is built. These predefined schemas use terms from already existing ontologies, like PIMO or the NEPOMUK ontologies, if possible (cf. Section 4.4). More details about data acquisition and representation is given in Section 4.3.1.

The models resulting from the two bottom layers are then interlinked with each other (cf. *Interlinkage & Consolidation Layer* in Figure 4.1). For instance, contact resources resulting from exposing all contacts of the user's phone book are interlinked with correspondences in which the contact is involved or with events (calendar entries) the person participates in. Moreover, location information can be combined with, for example, temperature information. The result of this step is a connected overall context model. Information about methods and algorithms for interlinkage are given in Section 4.3.1. As a next step, the model is consolidated. In order to achieve this, duplicate and unnecessary information is removed from the overall model. For instance, identical information about temperature is only kept once. Inconsistent information is made consistent by trying to detect and keep the correct/valid data item, and deleting the other one(s). Detailed information about aggregation and consolidation of the context model is given in Section 4.3.1.

Whenever statements are updated, added to, or deleted from the overall context model, the reasoning process starts. Our approach uses forward chaining rule-based reasoning (cf. Section 2.3.2). There is a local rule-base which holds predefined reasoning rules used by the reasoning engine. The RDF statements resulting from the reasoning engine are added to the overall model. More details about reasoning is given in Section 4.3.1.

The proposed system enables applications to perform situation identification. These may build a situation model representing situations like *going by car* and *sleeping* which are defined as assertions. Possible examples for such assertions may be the following: to find out whether the user is at work or not, the user could either tell the system the coordinates of their place of work, if they work always at the same place. Another possibility to identify a working situation may be a period of time in which the user is always at work. Whenever the user is - according to this definition - *not at work*, it is assumed that they have spare time. If the user is at work and there is an event entry in their calendar at a certain time, it is expected that the user is in a business meeting at this time. As with the situation *at work*, also the situation *at home* can be recognized by the coordinates of the user's home. Whenever the user is, for example, 100 meters around these coordinates, they are assumed to be *at home*.

Applications extending the proposed system benefit from it for all kinds of context-/ situation-aware functionalities. In order to recognize when the user's context model changes and react to these changes, an application has the possibility to individually subscribe to particular context change events and get a notification from the system whenever the event occurs. There are different alternatives of subscribing. Applications can either subscribe to predefined change events, like a change of the user's current geographical location or lighting conditions, or register individual queries, for which

they would like to have the result set monitored. More detailed information about subscription and notification is given in Section 4.3.1. Moreover, applications have the possibility to send so called "on demand" SPARQL queries to the system which sends back the result set. The system may reject queries from applications, if they violate privacy preferences. (cf. Section 4.3.1). In order to further secure the users' data, applications are prevented from fetching too much data from the context model, i.e. only a predefined number of entities are allowed to be fetched by one SPARQL query. If an application tries to get more than that, an exception is sent back to it. Applications may, furthermore, add RDF statements to the system's overall context model. Therefore, the application has to send the statements it likes to add as well as an RDF-Schema describing the additional data. This avoids that different applications use the same vocabulary for different purposes. For privacy reasons only fully trustworthy applications are allowed to add arbitrary statements with arbitrary namespaces. Further information about privacy is given in Section 4.3.1.

In the following, we exemplify the content of this section on the basis of the two scenarios introduced in Section 4.2:

- **Annotate/Group data items** In order to enable this function, the application has to add RDF statements to the system's context model whenever the user saves a new subject respectively adds items to an already existing subject. The first time it likes to add statements it has to additionally send an RDF-Schema which has to be proofed by the system. This means the system has to ensure that the additional data do not contradict with already existing data. Furthermore, the application uses "on demand" SPARQL queries every time the user likes to retrieve all elements of a stored subject. For the this functionality, no subscription to the system is necessary. The data items that can be added to the context model is limited by the data items that are already in the context model.

- **Remember locations** For this scenario to be accomplished, there have to be at least one context provider for location information and one data provider for calendar entries which create RDF graphs out of the data provided from their corresponding sensor respectively data source. An application implementing this functionality has to subscribe to our system for changes of calendar entries and changes of location. Every time the user either adds an annotated location or a calendar entry associated with an annotated location, the application has to check if the user is around an annotated location (by means of on-demand SPARQL queries) and the user has to be remembered of it, in order to take the action specified in the particular calendar entry. Our system has to send a notification to the application every time there is an update of calendar entries or locations.

All the other tasks that have to be done to accomplish the described scenario are handled by the application implementing this function.

4.3.1 Details of Approach

In this section, the most important constituents of our approach are described in more detail.

Acquisition

In the proposed system, the acquisition of data is performed by means of sensors and data sources. While, sensors are responsible for acquiring contextual information, data sources on the other hand extract and provide personal data items stored on the mobile phone, like e-mails, files, or contacts.

Physical sensors have their own built-in behaviour concerning the provision of information about data updates. There are the ones that immediately provide new data and others that have to be polled. Sometimes, it is possible to define conditions for automatic sensor updates, like a minimal difference between the old and the new value or a minimum time interval for changes. Data sources on the other hand either notify the system about changes or directly send the updated data to the system.

Depending on these specific properties of sensors and data sources, an appropriate context/data wrapper has to be created for each of them. This means, for example, a data wrapper that is attached to a data source which only sends notification whenever a data update is available, this data wrapper has to first fetch all the data from the data source and then compare it with the current data base in order to find out what the difference is between old and new data. On the other hand, a context wrapper attached to a sensor that has to be polled, has to send requests to the sensor on a regular basis.

All acquired data are then processed by context and data wrappers and as a result are represented as RDF models using controlled vocabulary and ontologies. Each of the wrappers has to always provide a model of the same schema that is predetermined by the system (cf. Section 4.4). However, a context/data provider is able to extend this predefined schema by adding individual statements as it delivers this extended schema to the system together with its first context/data model. Every time a context wrapper provides a model, the system has to compare the structure of the provided model with the predefined one. If they are not identical, for instance, if a statement is missing, the provided model is discarded. Examples of predefined schemas are provided in Section 4.4.

Whenever a new sensor is included into the system, an appropriate context wrapper has to be added too. It may be possible to use an existing one or a new one has to be created. Furthermore, there are optional components that are able to generate additional information out of data that are provided by context wrappers - so called *reactive* context wrapper. For example, a service that returns another representation of address information from the given longitude and latitude values. They always listen for changes in the data store, where all models are buffered before they are aggregated to one overall model, and start processing every time there is new data that they need for input. While such a component is processing data, the used models are locked in order to ensure that they cannot be manipulated by other components. If there arises an updated model of the one that is currently in use, the processing component gets informed, discards its previous computations, and restarts the process with the updated model. Only the new model is kept. In order to avoid deadlocks, a maximum execution time for these components is defined. If the computation is not finished within this period of time, the model is freed for other components and a possible delayed result is discarded. Reactive context wrappers are not allowed to update respectively manipulate a model more than once. They must also provide a model of a predefined schema. All these schemas use parts of already existing ontologies, like PIMO or the NEPOMUK ontologies, if possible (cf. Section 4.4).

Interlinkage and Consolidation

Interlinking and consolidating describes the processes of connecting the different "standalone" models and removing duplicate and unnecessary information from the overall model. First, the models provided from sensors and other data sources are interlinked. Therefore, resources that are assumed to be the same are combined to one resource with the same URI. For example, there is one contact resource that exists in the model provided by the contacts data provider and another contact resource which exists in the e-mail provider's context model and they both have the same e-mail address property. Hence, they can be assumed to be the same contact and be represented by the same URI. Resources of the same type have to be compared across the different models provided by the data and context providers. For each type of resource an identifying attribute or attribute combination is defined. If the values of them are identical in two or more resources, they are combined to one resource. For the example of contacts this means it is iterated over all statements of models containing contact resources except the contacts model itself. In the case of calendar entries, it is iterated over all resources that are of type social event and have attendees as properties. Then the attendees' e-mail addresses are compared to the ones in the contact model. If there is a contact with an equivalent

e-mail address in the model, the contact resource attached to the social event item is replaced by the contact resource from the contact model. Analogous to this example, this is likewise done with e-mail and phone call models.

Then, redundant and inconsistent information is removed from the overall model. This means, for instance, identical information about temperature is only kept once. Inconsistent information is made consistent by trying to detect and keep the correct/valid statement, and deleting the other(s). Therefore, each model which is generated by a data respectively context provider adds a statement about the data's origin to the model. In order to decide which of the inconsistent data items is the most trustworthy one, there may be a central web database, which holds information about the reliability of sensors. This information is provided by people that have experience with certain sensors. They have the possibility to evaluate them regarding their experienced reliability. Our system, then fetches the information from the database, calculates the arithmetic average of all provided reliability values, and takes it as a means to decide which of two conflicting statements is more likely to be the right one. Another possibility for detecting which information is more reliable than another is to evaluate sensors on the basis of their accuracy specified by the producer. For example, a local positioning system, e.g. by GSM that has an accuracy of plus or minus one kilometer may be not as reliable as one with an accuracy of plus or minus 50 meters.

If there are two statements in the overall model that represent the same information, only one of them is kept. Which means, for example, if two different environmental temperature sensors provide temperature information that differ not more than one degree, they are assumed to be identical. In cases of sources like e-mails or phone calls, only totally identical statements can be consolidated. However, there may be data sources that always deliver a model with all its data items instead of a model including only the updated respectively new data items. In such a case, already existing statements need to be consolidated as well.

Reasoning

In order to provide additional data to the so-called low-level sensor data, we use a reasoning engine which deduces higher level context data. In our case, this is achieved by evaluating reasoning rules using the forward chaining technique. The algorithm loops through all reasoning rules stored in the rule base as long as it can find any where the antecedent evaluates to true. Every time such a rule is found, the new information is inferred and the process is going on until no rule can infer new information any more.

In the rule base, which is located on the mobile device, predefined rules are persistently saved. The resulting statements are added to the overall model.

In the following, there is an example for a predefined rule:

```
1  [atHome: (?location rdf:type pimo:Location)
2           (?location geo:long "48.21302")
3           (?location geo:lat "16.360686")
4           ->
5           (<http://www.myns.at/users/me> myns:inSituation myns:atHome)]
```

CODE 4.1: Example rule for a rule-based reasoner

This example rule is a simplified rule for deducing that the user is "at home". The rule's antecedent evaluates to true if there is a resource of type `pimo:Location` which has a property `geo:long` that has the value 48.21302 and a property `geo:lat` that has the value 16.360686. If all three conditions are true, a new statement which states that the user is in the situation "at home" can be added to the overall model. Every rule begins with a name ("atHome" in the above example). The antecedent of the rule shown in Code 4.1 consists of three parts, implicitly connected by a logical AND operator. The arrow is the connector between the antecedent and the consequence part, which only consists of one statement in this example.

Provision of context history

As mentioned in Section 4.1 context history is provided by our system, in order to facilitate situation identification for applications. Therefore, every piece of contextual data is stored together with a timestamp for a certain period of time until it gets outdated. A data item is considered valid within a specific time frame, which is predefined by the system. Of course, not every piece of data can be deleted after the same time period. Information is not deleted as long as there is no update for it. We decided to use a system-defined time frame, in order to ensure that the amount of data does not exceed the available memory capacity. Applications that like to make use of this feature can fetch historical data by sending a request including either a SPARQL query or a resource type together with the desired period of time. If a request for a specific resource type arrives, the system sends back all RDF statements where this resource is used as a subject in the specified period of time. In the other case the result of the SPARQL query in consideration of the period of time will be returned.

Subscription and Notification

As mentioned in Section 4.1 it is possible for external applications to extend the mobile Semantic Desktop infrastructure. By using the proposed system, applications are relieved from acquiring, representing, and aggregating contextual data on their own. In order to optimize their performance, applications have the possibility to individually define which kind of data they would like to use. Therefore, a mechanism that proactively notifies them about changes on data level is implemented. Applications can either subscribe to changes of specific context types, such as the change of the user's current geographical location respectively a change of the current lighting conditions, or register a SPARQL query (cf. Section 2.1.3). This is achieved by implementing a SPARQL interface for our system, enabling applications to define their individual queries, for which they would like to have the result set monitored. These queries are saved individually for each application and whenever the result for a query changes, the registered application is informed. Subscriptions for changes of predefined context types apply to changes of the value of this type. For changes of the location this means the longitude and/or latitude value changes. The subscription happens by sending an application ID together with the desired ID of the context change event to the system, which stores the entries as tuples consisting of application ID and context change event ID. In order to subscribe with SPARQL queries, the application has to send its application ID together with a syntactically correct SPARQL query. After receiving the request, the system checks whether the syntax of the SPARQL query is correct or not. If it is not correct, the system sends back an error message. Otherwise, the query is stored together with the application ID. Whenever a change happens in the overall context model, the system has to check whether there is a subscription for the current change event or not. If there is a subscription concerning the change, a notification has to be sent. A notification is a message including information about which change event occurred and the new data. The information about the kind of change event is either a SPARQL query or a flag which indicates a predefined context change event. There are two kinds of changes, which are structural changes and changes of instance values. In the case of a subscription for a change of a particular context type, only instance value changes are notified.

Privacy

Users have the possibility to prevent specific context/data providers from exposing their data in order to preserve the users' privacy. This means that the data of such a provider is not included into the overall model. Furthermore, a level of privacy can be specified

for all types of data. For example, level 1 data types are allowed to be provided to all applications, level 2 data types can be provided to medium trustworthy applications, and level 3 data types are only allowed to be provided to fully trustworthy applications. This can be achieved by adding a statement about data/context sources, which represents the level of privacy for all data that is provided by this source. An application's level of trustworthiness can be stated at the time of application installation. This classification can be stored as a parameter when an application is registering at the system. If an application which has no permission to get information about particular entities tries to fetch such entities by sending "on demand" SPARQL queries, the system sends an exception to the application which tells it that it has no authorization for getting the requested data. The same happens if an application tries to subscribe for change events for which it is not authorized. Moreover, only fully trustworthy applications are allowed to add statements with arbitrary namespaces to the context model. All the others have to use their own application-specific namespace for the resources and properties they like to add.

4.4 Context Model Schema

In this section, the design of an ontology-based representation schema for contextual information and data items in a mobile Semantic Desktop environment that makes use of existing ontologies is introduced. We re-use already existing ontologies in order to do not have to create a whole new ontology from scratch. With this data model, information about the different data items on a mobile phone can be stored in a homogeneous way by adopting Semantic Web technologies. It contains information that can be used by a mobile device to be able to adapt to changing user situations, whereas the user plays a central role. Furthermore, the different data items can be related to each other, for instance, to indicate that they belong to the same project. The described approach including its data model enables the realization of applications for mobile devices which change their behaviour according to the current situation of the user or provide user- and context-specific support for them. The context model enables the representation of a multiplicity of context sources. The big difference to other existing context models is the combination of personal information management and context-sensitiveness, which is accomplished by reasoning (cf. Section 2.3).

In the following, every data source considered is described regarding its classes and properties, by means of example RDF statements and graphical representations. For this purpose, the Turtle[1] serialization format, as well as RDFS models are used. In order

[1] cf. `http://www.w3.org/TeamSubmission/turtle/`

to make the code more readable, namespace prefixes are applied to unambiguously refer to the appropriate concepts. All used namespaces with their corresponding URIs are shown in Table 4.1. At the end of this section, we depict one example ontology instance corresponding to one of the scenarios introduced in Section 4.2.

Prefix	URI
ccpp	`http://www.w3.org/2002/11/08-ccpp-schema#`
env	`http://www.cs.kuleuven.be/~distrinet/projects/CoDAMoS/2005/01/Context.owl#`
geo	`http://www.w3.org/2003/01/geo/wgs84_pos#`
myns	`http://www.myns.at#`
nco	`http://www.semanticdesktop.org/ontologies/2007/03/22/nco#`
nie	`http://www.semanticdesktop.org/ontologies/2007/01/19/nie#`
nmo	`http://www.semanticdesktop.org/ontologies/2007/03/22/nmo#`
owl	`http://www.w3.org/2002/07/owl#`
pimo	`http://www.semanticdesktop.org/ontologies/2007/11/01/pimo#`
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs	`http://www.w3.org/2000/01/rdf-schema#`
xsd	`http://www.w3.org/2001/XMLSchema#`

TABLE 4.1: Namespaces used for describing the context model schema

On the mobile phone, we have to organize the following pieces of data, which are described in the subsequent sections. They are directly stored on the Android device, to be able to relate them to users' situations (cf. Section 4.4.12):

- calls
- e-mails
- SMS
- contacts
- calendar entries
- files

In order to transfer these data units to a data model, which is applied to enable context awareness on a mobile phone, we use existing ontologies, such as the NEPOMUK Message Ontology[2], the Personal Information Model Ontology[3], the NEPOMUK Information Element Ontology[4], and the CoDAMos Ontology[5] and bring them together to form one big context model schema for mobile phones. Furthermore, some new elements

[2] cf. `http://www.semanticdesktop.org/ontologies/nmo/`
[3] cf. `http://www.semanticdesktop.org/ontologies/pimo/`
[4] cf. `http://www.semanticdesktop.org/ontologies/nie/`
[5] cf. `http://distrinet.cs.kuleuven.be/projects/CoDAMoS/2005/01/Context.owl`

for web pages, applications, settings, and situations had to be defined in order to map all relevant concepts that exist in a mobile environment.

4.4.1 Phone Call

A phone call is represented in the data model to see with whom the user is in contact and at which times. It is represented with the concepts of the NEPOMUK Message Ontology[6] as `nmo:Call`, which is a subclass of `nmo:Message` (cf. Figure 4.2). For each call, we can store the person who called (`nmo:from`), which has to be an instance of `nco:Contact`; the person, who was called (`nmo:to`), it has to be an instance of `nco:Contact` as well; and the date and time when the call came in or went out (`nmo:receivedDate`/`nmo:sentDate`) to be able to deduce in which situation (cf. Section 4.4.12) a user was on a call. Dates are represented as literals from type `xsd:dateTime`. It is possible to explicitly state in which situation the user was during a particular call, by adding the property `myns:inSituation` whose value is from type `myns:Situation`.

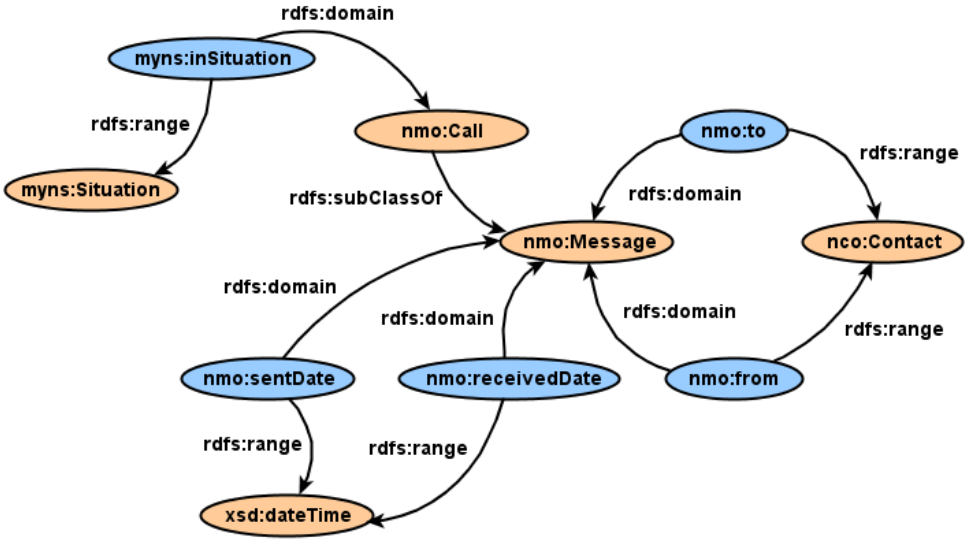

FIGURE 4.2: RDF-Schema of phone call entities

Every time a call comes in or goes out, it is inserted into the context model. In order to be able to insert the data into the model, it is necessary that the sensor is sending information about phone number, date and time of call, as well as type of call, i.e. if it is an incoming or outgoing call. If there is already a contact object with the caller's telephone number stored in the mobile phone's phone book, the model references to the representation of it, otherwise a new object is created and the user is asked whether

[6] cf. `http://www.semanticdesktop.org/ontologies/nmo/`

they like to add it to their phone book or not, by an android status bar notification[7]. An example phone call representation can be seen in Code 4.2.

```
1 <http://www.myns.at/calls/call3> rdf:type nmo:Call;
2     nmo:from <http://www.myns.at/com.android.contacts/contacts/37>;
3     nmo:to <http://www.myns.at/com.android.contacts/contacts/28>;
4     nmo:sentDate "2012-01-04T18:20:44"^^xsd:dateTime .
5 <http://www.myns.at/com.android.contacts/contacts/37> rdf:type nco:Contact .
6 <http://www.myns.at/com.android.contacts/contacts/28> rdf:type nco:Contact .
```

CODE 4.2: Example phone call representation

4.4.2 E-mail

E-mails are an important communication medium in project teams, we use them in the ontology to be able to, for example, relate the correspondence about a particular topic to the associated project. E-mails are added to the context model, every time the user receives or sends one. In order to get the information that is needed to represent an e-mail using RDF statements, in the first instance, there has to be a listener, which recognizes new e-mails. Therefore, the e-mail provider has to provide an API for accessing information about e-mails. All information that should be represented is extracted from the received e-mail objects. An e-mail is represented by the class `nmo:Email` and is also a subclass of `nmo:Message` (cf. Figure 4.3). As for the class `nmo:Call` we can represent the person who wrote the e-mail, the people who received the e-mail (all instances of `nco:Contact`), and the date and time when the e-mail came in or was sent. The object of the `nmo:sentDate` property is a literal of the `xsd:dateTime` data type. For e-mail messages, we additionally have the information of who got a copy (`nmo:cc`) or blind copy (`nmo:bcc`) of the email (all instances of `nco:Contact`), what is the message's subject (`nmo:messageSubject`) and if it is a reply to a previous message (`nmo:inReplyTo`) to be able to identify connected correspondence. In order to locate an e-mail, which may not be stored directly on the mobile phone, we furthermore, store the location information of an e-mail (`myns:storageAddress`), which is from type `rdfs:Resource`.

As soon as an e-mail is deleted, the property `myns:resourceDeleted` has to be set to "true". The contact objects for the properties are identified by their e-mail address. If there already exists a contact with the desired e-mail address, it is referenced to this object, otherwise a new one has to be created. If the e-mail address does not exist in the mobile phone's address book, the user gets an android status bar notification, which suggests to save the e-mail address to an existing contact or create a new one. In the following there is an example representation of an e-mail entry:

[7] cf. `http://developer.android.com/guide/topics/ui/notifiers/notifications.html`

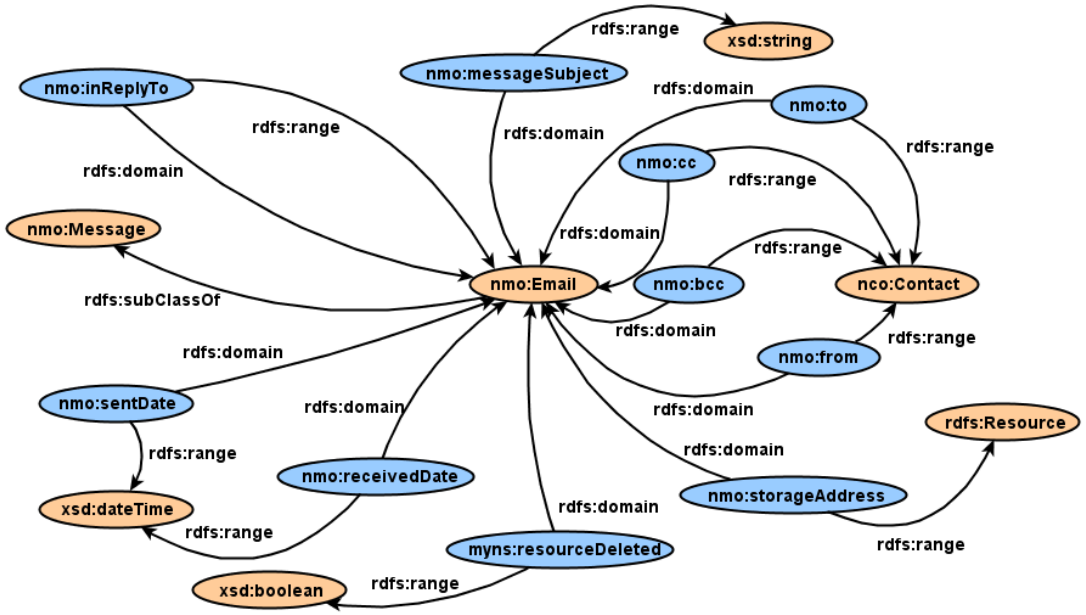

FIGURE 4.3: RDF-Schema of e-mail entities

```
1  <http://www.myns.at/emails/email3> rdf:type nmo:Email;
2      nmo:from <http://www.myns.at/com.android.contacts/contacts/37>;
3      nmo:to <http://www.myns.at/com.android.contacts/contacts/28>;
4      nmo:cc <http://www.myns.at/com.android.contacts/contacts/33>;
5      nmo:cc <http://www.myns.at/com.android.contacts/contacts/79>;
6      nmo:bcc <http://www.myns.at/com.android.contacts/contacts/67>;
7      nmo:sentDate "2012-01-04T18:20:44"^^xsd:dateTime;
8      nmo:messageSubject "diploma thesis"^^xsd:string;
9      nmo:inReplyTo <http://myns.at/emails/email18>;
10     myns:storageAddress <https://mail.google.com/mail/?shva=1#inbox/
11     198ac7648b8211e3> .
12 <http://www.myns.at/com.android.contacts/contacts/37> rdf:type nco:Contact .
13 <http://www.myns.at/com.android.contacts/contacts/28> rdf:type nco:Contact .
14 <http://www.myns.at/com.android.contacts/contacts/33> rdf:type nco:Contact .
15 <http://www.myns.at/com.android.contacts/contacts/79> rdf:type nco:Contact .
16 <http://www.myns.at/com.android.contacts/contacts/67> rdf:type nco:Contact .
17 <http://myns.at/emails/email18> rdf:type nmo:Email .
```

CODE 4.3: Example e-mail representation

4.4.3 SMS Message

The third class that is a subclass of `nmo:Message` is `nmo:SMSMessage` and we store the same properties for `nmo:SMSMessage` as for `nmo:Call`, plus the `nmo:inReplyTo` property that is also used for `nmo:Email`. SMS messages are inserted into the model, whenever the user receives or sends an SMS message. All relevant data is extracted from the Android's SmsMessage objects.

4.4.4 Contact

Another important data unit in the context model is the contact. Contacts are in most cases persons or organizations one is in contact with. We use them in our context model to be able to, for example deduce project team members of a current project and uniquely refer to people. Therefore, we use `nco:Contact` (cf. Figure 4.4). For each contact in the ontology the fullname (`nco:fullname`, which is a literal of type `xsd:string`), phone numbers (`nco:hasPhoneNumber`), the postal address (`nco:hasPostalAddress`), e-mail addresses (`nco:hasEmailAddress`) and website-URLs (`nco:websiteUrl`) can be represented. If the resource is from type `nco:PersonContact`, which is a subclass of `nco:Contact`, additionally the first name (`nco:nameGiven`), the surname (`nco:nameFamily`), and a name addition (`nco:nameAddition`) which are all represented as literals of type `xsd:string` can be defined. In order to identify a contact, the telephone number or the e-mail address is used.

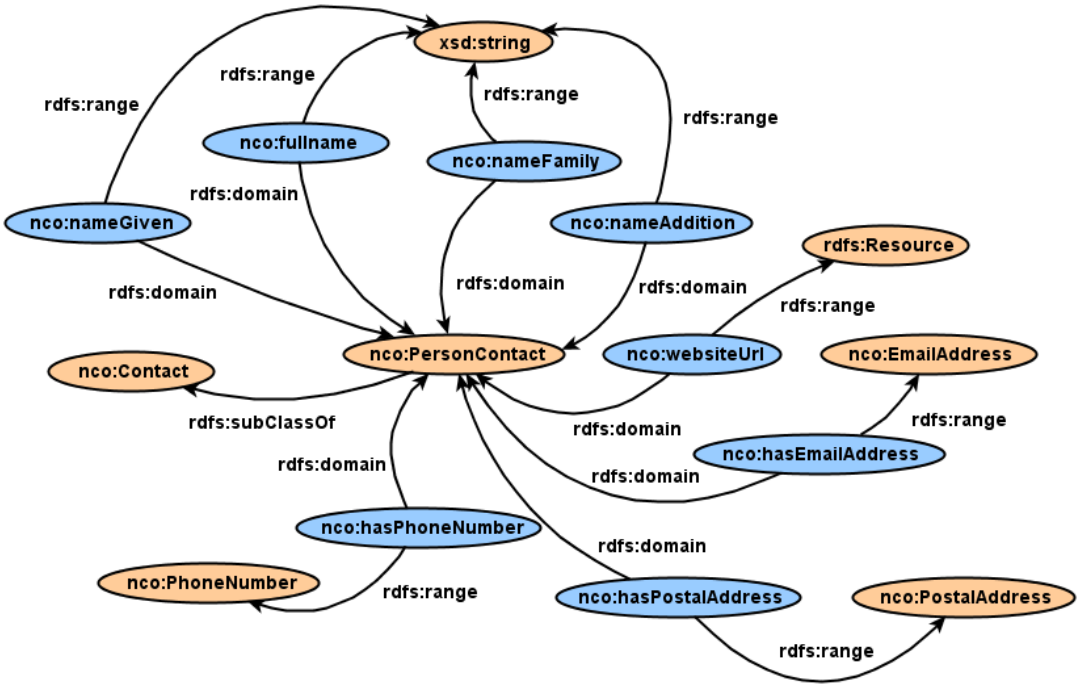

FIGURE 4.4: RDF-Schema of contact entities

Contacts are inserted into the context model, whenever the user is adding a new contact to their mobile phone's address book. They are accessed directly from the internal storage of the mobile phone. All relevant information is provided by the Android's internal *Contacts Provider*. Here is an example of an `nco:PersonContact` representation:

```
<http://www.myns.at/com.android.contacts/contacts/37> rdf:type
        nco:PersonContact;
    nco:fullname "Carina Christ"^^xsd:string;
    nco:nameGiven "Carina"^^xsd:string;
    nco:nameFamily "Christ"^^xsd:string;
    nco:nameAddition "BSc"^^xsd:string;
    nco:hasPhoneNumber <http://www.myns.at/com.android.contacts/data/
        phones/119>;
    nco:hasPostalAddress <http://www.myns.at/com.android.contacts/data/
        postals/27>;
    nco:hasEmailAddress <http://www.myns.at/com.android.contacts/data/
        emails/45>;
    nco:websiteUrl <http://www.my-hp.at> .
<http://www.myns.at/com.android.contacts/data/phones/119> rdf:type
        nco:PhoneNumber .
<http://www.myns.at/com.android.contacts/data/postals/27> rdf:type
        nco:PostalAddress .
<http://www.myns.at/com.android.contacts/data/emails/45> rdf:type
        nco:EmailAddress .
```

Code 4.4: Example person contact representation

4.4.5 Calendar Entry

One of the most important data stored on the mobile phone for situation-identification of users is the calendar entry since it can provide important information about where a user is at a certain point in time. The most suitable concept for this is `pimo:SocialEvent`, which is a subclass of `pimo:Locatable`. This is the reason why it can have the property `pimo:hasLocation`, which indicates the location where the event takes place. Objects of this property are from type `pimo:Location` (cf. Section 4.4.7).

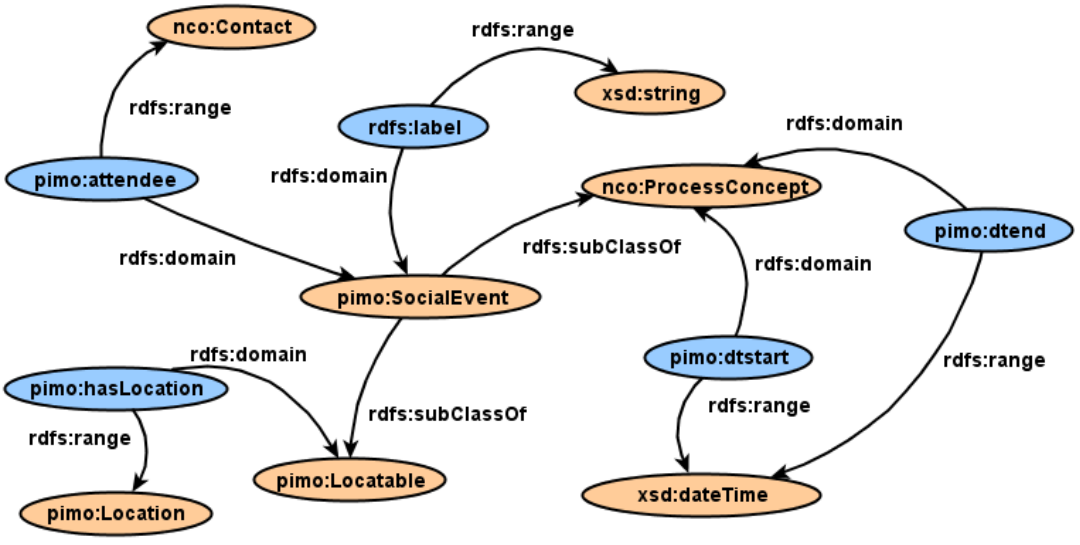

Figure 4.5: RDF-Schema of social event entities

Furthermore, it can have the property `pimo:attendee` to show who is participating in the event. The objects of this property are from type `nco:Contact`. If there does not exist a contact with the specified e-mail address in the address book of the mobile phone and in the model, the user gets a status bar notification, which suggests to add the e-mail address to an existent contact or create a new one and a new contact resource has to be created and inserted into the model. Another superclass of `pimo:SocialEvent` is `pimo:ProcessConcept`, which provides properties for the start (`pimo:dtstart`) and end time (`pimo:dtend`) of an event. Start and end time are represented as literals of type `xsd:dateTime`. Each `pimo:SocialEvent` can moreover be labelled with `rdfs:label` to give them a human-readable title, which has to be from type string. A resource of type `pimo:SocialEvent` is added to the model, every time the user adds a calendar entry. All data needed, is provided by the Android's *Calendar Provider*. In the following, an example of a calendar entry representation including its properties is shown:

```
1 <https://www.google.com/calendar/feeds/user/private/full/587> rdf:type
2   pimo:SocialEvent;
3     pimo:dtstart "2012-04-13T18:00:00"^^xsd:dateTime;
4     pimo:dtend "2012-04-13T22:00:00"^^xsd:dateTime;
5     rdfs:label "Business Dinner"^^xsd:string;
6     pimo:attendee <content://com.android.contacts/contacts/128>;
7     pimo:attendee <content://com.android.contacts/contacts/56> .
8 <http://www.myns.at/com.android.contacts/contacts/128> rdf:type pimo:Person .
9 <http://www.myns.at/com.android.contacts/contacts/56> rdf:type pimo:Person .
```

Code 4.5: Example event representation

4.4.6 File

Furthermore, we have to store files in the semantic data model. As mentioned above, data units can be related to projects and suggested to the user by a context-aware mobile phone, for example when they are preparing for a project team meeting.

Files can be represented by means of the class `nie:InformationElement`. For every `nie:InformationElement`, the creation date (`nie:contentCreated`), the title (`nie:title`) and the size (`nie:contentSize`) of the file, as well as the MIME-type (`nie:mimeType`) and the date when it was last modified (`nie:contentLastModified`) can be stored. The objects of the properties `nie:contentCreated` and `nie:contentLastModified` have to be from type date-time. Title and MIME-type are literals from type string, and the literals corresponding to `nie:contentSize` have to be from type integer. An information element is added to the context model every time a file listener notices a new file on the mobile phone. In Code 4.6 there is an example instance of a model delivered by a file data provider.

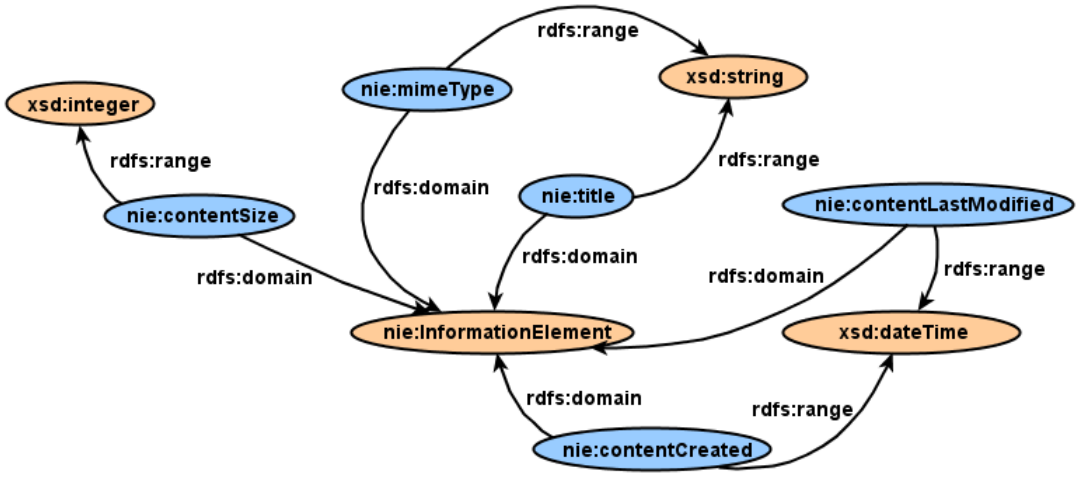

FIGURE 4.6: RDF-Schema of information element entities

```
1 <http://www.myns.at/com.android.files/html/56> rdf:type nie:InformationElement;
2     nie:title "Project Presentation v2.0"^^xsd:string;
3     nie:contentCreated "2011-12-02T18:07:58"^^xsd:dateTime;
4     nie:contentLastModified "2012-02-14T17:06:32"^^xsd:dateTime;
5     nie:contentSize "78"^^xsd:integer;
6     nie:mimeType "text/html"^^xsd:string .
```

CODE 4.6: Example information element representation

All these various concepts can be related to each other in order to express, for example that they belong to the same project. There are two possible ways to achieve this by using concepts of the NEPOMUK Ontologies. It can either be achieved by relating the classes - that belong together - with each other by using `nie:relatedTo`, or by associating them with a shared topic (`pimo:Topic`).

4.4.7 Location, Temperature, and Lighting

Beside the concepts described so far, facts like location, temperature, and lighting, have to be included in the model. The location information is important for many kinds of applications that are able to adapt their behaviour according to the user's current location. For example, when the user is at the doctor, the mobile phone is muted automatically. In order to deduce whether a user is outdoors or indoors the information about the current environment temperature may be helpful, amongst others. To represent users' locations we use `pimo:Location` in combination with the Geo Vocabulary[8]. A location instance consists of at least two values of type double, whereas one of them represents

[8] `http://www.w3.org/2003/01/geo/#vocabulary`

the longitude value and the other the latitude value. Additionally a timestamp, which is from type long, may be added.

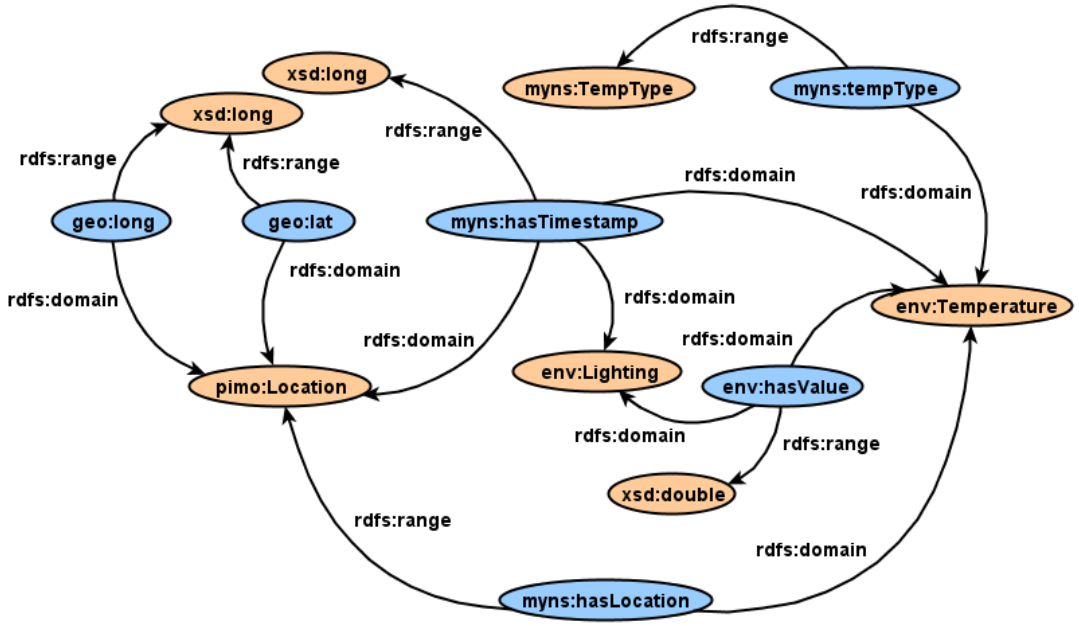

FIGURE 4.7: RDF-Schema of location, temperature, and lighting entities

The user can tell the system in which intervals they would like to make an update about their location. This can either be specified as a time-interval in seconds or as a spatial difference in meters. Location information is delivered, for example, by the built-in location sensor of an Android device. In the following, there is an example representation of a location instance:

```
1  <http://www.myns.at/locations/256478922> rdf:type pimo:Location;
2      geo:long "48.21302"^^xsd:double;
3      geo:lat "16.360686"^^xsd:double;
4      myns:hasTimestamp "2456879144755"^^xsd:long .
```

CODE 4.7: Example location representation

For the representation of lighting and temperature the concepts of the CoDAMoS Ontology [PdBW+04] are used. It provides, amongst others, classes for representing the environmental conditions, location, and tasks or activities of the user. For the use in this data model the concepts that can be seen in Code 4.8 are reused. Additionally to the concepts used in CoDAMoS we also use a timestamp (`myns:hasTimestamp`) and a location (`myns:hasLocation`) property. For temperature information, furthermore, a property `rdf:tempType`, which indicates whether the given temperature information comes from an indoor, outdoor, or built-in sensor, has to be included. The object of this property has to be one of `myns:OutdoorTemp`, `myns:IndoorTemp`, or `myns:BuiltInTemp`

which are from type `myns:TempType`. Information about lighting and temperature of the user's environment can, for example, be delivered by the built-in lighting and temperature sensors of the Android mobile phone. The user may define a time-interval for the frequency of updates or a minimum offset from the previous value that triggers an update.

```
<http://www.myns.at/lightings/6545684894894654> rdf:type env:Lighting;
    env:hasValue "150"^^xsd:float;
    myns:hasTimestamp "2456879144755"^^xsd:long;
    myns:hasLocation <http://www.myns.at/locations/54689798798> .
<http://www.myns.at/locations/54689798798> rdf:type pimo:Location .

<http://www.myns.at/temperature/84894894654> rdf:type env:Temperature;
        env:hasValue "27"^^xsd:float;
        myns:hasTimestamp "2456879144755"^^xsd:long;
        myns:hasLocation <http://www.myns.at/locations/54689798798>;
        myns:tempType myns:OutdoorTemp .
```

Code 4.8: Example lighting and temperature representation

4.4.8 Task and Activity

As mentioned before, the CoDAMoS ontology, moreover, defines concepts for user tasks and activities in its ontology. An `env:Activity` represents the action a user is performing at a certain moment. This is for example "e-mail correspondence". In order to find out the current activity of a user, there has to be a listener on active applications.

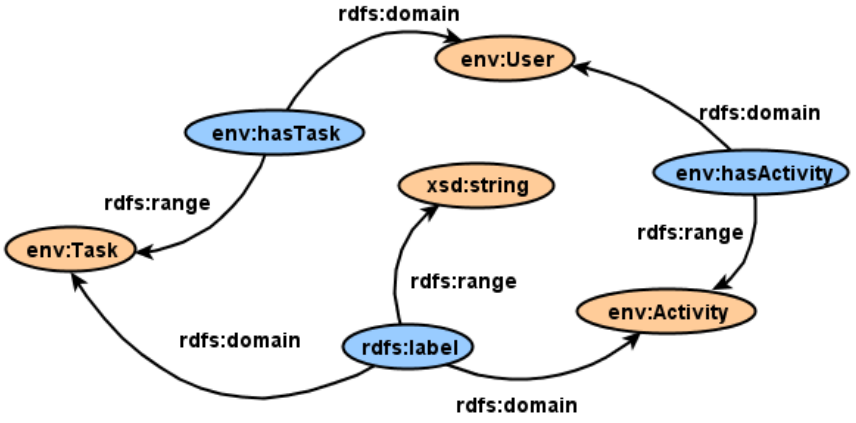

Figure 4.8: RDF-Schema of task and activity entities

Furthermore, applications have to provide metadata on what there purpose is, to be able to extract information about the user's current activity. An `env:Task` describes, for example, a task that has to be done from a to-do list. Tasks and activities always

have a human readable title, which is represented by the property `rdfs:label` and a value from type string. An example that represents a user's activities and tasks is shown in the following:

```
1 <http://www.myns.at/users/me> rdf:type env:User;
2         env:hasActivity <http://www.myns.at/activities/6545684894894654>;
3         env:hasTask <http://www.myns.at/tasks/6505684894894671> .
4 <http://www.myns.at/activities/6545684894894654> rdf:type env:Activity;
5         rdfs:label "e-mail correspondence"^^xsd:string .
6 <http://www.myns.at/tasks/6505684894894671> rdf:type env:Task;
7         rdfs:label "prepare meeting for tomorrow"^^xsd:string .
```

CODE 4.9: Example activity and task representation

4.4.9 Web Page

In order to gain information about users' interests respectively information needs (in particular situations), the browsing behaviour should also be represented in the ontology. For this purpose, new concepts have to be introduced.

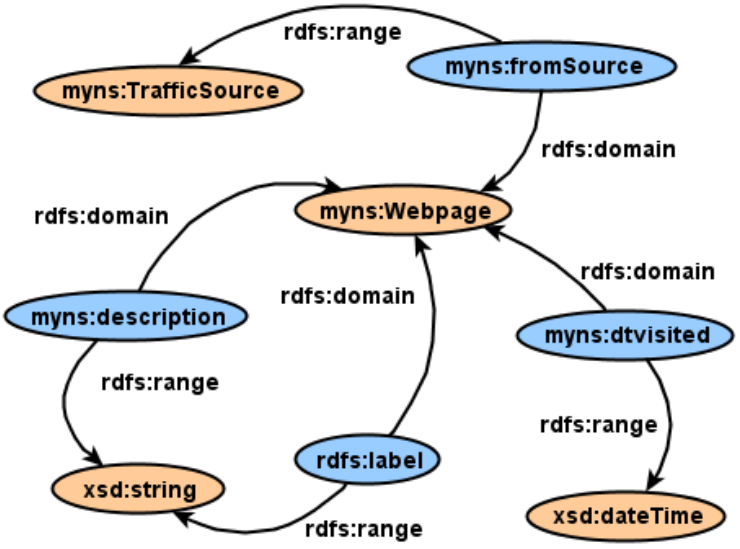

FIGURE 4.9: RDF-Schema of web page entities

We defined a class `myns:Website`, with the properties `rdfs:label`, `myns:description`, `myns:dtvisited`, and `myns:fromSource`. The values of the properties `rdfs:label` and `myns:description` are from type string. The objects of the property `myns:fromSource` have to be from type `myns:TrafficSource`, whereas a traffic source can either be *direct* (`myns:DirectTraffic`), *search* (`myns:SearchTraffic`), or *referral* (`myns:Referral-Traffic`). Values of the property `myns:dtvisited` have to be from type date-time. An example is shown in the following:

```
1  <http://www.google.com> rdf:type myns:Webpage;
2      rdfs:label "Google"^^xsd:string;
3      myns:description "Suchmaschine"^^xsd:string;
4      myns:dtvisited "2012-03-02"^^xsd:dateTime;
5      myns:fromSource myns:DirectSource .
```

CODE 4.10: Example web page representation

The property `rdfs:label` adds a human readable label to the website URL. With `myns:description` a human readable description for the web page can be defined. By means of `myns:dtvisited` the date and time when the page was visited can be stored. `myns:fromSource` indicates whether the page was reached via a link from another page (`myns:ReferralTraffic`), via a search engine (`myns:SearchTraffic`) or directly by typing it into the browser's address bar (`myns:DirectTraffic`).

4.4.10 Application

For context-aware adaptation in mobile environments, for some scenarios it may be important to know, when which application is executed. Therefore, this information is stored in the ontology. The following classes and properties were introduced for this purpose.

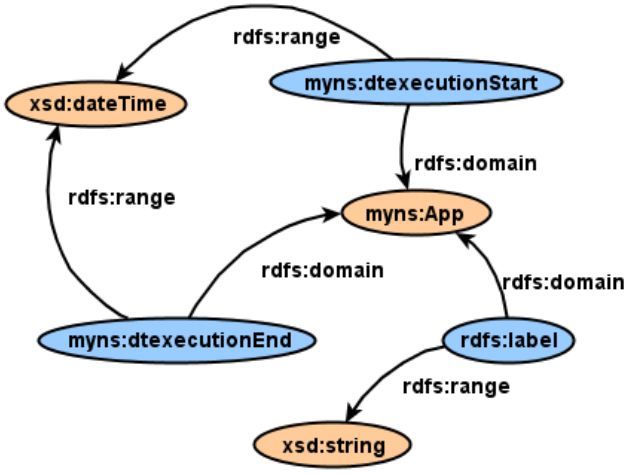

FIGURE 4.10: RDF-Schema of application entities

With `myns:App` we represent applications, which can have the properties `rdfs:label`, `myns:dtexecutionStart` and `myns:dtexecutionEnd`, whereas `rdfs:label` associates a human readable label, represented as a literal of type string, to the resource and `myns:dtexecutionStart` respectively `myns:dtexecutionEnd` represent the start/end

date and time, which are represented as literals from type date-time. Therefore, metadata from active applications have to be extracted. One possible example could be the following:

```
1 <http://www.myns.at/apps/345739458345> rdf:type myns:App;
2     rdfs:label "Google Maps"^^xsd:string;
3     myns:dtexecutionStart "2012-03-02T20:14:57"^^xsd:dateTime;
4     myns:dtexecutionEnd "2012-03-02T20:41:08"^^xsd:dateTime;
```

CODE 4.11: Example application representation

4.4.11 User Preference

Furthermore, we have to notice certain user preferences. For example, in which situation the user uses the vibration mode or the silent mode. This is important for applications that aim to support the user in making different phone settings in specific user situations automatically. Therefore, we reuse and extend the vocabulary of CC/PP.

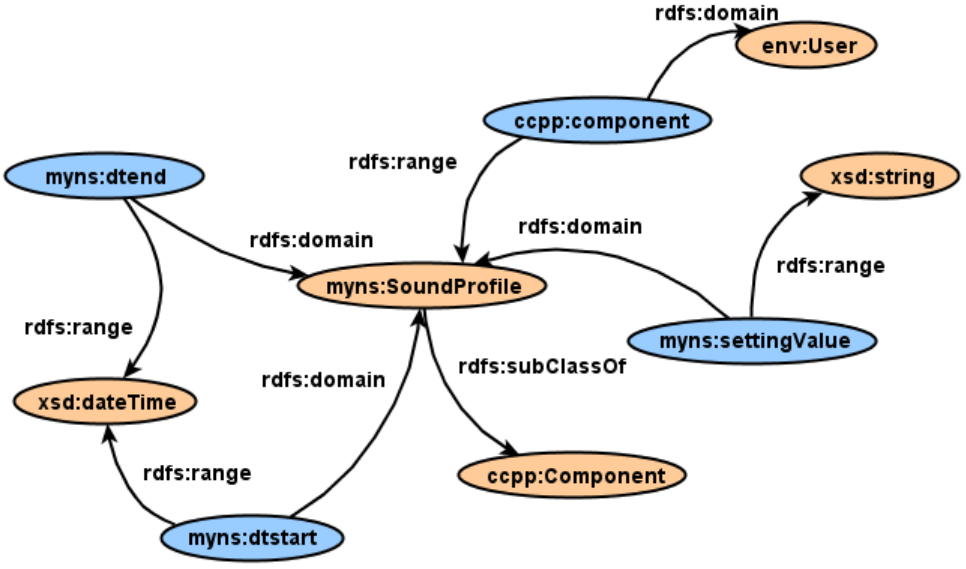

FIGURE 4.11: RDF-Schema of user preference entities

`myns:SoundProfile` indicates the type of the setting that is described. It can have various types that represent for example the "sound profile" setting or "Wi-Fi" setting, which are subclasses of `ccpp:Component`. Every type of setting has at least a `myns:settingValue` according to its possible setting options, which is of type string. In order to deduce preferred user preferences in different situations, the dates and times when the setting is changed are important. These are represented by `myns:dtstart` and

`myns:dtend`, which are represented as literals of type date-time. In the following, there is an example for the sound profile setting:

```
1 <http://www.myns.at/profile/user1> ccpp:component myns:SoundProfile .
2 <myns:SoundProfile> rdfs:subClassOf ccpp:Component;
3     myns:settingValue "silent"^^xsd:string;
4     myns:dtstart "2012-03-15T14:52:12"^^xsd:dateTime;
5     myns:dtend "2012-03-15T18:40:56"^^xsd:dateTime .
```

CODE 4.12: Example setting representation

4.4.12 Situations

Our system should furthermore be able to recognize simple user situations and represent them in the context model. A simple abstraction for this purpose would be the following situations: "at work" and "spare time", whereas "at work" can be further divided into "business meeting" and "available at work". "Spare time" may be divided into "at home" and "en route". For each time-related context/data item a statement indicating the current situation of the user may be added by means of the property `myns:inSituation`.

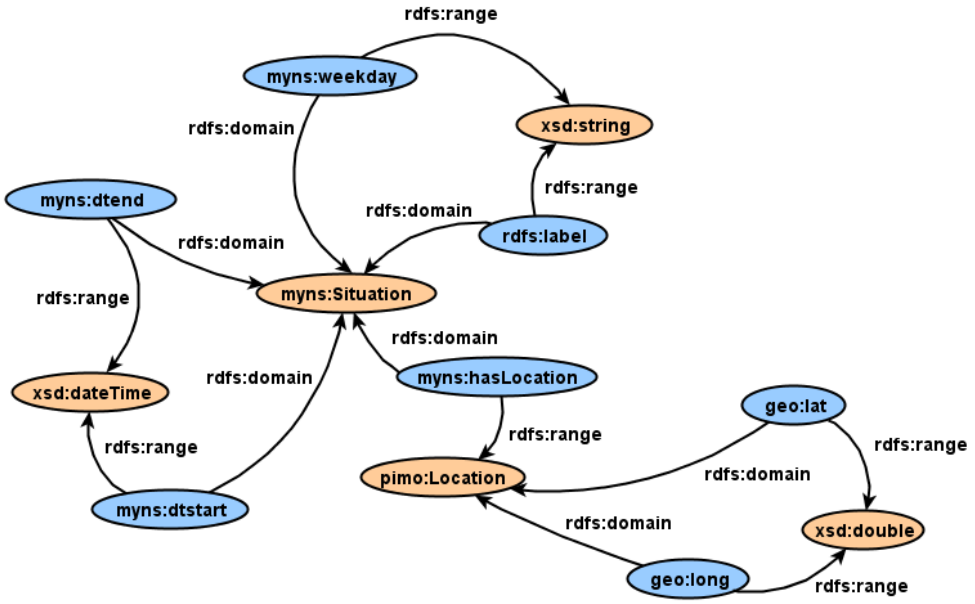

FIGURE 4.12: RDF-Schema of situation entities

To find out whether the user is at work or has spare time, the user could either tell the system the coordinates of their place of work, if they work always at the same place. Another possibility to identify a working situation may be a period of time in which the user is always at work. Whenever the user is not "at work", it is assumed that they have spare time. If the user is at work and there is an event in their calendar at

a certain time, it is expected that the user is in a meeting at this time, otherwise they are available. As with the situation "at work", also the situation "spare time" can be recognized by the coordinates of the user's home. Whenever the user is, for example 100 meters, around this coordinates, they are assumed to be at home, otherwise they are "en route".

```
<http://www.myns.at/situations/situation1> rdf:type myns:Situation;
    rdfs:label "at work"^^xsd:string;
    myns:weekday "Monday"^^xsd:string;
    myns:weekday "Tuesday"^^xsd:string;
    myns:weekday "Wednesday"^^xsd:string;
    myns:dtstart "2013-03-15T09:00:00"^^xsd:dateTime;
    myns:dtend "2013-03-15T13:00:00"^^xsd:dateTime;
    myns:hasLocation <http://www.myns.at/locations/346418922> .
<http://www.myns.at/locations/346418922> rdf:type pimo:Location;
    geo:long "-122.084095"^^xsd:double;
    geo:lat "37,422006"^^xsd:double .
```

Code 4.13: Example situation representation

4.4.13 Origin of Data

All data that comes from hardware sensors, furthermore, have to include a statement about their origin. This means, all these resources have a property `myns:originatesFrom` with a value from type `myns:Origin`, which represents the sensor itself.

4.4.14 Overall Model

In Figure 4.13 we combined all the individual models of this section and show how they are interlinked with each other.

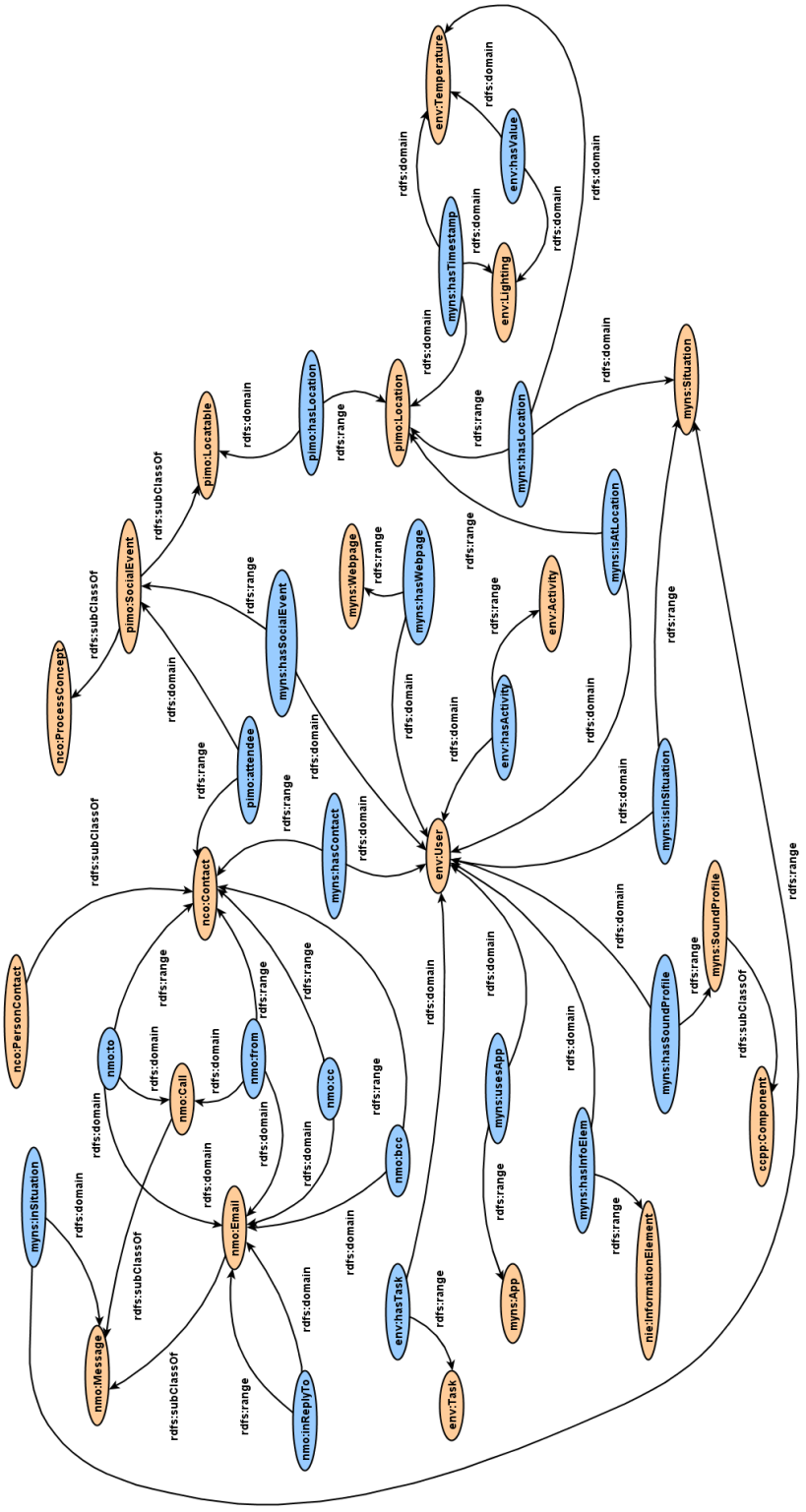

FIGURE 4.13: RDF-Schema of the overall context model

4.4.15 Example

The graph in Figure 4.14 shows part of an example instance context model that may result from the application scenario described in Section 4.2.2. There is one resource of type `pimo:SocialEvent` which is associated with an instance of `nco:PersonContact` and an instance of `pimo:Location`. The location's property `myns:locationType` indicates that the location is from type `myns:Supermarket`. `myns:Supermarket` maybe defined as a place where one can buy foodstuff and beverages. The resource of type `pimo:SocialEvent` has a property `myns:locationNeeded` with the value `myns:Supermarket` too. This means that for this entry to complete a location of type `myns:Supermarket` is necessary.

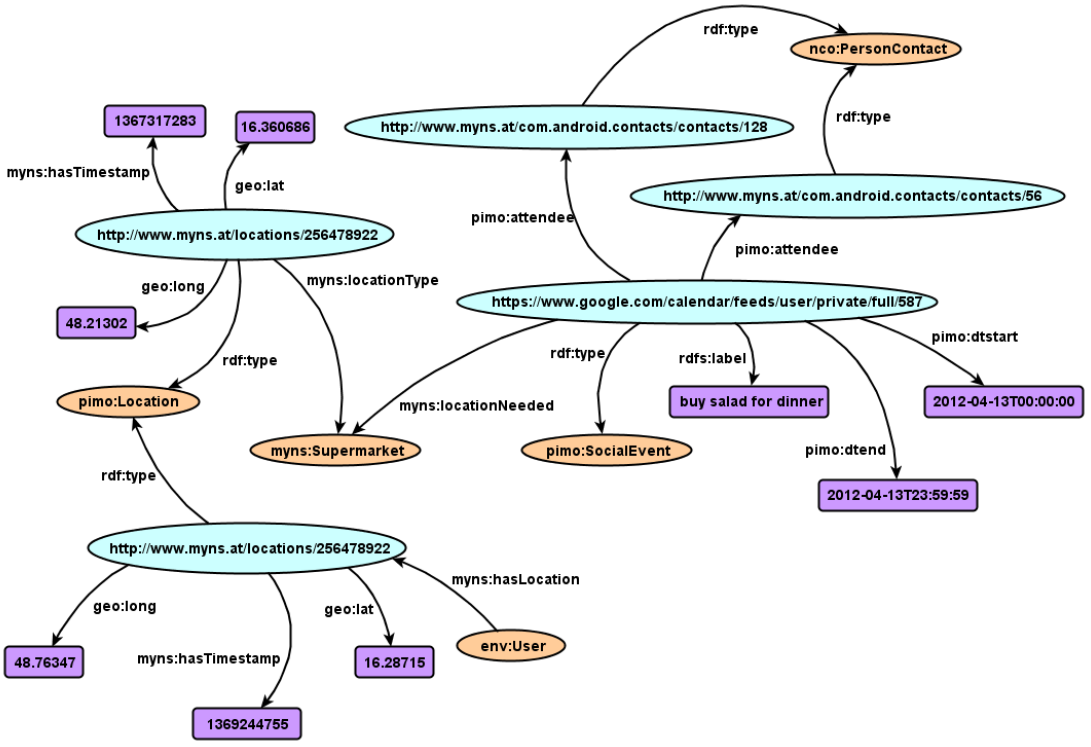

FIGURE 4.14: RDF-Schema of situation entities

4.5 Architecture

In this section, the architecture of the situation-aware mobile Semantic Desktop regarding its individual components is described. We especially consider the above described requirements and design considerations. At the end of this Section we exemplify the functionalities of the individual components by means of two scenarios introduced in Section 4.2. Concrete implementation details, are described in Chapter 5.

4.5.1 Architecture - Overview

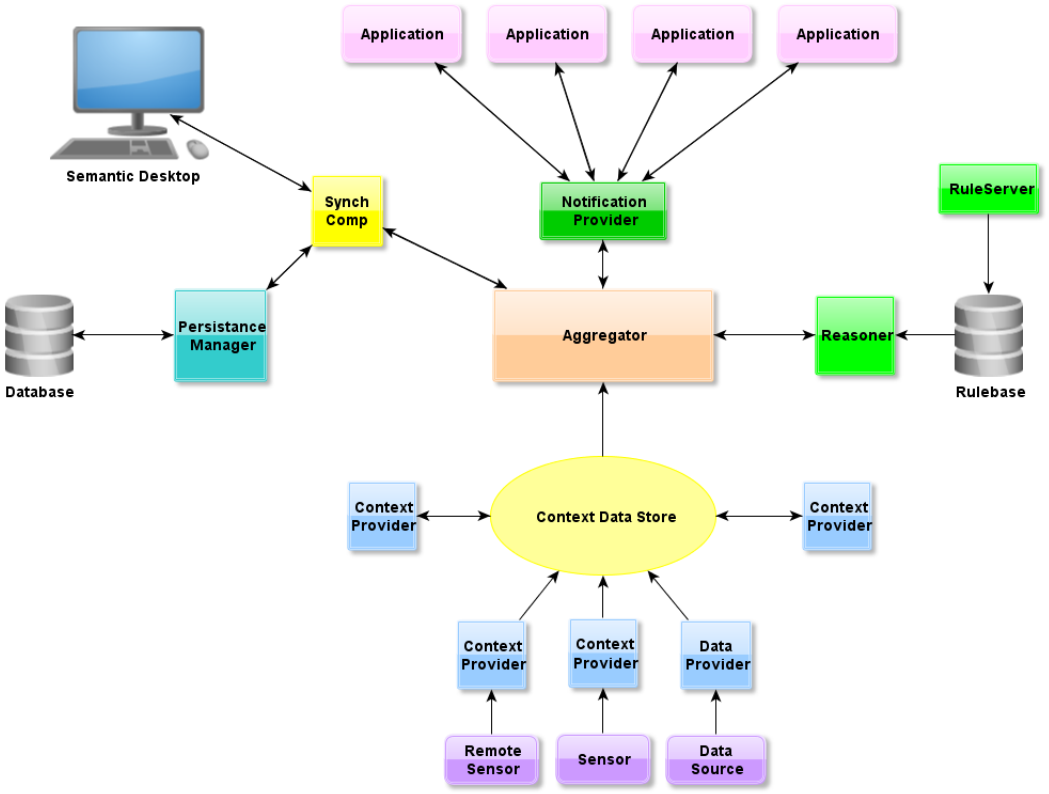

FIGURE 4.15: Architecture: overview of components

As shown in Figure 4.15, the architecture is very modular. The different modules communicate over defined interfaces (cf. Chapter 5). Hence, single modules can easily be exchanged without affecting the others, provided that they implement the desired interfaces.

On the lowest level, there are all the different *sensors* respectively *data sources* with its corresponding *context* and *data providers* (cf. Figure 4.15). Each of them works independently. *Context providers* and *data providers* process the data they get from the sensor respectively data source to which they are attached and create a context model using these data. More information about sensors, data sources, context and data providers is given in Section 4.5.2 and 4.5.3.

The resulting models are then added to the *context data store*. Context models, which are delivered to the context data store may be used by further context providers. When data processing is finished within the context data store, the models are transferred to the aggregator component. The *aggregator* merges all models from the different providers and delivers the whole model to the *persistence manager*. Furthermore, the aggregator component provides data for the *reasoner*. In order to persist the contextual data, a

local, lightweight database is used. Additional information about the user's context can be deduced by means of a rule-based reasoner, which communicates with the rule base that stores all defined rules used by the reasoner. The *reasoner* transfers its results back to the aggregator component. Detailed information about rules and reasoning is given in Section 4.3.1.

Applications have the possibility to subscribe to particular context change events. That means, they get a notification from the *notification provider* whenever a change event occurs for which they have subscribed. Detailed information about subscription and notification is given in Section 4.3.1.

All components of the infrastructure are located on the mobile phone, hence there is no need for an external server that manages or stores data. However, it is possible to synchronize the data of the mobile Semantic Desktop with the one on a server or the home system, whenever a connection can be established. To keep them synchronized, a *synchronization component* is used, which also synchronizes the database with the aggregation component (cf. Section 4.5.7). It is not necessary that any context/data provider delivers data to the aggregator, but the system does not make sense until there are at least two active context/data providers.

4.5.2 Sensor/Data Source

Usually a *sensor* is described as a physical component that measures a physical variable like temperature, illumination, sound, or pressure [itw]. However, for this work a sensor can either be a conventional hardware sensor located on the mobile phone itself or be an external/remote sensor. It can also be a web service. In contrast to a sensor, a *data source* provides data items that affect personal information management like e-mails, files, or contacts to the mobile Semantic Desktop. Each sensor respectively data source is attached to its own *context/data provider* (cf. Section 4.5.3), which processes the acquired data and creates an RDF model. Therefore, it generates RDF statements using the raw sensor data. The resulting models are then delivered to the context data store (cf. Section 4.5.4). There are sensors/data sources which send the updated data directly to their attached providers and some which only notify the providers when updated data is available.

4.5.3 Context/Data Provider

A *context/data provider* listens to changes in its underlying sensor/data source and creates RDF statements of the data acquired from it. If a sensor/data source just

notifies about changes, the attached provider has to fetch data from it before it is able to process them. Each provider builds an individual model, according to a predefined schema that depends on the type of data and is defined by the system and the developer of the context/data provider (cf. Section 4.3.1). In the end, the resulting model is passed to the context data store (cf. Section 4.5.4). A context/data provider is either attached to one particular sensor respectively data source or exists without an attached component. This can be decided dynamically, when including new components. These components implement the requirement of *context representation* which is mentioned in Section 4.1). Context/data providers with no attached component are so-called *reactive context providers*. They react to changes in the context data store and infer new data, embodied in RDF statements. An example for a reactive context provider would be one that turns geographical coordinates into textual address information (city, street, ...) every time the location context provider delivers its data. Whenever a new sensor respectively data source is included into the system, there has to be a new context/data provider as well. Every additional context/data provider has to be installed and registered by the user. Therefore, an additional installation component is used.

4.5.4 Context Data Store

In the *context data store* all instance data from context and data providers are buffered till they are delivered to the aggregation component (cf. Section 4.5.5). These models can be used by reactive context providers (cf. Section 4.5.3) in order to infer new data. They fetch a data model and lock it so that no other provider has access during the processing. If there arises an updated model of the same type in the meanwhile, the context provider gets informed, discards its previous computations, and starts again with the new model (cf. Section 4.3.1). When no context provider is able to infer new data any more, that means when no change happens for a certain time, the model is delivered to the aggregation component. Only the most recent model of one type is delivered, the others are discarded. This may result in a lack of completeness. However, it is not necessary nor useful to keep every short-term change in context.

4.5.5 Aggregator

The *aggregator* obtains all context models from the context data store (cf. Section 4.5.4) and aggregates them to one uniform model, by removing redundant statements and combining the standalone models. These processes are further described in Section 4.3.1 and enable *personal information management* (cf. Section 4.1). It is connected to the synchronization component (cf. Section 4.5.7), which is responsible for synchronizing

the database with the aggregated overall model in the main memory. Furthermore, it provides the data from the context model to a rule-based reasoner (cf. Section 4.5.10) that deduces higher-level information from low-level information (cf. Section 4.3.1). The reasoner is always listening for changes in the model and checks all its rules, every time a change happens. A *change* in the aggregated model is detected by a listener whenever the number of statements changes, i.e. a statement is either added or removed, or when a statement was updated. Moreover, it provides information to the notification provider (cf. Section 4.5.12), which is responsible for notifying applications about changes in the context. Every time a model is added to the aggregated model, the aggregation component sends a notification, in the form of a flag, to the notification provider, which tells it what kind of data had changed. Whenever a new sensor respectively context provider is added to the system, the aggregator has to be informed about the kind of data it provides in order to create a new flag for change event notifications. However, it is not necessary that data from every context provider that is registered is included in the model, for example if one sensor is disabled or currently not used, for example a brightness sensor that is disabled when the display is turned off. During the time when the aggregated model gets updated, there exist two versions of the model, the one that is currently getting updated and a "finished" one, whereas the latter is used for applications that are fetching data from the model during the update process.

4.5.6 Persistence Manager

The *persistence manager* is connected to the database and the synchronization component (cf. Section 4.5.7), which is responsible for keeping database and aggregation component (cf. Section 4.5.5) synchronous. It writes the aggregated context model to the database. The model is saved in the database at regular intervals, when the current workload is under a certain threshold, and before the mobile phone is turned off or the system is turned off. Furthermore, it provides data from the database for the aggregator. The aggregation component expresses its information needs by means of SPARQL queries.

4.5.7 Synchronization Component

The *synchronization component* is responsible for keeping database (cf. Section 4.5.9) and aggregator (cf. Section 4.5.5), as well as Semantic Desktop (cf. Section 4.5.8) and aggregator synchronous. In order to synchronize the aggregator with the database, all statements that exist in the "old" model, but do not exist in the "new" one are deleted from the database and all statements that only exist in the "new" model are inserted into

the database. A synchronization with the Semantic Desktop component is only possible when a connection to the Semantic Desktop on the home PC can be established.

4.5.8 Semantic Desktop Component

The *Semantic Desktop* component represents a Semantic Desktop on a home PC or a server. It is possible to synchronize a conventional Semantic Desktop with the mobile one via a synchronization component (cf. Section 4.5.7). The first time a synchronization happens, this has to be done manually because there has to be an ontology mapping. Furthermore, the resource IDs as well as the addressing scheme have to be unified. After the first manual synchronization, subsequent synchronizations can be performed automatically using the mapping that was created the first time. When synchronizing two sets of data, it is likely that conflicts occur. For example, two contacts have different IDs but the same telephone number or e-mail address. In this case, there are different approaches for dealing with conflicts between the two Semantic Desktops [KC]:

- *originator win* The RDF statement of the originator is taken.
- *recipient win* The RDF statement of the recipient is taken. In our case, the recipient is the device which initiates the synchronization.
- *client win* The RDF statement of the client device is taken. For this work, the client is the mobile Semantic Desktop.
- *server win* The RDF statement of the server is taken. For this approach, the server is the Semantic Desktop on the home PC or server.
- *recent data win* The RDF statement which was modified last is taken.
- *duplication* Both RDF statements are kept.

The user is able to choose their favourite strategy. The point in time when a synchronization takes place can be decided by the user either manually or by setting a synchronization interval. It is also possible to set fixed rules, like "synchronize every time a connection to a WIFI gets established", in order to automatically initiate synchronization.

4.5.9 Database

The *database* stores the aggregated context model in a persistent way. Therefore, the two components have to be synchronized (cf. Section 4.5.7). That means the synchronization

component gets data either from the aggregator or from the database and compares them with each other. In the database, all RDF statements including context history and ontology description are stored and can be accessed at any time because its located on the mobile phone itself. The database enables the realization of *providing historical context data* (cf. Section 4.1).

4.5.10 Reasoner

The reasoning component (called *reasoner*) performs inferencing based on rules stored in the rule base (cf. Section 4.5.11). Therefore, it listens to changes in the aggregated model and goes through all the rules stored in the rule base whenever the model changes to see if it is able to infer new statements. For this approach, forward chaining (cf. Section 2.3.2) is used that means it loops through all inference rules as long as it can find one where the antecedent is true. Every time such a rule is found, the new data is inferred and the process continues until no rule can infer new data any more. The resulting statements of this process are added to the overall model as well. The reasoner can be used to facilitate *situation identification* (cf. Section 4.1). More detailed information about the reasoning process is given in Section 4.3.1.

4.5.11 Rulebase

In the *rule base* all inferencing rules are stored persistently on the device. They are predefined by the system. The rules are executed by the reasoning component using the forward chaining algorithm. As a syntax for the rules Jena syntax is used. Further information about reasoning rules is given in Section 4.3.1.

4.5.12 Notification Provider

The *Notification Provider* is the connector between applications (cf. Section 4.5.13) and mobile Semantic Desktop. It is able to communicate with the *Aggregator* via local service invocation as well as with applications via remote service invocation (cf. Section 5.1.2). The aggregator component tells the notification provider whenever a change has happened and what change has happened in the overall context model, by sending an appropriate message (cf. Section 4.3.1). Applications can subscribe to arbitrary context change events in order to get a notification whenever this particular context has changed. A context change event is either the change of a particular context instance, for example *location* or *lighting*, or an application-defined SPARQL query. This means, applications can register a SPARQL query, which they would like to have monitored.

Every time the result of the query changes, the registered application gets informed. For the management of subscriptions and notifications, the notification provider is responsible (cf. Section 3.1). It listens for changes in the context model and sends notifications to applications after it received the information from the aggregation component. Further information about subscription and notification is given in Section 4.3.1. Furthermore, applications have the possibility to send so called "on demand" SPARQL queries to the system. These are also managed by the notification provider. It fetches the result set from the context model and sends it back to the application from which the request came from. If applications like to add RDF statements to the overall context model of the system, they have to do this via the notification provider as well. They send their statements to the notification provider, which verifies if the provided data is contrary to already existing data and adds it to the model if not. More information about "on demand" SPARQL queries and adding RDF statements regarding privacy are given in Section 4.3 and 4.3.1.

4.5.13 Application

Applications, which are based on the mobile Semantic Desktop infrastructure, only communicate with the system via the notification provider (cf. Section 4.5.12). As mentioned in Section 4.5.12, the notification provider manages subscriptions from applications. When a change happens, all applications that have subscribed to the event get a notification as well as the new data. Furthermore, applications are allowed to add RDF statements to the system's context model and get information on demand by sending SPARQL queries.

4.5.14 Examples

In the following, we exemplify the content of this section on the basis of the two scenarios introduced in Section 4.2. It will be described what components of the described system play a role in the example scenarios and what they are responsible for.

- *Annotate/Group data items* For this function different data sources are needed, the more the merrier. All of them have to have a data provider associated which creates RDF models out of the acquired data. The resulting models are then delivered to the *Context Data Store* which has nothing to do for this example. Hence, the models are forwarded to the *Aggregator* which interlinks the standalone models to one overall context model. The application uses the *Notification Provider* to add the RDF statements regarding the classification of data items into different

subjects. It sends the statements to the notification provider which integrates them into the aggregated model. For retrieving data associated to particular subjects, the application sends a SPARQL query to the notification provider which then sends the result of the query back to the application. The application, then, shows them to the user.

- *Remember locations* For this application scenario at least a location sensor and a calendar entry data provider are needed. They send their acquired data to their associated context/data provider, which creates RDF models that represent the sensors' data. Afterwards the models are delivered to the *Context Data Store*. For this scenario nothing will happen in this component. The models are directly forwarded to the *Aggregator* which combines the two standalone models in order to achieve one overall model. The application uses the *Notification Provider* to subscribe for changes in the context model. As mentioned in Section 4.3 this application has to subscribe for location changes and calendar entry changes. Furthermore, it has to execute various on-demand SPARQL queries in order to check if the user is near an annotated location. Whenever the user annotates a location, the application generates RDF statements representing longitude and latitude as well as type of location and timestamp and sends them to the notification provider, which is responsible for adding them to the aggregated model. If the notification provider recognizes changes in the context model regarding location or calendar entries, it notifies the application and sends the changed pieces of data to it. Afterwards, the application fetches all annotated locations as well as all events that are tagged with such a location and start within the next two days. Then, it checks if the user is near a location of a tagged event that starts within the next two days. If so, the application notifies the user.

4.6 Graphical Representation of Scenarios

In this section, we provide activity diagrams depicting the course of action of the two use cases initially described in Section 4.2.1 and 4.2.2. They represent all tasks that have to be performed by user, application and system in order to accomplish the use cases. They should furthermore illustrate the interactions between our system and an extending application.

Yellow nodes represent actions executed by the user, blue nodes show tasks that are performed by the application implementing the described use case, and green nodes represent activities executed by our system. Although the system may already run before the user starts the application, we assume this action as a starting point.

4.6.1 Annotate/Group data items

Figure 4.16 shows the activity diagram for the *annotate/group data items* use case.

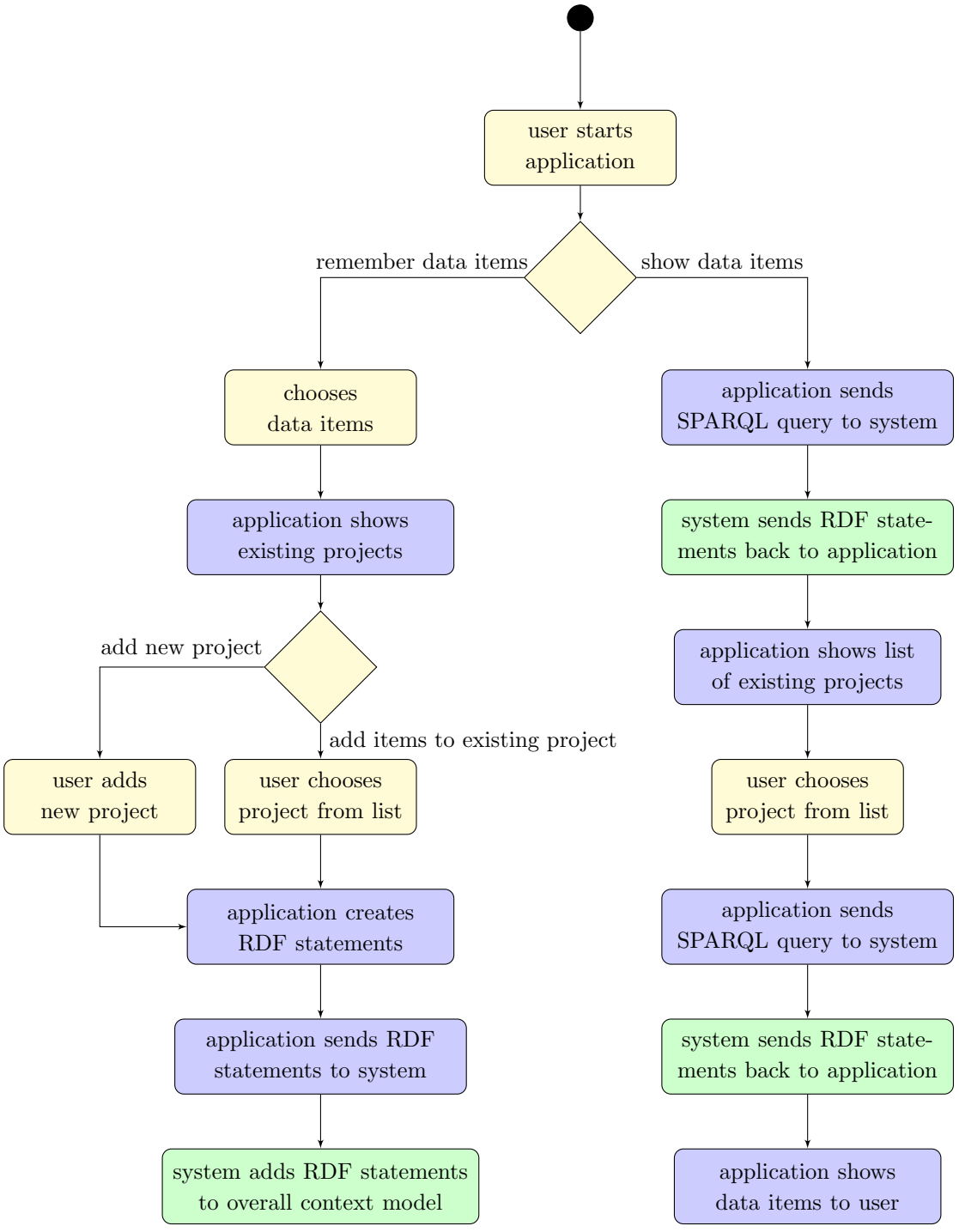

FIGURE 4.16: Activity diagram for scenario "Annotate/group data items"

If the user wants to see the data items stored in a specific project (right branch in Figure 4.16), first the application has to send a SPARQL query to the system in order to receive all existing projects. Then, the application shows them to the user and they select one

project. After selecting a project, the application sends another SPARQL query to the system in order to retrieve all data items related to the chosen project. Afterwards the application shows all of them on the display.

If the user wants to store data items in a new or already existing project (left branch in Figure 4.16), they first choose a data item they would like to add and tell the system to add this item to a project. Therefore, they have to either choose an existing one from the list or add a new one first. Then the application creates RDF statements and sends them to the system which adds them to the context model.

4.6.2 Remember locations

Figure 4.17 shows the activity diagram for the *remember locations* use case. Once the application is started, it automatically subscribes for location and calendar change events. From that time on, the user is able to annotate locations and adding calendar entries using them. If the user annotates a location, the application creates RDF statements, sends them to the infrastructure which adds them to the overall model and notifies the application that something regarding location has changed in the context model.

After receiving the notification, the application fetches all annotated locations as well as all events using such a location and starting in less than two days. Then, it calculates the distance between the user's current location and all locations used in events. If the linear distance is less than one kilometer, the application notifies the user.

Every time the user changes their location or adds calendar entries (including annotated locations or not), the corresponding *Context Providers* creates RDF statements and delivers them to the *Aggregator* which adds them to the overall context model. Then, the *Notification Provider* notifies the application about the change and the application starts again with fetching annotated location and calendar entries, calculating distances and notifying the user if they are near a relevant location.

The shown activity diagram has no final state because application and system are always running in background as long as they are not terminated by the user.

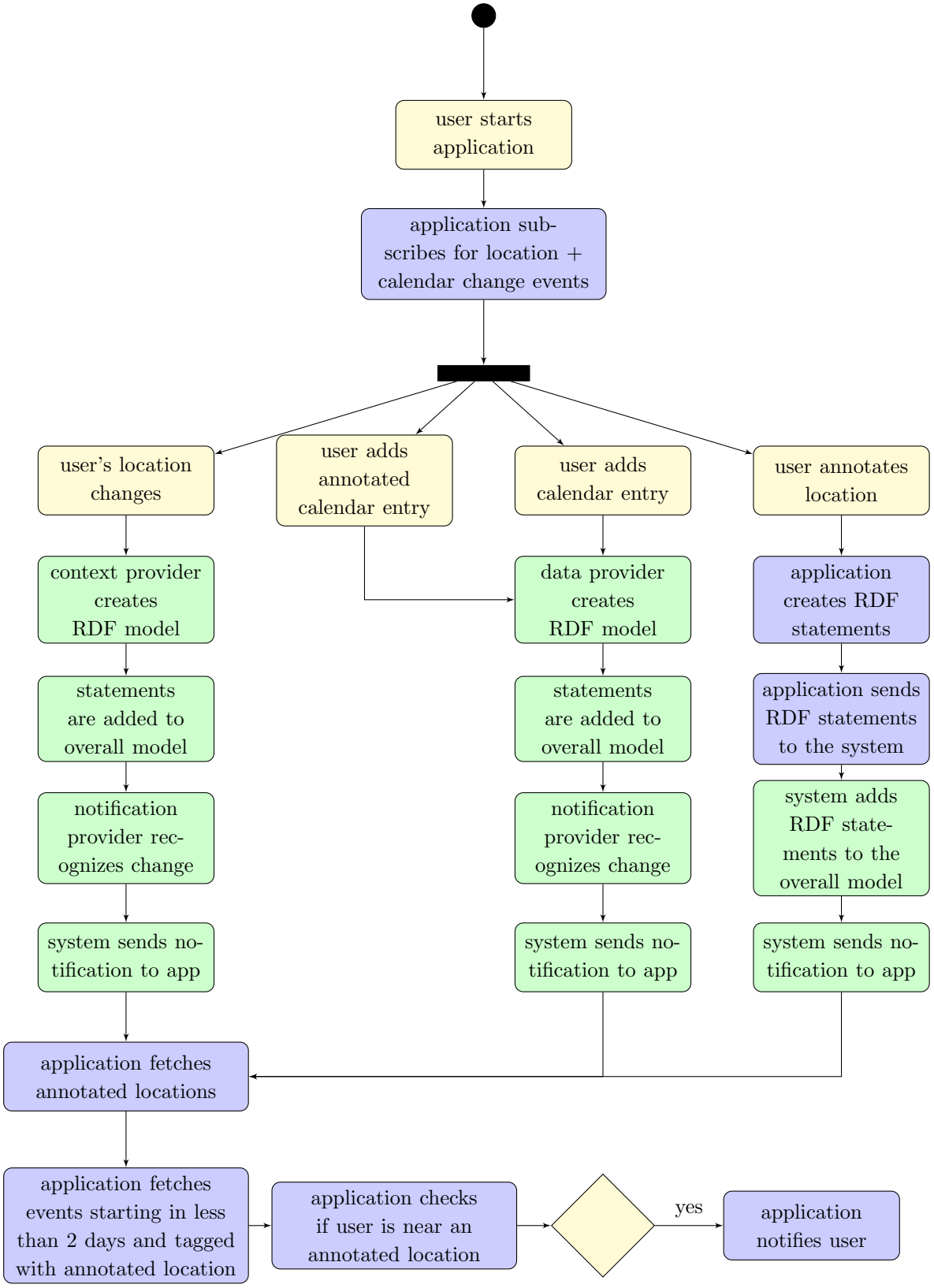

FIGURE 4.17: Activity diagram for scenario "Remember locations"

4.7 Summary

In this chapter, we first discussed the main characteristics of our mobile Semantic Desktop infrastructure's approach. We stated that it is important for our approach to support personal information management and provide context history in order to facilitate reasoning. Furthermore, we decided to use a subscribe-notify mechanism as a communication channel between the infrastructure and applications that extend it. We aimed to build a context model schema for representing data items as well as contextual information. Moreover, we decided to use a reasoning mechanism in order to facilitate situation identification and to provide mechanisms that enable users to individually define privacy principles on their own.

Afterwards, we introduced two use cases that are enabled by the proposed system and used them to exemplify our approach in each section.

Then, we described the mobile Semantic Desktop infrastructure itself. Therefore, we first functionally divided the different processing steps into layers and described each of them. Afterwards, we described the most important aspects of the approach in more detail. These are data acquisition, aggregation, consolidation and interlinkage of data, provision of context history, the subscription and notification mechanism, and privacy mechanisms.

We, moreover, introduced the design of an ontology-based representation schema for contextual information and data items that exist in a Semantic Desktop environment on mobile phones which makes use of already existing ontologies, like PIMO and the NEPOMUK ontologies. However, not every piece of information that is important for this approach can yet be represented with existing vocabulary, hence we also had to introduce some new vocabulary. With the described schema it is possible to represent both personal data items and context information and interconnect them with each other. Finally, we described the infrastructure regarding its individual components and provide activity diagrams representing the course of action for our two example scenarios.

Chapter 5

Implementation

In this chapter, the prototype implementation of a context-aware mobile Semantic Desktop infrastructure is discussed. Therefore, first some enabling components used for the prototype implementation are introduced (cf. Section 5.1, 5.2, and 5.3). In Section 5.4.1, we provide an overview class diagram including both the infrastructure's and the application's classes. The class design of the prototype implementation is described in Section 5.4.2. As a proof of concept, moreover, an application using the mobile Semantic Desktop infrastructure has been developed and its classes are described in Section 5.4.3. Finally, in Section 5.5, we discuss problems during the implementation process and limitations of the prototype.

For the prototype infrastructure a GPS context provider exposing data gained by the mobile phone's built-in GPS sensor, a calendar data provider processing data from the user's calendar, and a contacts data provider service using data from the user's phone book have been implemented. Furthermore, context data store, aggregator, and notification provider have been implemented.

The application using the mobile Semantic Desktop infrastructure implements the functionality initially introduced in Section 4.2.2. By means of this application a user can annotate locations with predefined types of locations, which are *supermarket*, *florist*, *oculist*, *general practitioner*, *ear specialist*, and *dentist*. If a user adds an annotation to a location, the application creates RDF statements representing the location's longitude and latitude and the location type. Then, it sends them to the infrastructure which adds them to the overall context model. There may be various supermarkets and florists stored in the context model, but there is only one location for each doctor type allowed.

Once there are annotated locations stored in the system, the user can use them for their calendar entries as an additional information. If the user adds a calendar entry

using an annotated location of type *supermarket* or *florist*, the application notifies the user whenever they are near a supermarket/florist, but only if the particular event starts within the next two days. If the user adds an event with a location of type *oculist*, *general practitioner*, *ear specialist*, or *dentist* and this event begins within the next two days, the application calculates the time the user needs to go to this location and sets a reminder fifteen minutes before the user has to leave in order to arrive at their appointment in time.

We paid attention to developing a system that does not negatively affect the user's work and keeps the user involvement minimal. Hence, the system runs as a background service that does not necessarily need input from the user. We chose status bar notifications for informing respectively communicating with the user.

The described system is only running on mobile phones running Google's Android operating system version 2.2 including the Android-specific data structures for contacts, the Google's calendar application, and the built-in GPS sensor. Furthermore, the system has to be granted the necessary permissions in order to be able to access the needed data.

5.1 Used Android Components

In this section the most important components that we use from Android's API for the prototype implementation are introduced and its adoption for our approach is described.

5.1.1 Android Activity

In Google's Android operating system an *Activity* is part of an application and provides a graphical user interface. Usually an application includes more than one activity and each of them can start another one. We only use one activity for our infrastructure in order to start services and another one for the application extending the infrastructure. Before an activity is started the first time, it has to be created. There are different states an activity may have. After starting the activity, it is in the *resumed* state. If another activity is in the foreground partly covering the other one, the activity which is in the background is *paused*. However, when the new activity in the foreground wholly covers the other one - i.e. it is not visible any more - the activity in the background is *stopped*. A stopped activity is still retained in memory including all state information as long as it is not destroyed by the system. Our system stops working - i.e. stops all services - if the activity is destroyed.

In order to create an Activity, the class `Activity` or a subclass of it has to be extended and callback methods need to be implemented. The only one you *must* implement is `onCreate()`, where initializations should be done and `setContentView()` must be called. Every Activity has to be declared in the manifest file, by adding an `<activity>` tag with the `android:name` attribute.

5.1.2 Android Services

"*A Service is an application component that can perform long-running operations in the background and does not provide a user interface.*" [Devd] A *service* can either be *started* or *bound*. Services can be started from other applications or components of the same application. It is running as long as it is not stopped (by `stopService()`), but usually it is stopping itself by `stopSelf()` after performing its task. On the other hand, a bound service is running as long as at least one component is bound to it. A component that is bound to a service can interact with it via a client-server interface. Generally one service can be used in both ways providing that the appropriate callback methods - onStartCommand() for started services and onBind() for bound services - are implemented. If a service is both started and bound, the methods `stopService` and `stopSelf()` are not able to stop the service as long as it is bound to a component [Devd].

Like activities, all services have to be declared in the manifest file. In order to declare a service, a `<service>` tag has to be added, including the `android:name` attribute. If the service should only be used within the application in which it is declared, the `android:exported` attribute can be set to `false`. In this work, we only use bound services, therefore we only elaborate on them, in the following [Devd].

Bound Services

An application component can bind to a service by calling `bindService()`, which takes three parameters. The first one is an `Intent` naming the service, the second is the `ServiceConnection` object, and the last is a flag, which is `BIND_AUTO_CREATE` in our case. This means, the service is only created, if it is not already alive. The only components that can bind to a service are activities, content providers, and services. As mentioned before, in order to create a bound service the `onBind()` callback method must be implemented, which returns an interface for communicating with the service. A bound service is automatically destroyed when it is not bound to any component, hence it need not be stopped manually, except it has been started before. If a component does not need the service any more, it closes the connection by calling `unbindService()`. When creating

a bound service, the communication interface has to be implemented. This must be an implementation of `IBinder`. For a bound service the following lifecycle callback methods are relevant: `onCreate()`, `onBind()`, `onUnbind()`, `onRebind()`, and `onDestroy()`. If a component binds to a service, it must implement `ServiceConnection` including the methods `onServiceConnected()` and `onServiceDisconnected()`. When the system has established the connection between the two components, `onServiceConnected()` is called. `onServicesDisconnected()` is called when the connection to the service is unexpectedly lost.

There are different ways, the interface can be defined. For our implementation we extended the `Binder` class, respectively - for the remote services - used AIDL. Extending the `Binder` class is the most common method to implement the interface. It can be used for services that are only used from the own application. Using this method allows the bound component to directly access all public methods of the service. However, the service and client must be in the same process. Using AIDL needs a `.aidl` file, which defines the programming interface that is extended within the service. The service is then able to handle requests simultaneously.

Remote Services and AIDL

Android Interface Definition Language (AIDL) is used to define a programming interface between a service and a client. "*On Android, one process cannot normally access the memory of another process. So to talk, they need to decompose their objects into primitives that the operating system can understand, and marshall the objects across that boundary for you.*"[Deva] AIDL is used, if a service should be used by remote applications and requests have to be answered simultaneously. Otherwise, the implementation of `Binder` or the usage of `Messenger` is recommended. AIDL files are written in Java syntax and have to be saved within the application in which the service is defined, as well as in the remote application. If the file is stored in a package, the packages of both applications must have the same name. The interface has to be implemented within the service and exposed by returning the implementation from the `onBind()` method. The AIDL file only includes method signatures, there cannot be static variables in it. When using AIDL, the service has to be thread-safe [Deva].

5.1.3 Android Permissions

Using Android's operating system applications do not have any permissions for operations that may affect the operating system, user, or other applications adversely. These are, for example, reading and writing of user's private data respectively files from other

applications or network access. Hence, application developers have to manifest all permissions the application needs and the user has to grant them at the time of installation. Permissions cannot be granted at runtime [Devc].

Permissions have to be declared in the `AndroidManifest.xml` using `<uses-permission>` tags. The implemented prototype needs the following permissions:

```
1 <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
2 <uses-permission android:name="android.permission.READ_CALENDAR" />
3 <uses-permission android:name="android.permission.READ_CONTACTS" />
4 <uses-permission android:name="android.permission.INTERNET" />
5 <uses-permission android:name="android.permission.GET_ACCOUNTS" />
```

Code 5.1: AndroidManifest.xml

5.2 Androjena

Androjena[1] is a porting of Jena[2] to the Android[3] platform [Goo11]. It includes all libraries of Jena and is optimized for mobile devices with less processing power than desktop computers. With Androjena it is possible to read, process and write RDF data in different serialization formats and handle OWL and RDFS ontologies [Sea10]. Furthermore, a rule-based inference engine for reasoning is included. For an implementation of a SPARQL query engine, ARQoid[4] may be used, which is based on ARQ 2.8.3 [Goo10]. If a SPARQL database is needed, TDBoid can be used, which is based on TDB library version 0.8.5[5].

There are also a few other implementations for processing RDF data on mobile devices, like Mobile RDF[6] and μJena[7]. However, according to [ZS11], Androjena is best suited for the usage on mobile phones running Android's operating system, because it provides the most functionality and is much faster with processing a large amount of triples. That is the reason why we have decided to use Androjena for the implementation of the mobile Semantic Desktop infrastructure.

[1] cf. `http://code.google.com/p/androjena/`
[2] cf. `http://incubator.apache.org/jena/`
[3] cf. `http://www.android.com/`
[4] cf. `http://code.google.com/p/androjena/wiki/ARQoid`
[5] cf. `http://code.google.com/p/androjena/source/browse/trunk/tdboid/README?r=75`
[6] cf. `http://www.hedenus.de/rdf/index.html`
[7] cf. `http://poseidon.elet.polimi.it/ca/?page_id=59`

5.3 Directions API

For the implementation of the prototype we use the MapQuest[8] Directions Web Service[9] in order to calculate the time needed to drive from the user's current location to a doctor. This web service provides directions between two or more locations/waypoints. There are up to 50 locations allowed for one HTTP request. The entire distance of all location pairs must not exceed 8000 miles linear distance. Parameters can be provided as key-value pairs, in XML, or JSON format. Mandatory parameters are *key*, *from*, and *to*. For our requests, we furthermore use the *unit* parameter which can either have the value *m* for miles or *k* for kilometers and the *outFormat* parameter which can either be *json* or *xml*. When using key-value pairs as input, the location parameters can either be a "single-line" string or longitude and latitude values separated by comma [Map]. In the following an example request is shown:

```
http://www.mapquestapi.com/directions/v1/route?key=Fmjtd%7Cluub2hur25%
2Cb0%3Do5-9ut2q4&from=48.213407,16.360248&to=48.220146,16.356072&unit=
k&outformat=json
```

From the JSON response we only use the *time* part of the *route* object, which indicates the calculated elapsed time in seconds for the route.

5.4 Class Design

In this section we describe all classes that have been implemented for the prototype of the mobile Semantic Desktop infrastructure as well as for the application prototype.

5.4.1 Overview

Figure 5.1 shows an overview of all involved classes and the relationships between them, both from the infrastructure and from the application implementation. Details regarding methods and properties are provided in Section 5.4.2 and 5.4.3.

[8] `http://developer.mapquest.com/`
[9] `http://www.mapquestapi.com/directions/`

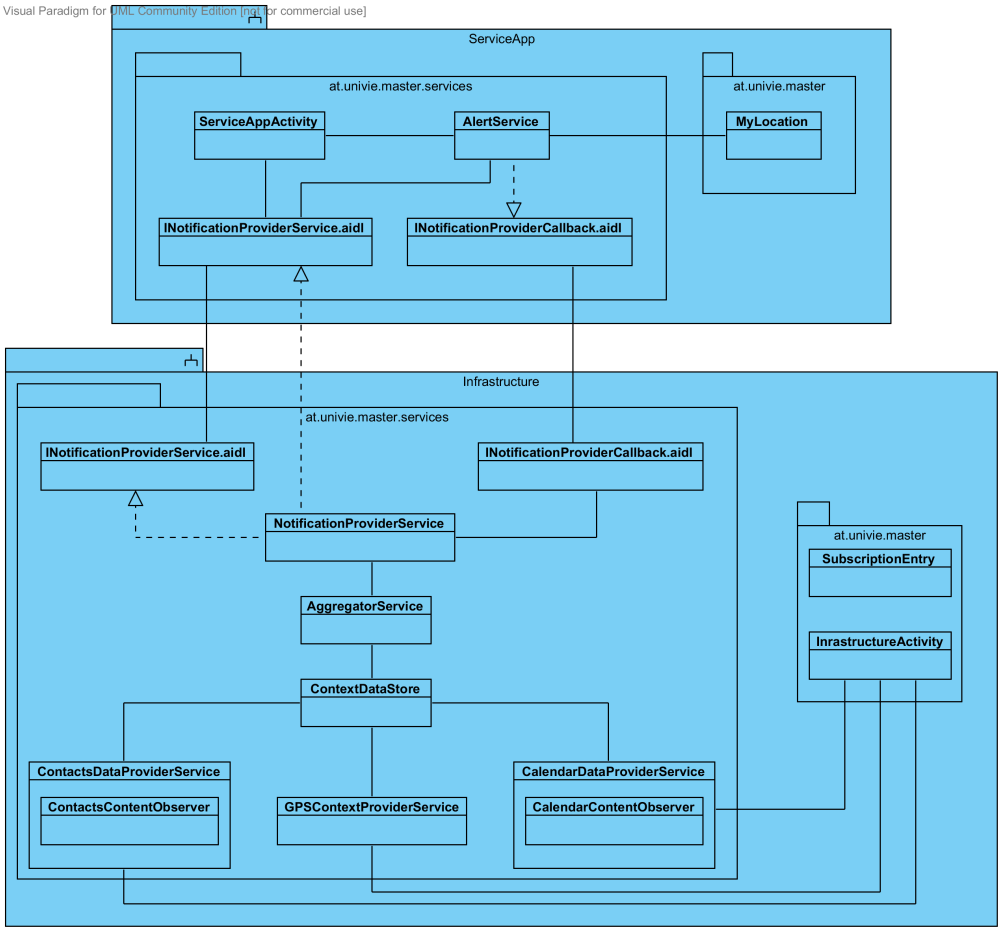

FIGURE 5.1: Class diagram of the whole implemented system

5.4.2 Infrastructure

In this section, all classes that have been implemented for the mobile Semantic Desktop infrastructure are described. Figure 5.4 shows all of them including its attributes, methods, and the relationships between them.

InfrastructureActivity

The `InfrastructureActivity` class is responsible for starting and stopping the application. Once the user taps the start button (cf. Figure 5.2), `CalendarDataProviderService`, `ContactsDataProviderService`, and `GPSContextProviderService` are started/bound using the `doBindService` method. These context providers then bind to the `ContextDataStore`. If the user taps the stop button (cf. Figure 5.3), the activity calls the `doUnbindService` method, which unbinds from all used context providers. Furthermore, the user has the possibility to manually add all events from the calendar to the infrastructure using the *Add Events* button (cf. Figure 5.3). If the user does not add

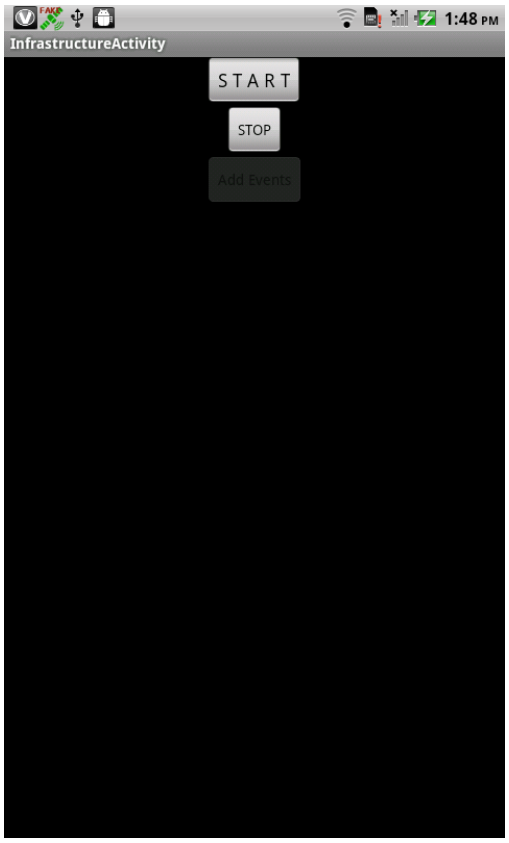

FIGURE 5.2: Infrastructure start screen

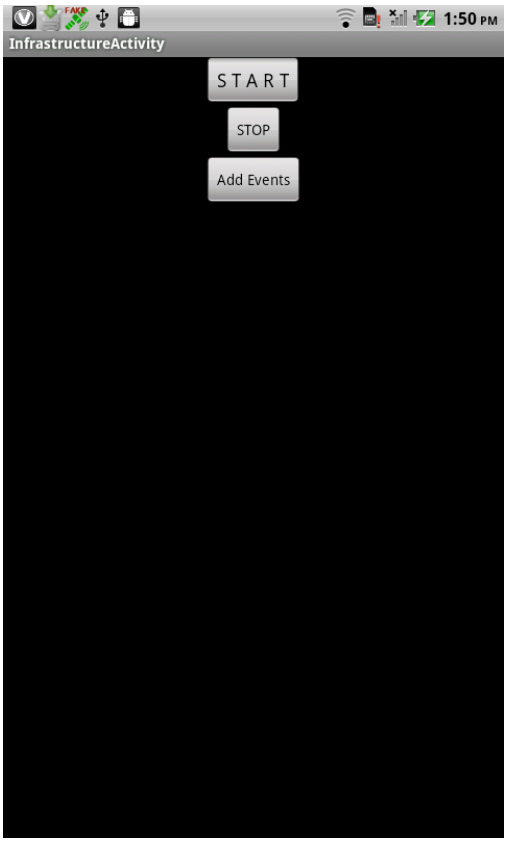

FIGURE 5.3: Screen of running infrastructure

the events manually they are automatically added when they first add/modify/delete an event in the calendar.

GPSContextProviderService

The `GPSContextProviderService` is a bound background service that receives data from a GPS sensor and processes them. If the provider is started, it creates an object of type `LocationManager`, which requests location updates. Our location manager recognizes location changes of at least 50 metres at most every ten seconds. Furthermore, the `GPSContextProviderService` binds to the `ContextDataStore` after registering the location manager. Whenever the GPS sensor notices a change in location of the mobile device, the `GPSContextProviderService` class generates RDF statements describing the new location. For this purpose, the `onLocationChanged` event handler with the parameter `location` is called. The resulting model includes statements about the latitude and longitude of the location, as well as a timestamp. Then it is delivered to the `ContextDataStore`, where it may be further processed and finally transferred to the `AggregatorService`.

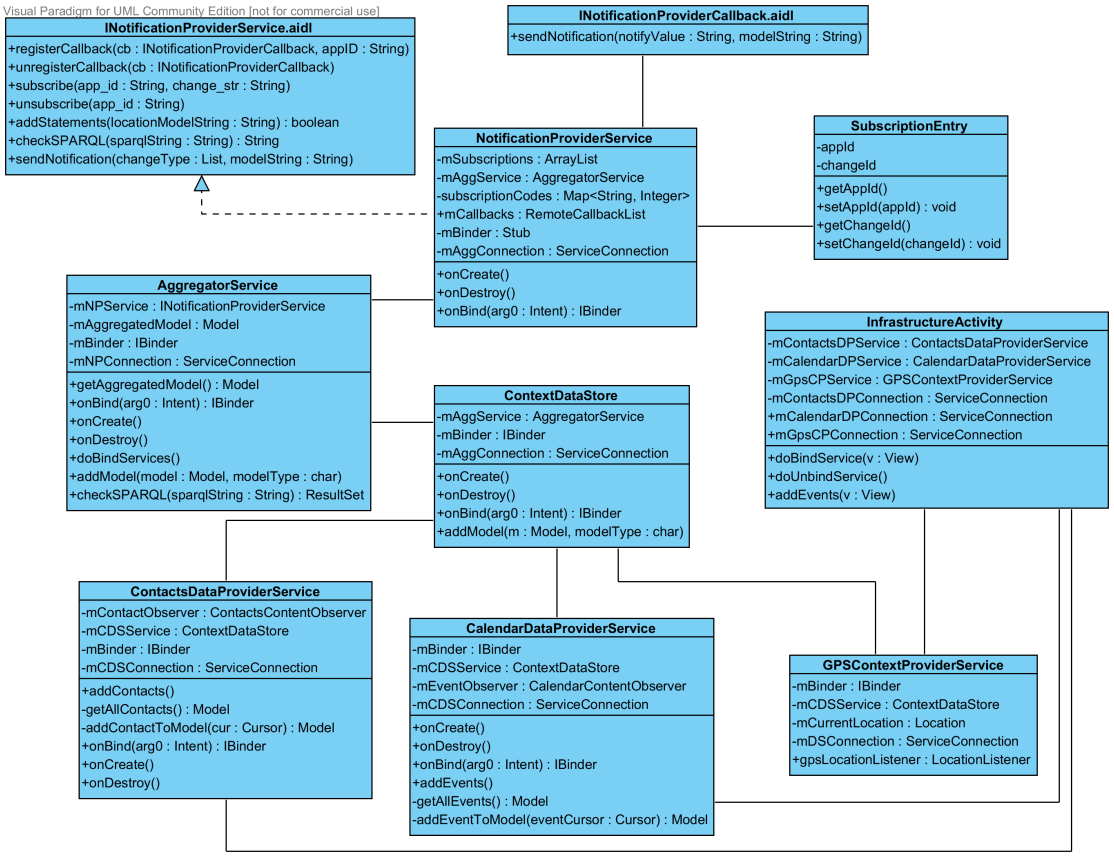

FIGURE 5.4: Detailed class diagram of the infrastructure

CalendarDataProviderService

The `CalendarDataProviderService` is a bound background service that is in charge of processing data received from the `CalendarContentObserver`. While creating the service a `ContentObserver` that observes changes in the calendar data source is registered and the service binds to the `ContextDataStore`. Whenever the `CalendarContentObserver` notices a new or edited calendar entry, it informs the `CalendarDataProviderService` class, which generates RDF statements describing all calendar entries. For this purpose, the `addEventToModel` method is called. Within this method, RDF statements providing information about people invited to the event, start and end time, as well as an event-URL and a location string, are generated. For each attendee a temporary `PersonContact` is created, which may be replaced by already existing ones with the same e-mail address in the `ContextDataStore` class or otherwise the user gets a notification which suggests them to add the contact to the address book. The result of this method is delivered to the `ContextDataStore`, where it may be further processed and finally transferred to the `AggregatorService`. The user has the possibility to add all calendar entries to the system in advance by pressing the *Add Events* button on the

`InfrastructureActivity` (cf. Figure 5.3). This triggers the `addEvents` method which again calls the `addEvents` method of the `CalendarDataProviderService`.

ContactsDataProviderService

The `ContactsDataProviderService` is a bound background service that is used for generating RDF statements for new or modified contacts. It receives information from the `ContactsContentObserver`. If the service is started, it binds to the `ContextDataStore` and registers a `ContentObserver` that listens for changes in the contacts data source. Whenever the `ContactsContentObserver` detects a new or edited contact in the mobile phone's address book, it informs the `ContactsDataProviderService`, which generates RDF statements describing all user's address book entries. Therefore, the `addContactToModel` method is called. Within this method, RDF statements providing information about people's name, telephone number(s), e-mail address(es), and postal address are represented. The resulting model is then delivered to the `ContextDataStore`, where it may be further processed and finally transferred to the `AggregatorService`. The `AggregatorService` only adds contacts to the overall model which do not exist so far.

ContextDataStore

The `ContextDataStore` class is a bound background service. Context models, which are delivered to the context data store may be used by further context providers (but not in this prototype implementation). While starting the service it binds to the `AggregatorService`. When data processing is finished within the context data store, the models are transferred to the `AggregatorService`. If there is no reactive context provider - like in the proposed prototype - that makes use of the model, the models are immediately transferred to the aggregator component. This is done by calling the `addModel` method of the `AggregatorService`.

AggregatorService

The `AggregatorService` class is a bound background service that consolidates the data models from the various context and data provider services to one model. This is done by the `addModel` method. This method detects which context types have changed and informs the `NotificationProviderService` about these changes.

The `checkSPARQL` method is responsible for returning result sets from SPARQL queries coming from applications using the infrastructure.

SubscriptionEntry

The `SubscriptionEntry` class is used to store subscriptions from applications. It consists of an application ID with corresponding subscription code.

NotificationProviderService

The `NotificationProviderService` is a remote service (see Section 5.1.2). It offers functions to register and unregister applications (`registerCallback` and `unregisterCallback`), to subscribe and unsubscribe (`subscribe` and `unsubscribe`) for change events as well as methods to add statements to the overall context model `addStatements`, to check SPARQL queries against the overall model `checkSPARQL`, and to send notifications to subscribed applications `sendNotification`. An application that likes to get notifications from the system has to register with its application ID first. In order to subscribe for change events two parameters have to be provided whereas one indicates the application ID and the other indicates the change event. Whenever a change of the desired type happens, the `NotificationProviderService` informs all applications that have subscribed for the event and at the same time delivers the new data. This is accomplished by the `sendNotification` method.

By means of the `addStatements` method application-specific RDF statements can be added to the overall context model by providing the statements as a string representation. The `checkSPARQL` method can be used by registered and authorized applications for fetching RDF statements that meet certain requirements. Therefore, a valid SPARQL query has to be provided as a parameter.

If a registered application is destroyed, first the `unsubscribe` method is used to unsubscribe from change events and afterwards the `unregisterCallback` method is called. The `NotificationProviderService` binds to the `AggregatorService` at the time of starting.

INotificationProviderService.aidl

The `INotificationProviderService` AIDL file defines an interface between our system and an application extending it. It includes signatures of all methods that can be used by applications (cf. Code 5.2). The methods are implemented in the `NotificationProvider` component of the proposed system.

```
1  package at.univie.master.services;
2
3  import at.univie.master.services.INotificationProviderCallback;
4
5  interface INotificationProviderService {
6          // registering a callback interface with
7          void registerCallback(INotificationProviderCallback cb, String appID);
8          //Remove a previously registered callback interface.
9          void unregisterCallback(INotificationProviderCallback cb);
10         void subscribe(String app_id, String change_str);
11         void unsubscribe(String app_id);
12         boolean addStatements(String locationModelString);
13         String checkSPARQL(String sparqlString);
14         void sendNotification(in List<String> changeType, String modelString);
15 }
```

Code 5.2: INotificationProviderService.aidl

INotificationProviderCallback

The `INotificationProviderCallback` AIDL file defines an interface between the application and our system. It is used to provide callback functions for applications in order to get answers from the system. In our case, it is used to inform applications about changes in the infrastructure's context model. Therefore, the application implements the interface's `sendNotification` method.

5.4.3 Application

Figure 5.5 shows all classes that have been implemented for the application including its attributes, methods, and the relationships between them.

ServiceAppActivity

The `ServiceAppActivity` class is mainly responsible for starting and stopping the application. Once the application is started respectively the Activity is created, it binds to the `AlertService` with the `bindLocalService` method. The `AlertService` then binds to the `INotificationProviderService` and registers for changes in the context model. Furthermore, the user interacts with the `ServiceAppActivity` whenever they add annotated locations to the system. Therefore, the `saveLocation` method calls the `AlertService`'s `addStatements` method with the location's name and type as parameters. The `listTaggedLocations` method is called from the `onCreate` method every time the `ServiceAppActivity` is created and every time a new location is added. It is

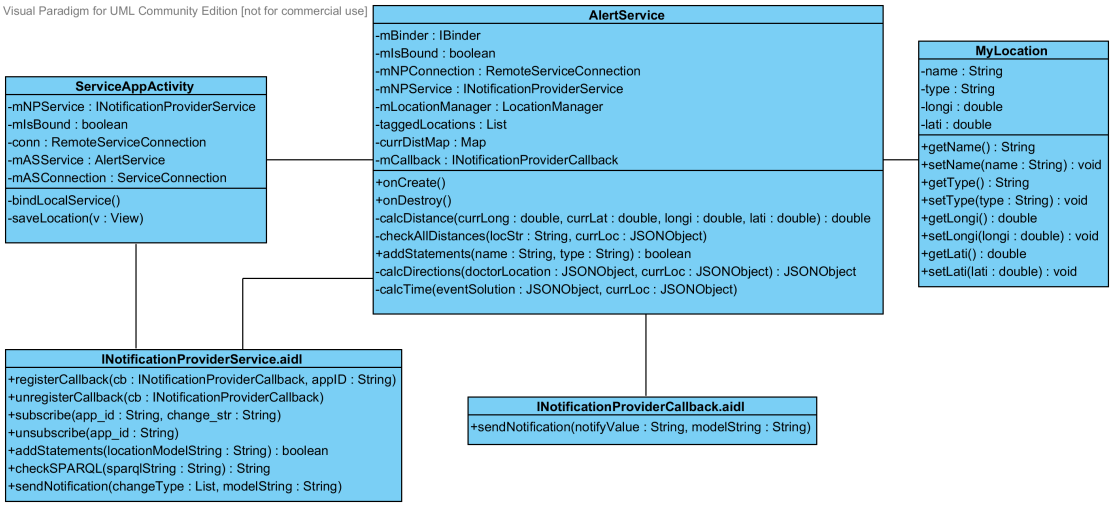

FIGURE 5.5: Detailed class diagram of the application

responsible for showing a list of all tagged locations in the GUI and therefore calls the `AlertService`'s `getAllTaggedLocations` method. The Figures 5.6 and 5.7 show the GUI of the application.

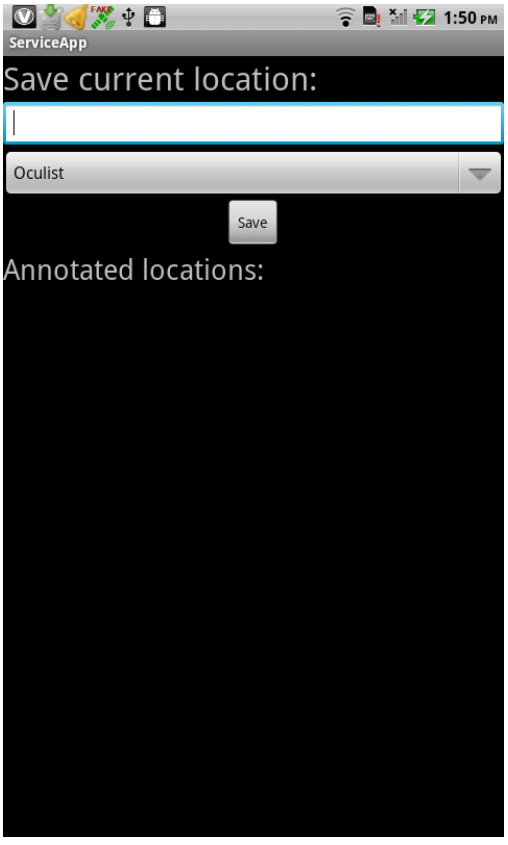

FIGURE 5.6: Application start screen

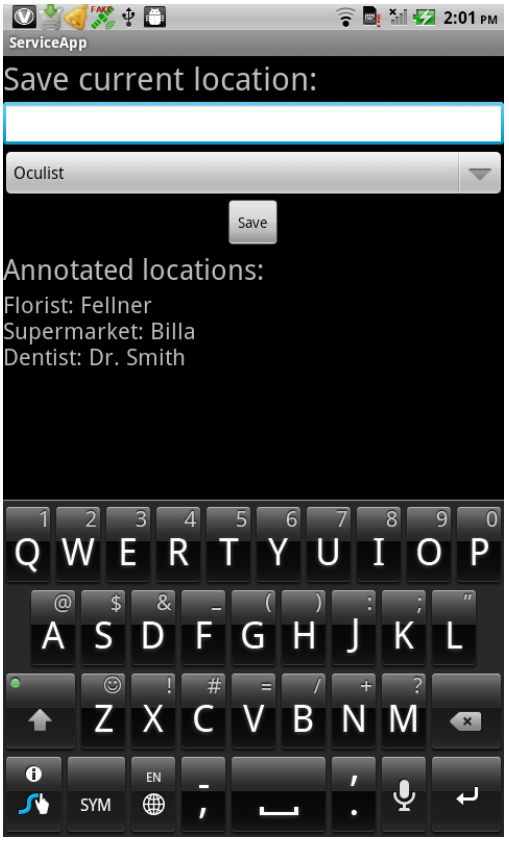

FIGURE 5.7: Application screen during use

AlertService

When the `ServiceAppActivity` binds to the `AlertService` it is started and binds to the remote `NotificationProviderService`. After the connection is established the `AlertService` subscribes for location and calendar change events. While creating the `ServiceAppActivity` the `getAllTaggedLocations` method is called. It calls the `NotificationServer`'s `checkSPARQL` query and sends the result back to the `ServiceAppActivity` where all tagged locations are shown in a list (cf. Figure 5.7).

The `addStatements` method gets the user's input from the `ServiceAppActivity` and generates RDF statements representing longitude, latitude, timestamp, location type, and location name. These statements are converted into a string representation and delivered to the `NotificationProviderService` using its `addStatements` method. Whenever the `AlertService` is notified by the `NotificationProviderService` the `sendNotification` method implemented from the `INotificationProviderCallback` is executed, which then checks if the user has to be informed of a nearby supermarket or florist respectively calculates the distances and time the user needs to go to a dentist, general practitioner, oculist, or ear specialist in order to set the appropriate reminders. As mentioned in Section 5.3 the distances to doctors are calculated by the MapQuest[10] Directions Web Service. Distances to supermarkets and florists are linear distances calculated with the following formula:

$$distance = \frac{arccos(sin(lat_a) * sin(lat_b) + cos(lat_a) * cos(lat_b) * cos(long_b - long_a))}{360} * 40000$$

The *nMn* `NotificationManager` is used for notifying the user of nearby supermarkets or florists.

MyLocation

Objects from type `MyLocation` hold annotated locations. These objects represent the name and type of locations as well as longitude and latitude values.

5.5 Problems & Limitations

In this section we discuss the limitations of the implemented prototype as well as the problems that occurred during the implementation process.

- For the prototype only a subset of the components described in Section 4.5 has been implemented. These are one context provider (*GSPContextProviderService*), two

[10] `developer.mapquest.com`

data providers (*CalendarDataProviderService* and *ContactsDataProviderService*), the context data store, the aggregator, and the notification provider as well as the *INotificationProviderService* interface. This means, there is no reasoning and no persistence management implemented. Hence, all data is only available as long as there is enough working memory respectively as long as the mobile phone is on.

Furthermore, there is no algorithm implemented that detects outdated data and removes it from the context model. This means, context and data providers are continuously adding data to the context model, but old data is never deleted.

- By default, AIDL only supports primitive data types, strings (`String`), character sequences (`CharacterSequence`), lists (`List`), and maps (`Map`) whereas all elements of lists and maps must be one of the supported data types. Hence, objects of type `Model` or result sets delivered by SPARQL queries (`ResultSet`) cannot be used as a parameter for methods declared in the `INotificationProvider-Service`. As a consequence, we transform models and result sets from SPARQL queries into string representations before they are passed on to AIDL methods. RDF statements are delivered as strings in XML syntax and SPARQL results in JSON syntax. They have to be reconverted into objects after delivering.

- For the prototype the use of annotated locations within calendar entries has to be accomplished by means of the location field. The user has to put in the desired type of location, like "supermarket" and save it. Ideally, there should be an additional drop-down field in the event's form where the user can choose one of their stored locations.

- The user does not have the possibility to disable notifications of nearby supermarkets or florists and reminders of doctor locations. This means the user gets notified every time they are near an annotated supermarket/florist as long as the event has not started. Same happens with dismissed reminders. They appear again and again if the location of the user changes as long as the event has not started.

- The prototype application does not warn the user if they are adding more than one location of the same doctor type.

- There is no functionality for deleting or editing already existing annotated locations implemented in the prototype application.

- The infrastructure as well as the application stop running after a while not using them. This is caused by the operating system which automatically destroys applications if it needs memory for other applications [Devb]. Given that the bound background services only run as long as any component is bound to them and

only the activities are bound to them, they are also stopped when an activity is destroyed. When the user presses the back button on its mobile phone during the infrastructure or application activity is in the foreground the application is destroyed as well.

- As mentioned at the beginning of this chapter, the prototype is only running on mobile phones running Google's Android operating system version 2.2, having Google's calendar application installed and the built-in GPS sensor enabled.

5.6 Summary

In this chapter, the prototype implementation of a context-aware mobile Semantic Desktop infrastructure is discussed. After introducing the functionalities of it, some enabling components used for the prototype implementation are introduced. These are Android's activities, services, AIDL, and permissions, as well as Androjena which is used for processing RDF data and handling SPARQL queries, and MapQuest Directions Web Service used for the calculation of distances between different locations. Then, the class design of the prototype infrastructure - consisting of a subset of the components described in Chapter 4 - and an application using the infrastructure is described. The application implements the functionality introduced in Section 4.2.2. The aim was to develop a system that does not negatively affect the user's work and keeps the user involvement as minimal as possible. This is achieved by using background services and status bar notifications. The described system is developed for mobile phones running Google's Android operating system. At the end of this chapter, problems that occurred during the implementation process and limitations of the prototype implementation are discussed.

Chapter 6

Evaluation

In this chapter, the process of a light evaluation of the prototype implementation of an application using the proposed system and its results are discussed. We accomplished light evaluation by choosing four persons who tested the application and answered a questionnaire (cf. Appendix A) afterwards. All of them are familiar with the use of smartphones. The questionnaire includes questions where the user has to evaluate functionalities and qualities of the application, as well as open-ended questions.

6.1 Test Use Cases

The test users had to accomplish the following use cases:

- Add all calendar entries to the system.
- Annotate at least one location of type doctor and one supermarket/florist.
- Add a calendar entry using the doctor location added before.
- Add a calendar entry using the supermarket/florst added before.
- Show the directions to a nearby supermarket/florist.

All test persons managed to accomplish all tasks.

6.2 Questionnaire

One part of the questions from our questionnaire (8-10, 13 a, and 14 a) are adopted from ISONORM 9241/110 [1]. We use a scale with four possible values, which are *agree*, *rather agree*, *rather disagree*, and *disagree*. Questions 1-7, 11, and 12 evaluate specific functionalities and qualities of our application. Questions 13b, 14b, 15b, and 16 give the test users the opportunity to give us ideas for improvement. Question 15 a tells us if such an application would be used from the test users.

The whole questionnaire is included in Appendix A.

6.2.1 Analysis

Two of the four test users agree that the calculation of distances from current location to supermarkets and florists is very precise, two of them rather agree. All test persons agree that the calculated time of travel to doctors is realistic. One person agrees that the application reacts quickly to changes of location, all the others rather agree. Three persons agree that notifications in the mobile phone's status bar are sufficient, one of them rather agrees. Two of four test persons agree that saving locations in the system is intuitive, the others rather agree. Two users think the reminder for consultations 15 minutes before departure should happen more than 15 minutes before, the other two think 15 minutes are adequate. Requests for location changes are carried out often enough say three of four test persons, one stated that requests should happen more often. All of the users agree that the meaning of texts/lables is understandable and do not cause misinterpretations. One person agrees that the application provides sufficient feedback which is understandable and helpful, two others rather agree and one person rather disagrees. With the statement that there is not much time needed for learning how to use the application one test user agrees, the others rather agree. One of the users agrees that the application does not cause more effort than it profits, the others rather agree. Three of four test persons agree that the application responds quickly, one rather agrees. With the statement that the application provides all functionality that is needed to efficiently accomplish the use cases one of the test users agrees, one rather agrees, and two rather disagree. The test users wished to have the following additional functionalities:

- possibility to decide how long before consultations the reminder should be set,
- help texts to explain functionality,

[1] cf. `http://www.seikumu.de/de/dok/dok-echtbetrieb/Fragebogen-ISONORM-9241-110-S.pdf`

- possibility to set location of points of interest manually (not only use current location),
- show directions to doctors too,
- more different doctors respectively provide possibility to add individual ones,
- edit and delete functionality,
- possibility to register more than one medical per category,
- error messages should be more clearly, I did not recognize them.

One of the test users agrees that the application does not require needless inputs/work steps in order to achieve an aim, two rather agree, and one rather disagrees. Test users stated the following things as needless work steps:

- start infrastructure application,
- adding events manually,
- setting the location in calendar textually (instead of drop-down),
- it would be fine to have a drop-down listing all possible categories for the location field in the calendar

One of the test persons would not use such an application at all, two others would probably use it and one is sure that he/she would use it. Reasons for usage are:

- The application is advantageous for appointments which I often forget; because of the additional 15 minutes one has enough time to take a shower and prepare for the appointment.
- I would use the application because I often forget my consultations.
- It is useful for consultations but I would not use the supermarket functionality because I always go to the same supermarkets to which I have to drive on purpose and so it is not interesting for me if there is another supermarket around.

The person who would not use such an application stated that he/she does not use calendars at all. The following additional ideas for improvement were given:

- not only calculate driving distance, but also walking distance;

- it would be useful to have the possibility to share locations with other users and to decide which locations of other users I would like to use myself, therefore locations whose coordinates differ minimally should be recognized to be the same;
- administration should be simplified (e.g. input of doctor type in calendar)

Statement	**agree**	**rather agree**	**rather disagree**	**disagree**
The calculation of distances from current location to supermarkets and florists is very precise.	50 %	50 %	0 %	0 %
The calculated time of travel is realistic.	100 %	0 %	0 %	0 %
The application reacts quickly to changes of location.	25 %	75 %	0 %	0 %
Notifications in the mobile phone's status bar are sufficient.	75 %	25 %	0 %	0 %
Saving locations in the system is intuitive.	50 %	50 %	0 %	0 %
The meaning of texts/labels is understandable and do not cause misinterpretations.	100 %	0 %	0 %	0 %
The application provides sufficient feedback which is understandable and helpful.	25 %	50 %	25 %	0 %
There is not much time needed for learning how to use the application.	25 %	75 %	0 %	0 %
The application does not cause more effort than it profits.	25 %	75 %	0 %	0 %
The application responds quickly.	75 %	25 %	0 %	0 %
The application provides all functionality that is needed to efficiently accomplish the use cases.	25 %	25 %	50 %	0 %
The application does not require needless inputs/work steps in order to achieve an aim.	25 %	50 %	25 %	0 %
	yes	**probably**	**probably not**	**no**
I would use such an application.	25 %	50 %	0 %	25 %

TABLE 6.1: Overview of the test users answers

The reminder for consultations 15 minutes before departure	is adequate	should happen more than 15 minutes before	should happen less than 15 minutes before
	50 %	50 %	0 %
Requests for location changes	should happen more often	should happen less often	are carried out often enough
	25 %	0 %	75 %

TABLE 6.2: Overview of the test users answers of questions 6 and 7

6.3 Conclusion

In summary, the results of the initial study are good. The light evaluation showed that the most important technical requirements for the realization of a mobile Semantic Desktop infrastructure are given and performance is acceptable. Certainly, there is room for improvement.

Furthermore, there is one idea from a test user that was not considered at all in this work. Nevertheless, the idea of sharing locations with other users may be an interesting issue for future work. Some other ideas were already discussed in Section 5.5, such as a drop-down list for the location field in the calendar, as well as edit and delete functionality. It may also be helpful for users to have the possibility to manually add the address or coordinates of a location one would like to add and not always use the current location respectively give users the possibility to make their own configurations in order to influence the behaviour of the application.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

In this work, we developed a mobile Semantic Desktop infrastructure that facilitates the development of context-aware semantic personal assistance applications. Therefore, we first provided all important background information and evaluated already existing related work. Then, we introduced our approach from different angels of view and the context model we built for representing data. As a proof of concept, a prototype of the infrastructure and an application extending the infrastructure have been implemented and described. Finally, the application has been tested from four different test users which tried the whole functionality of it and afterwards evaluated the application by means of a questionnaire. In the following, we discuss the drawn conclusions, achievements, and limitations of this work.

The implementation process and the initial study showed that we are on a good path in order to develop a mobile Semantic Desktop infrastructure. One of the subgoals of this work was to acquire data from the mobile phone and its environment and expose them as an RDF graph with as little user participation/input as possible. This aim has been accomplished by implementing all context and data providers as background services. Hence, there is no user input necessary for the acquisition of data. Another subgoal was the implementation of an interface for the communication between infrastructure and applications that make use of the infrastructure. This has been accomplished by using remote service invocation. AIDL files are used to define the interface. These have to be included in the infrastructure application as well as in the extending application. For the definition of a context model that represents all data that is needed we reused already existing ontologies, like parts of the NEPOMUK Ontologies [Con] and CC/PP [ccp07],

and combined them to one overall model by including our own additional properties and objects (cf. Section 4.4).

The result of the research regarding different reasoning algorithms is that rule-based reasoning is best suited for the use with mobile devices. There are two different types of rule-based reasoning, these are forward chaining and backward chaining. We decided to use forward chaining for our approach because it is best suited for deducing high-level context information from low-level context information and it does not need much resources which is beneficial for mobile applications.

The implementation process showed that there are some issues that do not work straight forward in Android and hence workarounds had to be found. For instance, the Android's standard data providers of calendar entries and contacts do not provide information about what has changed, but only that anything has changed. Hence, the whole events/contacts submodel has to be rebuilt if a change has happened and only the differences between the two models are added to the aggregated model. Consequently, deleted events or contacts are not considered in the prototype implementation. During the implementation process, we found out that persisting RDF models as text files on the mobile phone respectively the SD card is possible, whereas it is recommended to use an SD card, because the file may get very big and the internal storage of the mobile phone is limited.

7.2 Future Work

Our approach describes a synchronization component which is responsible for synchronizing the mobile Semantic Desktop infrastructure with a Semantic Desktop on a PC or server. Therefore, an appropriate synchronization algorithm as well as a mapping scheme for the two context models has to be researched. Furthermore, a mechanism for unifying the resources' IDs and the addressing scheme has to be explored. Finally, it has to be investigated how to implement these mechanisms efficiently.

Moreover, the management of context history is not yet implemented in the infrastructure. Therefore, adequate lifetimes for each type of information stored in the context model has to be investigated and it has to be explored how data items that are not used any more can be identified reliably. Maybe for the persistence of data an SQLite database should be used instead of a text file.

Privacy is another issue that requires more intensive research. It has to be considered how this mechanism can be implemented efficiently and transparent for the user and how users can be able to decide which data they like to expose or not on their own.

The user evaluation offered another issue for future work which is the sharing of content between different users. For example, sharing annotated locations with other users. Therefore, a central storage for location information, as well as a connection from the application to this storage is needed. Furthermore, due to privacy reasons, this functionality should be optional and there has to be an algorithm that recognizes identical locations.

Appendix A

Questionnaire

1. The calculation of distances from current location to supermarkets and florists is very precise.

agree	rather agree	rather disagree	disagree
□	□	□	□

2. The calculated time of travel is realistic.

agree	rather agree	rather disagree	disagree
□	□	□	□

3. The application reacts quickly to changes of location.

agree	rather agree	rather disagree	disagree
□	□	□	□

4. Notifications in the mobile phone's status bar are sufficient.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

5. Saving locations in the system is intuitive.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

6. The reminder for consultations 15 minutes before departure:

☐ is adequate
☐ should happen more than 15 minutes before
☐ should happen less than 15 minutes before

7. Requests for location changes:

☐ should happen more often
☐ should happen less often
☐ are carried out often enough

8. The meaning of texts/labels is understandable and do not cause misinterpretations.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

9. The application provides sufficient feedback which is understandable and helpful.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

10. There is not much time needed for learning how to use the application.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

11. The application does not cause more effort than it profits.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

12. The application responds quickly.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

13 a. The application provides all functionality that is needed to efficiently accomplish the use cases.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

13 b. I wished to have the following additional functionalities:

_____

_____

_____

14 a. The application does not require needless inputs/work steps in order to achieve an aim.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

14 b. Needless work steps:

_____

_____

_____

15 a. I would use such an application.

agree	rather agree	rather disagree	disagree
☐	☐	☐	☐

15 b. Reasons:

_____

_____

_____

_____

16. Additional remarks:

_____

_____

_____

_____

Appendix B

Manual

In the following the installation and usage of the prototype implementation is described. The infrastructure and application are developed for mobile phones running Google's Android operating system version 2.2 and needs Google's calendar application.

B.1 Installation

1. Download and install the Google Android SDK program[1] and the Android USB drivers[2].

2. Modify your mobile phone's settings to allow the installation of applications from unknown sources:

   (a) Under *Settings* select *Applications* and enable *Unknown sources* (cf. Figure B.1).

   (b) Under *Settings* select *SD card & phone storage* and disable *Mass storage only* (cf. Figure B.2).

   (c) Under *Settings* select *Applications*, select *Development* and enable *USB Debugging* (cf. Figure B.3).

[1] Download link: `http://code.google.com/android/intro/installing.html`
[2] Download link: `http://dl.google.com/android/android_usb_windows.zip`

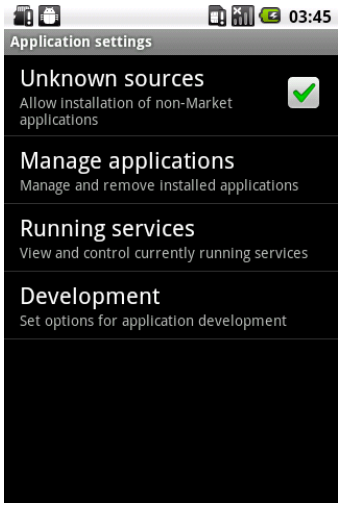

FIGURE B.1: Application settings screen (a)

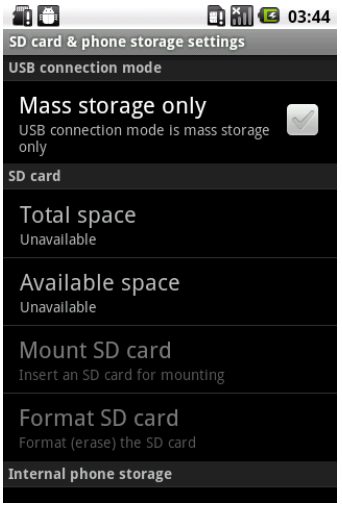

FIGURE B.2: SD card & phone storage settings screen (b)

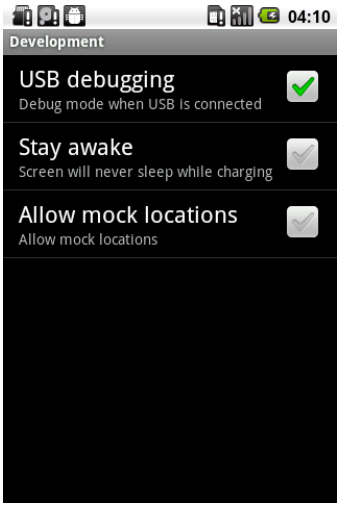

FIGURE B.3: Development settings screen (c)

3. Connect your mobile phone with your computer via USB.

4. Open the operating system's command prompt and type:
   `adb install <path>\CASPA_v2.0\bin\CASPA_v2.0.apk` and
   `adb install <path>\ServiceApp\bin\ServiceApp.apk` and press enter.

   (a) Windows: Click the *Start* button, type `cmd`, and press enter.

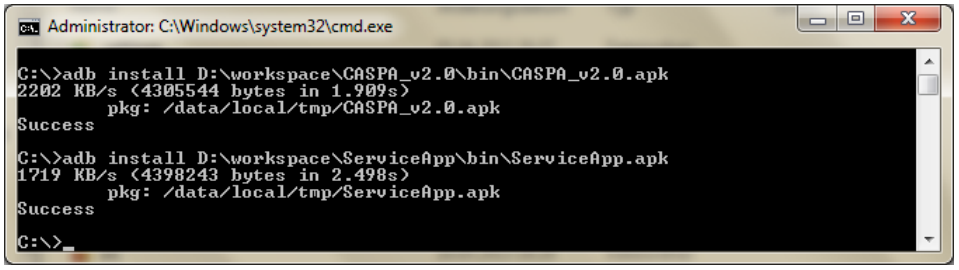

FIGURE B.4: Command prompt with install command

   (b) OS X: Open *Finder*, select *Applications*, double click on *Terminal*.

   (c) Linux: Right-click on the desktop and choose *Open Terminal* from the menu.

B.2 Use Cases

Start infrastructure

Start InfrastructureActivity and press the *START* button.

Add all calendar entries to the system

If you like to immediately add all events to the infrastructure's context model, press the button *Add Events*. Otherwise, all events are added to the model as soon as you add, edit, or delete a calendar entry.

Annotate location

Assumption: The infrastructure is started and the GPS sensor is enabled.

1. Start ServiceAppActivity
2. Type the name of the location into the text field.
3. Choose the type of the location from the dropdown field.
4. Press the *Save* button.

Add calendar entry using an annotated location

1. Start calendar application.
2. Open the *add event* view.
3. Fill in all necessary information and fill in the type of location in the *Where* field of the view (using the same spelling as in the ServiceApp).
4. Save the event.

Show directions to next supermarket/florist

Assumption: There is a location of type supermarket/florist added to the system using the ServiceAppActivity. There is an event using this type of location added to the calendar. A stored supermarket/florist is within a radius of one kilometer linear distance.

1. Touch the status bar notification from the ServiceApp saying that there is a supermarket nearby.
2. Touch *Maps* in the following dialog.
3. Google Maps opens and shows the route.

Stop acquisition of data

Stop all background services by pressing the *STOP* button in the InrastructureActivity.

Bibliography

[ACG10] Bruno Antunes, Francisco Correia, and Paulo Gomes. Towards a software developer context model. In Jrg Cassens, Anders Kofod-Petersen, Marielba Silva Zacarias, and Rebekah K. Wegener, editors, *Proceedings of the Sixth International Workshop on Modeling and Reasoning in Context*, volume 618 of *CEUR Workshop Proceedings*, pages 1–12. CEUR-WS, August 2010.

[AD99] Gregory D. Abowd and Anind K. Dey. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307. Springer-Verlag, 1999.

[Alt] Altova. What is the semantic web? online: `http://www.altova.com/semantic_web.html`.

[BBH+10] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010.

[BCQ+07] Cristiana Bolchini, Carlo A. Curino, Elisa Quintarelli, Fabio A. Schreiber, and Letizia Tanca. A data-oriented survey of context models. *SIGMOD Rec.*, 36:19–26, December 2007.

[BDR07] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007.

[BG07] Irad Ben-Gal. *Bayesian Networks*. John Wiley and Sons, 2007.

[BGM04] Dan Brickley, R.V. Guha, and Brian McBride. RDF vocabulary description language 1.0: RDF schema. Technical report, February 2004.

[BL03] Tim Berners-Lee. Iswc2003 keynote. online: `http://www.w3.org/2003/Talks/1023-iswc-tbl/`, 2003. Accessed January 2012.

[BLFM05] T Berners-Lee, R Fielding, and L Masinter. Uniform resource identifier (uri): Generic syntax. Technical report, 2005.

[BLHH+06] Tim Berners-Lee, Wendy Hall, James A. Hendler, Kieron O'Hara, Nigel Shadbolt, and Daniel J. Weitzner. A framework for web science. *Foundations and Trends in Web Science*, 1(1), 2006.

[Bru04] Patrick Brunner. Yasec - yet another semantic web crawler: Implementation eines semantikfreundlichen web-crawlers, November 2004.

[BvHH+04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference. W3c recommendation, W3C, February 2004.

[ccp07] Composite capability/preference profiles (cc/pp): Structure and vocabularies 2.0. W3C working draft, W3C, April 2007.

[CCRR02] James L. Crowley, Jolle Coutaz, Gatan Rey, and Patrick Reignier. Perceptual components for context aware computing. In *UBICOMP 2002, International Conference on Ubiquitous Computing, Goteborg*, pages 117–134, 2002.

[CD91] Corkill and Daniel D. Blackboard systems. *AI Expert*, 6(9), January 1991.

[CF] Fulvio Corno and Laura Farinetti. Rule-based reasoning in the semantic web. online: `http://www.slideshare.net/fulvio.corno/rulebased-reasoning-in-the-semantic-web`. Accessed January 2012.

[Con] Nepomuk Consortium. Oscaf/nepomuk ontologies. online: `http://www.semanticdesktop.org/ontologies/`.

[dec] Decision making. online: `http://www.managers-net.com/decisionmaking.html`. Accessed March 12 2012.

[Deva] Android Developers. Android interface definition language (aidl). Technical report. Accessed July 2012.

[Devb] Android Developers. Application fundamentals. Technical report. Accessed November 2013.

[Devc] Android Developers. Permissions. Technical report. Accessed September 2013.

[Devd] Android Developers. Services. Technical report. Accessed July 2012.

[dFRRR04] Cristiano di Flora, Oriana Riva, Kimmo Raatikainen, and Stefano Russo. Supporting Mobile Context-Aware Applications through a Modular Service Infrastructure. *Poster presented at the 6th International Conference on Ubiquitous Computing (UbiComp'04)*, 7-10 September 2004.

[FC06] Patrick Fahy and Siobhan Clarke. CASS - middleware for mobile Context-Aware applications. *University of Pennsylvania Law Review*, 154(3):477+, January 2006.

[FHMV95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[GKPZHW04] Tao Gu, Hung Keng Pung, Da Qing Zhang, and Xiao Hang Wang. A middleware for building context-aware mobile services. In *In Proceedings of IEEE Vehicular Technology Conference (VTC)*, 2004.

[Goo10] Google. Arqoid. online: `http://code.google.com/p/androjena/wiki/ARQoid`, September 2010. Accessed March 8 2012.

[Goo11] Google. androjena. online: `http://code.google.com/p/androjena/`, 2011. Accessed March 8 2012.

[Gru95] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928, December 1995.

[GWPZ04] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An ontology-based context model in intelligent environments. In *In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 270–275, 2004.

[HI06] K. Henricksen and J. Indulska. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2(1):37–64, February 2006.

[HIR02] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *Proceedings of the First International Conference on Pervasive Computing*, pages 167–180, London, UK, 2002. Springer-Verlag.

[HIR03] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Generating context management infrastructure from high-level context models. In *4th International Conference on Mobile Data Management, Melbourne, Australia*, 2003.

[HM08] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases, Second Edition (The Morgan Kaufmann Series in Data Management Systems)*. Kaufmann, Morgan, 2 edition, March 2008.

[HMS+10] Peter Haase, Tobias Mathäß, Michael Schmidt, Andreas Eberhart, and Ulrich Walther. Semantic technologies for enterprise cloud management. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part II*, ISWC'10, pages 98–113, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hog11] Aidan Hogan. *Exploiting RDFS and OWL for Integrating Heterogeneous, Large-Scale, Linked Data Corpora*. PhD thesis, April 2011.

[HSP+03] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann, and Werner Retschitzegger. Context-awareness on mobile devices - the hydrogen approach. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10 pp.+, 2003.

[itw] Sensor. online: `http://www.itwissen.info/definition/lexikon/Sensor-sensor.html`. Accessed May 5 2012.

[Jon07] William Jones. *Keeping found things found*. Kaufmann, Morgan, November 2007.

[KC] YounSoo Kim and Hoon Choi. Data synchronization and conflict resolution for mobile devices. http://strauss.cnu.ac.kr/research/sync/paper/11_SyncML_ConflictResolution.pdf. Accessed September 2012.

[KMK+03] Panu Korpipää, Jani Mäntyjärvi, Juha Kela, Heikki Keränen, and Esko-Juhani Malm. Managing context information in mobile devices. *Pervasive Computing, IEEE*, 2(3):42–51, 2003.

[Kol92] Janet L. Kolodner. An introduction to case-based reasoning. *Artificial Intelligence Review*, 6:3–34, March 1992.

[Map] MapQuest. Open directions service developer's guide. Technical report. Accessed September 2013.

[Mir08] Otto Mirko. Ontologien zur semantischen suche in einem bestand von dokumenten, September 2008.

[MKP05] Marius Mikalsen and Anders Kofod-Petersen. Representing and reasoning about context in a mobile environment. *REVUE D'INTELLIGENCE ARTIFICIELLE (RIA)*, 19:479–498, 2005.

[MM04] Frank Manola and Eric Miller. RDF primer. W3C recommendation, W3C, February 2004.

[MS01] Alexander Maedche and Steffen Staab. Ontology learning for the semantic web. *IEEE Intelligent Systems*, 16(2):72–79, March 2001.

[NF] Petteri Nurmi and Patrik Floren. Reasoning in context-aware systems. Accessed February 2012.

[NM01] Natalya F. Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology. Online, 2001.

[PdBW+04] Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an extensible context ontology for ambient intelligence. *Lecture Notes in Computer Science*, 3295:148–159, October 2004.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, World Wide Web Consortium, January 2008.

[QHK03] Dennis Quan, David Huynh, and David R. Karger. Haystack: A platform for authoring end user semantic web applications. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 738–753. Springer, 2003.

[Ram07] Daniel Ramage. Hidden markov models fundamentals. online: `http://cs229.stanford.edu/section/cs229-hmm.pdf`, December 2007. Accessed March 10 2012.

[RC04] Gatan Rey and Jolle Coutaz. The contextor infrastructure for context-aware computing. In *18th European Conference on Object-Oriented Programming (ECOOP 04), Workshop on Component-oriented approach to context-aware systems*, 2004.

[RSS+10] Yves Raimond, Tom Scott, Patrick Sinclair, Libby Miller, Stephen Betts, and Frances McNamara. Case study: Use of semantic web technologies on the bbc web sites, January 2010. Accessed May 31 2012.

[Sau03] Leo Sauermann. The gnowsis – using semantic web technologies to build a semantic desktop. Diploma thesis, Technical University of Vienna, 2003.

[Sau05a] Leo Sauermann. The gnowsis semantic desktop for information integration. In *I-KNOW 2005 conference*, 2005.

[Sau05b] Leo Sauermann. The semantic desktop - a basis for personal knowledge management. In Hermann Maurer, Cristian Calude, Arto Salomaa, and Klaus Tochtermann, editors, *Proceedings of the I-KNOW 2005. 5th International Conference on Knowledge Management*, pages 294 – 301, 2005.

[Sau06] Leo Sauermann. Semantic desktop – der arbeitsplatz der zukunft. In Andreas Blumauer and Tassilo Pellegrini, editors, *Semantic Web – Auf dem Weg zur vernetzten Wissensgesellschaft*, pages 161–176. Springer Verlag, 2006.

[SBD05] Leo Sauermann, Ansgar Bernardi, and Andreas Dengel. Overview and outlook on the semantic desktop. In Stefan Decker, Jack Park, Dennis Quan, and Leo Sauermann, editors, *Proceedings of the 1st Workshop on The Semantic Desktop at the ISWC 2005 Conference*, volume 175 of *CEUR-WS.org*, pages 1 – 18. CEUR-WS, November 2005.

[SCV07] Leo Sauermann, Richard Cyganiak, and Max Völkel. Cool uris for the semantic web. Technical memo, DFKI GmbH, February 2007.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, CHI '99, pages 434–441, New York, NY, USA, 1999. ACM.

[SDCD09] Ahmet Soylu, Patrick De Causmaecker, and Piet Desmet. Context and adaptivity in pervasive computing environments: Links with software engineering and ontological engineering. *Journal of Software*, 4(9):992–1013, 2009.

[Sea10] Andy Seaborne. Jena, a Semantic Web Framework. online: `http://wiki.apache.org/incubator/JenaProposal`, November 2010. Accessed March 8 2012.

[Sie09] David Siegel. *Pull: The Power of the Semantic Web to Transform Your Business*. 2009.

[SLP04] Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004.

[Spo] Manu Sporny. Intro to the semantic web. online: `http://www.youtube.com/watch?v=OGg8A2zfWKg`. Acceessed January 2012.

[SRMR+07] M. Sasikumar, S. Ramani, S. Muthu Raman, KSR Anjaneyulu, and R. Chandrasekar. *A Practical Introduction to Rule Based Expert Systems*. Narosa Publishing House, 2007.

[SS] Christos Stergiou and Dimitrios Siganos. Neural networks. online: `http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html`. Accessed March 12 2012.

[ST94] Bill Schilit and M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8:22–32, 1994.

[SvED07] Leo Sauermann, Ludger van Elst, and Andreas Dengel. Pimo - a framework for representing personal information models. In Tassilo Pellegrini and Sebastian Schaffert, editors, *Proceedings of I-MEDIA '07 and I-SEMANTICS '07 International Conferences on New Media Technology and Semantic Systems as part of TRIPLE-I 2007*, pages 270–277. Know-Center, Austria, September 2007.

[SvEM09] Leo Sauermann, Ludger van Elst, and Knud Mller. Personal information model (pimo). Technical report, February 2009.

[Tau08] Joshua Tauberer. What is rdf and what is it good for?, January 2008. Accessed May 31 2012.

[UG96] Mike Uschold and Michael Grüninger. Ontologies: Principles, methods, and applications. *Knowledge Engineering Review*, 11(2):93–155, 1996.

[VW51] Georg H. Von Wright. *An essay in modal logic*. 1951.

[Wal04] Hanna M. Wallach. Conditional random fields: An introduction. Technical report, University of Pennsylvania, February 2004.

[WW08] Wolfgang Woerndl and Maximilian Woehrl. Semodesk: Towards a mobile semantic desktop. *Information Systems Journal*, pages 1–6, 2008.

[WZGP04] Xiao Hang Wang, Da Qing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *Proceedings*

of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops, pages 18–22, Washington, DC, USA, 2004. IEEE Computer Society.

[YDM11] Juan Ye, Simon Dobson, and Susan McKeever. Situation identification techniques in pervasive computing: A review. *Pervasive and Mobile Computing*, January 2011.

[ZS11] Stefan Zander and Bernhard Schandl. Context-driven rdf data replication on mobile devices. *Semantic Web Journal Special Issue on Real-time and Ubiquitous Social Semantics*, 1(1), 2011.

[ZS12] Stefan Zander and Bernhard Schandl. Semantic web-enhanced context-aware computing in mobile systems: Principles and application. In A.V.Senthil Kumar, editor, *Mobile ComputingTechniques in Emerging Markets: Systems, Applications and Services*. IGI Global, 2012.

Curriculum Vitae

Personal Data

Name:	Carina Christ, BSc
Date of Birth:	January 28, 1987
Place of Birth:	Mistelbach, Austria

Education

Oct 2006 - Sept 2009	Undergraduate Studies in Media Information Technology at **University of Vienna**, Vienna Thesis' title: Process Editor for the Composition of Web Services — Supervisor: Univ.-Prof. Dipl.-Ing. Dr. techn. Erich Schikuta
June 2006	General Qualification for University Entrance at **HTL Donaustadt (Department for EDP & Organization)**, Vienna Final project: E-Learning Platform for Financial Accounting

Work Experience

Sept 2010 - present	Software Engineer at LFRZ GmbH, Vienna Developing Web GIS applications.
June 2007 - Aug 2012	Software Engineer at iT-Knowledge GmbH, Vienna Part-time employee for 12 hours per week.
July 2004	Internship at IT-Austria GmbH, Vienna
July 2003	Internship at Siemens AG Austria, Vienna
July 2002	Internship at MA 14, Vienna