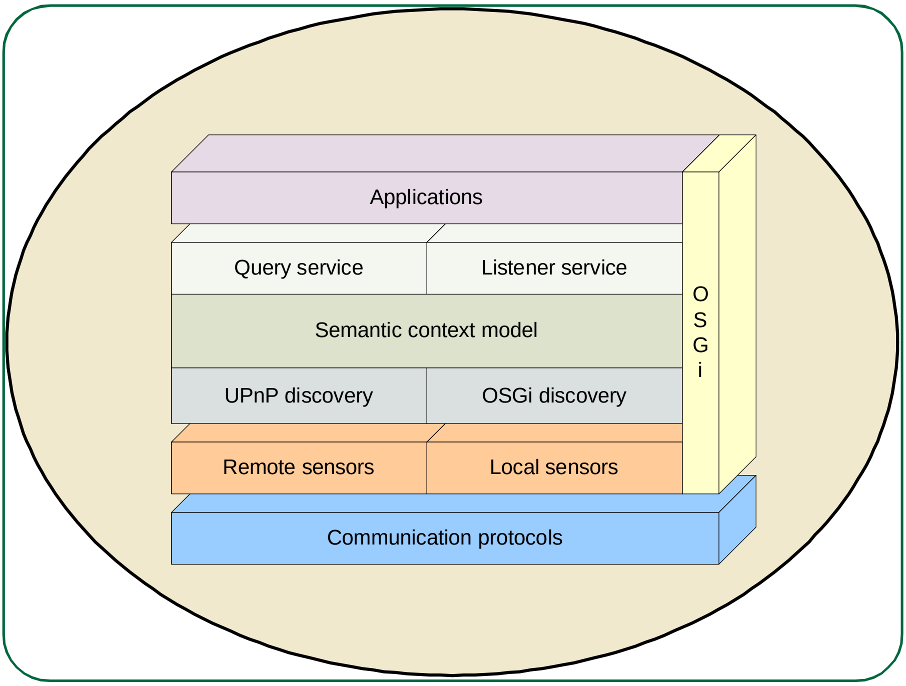
ESPOO 2007

VTT PUBLICATIONS 651



Juhani Laitakari

Dynamic context monitoring for adaptive and context-aware applications

VTT PUBLICATIONS 651

Dynamic context monitoring for adaptive and context-aware applications

Juhani Laitakari

ISBN 978-951-38-7036-2 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 978-951-38-7037-9 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 3, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 3, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 3, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax +358 20 722 4374

VTT, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Technical Research Centre of Finland, Kaitoväylä 1, P.O. Box 1100, FI-90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Text preparing Tarja Haapalainen

Edita Prima Oy, Helsinki 2007

Laitakari, Juhani. Dynamic context monitoring for adaptive and context-aware applications [Dynaaminen kontekstin monitorointipalvelu adaptiivisille ja kontekstitietoisille sovelluksille]. Espoo 2007. VTT Publications 651. 111 p. + app. 8 p.

Keywords service architecture, context acquisition, Open Services Gateway initiative, Universal Plug and Play

Abstract

The field of ubiquitous computing has recently proliferated with a view to providing applications and services that are able to adapt to the rapidly changing situations in dynamic environments and act accordingly. The seamless adaptation to contexts and the alterations to behaviour require the applications to implement mechanisms for acquiring the context information. The required context information is usually diverse and scattered throughout the environment. On account of this, the processing of the context information and its compilation from separate sources is a requirement for the applications to reach adequate context-awareness for successful adaptation. To facilitate the development of context-aware applications, service-oriented architectures for supporting the context-awareness have emerged.

In this work the research problem was to find a solution for dynamic acquisition and representation of distributed context information and its efficient provisioning for ubiquitous applications. As a solution to the research problem this work provides a service architecture called Context Monitoring Service (CMS), which utilizes a dynamically evolving semantic model of context information that the applications can access. A requirement analysis for such architecture was carried out by a literature review in the field of context-awareness. The architecture of the CMS was designed according to the identified requirements and a prototype implementation was created for validation purposes. The prototype implementation successfully validated the architecture's functionality and also opened issues for future research and development in this field.

Laitakari, Juhani. Dynamic context monitoring for adaptive and context-aware applications [Dynaaminen kontekstin monitorointipalvelu adaptiivisille ja kontekstitietoisille sovelluksille]. Espoo 2007. VTT Publications 651. 111 s. + liitt. 8 s.

Avainsanat service architecture, context acquisition, Open Services Gateway initiative, Universal Plug and Play

Tiivistelmä

Kaikkialla läsnä olevan tietotekniikan aihealue on hiljattain kasvanut räjähdysmäisesti, näkemyksenään tuottaa sovelluksia ja palveluita, jotka pystyvät mukautumaan nopeasti muuttuviin olosuhteisiin ja toimimaan niiden mukaisesti. Saumaton mukautuminen olosuhteisiin ja käyttäytymisen muokkaaminen vaativat sovelluksilta mekanismeja kontekstitiedon keräämiseen ja prosessoimiseen. Vaadittu kontekstitieto on yleensä monimuotoista ja hajallaan ympäristössä, minkä vuoksi sovelluksilta vaaditaan tiedon prosessointia ja kokoamista useista lähteistä, jotta onnistunut mukautuminen saavutetaan. Helpottaakseen kontekstitietoisten sovellusten kehittämistä kehityssuuntana ovat olleet palvelulähtöiset arkkitehtuurit kontekstitietoisuuden tukemiseen.

Tässä työssä tutkimusongelmana on ollut löytää ratkaisu hajautetun kontekstitiedon dynaamiseen keräämiseen ja sen kuvaamiseen sekä tiedon tehokkaaseen välittämiseen mukautuville ja kontekstitietoisille sovelluksille. Tämä työ esittelee ratkaisuna tutkimusongelmaan palveluarkkitehtuurin nimeltään Context Monitoring Service (CMS). CMS hyödyntää dynaamisesti kehittyvää semanttista mallia kontekstitiedosta, joka tarjotaan sovellusten käyttöön palvelun kautta. Tällaisen arkkitehtuurin vaatimusmäärittely suoritettiin laajalla katsauksella kirjallisuuteen kontekstitietoisuuden saralla. CMS-arkkitehtuuri suunniteltiin vaatimusmäärittelyn mukaisesti ja arkkitehtuurista toteutettiin prototyyppi toiminnallisuuden vahvistamista varten. Prototyyppi vahvisti arkkitehtuurin toimivuuden onnistuneesti ja avasi myös uusia tutkimusaiheita tällä aihealueella.

Preface

This thesis was completed in the Software Architectures team of the Software Architectures and Platforms research center at VTT Technical Research Centre of Finland. The research work for the thesis is part of the research done in the ITEA ANSO (Autonomic Networks for SOHO users) project.

First of all, I would like to thank all the co-workers that have shared their expertise and contributed to this work. Secondly, my deepest gratitude goes to Mr. Daniel Pakkala for guiding me through this work by providing valuable reviews, discussions and support during the work. I would also like to thank my supervisor at the university, Professor Tapio Seppänen, and the work's 2nd reviewer, Professor Jukka Riekki, for the comments and guidance.

Finally, I thank all the people that have supported and encouraged me during the research and writing process of this thesis.

Oulu, November 17, 2006

Juhani Laitakari

Contents

Abstract ... 3

Tiivistelmä ... 4

Preface ... 5

Abbreviations ... 9

1. Introduction ... 12

2. Related research and technologies ... 14
    - 2.1 Introduction to context-awareness ... 14
        - 2.1.1 Context information ... 15
        - 2.1.2 Representation of context and ontologies ... 16
        - 2.1.3 Context-awareness ... 17
        - 2.1.4 Context monitoring service ... 18
    - 2.2 Technologies ... 19
        - 2.2.1 Extensible Mark-up Language ... 20
        - 2.2.2 XML Schema ... 21
        - 2.2.3 Resource Description Framework ... 22
        - 2.2.4 RDF Schema ... 23
        - 2.2.5 Web Ontology Language ... 24
        - 2.2.6 RDF Data Query Language ... 25
        - 2.2.7 Jena Semantic Web Framework ... 25
        - 2.2.8 Open Services Gateway Initiative ... 26
        - 2.2.9 Universal Plug and Play ... 28
    - 2.3 Architectures of existing context monitoring services ... 31
        - 2.3.1 CoBrA ... 31
        - 2.3.2 Semantic Space ... 33
    - 2.4 Terminology ... 34

3. Design of Context Monitoring Service ... 36
    - 3.1 Introduction to service ... 36
    - 3.2 Requirements scenario for the design ... 38
        - 3.2.1 Deployment environment ... 39
        - 3.2.2 Communications ... 40

3.2.3 Functionality 40
3.2.4 Identified requirements 41
3.3 Context model 43
3.4 Ontologies 45
3.4.1 Upper-level ontology 45
3.4.2 Lower-level ontologies 47
3.5 Reasoner 49
3.5.1 Rules for Reasoner 50
3.6 Querying of the context model 51
3.7 Conditional eventing 52
3.8 Model updater devices 54
3.9 Dynamic discovery and advertisement of model updater devices 56
3.9.1 Local model updater sensors 56
3.9.2 Remote model updater devices 56
3.10 OSGi bundle configuration 58

4. Architectural design 59
4.1 Solution architecture 59
4.1.1 Service layer 61
4.1.2 Model updater device advertisement and discovery layer 62
4.1.3 Model updater sensor layer 64
4.2 Architecture of the service 65
4.2.1 Local service interfaces 67
4.2.2 Remote service interfaces 71
4.2.3 Local management interfaces 74
4.2.4 Remote management interfaces 75
4.2.5 Local control interfaces 76
4.2.6 Remote control interfaces 78

5. Prototype implementation and testing 79
5.1 Prototype implementation 79
5.2 Configuration 80
5.2.1 Hardware 81
5.2.2 Software 82
5.2.3 Overall view 84
5.3 Validation scenario and use cases 85
5.3.1 Scenario overview 86

5.3.2 Dynamic discovery and advertisement 87
5.3.3 Context model querying 88
5.3.4 Conditional eventing 90
5.4 Evaluation of prototype 93
5.4.1 Service layer 93
5.4.2 Model updater device advertisement and discovery layer 94
5.4.3 Model updater sensor layer 94

6. Discussion 96
6.1 Comparison with existing architectures 96
6.2 Strengths and weaknesses 98
6.2.1 Single component 98
6.2.2 System as a whole 99
6.3 Development targets 101

7. Conclusion 103

References 105

Appendices

Appendix 1: Example of a model updater device UPnP device description
Appendix 2: Example of a model updater device UPnP service description
Appendix 3: UML sequence diagram of the conditional rule listener functionality
Appendix 4: UML sequence diagram of the instance listener functionality
Appendix 5: UML sequence diagram of the context model query utilization
Appendix 6: UML activity diagrams of dynamic discovery and advertisement
Appendix 7: UML activity diagram of context model query
Appendix 8: UML activity diagram of conditional eventing

Abbreviations

API Application Programming Interface, interface to existing applications

C-CMS Central Context Monitoring Service, component of the context monitoring service that contains the context information of the environment’s overall structure

CDC Connected Device Configuration, a framework for building J2ME applications

CLDC Connected Limited Device Configuration, defines the base set of application programming interfaces and a virtual machine for resource-constrained devices

CMS Context Monitoring Service, the service developed in this thesis.

CoBrA Context Broker Architecture, architecture for supporting context-aware applications

DCP Device Control Protocol, UPnP specification standardized by the UPnP forum

DHCP Dynamic Host Control Protocol, a client-server networking protocol used for allocating IP addresses to hosts

GENA General Event Notification Architecture, HTTP notification architecture developed by Microsoft

GSE Generic Service Element, architectural concept for middleware services

GUI Graphical User Interface, graphical interface for the user to interact with a computing system

HTML	Hypertext Mark-up Language, a mark-up language for representing data, such as web pages
HTTP	Hypertext Transport Protocol, application layer protocol
IP	Internet Protocol, network protocol
J2ME	Java 2 Micro Edition, collection of Java APIs for the development of software for resource-constrained devices
J2SE	Java 2 Standard Edition, collection Java APIs for most Java programs
JAR	Java Archive, file format used to package Java applications
JVM	Java Virtual Machine, a virtual machine that executes Java byte code
MIDP	Mobile Information Device Profile, profile of J2ME for connected limited device configuration
MOM	Message-Oriented Middleware, client/server architecture that supports asynchronous calls between the client and server applications
OSGi	Open Services Gateway initiative (nowadays OSGI Alliance), corporation comprised of technology innovators and developers focused on developing open service gateway technology
OWL	Web Ontology Language, a language for defining machine interpretable vocabularies specified by W3C
PC	Personal Computer, a term for a computer that is suitable for personal use
RDF	Resource Description Framework, W3C specification for a metadata model
RDF-S	RDF Schema, a language to structure RDF resources

RDQL	RDF Data Query Language, a query language for RDF
RF	Radio Frequency, portion of the electromagnetic spectrum in which electromagnetic waves can be generated
S-CMS	Specific Context Monitoring Service, component of the context monitoring service that contains more detailed context information on the deployment environment than C-CMS
SOAP	Simple Object Access Protocol, application layer protocol specified by W3C
SPARQL	SPARQL Protocol and RDF Query Language, a query language for RDF
SSDP	Simple Service Discovery Protocol, basis discovery protocol of UPnP
ULCO	Upper-Level Context Ontology, existing ontology for more abstract context information
UML	Unified Modelling Language, object-based modelling technology
UPnP	Universal Plug and Play, a set of protocols that allows devices to seamlessly connect to each other
URI	Uniform Resource Identifier, unique identifier for resources, particularly in WWW
URL	Uniform Resource Locator, identifier for resource locations
W3C	World Wide Web Consortium, standardization organization for web technologies
WWW	World Wide Web, application layer protocol widely used for the Internet
XML	Extensible Mark-up Language, information representation technology

1. Introduction

The motivation for this work derives from the concept of ubiquitous computing [1] in which the computational resources are graciously integrated with the human users so that their services are present everywhere and the required human intervention in providing their functionality is minimal. For an efficient implementation of this concept the services need to adapt to the changing conditions of the surrounding environment [2]. This requirement for the adaptation is also a key issue in other fields, such as nomadic computing [3]. Nomadic computing is defined as providing anyplace Internet access by exploiting the surrounding information for the adaptation.

Context-awareness establishes a base for the adaptation, which can only be achieved with appropriate acquisition of the related context information. However, the constitution of the context information required to support the adaptation might need a compilation of information from separate sources in different physical locations. The mechanisms for acquiring and processing the distributed context information are switching from the application side to the middleware. The trend in facilitating the development of context-aware applications is towards service-oriented solutions, which removes the restraint of supporting context-awareness from the application [4]. The fundamental research problem of this thesis derives from this development facilitation trend along with the requirement for context-awareness support in different fields of computing. The problem is to find a solution for dynamic acquisition and representation of distributed context information and its efficient provisioning for the applications in order to enhance their adaptivity and context-awareness.

To solve the research problem, the goal is to identify the requirements and functionalities for a service architecture that enables context-awareness by compiling the distributed context information from separate sources and providing the applications with access to it. The only way to find a solution to this is a wide literature review of the state-of-the-art research and technologies regarding context-awareness, service provisioning and closely related architectures for supporting context-awareness. In particular, the goal is to develop and validate a novel service architecture that supports context-awareness and whose design is based on the state-of-the-art technologies in the field of ubiquitous computing. The architecture's novelity value derives from the concept of a dynamically evolving semantic model of context information that it utilizes to provide a solution for the research problem.

The domain of the work, related technologies and research, including a couple of architecture solutions for supporting context-awareness, are presented in the literature review in Chapter 2. The requirements are summarized and the design of such a service is presented in Chapter 3. According to the design, an architecture for the service and its internal components is contemplated and presented in Chapter 4. The prototype implementation of the service architecture itself and an application requiring context-awareness support are created to validate the design. The architecture validation is presented in Chapter 5.

Once the design and validation are presented, the outcome of this work is discussed in Chapter 6, which covers the comparison with existing architecture solutions and dissertation of the architecture's strength and weaknesses. Chapter 6 covers discussion of the development targets that are derived from the architecture's dissertation. Finally, the conclusions from the work are presented in Chapter 7.

2. Related research and technologies

The results of the research related to this thesis are presented in this chapter. The research focused on the field of context-awareness and the technologies related to enabling context-awareness. Context and context-awareness are discussed at the beginning of the chapter and then the related enabling technologies are presented. Existing systems that provide the applications services for enabling their context-awareness are also presented at the end of this chapter to give insight into the previous research done in the field of context-awareness.

2.1 Introduction to context-awareness

Ubiquitous computing was introduced in 1991 by M. Weiser [1] and set forth a vision of people and an environment augmented together with computational resources that provide services everywhere. To achieve the essence of the vision, in which the computing environment is gracefully integrated with human users, ubiquitous systems need to be context-aware [2]. As the computational resources augment the people and environment together, both of which can change their context rapidly, the services must adapt to new situations in order to be able to provide services that are suitable for the current situation. This is where context-awareness becomes the key requirement for the ubiquitous computing systems [5].

In addition to ubiquitous computing, other similar fields of computing, such as pervasive and nomadic computing [3], have also identified context-awareness as a requirement for the systems belonging to the field. Pervasive computing is stated in [5] to be just a new term for ubiquitous computing, but it is recognized and used in the literature as a field of computing on its own. Nomadic computing defines a concept of providing Internet access for the user anyplace and anytime by utilizing portable computing devices in conjunction with communications technologies. But, common to all computing fields, the successful achievement of their concepts requires the systems to be context-aware.

To enable context-awareness the system needs to process and represent the context information it gathers from the environment. Context information, its methods of representation and the concept of context-awareness are discussed in the following sections; the definition of a context monitoring service is explained at the end of this section.

2.1.1 Context information

Context information is important for applications that need to adapt to situations in which the user's context is rapidly changing [6]. It is gathered from the environment surrounding the system to be utilized for enabling better adaptability of the applications. But in order to utilize context information efficiently we need to understand what the context is and how it can be used.

Researchers have come up with several different definitions for the term 'context', but most of them are synonyms for the word or the term is explained by examples. In the work of Schilit and Theimer [7] the definition of context is given by an example where the context is referred to as location, identities of nearby people and objects, and changes to these objects. This definition does not explain whether a type of information, which is not listed in the definition, is context or not. Other definitions that provide synonyms for the context are difficult to apply in practice. Examples of the synonym definitions in [8] and [9] refer to context as the environment or situation.

In [6] Dey et al. state that the context definitions created by Schilit et al. [10], Dey et al. [11] and Pascoe [12] are closest to the spirit they desire, but are too specific and cannot be used. The more comprehensive definition for the context is presented as "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 used and an application, including the user and applications themselves." [6]. For the primary context types, Dey et al. identify location, identity, time and activity that characterize the situation of a particular entity. These context types act as indices for other sources of context information and answer the questions of who, what, when and where [6].

2.1.2 Representation of context and ontologies

A uniform representation of the context is required to enable understanding of context information between different computational units and applications. To address this, ontologies have been studied to provide the sharing and understanding of the context. In one definition the ontology has been described as an explicit specification of conceptualization, which was further elaborated into a formal, explicit specification of a shared conceptualization. The conceptualization is meant to refer to an abstract model of some phenomenon in the world that identifies the relevant concepts of that phenomenon. In this definition the types of concepts and the constraints they use should be explicitly defined. The ontology should also be machine-readable and capture consensual knowledge. These features, respectively, refer to formal and shared in the definition [13], [14].

Different domains of context information require specific ontologies for modelling the relevant concepts. However, it is an enormous task, if not impossible, to develop a comprehensive ontology that covers all the context information that can exist. The criteria for ontology development that have proven useful are summarized in [15]. The criteria and their explanations are provided below.

- Clarity and Objectivity, the ontology should provide objective definitions and natural language documentation to bring meaning to the defined terms.
- Completeness, a definition expressed in terms of necessary and sufficient conditions is preferable to a partial definition.
- Coherence, to allow consistent inferences with the definitions.
- Maximum monotonic extendibility, introduction of new terms to an existing ontology should not require revision of the existing definitions.
- Minimal ontological commitments, to make a minimal number of claims on the world being modelled, giving the freedom to specialize and instantiate the ontology as required.
- Ontological Distinction Principle, the classes in the ontology should be disjointed.

- Diversification of hierarchies, increase in the efficiency of multiple inheritance mechanisms.
- Modularity, different modules should have the minimum number of relationships to each other.
- Minimization of the semantic distance between sibling concepts, similar concepts should be grouped and represented using the same primitives.
- Standardization of names whenever possible.

Ontologies can be used as a mediator in communication between people and systems. They also support the design and development of knowledge-based software systems such as the Context Monitoring Service introduced in this work. The context information that the CMS acquires and processes can efficiently be represented with ontologies and shared further with other entities without losing any knowledge [16].

2.1.3 Context-awareness

Context-awareness is an ability of a computing system to utilize the relevant context information in the execution environment in order to alter its behaviour and act as expected in the current situation. There are several existing definitions of the concept of context-awareness that define the term from different points of view. However, Dey and Abowd have introduced a general definition in [6], which is informed by all existing definitions; “A system is context-aware if it uses context to provide relevant information and/or service to the user, where relevancy depends on the user’s task.” [6].

In [6], context-aware applications have been divided into four different categories according to the purpose for which they utilize the context-awareness: proximate selection, automatic contextual reconfiguration, contextual command and context-triggered action applications.

- Proximate selection applications use context information for emphasizing the items that are relevant to the user’s context. For example, the nearest printers are shown first or emphasized in the printer selection list.

- Automatic contextual reconfiguration applications retrieve information for the user based on the available context. This system-level technique creates automatic binding to the available resource based on the current context. An example could be a museum tour guide application that shows additional information on the user's PDA for items the user is assumed to be looking at.
- Contextual command applications are executable services that are made available or their execution is modified due to the user's current context.
- Context-triggered actions applications are services that are executed automatically when the current state of the context matches the defined conditions.

Deployment of a context monitoring service that facilitates the development of applications comprising all the context-aware application categories is presented in this work. The context information and context-awareness is provided by the service but the decision on what to do with the information is left for the application developers. The Context Monitoring Service architecture is designed for utilization by applications belonging to all the context-aware application categories.

2.1.4 Context monitoring service

A definition of the context monitoring service is a service that doesn't utilize the context information it gathers and maintains to execute any tasks or processes but provides an interface for accessing the information to enable its utilization by applications. The applications that are context-aware utilize the context information of the surrounding environment to adapt their behaviour according to the current situation. The applications might also require the context information to provide relevant information or services to a human user or to another computing entity, where relevancy depends on the current task [6]. To achieve this, the application requires functionalities for gathering, processing and maintaining the context information. Deploying a context monitoring service that is responsible for the gathering and management of the context information helps the application to focus on other more important tasks. The context monitoring service can also provide the knowledge to several applications at the same time, thus enabling the adaptability of all applications that desire to be context-aware.

The context monitoring service should be deployed in such a way that it can provide the services to multiple clients. To address this, the context monitoring service needs to be deployed on a platform that holds the applications and the service. A promising platform for the deployment could be the MidGate [17] middleware service platform, which is described as “a service platform that provides communications middleware, a set of generic service elements and a dynamically re-configurable service framework that can be applied in the development and deployment of any application for a distributed service gateway based environment.” [17].

The MidGate architecture introduces a Generic Service Element (GSE) [18] called the Environment Monitoring GSE, which provides services for monitoring changes in the computing environment. The Environment Monitoring GSE is explained as hosting “a retrievable environment profile, which can be thought of as a snapshot of the dynamically changing context taken at the time of retrieval.” [17]. The context information is provided to the applications deployed in the MidGate by providing a service that allows listener registration for monitoring changes in the context.

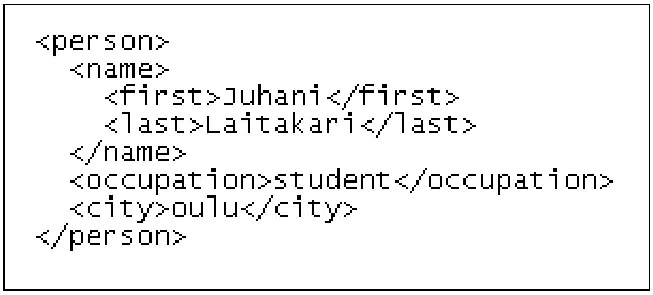
2.2 Technologies

This section reviews the technologies related to context representation, context-awareness and the context monitoring service designed in this thesis. Firstly, technologies used for representing the context information as standardised formats are discussed. The context information representation utilizes several mark-up languages to enable the human and machine interpretation of the information. Mark-up languages extend or utilize other existing mark-up language to provide a more informative description of the context. As the context can be represented in a descriptive way for humans and machines, a system needs to be created to hold and process the information. The mark-up languages presented in the subsections are recommendations by the World Wide Web Consortium (W3C) [19] and some are part of the Semantic Web technologies [20]. Semantic Web is a project that intends to create a universal medium for information exchange by creating computer-processable mark-up languages for documents on the World Wide Web (WWW). The Semantic Web technologies offer an approach to processes and information management, the fundamental principle of which is the creation and use of semantic metadata [21].

This section also reviews the technologies that provide frameworks to establish the services that provide the context information further to be used for adaptivity and context-awareness. The Jena Semantic Web Framework (Jena) is a feasible technology to be utilized for deploying the context information storage and processing capabilities. Furthermore, to provide the service interfaces to access the context information, service framework technologies like the Open Services Gateway Initiative (OSGi) are needed. Finally, the UPnP technology is presented, which can be seen as a key enabler for the dynamic acquisition of the distributed context information.

2.2.1 Extensible Mark-up Language

Extensible Mark-up Language (XML) [22] is a simple and flexible text format that was originally designed for large-scale electronic publishing. XML is widely used on the WWW as a data exchange format, and its role is growing larger all the time. The advantages of XML are that it is independent of platform, software and hardware configurations, and it is self-descriptive and interpretable by both humans and the machine. A simple example XML document is provided in Figure 1.



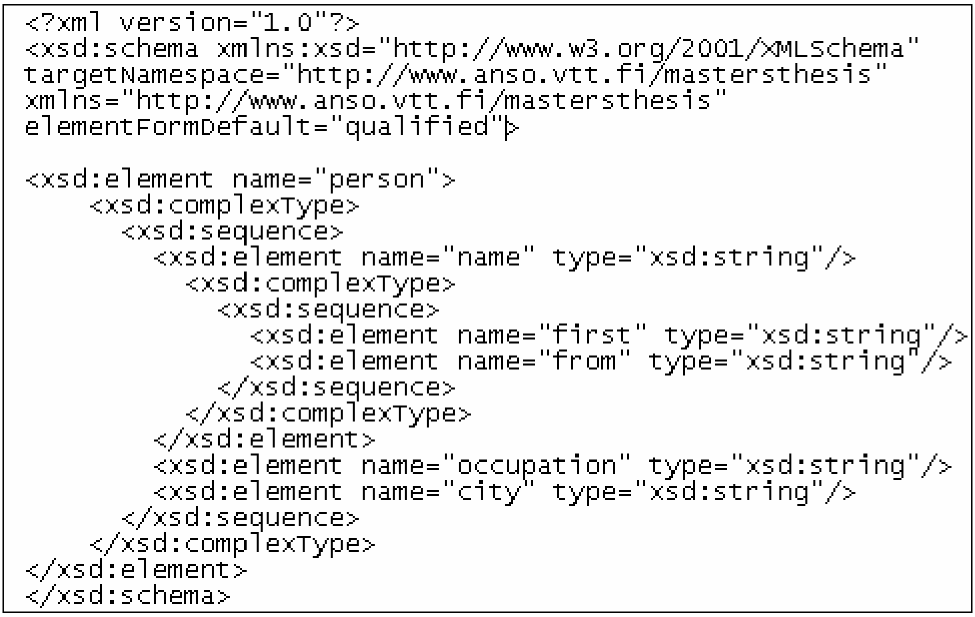
```
<person>
  <name>
    <first>Juhani</first>
    <last>Laitakari</last>
  </name>
  <occupation>student</occupation>
  <city>oulu</city>
</person>
```

Figure 1. Personal information stored in XML format.

The example document contains personal information such as name, occupation and city of a person stored in the XML format. From the example one can see that XML is a self-descriptive format and the machine can interpret the content using the tags marked as <tag>. The tags, however, are not specified by XML but by the author of the XML document.

2.2.2 XML Schema

XML Schemas are used to define the structure, context and semantics of an XML document [23]. The purpose of XML Schema is to define the class of XML documents that have the same semantics as those defined by the Schema [24]. XML Schema specifies typed definitions and element declarations under a particular namespace and can be seen as a vocabulary for XML documents of a similar class. XML documents that conform to a particular XML Schema are often called instance documents of the Schema [24]. The namespaces in XML Schema documents are used to distinguish the different type definitions and element declarations from each other. An XML Schema document for the XML instance document shown in Figure 1 is illustrated in Figure 2.



```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.anso.vtt.fi/mastersthesis"
xmlns="http://www.anso.vtt.fi/mastersthesis"
elementFormDefault="qualified">

<xsd:element name="person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="first" type="xsd:string"/>
              <xsd:element name="from" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="occupation" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```

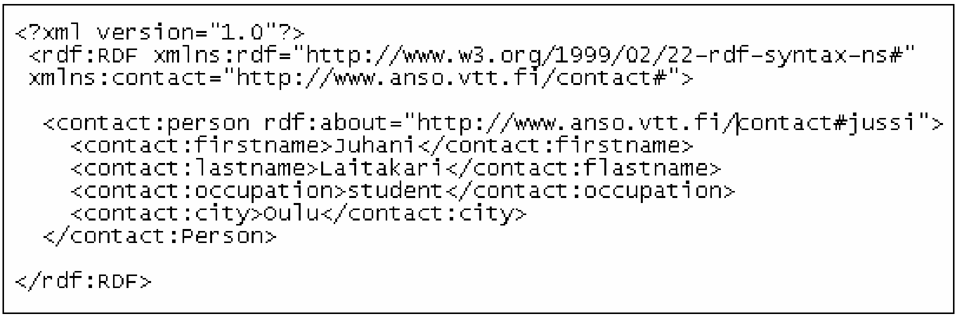
Figure 2. An example XML Schema document.

The first part of the XML Schema document is the section that defines the namespaces used in the schema. The XML instance document must also define the same namespace and have a reference to the Schema document. The type of “person” element is complex because it contains other elements like, in this example, the other complex element “name” and the simple elements “occupation” and “city”. In this example a built-in data type “string” is defined for all the simple elements, meaning that the values of these elements are represented as strings.

2.2.3 Resource Description Framework

Resource Description Framework (RDF) is a language designed to describe the information on resources in WWW [25]. However, RDF can used to represent things that aren’t even retrievable from the Web, such as physical objects or the context in general. RDF is based on an idea of identifying the resources by Uniform Resource Identifiers (URI) and describing the resources in terms of properties and property values. A unique URI is defined for identification of each resource or property. Properties represent the attributes of the resource and are expressed as values, such as numeric or string value. Resources, properties and their values are represented as triples, similar to the subject, verb and object of a simple sentence.

RDF uses a special RDF/XML format to record or exchange the resources with properties and values that enable machine processing of the RDF documents. An example of an RDF/XML format document is provided in Figure 3.



```
<?xml version="1.0"?>
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:contact="http://www.anso.vtt.fi/contact#">

  <contact:person rdf:about="http://www.anso.vtt.fi/contact#jussi">
    <contact:firstname>Juhani</contact:firstname>
    <contact:lastname>Laitakari</contact:flastname>
    <contact:occupation>student</contact:occupation>
    <contact:city>Oulu</contact:city>
  </contact:Person>

</rdf:RDF>
```

Figure 3. Example of RDF/XML formatted RDF document.

The example shows the same details of a person in RDF. In the example the upper part of the figure is the namespace definitions for the RDF document and the lower part shows the resources, properties and their values. The resource in the example is a person who is identified by the URI *"http://www.anso.vtt.fi/contact#jussi"*.

RDF provides a common framework for describing the context data, thus the data can be exchanged between applications without losing any data. RDF is a feasible language for representing the context knowledge data in the context monitoring services.

2.2.4 RDF Schema

RDF Schema (RDF-S) [26] is a language for describing RDF vocabularies that define classes and properties to describe other resources and properties expressed in RDF. As previously explained, RDF provides a way to express statements on resources using named properties and values. The problem with RDF is that it has no mechanisms for describing these properties, and the relationships between RDF properties and resources cannot be expressed in RDF. RDF Schema is used to establish relationships between the RDF resources, which are categorized into the domains and ranges of the properties. The properties have a domain that contains the possible classes of resources to which the property can be applied. The range of the property defines the group of values that can be assigned for a certain property.

A concrete example of domains and ranges of properties is a case where the RDF schema classes "Car" and "Bicycle" have a property called "hasTire". Now the "hasTire" property has a domain of the classes "Car" and "Bicycle", which means that the "hasTire" property can be assigned to all RDF resources that belong to these classes. The range of the property "hasTire" can, for example, include the RDF Schema classes "BigTire" and "SmallTire". Hence the RDF instances of these classes can be assigned to the "hasTire" property.

RDF and RDF Schema together offer potential methods to describe the context information in the context monitoring service. Some ontology languages such as Web Ontology Language (OWL) are already based on RDF and RDF Schema languages.

2.2.5 Web Ontology Language

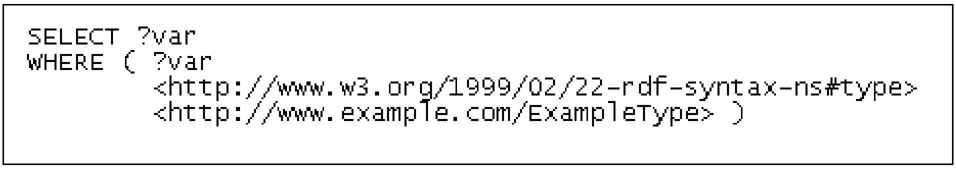
Web Ontology Language is a language for defining and instantiating formal ontologies that provide greater machine interpretability of Web content than is supported by XML, XML Schema, RDF and RDF Schema [27], [28]. The difference between ontology and, for example, XML schema is that ontology is knowledge representation, not a message format. Most industry-based Web standards consist of message formats and protocol specifications that have given operational semantics but no support for reasoning outside the transactional context. An example from [28] illustrates the semantics of an industry-based Web standard message: "Upon receipt of this *PurchaseOrder* message, transfer *Amount* dollars from *AccountFrom* to *AccountTo* and ship *Product*." The difference with ontologies is that it cannot be concluded from this message that if the *Product* type is Chardonnay, it must also be white wine.

OWL was developed as vocabulary extension of RDF and it also depends on the constructs defined by RDF Schema and XML Schema data types. OWL language is divided into three different sublanguages – OWL Lite, OWL DL and OWL Full – which provide increasingly expressiveness respectively. OWL Lite is intended for lightweight implementations and provides limited expressiveness compared with the other two. OWL DL guarantees computational completeness and is designed to support the existing description logic business segment OWL DL is named after. OWL Full does not guarantee any computational requirements but it does provide the maximum expressiveness and syntactic freedom of RDF. However, it is very unlikely that any reasoning software will be able to support every feature of OWL Full. [28]

OWL is one of the potential languages with which to represent knowledge and context information in context-aware systems. Designing an expressive ontology from scratch is a challenging effort and building a reasoning tool to support the ontology creation is even more complex. Research has revealed that several tools for developing ontologies that support different sublanguages of OWL can be found. Ontology editor tools include, for example, Protégé [29], TopBraid Composer [30] and SWOOP [31]. Protégé is seen as a potential ontology editor and is selected for use because it is an open-source software tool. Protégé can be utilized for developing new ontologies or extending existing ones for the context monitoring service described in this work.

2.2.6 RDF Data Query Language

RDF Data Query Language (RDQL) [32] is an SQL-like language for querying data models that are expressed in RDF. RDQL has evolved from several other RDF query languages, such as SquishQL [33], and includes the ideas presented in [34]. An RDF model is a graph that is often expressed as subject, predicate, object triples, and the RDQL query consists of a graph pattern as a list of these triples. The triples in the graph pattern are comprised of named variables and RDF values as URIs or literals. An RDQL query can contain constraints for the values in the query pattern and a list of variables that are returned in the result set of the query. Results for the RDQL queries are searched from both the predefined triples in the knowledge base and the virtual triples that are inferred from known triples. A simple example of an RDQL query is shown in Figure 4.



```
SELECT ?var
WHERE ( ?var
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://www.example.com/ExampleType> )
```

Figure 4. A simple example of an RDQL query pattern.

In the example the pattern "?var" is the variable into which the results are stored. The pattern can comprise several variables that are returned in the result set. Here the variable is set as a subject of the triple pattern, and the predicate and object are defined for conditions. This query returns all the triples that contain a subject having the predicate and object defined in the query pattern.

RDQL enables expressive queries to the context information model and can be either very detailed or comprise a wide range of information. RDQL can be seen as a feasible technology to provide access to context information presented as an RDF model.

2.2.7 Jena Semantic Web Framework

Jena Semantic Web Framework [35], or Jena, is a framework for creating Semantic Web applications. Jena is an open-source framework written in Java and provides a programmatic environment to develop semantic knowledge bases using the RDF, RDF Schema and OWL languages. The Jena framework also includes a rule-based inference engine that can be used to deduce new information from the predefined information in the knowledge base.

The Jena Semantic Web Framework consists of two versions of the implementation, Jena 1 and Jena 2. Jena 1 provides a rich Application Programming Interface (API) for manipulating RDF graphs and several tools are included around the API for support. The tools include modules for parsing and writing RDF/XML formatted ontology documents, and a query language support. The API also provides functions to store the RDF graphs to a persistent storage or in the memory [36].

The upgrade of Jena 1 to Jena 2 improved the framework by bringing additional functionality that supports RDF Schema and OWL, and new APIs were created for the developers to be able to access the ontologies and process the vocabularies. Jena 2 provides an extension point that allows development of new sources of data triples, which are dynamically created. The triple sources can be, for example, an inference engine that produces virtual triples of inferred information. The inference engine supports the semantics of RDF and OWL, and the engine can be extended with self-created inference rules [36].

The second version of the Jena Semantic Web Framework is a very feasible technology for creating the knowledge base for the context information the context monitoring service is designed to contain. Jena 2 provides support for reading and writing RDF/XML ontologies defined with the OWL language, which enables easy utilization of developed ontologies for the context monitoring service. The context information in the knowledge base can be accessed through the support for the RDF query languages. Jena also includes inference engines for RDF and OWL semantics and can be further extended with special inference rules.

2.2.8 Open Services Gateway Initiative

The Open Service Gateway Initiative [37] is an alliance forum focused on the interoperability of applications and services based on its component integration platform. The OSGi alliance has defined a framework and service platform specification for an open service gateway. The specification defines standardized APIs for designing the interfaces between the components of the service gateway. The components include the services, the OSGi framework, access to devices and service management. The specification does not define any API for the interface design between the services and applications that utilize them. The design of the service interfaces towards the applications is left open for the service developers. Currently, the OSGI Service Platform specification is in release 4 [38].

The OSGi technology provides a possibility to install updates or remove software components on the fly without having to interrupt the operation of the device [39]. These software components can be applications or libraries to be utilized by the other applications, and the components can be dynamically discovered inside the OSGI framework. The software components, or bundles, are Java Archive (JAR) packets that include the control interface, service code and service interfaces for the applications. The core component of the specification is the OSGi framework that provides the standardized environment for the software bundles. The OSGi framework is divided into four different layers, around which are the software bundles. The layered division of the OSGi framework is illustrated in Figure 5 from [39].

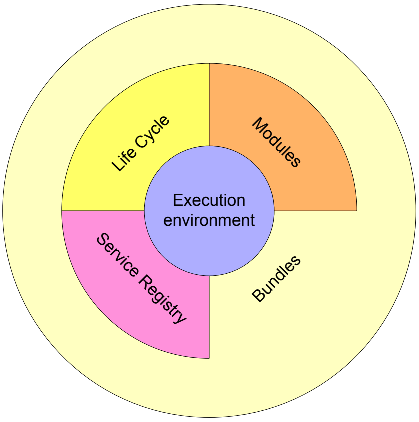


Figure 5. The layer division of the OSGi framework.

The execution environment specifies the Java environment. Valid execution environments include, for example, Java 2 Standard Edition (J2SE) and different

Java 2 Micro Edition (J2ME) configurations and profiles such as Connected Device Configuration (CDC), Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP). More information on the Java platform can be found in [40].

The modules layer defines the class loading policies by specifying the software bundle's private classes and the classes that are shared with other bundles. The Life Cycle module provides the functionalities to dynamically install, start, stop and uninstall the bundles. The module also keeps track of the dependencies between the operating bundles so that a bundle isn't run before the required library bundles are installed. The last module, the Service Registry module, provides the cooperation model for the software bundles. The Service Registry provides the bundle's service interfaces to other bundles and also notifies new registrations or removals of bundles with a number of defined events. [39]

2.2.9 Universal Plug and Play

Universal Plug and Play (UPnP) is an architecture for pervasive peer-to-peer network connectivity of different networked devices. Microsoft initiated the UPnP standard to extend the existing Microsoft Plug-and-Play peripheral model. The goal of the UPnP technology is to enable the advertisement, discovery and control of networked devices and consumer electronics [41]. UPnP enables the device to dynamically join a network, obtain an Internet Protocol (IP) address, present its own services and discover other devices in the network. Furthermore, UPnP is not only a technology but also a cross-industry initiative, which is embodied in the UPnP Forum [42] that develops Device Control Protocols (DCP) for standardized device interaction [43].

A UPnP network consists of devices, services and control points. A UPnP device is a container for different services the device provides; the device can also contain other nested devices. For example, a multifunctional printer can contain the printer and a scanner in the same device. Each UPnP device has an XML formatted device description that contains the details of the device. The device description also contains a pointer to the descriptions of the service the device provides. This service description is also in XML format and defines the details of the device's service. Each service the device contains has its own service description document. The service can include different actions to query or manipulate the state of the service. The state of the service is expressed as state variables, which are located in the state table of the service. Each service can contain several state variables representing different states of the service. In addition to the state table, a UPnP service also consists of a control server and an event server. The control server is used for receiving action requests from the controllers and executing them. The event server of the UPnP service notifies the interested parties of changes in the service state. An example of a UPnP network and its components is presented in Figure 6 [44, p. 10].

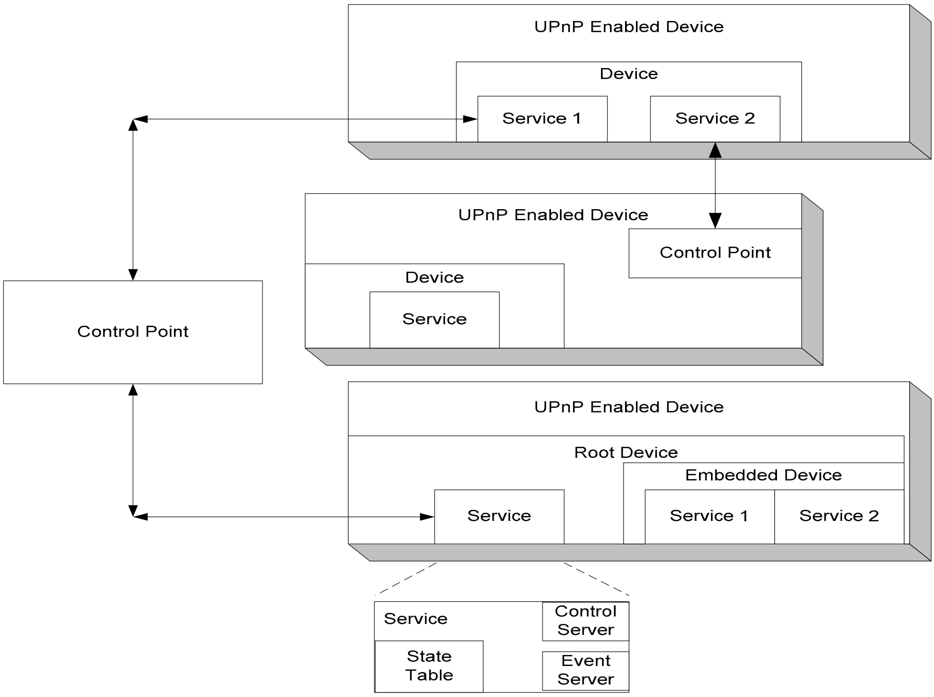


Figure 6. UPnP control points, devices and services.

UPnP control points operate as controllers of the UPnP network and are capable of discovering and controlling the services. A control point retrieves the device's device description document and the associated service descriptions. After this, the control point can control the device by invoking the actions introduced in the service description documents. Control points need to register to the service's event source in order to get the state change notifications [44].

The interaction of the UPnP network is divided into six different steps: addressing, discovery, description, control, eventing and presentation. Each of these steps, and the associated protocols, are now briefly described.

Addressing

Addressing is used to obtain the IP address for the devices from the DHCP (Dynamic Host Control Protocol) [45] server using a client that each device contains.

Discovery

The discovery step advertises the device's services by using the UPnP discovery Protocol, which is based on the Simple Service Discovery Protocol (SSDP) [46]. The device sends a discovery message to a standard multicast address to advertise its services and embedded devices. The control point listens to the address and receives the information on the devices. A newly joined control point also sends discovery messages to locate the interesting devices and messages.

Description

The control point uses the Uniform Resource Locator (URL) presented in the discovery message to retrieve the device's device description and service descriptions. The descriptions are retrieved by issuing a Hyper Text Transfer Protocol (HTTP) GET message to the device URL.

Control

The control point invokes service actions by sending XML formatted control messages to the device's URL. The required arguments and a return value of a service action are defined in the service description document. The protocol utilized for the device controlling purposes is the Simple Object Access Protocol (SOAP) [47].

Eventing

Through eventing the control point receives notifications on state changes to the service. The control point needs to subscribe to listen to the state changes of a state variable. The subscription must be renewed over a period defined in the subscription message. Events messages are General Event Notification Architecture (GENA) [48] NOTIFY messages that are sent using HTTP.

Presentation

In addition to controlling and eventing mechanisms, UPnP devices may also provide a web interface for controlling the services. The interface is a normal, public Hypertext Mark-up Language (HTML) page that the control point can present to a user. Access to the interface is enabled by delivering a hyperlink to the interface's web page in the device description document. The UPnP specification presents no strict requirements for the page except that the page needs to be written in HTML v.3.0 or later [49].

2.3 Architectures of existing context monitoring services

A lot of research has been done in the area of context-awareness and several services that enable context-awareness have evolved during the research. To provide insight for the reader, a couple of these context monitoring services have been chosen to be presented in this section. The selection criteria were that the existing systems provide architectural and functional solutions that conform to the requirements of the context monitoring service designed in this work.

2.3.1 CoBrA

Context Broker Architecture (CoBrA) [50] is an architecture for supporting context-aware systems in smart spaces. Central to the architecture is a broker that maintains the shared model of the context for all computing entities in the space. In addition to maintaining the model, the broker also is responsible for acquiring the context information from the sources that are unreachable by resource-limited devices, reasoning new context information from the acquired information, preserve the consistency of the knowledge in the model and protecting the user's privacy by controlling the sharing of the context information [51]. The design of the context broker is illustrated in Figure 7 from [51].

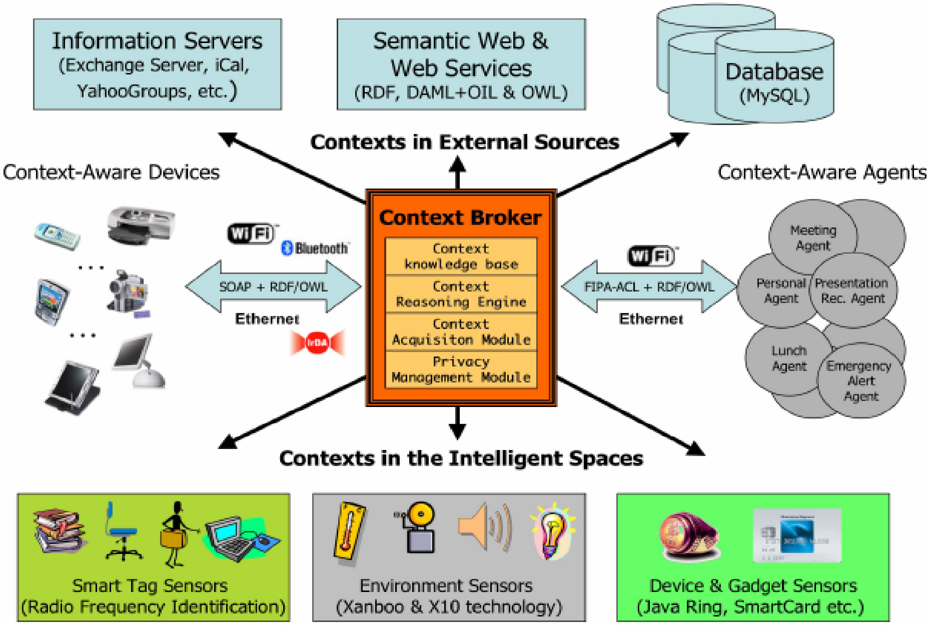


Figure 7. The design of the context broker in CoBrA.

The context broker in CoBrA contains four different functional components: context knowledge base, context reasoning engine, context acquisition module and policy management module [51]. The context knowledge base is a persistent storage of the contextual knowledge that provides an API with access to the information in the base. It also contains the ontologies of a specific smart space and some heuristic information associated with the space. The context reasoning engine is a reactive inference engine that reasons over the context information in the context knowledge base. The reasoning engine deduces new context information from the existing by using the ontologies. The inference engine also detects and resolves inconsistent information in the knowledge base by using heuristic knowledge. The context acquisition module is responsible for providing a middleware abstraction for the context acquisition. It shields each sensor's specific implementation for information acquisition from the high-level applications. The last functional component is the policy management module, which is a set of inference rules to deduce instructions for deciding on permissions for the shared context information and receiving notifications of context changes.

2.3.2 Semantic Space

Semantic Space is described by its developers as “a pervasive computing infrastructure for smart spaces that exploits Semantic Web technologies to support explicit representation, expressive querying and flexible reasoning of context in smart spaces” [52]. The Semantic Space infrastructure is focused on the explicit representation of context, context querying and context reasoning. To address these issues the infrastructure is designed to consist of different collaborating components that provide the required functionalities for context representation, querying and reasoning. These components, presented in Figure 8 from [52], are: context knowledge base, context query engine, context reasoner, context aggregator and different context wrappers.

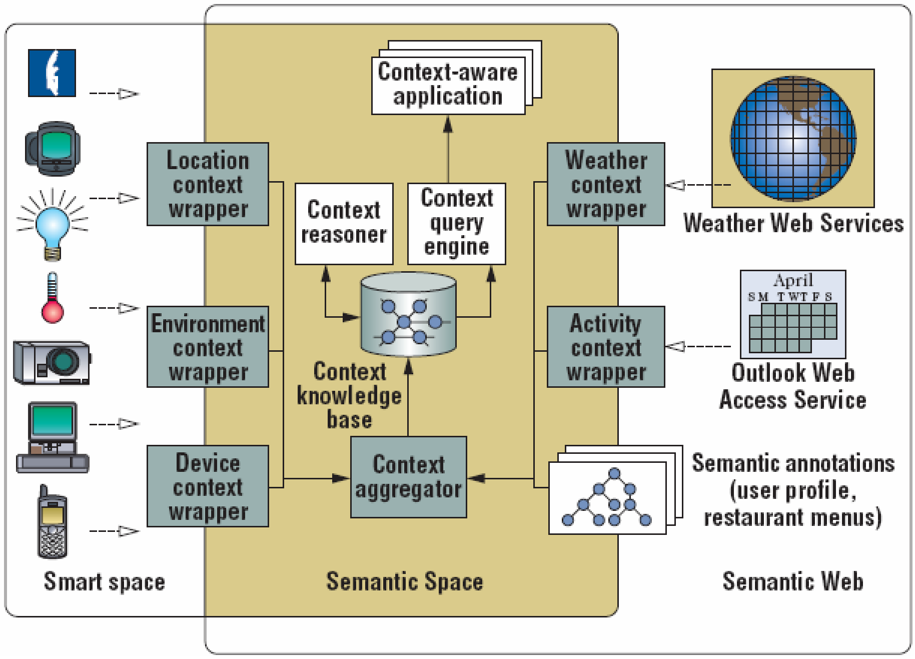


Figure 8. The Semantic Space context infrastructure.

The context knowledge base contains the context information represented as a context model, which is defined with different ontologies. To support different kinds of context information in the same context model, the Semantic Space developers have developed an Upper-Level Context Ontology (ULCO), which can be extended with more specific ontologies. The ULCO provides classes of real-world objects that link together to form a skeleton of a contextual environment for the context model. Access to the context model is enabled by the context query engine, which provides an abstract interface for the applications. The context model is queried with the RDQL query language, which supports expressive queries to the context information. The context reasoner infers abstract higher-level context from the information produced by the sensors. Semantic Space allows application-specific inference rules to be used in deducing new context information. Each application can add rules to the inference engine to receive the desired high-level context information. This can generate conflicting results, but the reasoner doesn't assert the information in the context knowledge base, thus avoiding conflict in a model. Context wrappers obtain the raw context data from the sensors and transform it to context mark-ups. The wrappers provide the abstraction of the data that different sensors produce and hence provide the unified interface for acquiring the data from all the sensors. The wrappers are implemented as UPnP services to enable dynamic joining to a Smart Space. The last component is the context aggregator, which discovers the context wrappers and gathers the context information from them. As the context wrappers are implemented as UPnP services, it is natural that the aggregator is deployed as a UPnP control point. The context aggregator can dynamically discover all the joining context wrappers and subscribe to receive the context data from them.

2.4 Terminology

For the convenience of the reader, the terms used in Chapter 2 and throughout this work are gathered and summarized in Table 1.

Table 1. The terminology used in this work.

Context-awareness	An entity’s ability to be aware of the surrounding context information and use the information to adapt its behaviour according the current context.
Context monitoring service	A system or process that provides the means for other entities to access the context information it possesses, thus enhancing their context-awareness.
Context information	Information on the circumstances and conditions surrounding an entity; represents the state of the entity’s surrounding environment.
Context model	A model of the context information where the semantic relations between the entities of the context information are modelled using some method of representation.
Inference engine / Reasoning engine / Reasoner	An engine that deduces new implicit context information from existing explicit context information according to the given inference or reasoning rules.
Knowledge Base	A database for knowledge management; provides the means for computerized collection and organization of knowledge.
Ontology	A language or vocabulary used to model the relationships and entities of context information to a context model.

3. Design of Context Monitoring Service

A context monitoring service that monitors the deployment environment's different context information and is also dynamic by its nature requires several different functionalities. These functional requirements and the design of the implemented Context Monitoring Service are covered in this chapter. The functionality of the CMS is decomposed into separate components and the structure of this chapter is organized accordingly. The first section provides an introduction to the service's functionalities and the following sections explain the service's different components and their deployment in detail.

3.1 Introduction to service

Context Monitoring Service is a service that acquires context information from its deployment environment and provides it for the applications to enable context-awareness. Acquisition of the context information is done by the different types of sensors that sense the conditions in the environment. Context information is stored to the semantic context model, which binds the information to a form defined by an ontology. To enable improved context awareness, a reasoning engine is assigned to the context model to deduce new information from the semantic relations of the information. A service interface to the model is needed to provide the data in the semantic context model for applications. This interface provides the mechanisms to make queries to the information in the model and register listeners with conditions that are triggers for notification events. The CMS is deployed into an OSGi framework, which provides reconfigurability by allowing addition and removal of software bundles in the framework in run-time. Thus applications can be installed and run without shutting the CMS down. The CMS is illustrated as a block diagram in Figure 9.

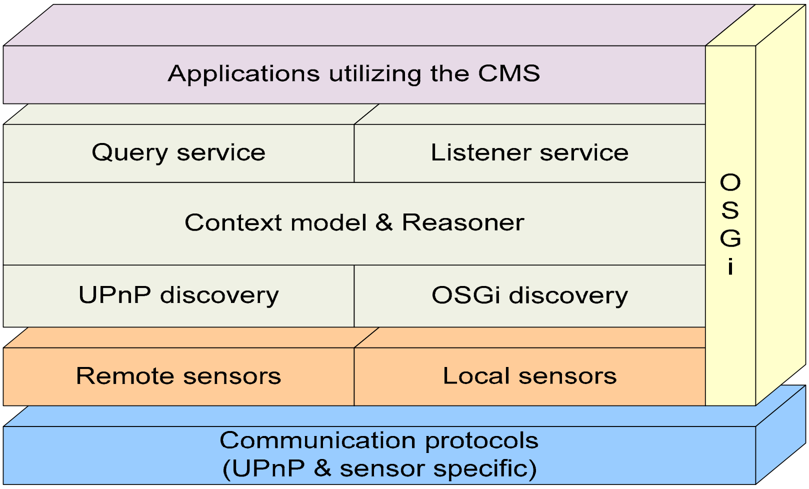


Figure 9. Illustration of the CMS as a block diagram.

Context information is dynamic by its nature, thus the service's context model must be updated after every state change. The context model receives context information from the sensors, which can be located in the same or a different remote computational unit. Locally connected sensors are discovered by the CMS via the OSGi service discovery and remote sensors are discovered via the UPnP device discovery. Remotely discoverable sensors are enabled by deploying the local sensors as UPnP devices. UPnP sensor devices have a special UPnP service that is used to receive the data from the sensors. The data the sensors provide is in raw form and needs to be manipulated to be suitable for adding to the context model. Sensor data is converted to the semantic statements that are suitable for altering the information in the context model.

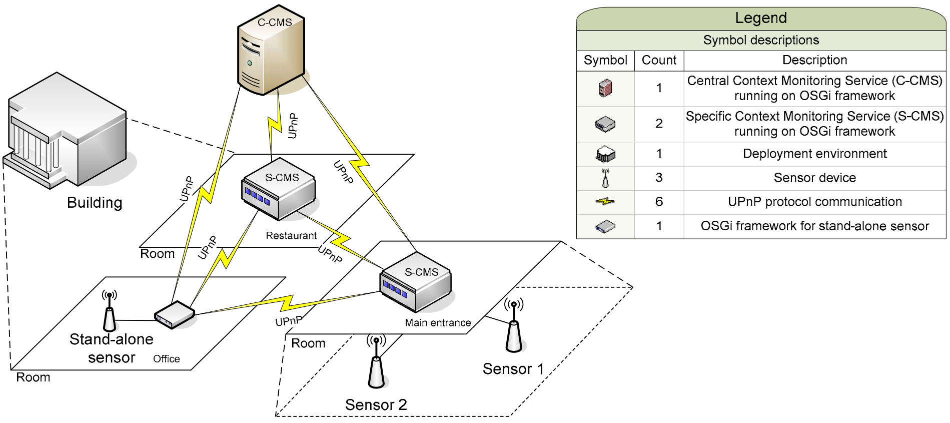
The Context Monitoring Service can be decoupled into the different functional components that are needed to provide the functionalities described above. The functional components of the Context Monitoring Service are listed in Table 2.

Table 2. Functional components of the Context Monitoring Service.

Component	**Description**
Context model	A knowledge base that contains the contextual entities and their relationships defined with ontologies.
Reasoner	An engine that deduces new information from the information in the context model.
Model querying service	Handles the queries to the context model and returns the results of the query.
Conditional eventing service	Manages the notification of listeners when a condition given in listener registration triggers the notification.
Model updater devices	Devices that provide new context information for the Context Monitoring Service.
Model updater device advertisement and discovery	Responsible for finding local and remote model updater devices and advertising the local model updater devices in the network as remote model updater devices. The data from the discovered model updater devices is provided further for context model updating.

3.2 Requirements scenario for the design

This section explains the scenario on which the design of the whole service is based. The requirements for the design are identified from this scenario. First, the structure of the scenario, the communications in it and the overall functionality are explained first, and the identified requirements for the design and their solutions are presented at the end of the section. Figure 10 shows the scenario's deployment environment in which the service is located and also illustrates the deployment aspects of the service.



Legend

Symbol descriptions

Symbol	Count	Description
	1	Central Context Monitoring Service (C-CMS) running on OSGi framework
	2	Specific Context Monitoring Service (S-CMS) running on OSGi framework
	1	Deployment environment
	3	Sensor device
	6	UPnP protocol communication
	1	OSGi framework for stand-alone sensor

Figure 10. Deployment environment of the Context Monitoring Service.

3.2.1 Deployment environment

A building containing several different sized rooms is the deployment environment for the Context Monitoring Service in this scenario. The CMS can be deployed as two separate components, which have slight differences in their configuration. The functionalities are exactly the same but the start-up setups of the components differ. The components of the CMS are the Central Context Monitoring Service (C-CMS) and the Specific Context Monitoring Service (S-CMS). The idea is that the C-CMS has a context model containing all the explicit and static information that cannot be sensed with the given sensors and needs to be configured manually. An example of this kind of information is the room configuration of the building and the information on the employee working in it. The context models of the S-CMS components only need to be configured with the room-specific explicit information the C-CMS does not yet contain. All the other information is received from the C-CMS or from the other S-CMS components. Thus the extension of the whole system is easy as possible. The deployment environment for the CMS components in the scenario is a building containing three different rooms. A mainframe inside the building is running a C-CMS component and three S-CMS components are deployed to rooms in the building.

Every room can have its own OSGi framework running the S-CMS. An S-CMS located in a room contains all the room's static context information, such as sensors deployed in the room, and the room's own information. Room information is used to map the room to the C-CMS's semantic model. The OSGi framework in the room can also be running room-specific controlling or maintenance applications that use the data from the S-CMS in the same OSGi framework. Sensors that are needed in the room are connected to the room's OSGi framework and controlling software bundles are installed in the OSGi framework. If the room is remarkably large and has different districts, special OSGi frameworks for controlling only the sensors can be deployed separately. The OSGi frameworks used for these sensors do not need any of the CMS components to be installed. Therefore, the sensors are called stand-alone sensors.

3.2.2 Communications

All the CMS components communicate with each other via UPnP. The sensors in the rooms can be connected direct to the computer running the OSGi framework with CMS or they can be deployed as stand-alone sensors. Directly connected sensors use their own communication protocols and the controlling software bundles must be installed in order to receive the sensor data. Stand-alone sensors consist of the OSGi framework, the sensor's software bundle, a possible driver for the sensor and a sensor physically connected to the computer. Stand-alone sensor devices communicate with other CMS components using the UPnP protocol.

The UPnP protocol enables the dynamic discovery of the connected devices in the whole system. In this scenario it means that the C-CMS and the S-CMS components automatically discover the newly connected S-CMS components and newly connected stand-alone sensors.

3.2.3 Functionality

If a new S-CMS is assigned to a room, it is discovered by the other CMS components. All the CMS components receive the base semantic model of the S-CMS, not the dynamic model compiled from other models of the CMS components. After receiving the model, the S-CMS starts listen to the changes in it. Before the assignment, the other CMS components only had information that the building contains a room, but had no information about the context of the room. Now, after the assignment and receipt of the S-CMS's semantic model, all the CMS components know about the room's detailed context information.

The received knowledge also depends on the sensors that have been deployed in the room, but now the other CMS components are aware of the contextual changes in the new room. For example, if a new directly connected sensor is deployed to the room, the room's S-CMS discovers it and starts to listen to it. Then the other CMS components listening to the S-CMS in the room become aware of the newly added sensor and receive the information it produces. The stand-alone sensors are discovered by all the CMS components in a similar way to new CMS components deployed to the environment.

When the new S-CMS is deployed, in addition to advertising its context model, it receives the semantic models of all the CMS components in the deployment environment. The deployed S-CMS extends its context model with the received models and starts listening to the changes in them. Thus the room's S-CMS becomes aware of context information on other rooms in the building. By listening to all the CMS components and stand-alone sensors, the room's own S-CMS notices all the changes occurring in the deployment environment.

3.2.4 Identified requirements

The scenario presented in Figure 10 was used to gather the requirements for the Context Monitoring Service. These requirements are taken into account when the CMS is designed and the solution should fulfil these requirements. The requirements are categorized to non-functional and functional requirements, where non-functional requirements apply when the system is not run and functional requirements cover the run-time of the system.

The requirements are derived from several quality attributes that are presented in [53]. Not all of the attributes are taken into account for the requirement analysis of the CMS, but the attributes that are seen as important for the design are selected. The derived requirements are presented along with the quality attributes that determine the requirement. The identified requirement descriptions and the design solutions to control the fulfilment of the requirements are listed in Table 3 and Table 4.

Table 3. Non-functional requirements for the design and their solutions.

Non-functional requirement description	**Solution**
Maintainability: As the each deployment environment is different to the other, the CMS should be able to easily modify or adapt to changing environments.	The utilization of ontologies provides a way to easily expand the deployment environment the CMS should cover. A new ontology to cover the environment can be designed and used to extend the existing ontologies.
Portability: The hardware can also vary between the deployment environments, thus the CMS should be easily run under different computing systems.	The CMS is designed to be run on the Java 2 platform, which can be easily ported to other computing systems.
Integrability: The separately developed applications should work together correctly.	Utilization of OSGi as a framework for the CMS and applications provides a standard way to collaborate with other components.

Table 4. Functional requirements for the design and their solutions.

Functional requirement description	**Solution**
Adaptability: As the deployment environment might expand during the system's run-time, the CMS should be able to easily and with minimum effort extend to cover the new environment.	Utilization of UPnP technology provides a way to expand the CMS with new CMS deployment components in run-time.
Interoperability: The CMS components and the stand-alone sensors of the CMS should use a way of communication that is understood by every component.	The use of UPnP technology in company with the ontologies as a message format provides a common way for deployment components to communicate with each other.
Availability: The CMS should be up and running as much as possible to preserve the context-awareness and the adaptability of the applications.	Utilization of the OSGi framework allows the installation of new components and applications without shutting the whole system down.

3.3 Context model

Context information enables the applications to adapt their behaviour according the current state in the context of the deployment environment. To be able to provide the context information for the applications the service must contain a model of the current state. This context model contains semantic representations of the context information that are different contextual entities and the relationships between them. A contextual entity can be, for example, a room of a house or a mobile phone in the room. The relationships between the entities could be that the room contains the phone and the phone is located in the room. A small example of entities and their relationships is illustrated in Figure 11.

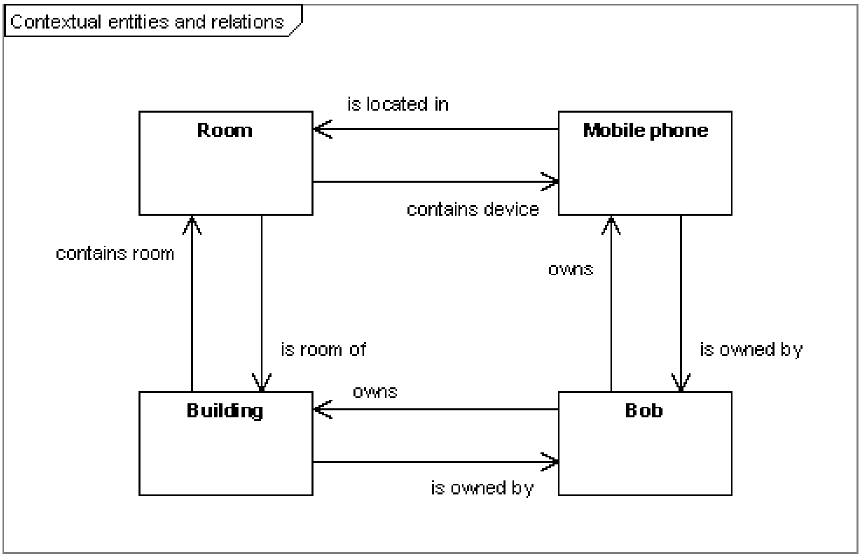


Figure 11. Example of contextual entities and the relationships between them.

The rectangular boxes in Figure 11 describe the contextual entities and the arrows between them are the relationships to each other. Starting from the entity Bob, it has relationships to the Mobile phone and Building entities, which defines that Bob is the owner of the Mobile phone and the Building. These two relationships define that the Mobile phone and Building entities also have relationships to Bob. If Bob owns an entity, the same entity is then owned by Bob. Relationships that have this kind of dependency on other relationships are called inverse relationships. Thus the “owns” relationship is the inverse relation of the “is owned by” relationship. In this example the Room entity also has inverse relationships to both the Building and the Mobile Phone entities. Room is a room in the Building and, inversely, the Building contains the Room. In the same manner, a room can contain devices – in this case the Mobile phone – and devices can be located in the room.

The context model is compounded of these entities and their relationships to each other. Entities and relationships are modelled in the context model as statements and all the data in the model is represented as RDF statements. A statement is an expression of the entity’s relationship to other entity in the form of subject, predicate and object. The syntax of the example statement expression could be <*Room, containsDevice, Mobilephone*> meaning that a room can contain a device, which in this case is a mobile phone.

The Jena Semantic Web Framework [35] is selected for the deployment of the context model for the service. As previously described, Jena is an open source Java-based framework for building Semantic Web applications and provides an RDF API with the possibility to read and write RDF in RDF/XML. An API is also provided for handling OWL-based graphs, which is the reason for selecting Jena to be used as a framework for the context model. Other features of the Jena framework are rule-based inference engine, query engine, and in-memory and persistent storage for the model. Persistent storing of data in the model is provided by an API to store and manage the data in a database. However, the current design is to keep the context model in the memory because it reduces the latency of changes done in the context model.

The rule-based inference engine of the framework is used as a reasoner to deduce new information from the model. Dynamic context information is received from discovered sensors and from the reasoner. The query engine of the Jena is used to provide applications with a possibility to query the data in the model. Jena provides a query engine for the RDQL query language, which is selected for utilization in the Context Monitoring Service.

3.4 Ontologies

Ontologies are needed to define the vocabulary for semantic representation of the context. Different domain-specific ontologies are designed for the context information in the service's deployment domain. The designed ontologies are divided into two categories: upper and lower-level ontologies. The upper-level ontology defines the more abstracted model, which is extended with more specific models defined with lower-level ontologies. The purpose of the abstract model is to define how the lower-level models extending the abstract model are related to each other.

Several separate ontologies need to be designed for the requirements of the Context Monitoring Service. First, the service needs an upper-level ontology to provide a base for the context model and define how the specific models are bound to the context model. Lower-level ontologies are needed to define the specific models that extend the abstract model. Lower-level ontologies are needed for each sensor to model the context information that the sensor is able to sense from the deployment environment.

In this work the OWL Web Ontology Language was chosen for the vocabulary to define the required ontologies. The reason for the selection is that OWL facilitates great machine interpretability and provides sufficient expressiveness for deducing new information. OWL DL was chosen from the OWL sublanguages of OWL Lite, OWL DL and OWL Full because OWL DL provides good computational properties while still retaining sufficient expressiveness.

3.4.1 Upper-level ontology

The upper-level ontology should only represent abstract context entities and their relationships to each other, thus providing an extendable base model for specific ontologies. The upper-level ontology presented here is designed to be extendable at least with the lower-ontologies needed in the Context Monitoring Service's deployment environment. The designed ontology's classes and their relationships are briefly explained in this section. The designed upper-level ontology for the service is illustrated in Figure 12 as a Unified Modelling Language (UML) class diagram.

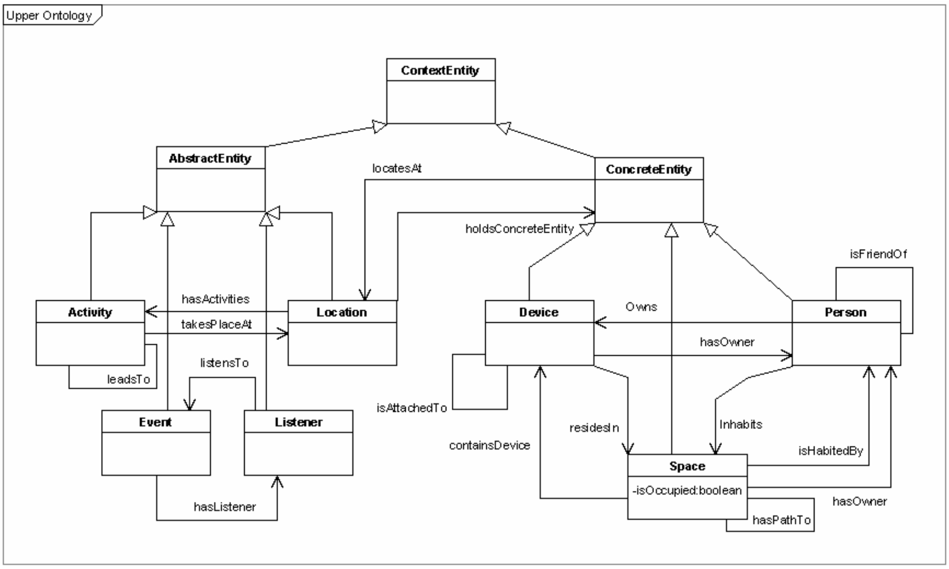


Figure 12. UML diagram of the upper-level ontology.

UML class diagrams are very feasible for illustrating ontologies because the ontologies consist of classes and the relationships between them. The UML class diagram elements used in Figure 12 are class descriptions, inheritance relationships between classes and dependency relationships between class instances. Every class that has some relationship to another class also contains attributes for the instances the relationship points to. An attribute can be defined to hold only one instance or multiple instances, depending on the relationship. Attributes that create the relationships between classes are left out to make the diagrams easier to read.

In this ontology the highest level is the ContextEntity class that other defined classes inherit directly or indirectly. The ContextEntity class is divided into two subclasses: AbstractEntity and ConcreteEntity. AbstractEntity is the base class for all entities that can be seen as abstract objects, such as Location, Activity and Event classes. For example, the Activity class can be extended with specific ontologies for scheduled activities or activities that a person might be involved with. The Event and Listener classes represented here are used for conditional eventing to represent triggered events and their listeners.

The ConcreteEntity class is base for all classes representing concrete contextual objects. It has a relationhips – “LocatedAt” – to the Location class because it can be assumed that concrete entities have a location in all circumstances. All classes that inherit the ConcreteEntity class also inherit the “LocatedAt” relationship. The Location class is assigned to have an inverse relationship – “holdsConcreteEntity” – back to ConcreteEntity because if an entity has a location, the same location obviously contains the entity. Other classes inheriting the ConcreteEntity class have similar inverse relationships to each other.

The Device class has a relationhips – “isAttachedTo” – back to itself, which represents the case if some instance inheriting the Device class is physically attached to some other Device class inheriting the instance. Other classes also have these kinds of relationships pointing back to themselves, like the Space and Person classes for example. The “hasPathTo” relationship of the Space class defines that a space can have a path that leads to another space, similar to the Person class where a person can be a friend of another person. All of these relationships are also inverse relationships, which means that if, for example, a person called John is a friend of Jane’s, then Jane is also a friend of John’s. An exception to this is the “leadsTo” relationship of the Activity class, which means that an activity can take place after some other activity. However, an activity can’t lead back to the previous one because the previous activity is a prerequisite for the next one.

3.4.2 Lower-level ontologies

The lower-level ontologies designed for the Context Monitoring Service are more specific ontologies with different attributes in the classes. Models defined with these ontologies are intended to extend the base model defined with the upper-level ontology. The designed ontologies are for different sensor devices and constructional semantics of the service’s deployment environment that, for example, describe a building with rooms and the area the building is standing on. Figure 13 describes the building ontology that has been designed for the context model’s base model.

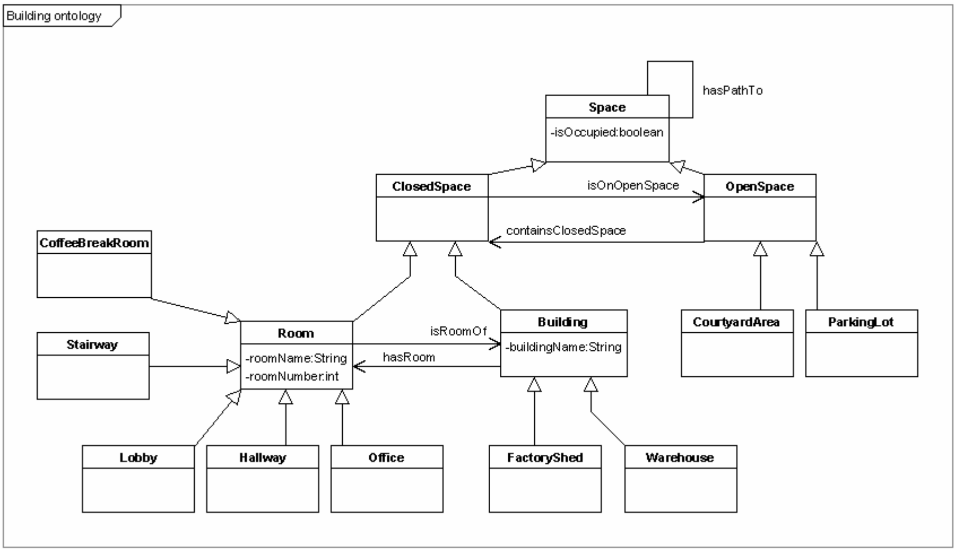


Figure 13. UML class diagram of the building ontology.

The ClosedSpace and OpenSpace classes represented in Figure 13 can be seen as an interface from where the building ontology binds to the upper-level ontology. They inherit the Space class from the upper-level ontology, thus enabling extendibility when the ontologies are combined. When a new model defined with the building ontology is bound to the base model defined with the upper-level ontology, all the relationships through the upper-level ontology to other bound models are assigned in the context model.

Each sensor used to gather the context information from the environment to update the context model has its own ontology. An example ontology designed for three particular sensors is described in Figure 14. The sensor ontology is extended as new sensors are introduced to the system. This sensor ontology's interface to the upper-level ontology is the Device class, which both ontologies define. The classes in the ontology also have several attributes to describe the data the sensors contain. Subclasses inherit all the attributes from the classes they extend. For example, the X10MotionDetector class extends the MotionDetector and X10Device classes, thus it contains both attributes typical for X10 devices and motion detectors. Note that the sensor ontology can be easily extended as new sensors are introduced to the system.

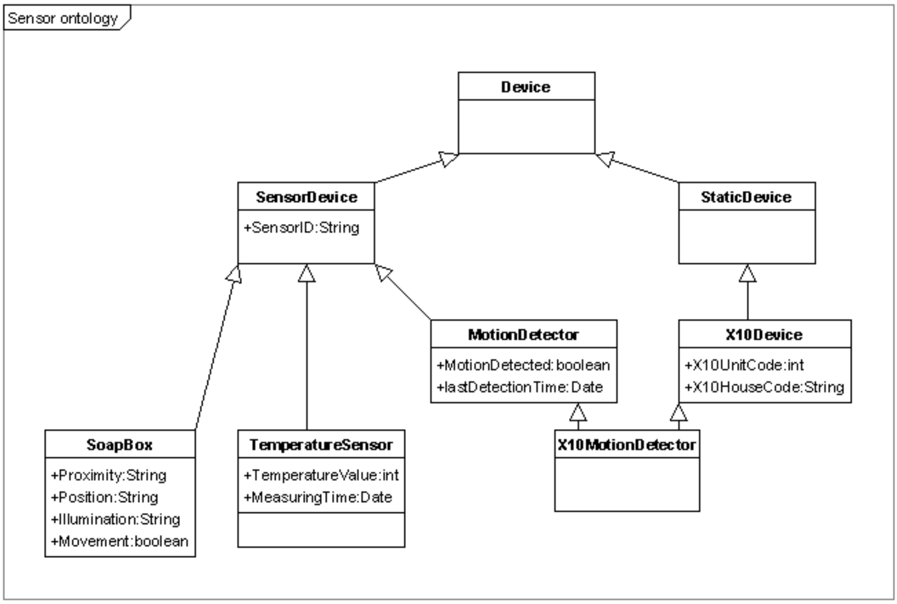


Figure 14. UML class diagram of the sensors ontology.

3.5 Reasoner

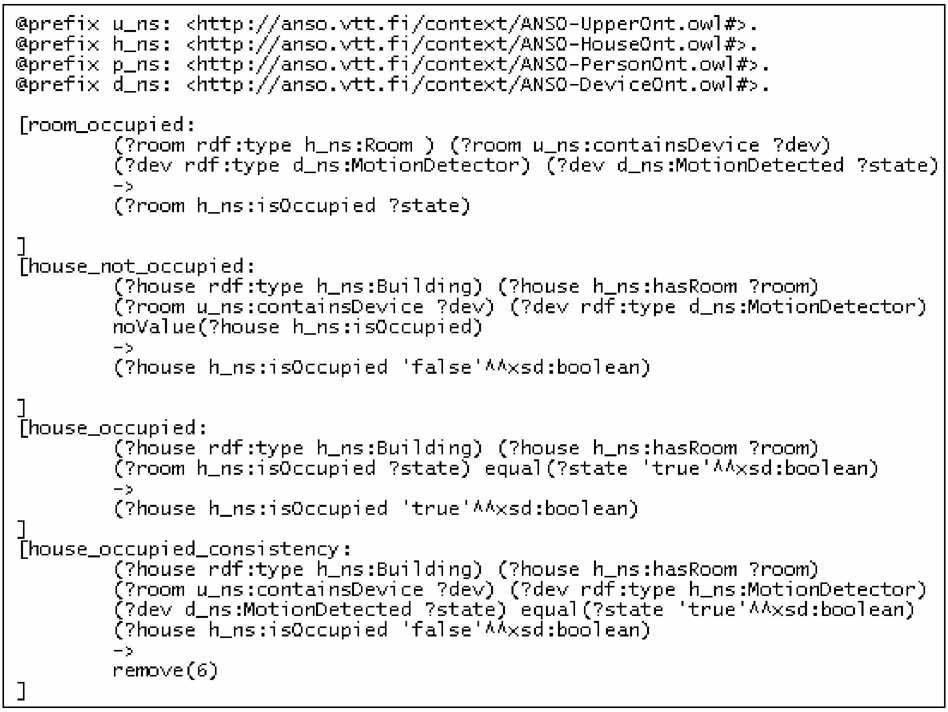
Reasoning over the context model semantic data is an important task for the Context Monitoring Service as it provides all the new information that can be deduced from the context model. Reasoning or inferring is done by the reasoning engine that is assigned to the context model. To deduce the new information, the reasoning engine uses predefined rules, which can be created by the user, or specific reasoning rules of some ontology language. For example, the Jena Semantic Web Framework contains specific reasoning engines that have reasoning rules for the RDFS and OWL languages.

The Reasoner produces new statements to the context model according the rules it has been given. The context model can only be held in the memory of the computer, so all inferred data would be lost on shutdown. The memory consumption could also be high when the context model grows extensively. If a database is used to establish persistent data storage for the context model, the inferred data will remain after shutdown and memory consumption will reduce. However, utilization of a persistent database requires time consuming storage and removal operations to the database.

The CMS is deployed in a dynamic environment where the changes occur rapidly, thus the inferred data is held in the memory to keep the CPU consumption to a minimum. The Reasoner monitors the context model and every time some change occurs in the model it deduces all the inferred data to the in-memory context model again.

3.5.1 Rules for Reasoner

The Jena Semantic Web Framework also provides a general-purpose rule engine that can be used for inference with custom rules. The Context Monitoring Service requires customized inference rules to enable improved context awareness. The Jena general-purpose rule engine is given a rule set that is designed to deduce new context information from the monitoring service's deployment environment. The created rule set is simple but provides insight for the general-purpose rule engine. The defined rules are shown in Figure 15.



```
@prefix u_ns: <http://anso.vtt.fi/context/ANSO-UpperOnt.owl#>.
@prefix h_ns: <http://anso.vtt.fi/context/ANSO-HouseOnt.owl#>.
@prefix p_ns: <http://anso.vtt.fi/context/ANSO-PersonOnt.owl#>.
@prefix d_ns: <http://anso.vtt.fi/context/ANSO-DeviceOnt.owl#>.

[room_occupied:
        (?room rdf:type h_ns:Room ) (?room u_ns:containsDevice ?dev)
        (?dev rdf:type d_ns:MotionDetector) (?dev d_ns:MotionDetected ?state)
        ->
        (?room h_ns:isOccupied ?state)

]
[house_not_occupied:
        (?house rdf:type h_ns:Building) (?house h_ns:hasRoom ?room)
        (?room u_ns:containsDevice ?dev) (?dev rdf:type d_ns:MotionDetector)
        noValue(?house h_ns:isOccupied)
        ->
        (?house h_ns:isOccupied 'false'^^xsd:boolean)

]
[house_occupied:
        (?house rdf:type h_ns:Building) (?house h_ns:hasRoom ?room)
        (?room h_ns:isOccupied ?state) equal(?state 'true'^^xsd:boolean)
        ->
        (?house h_ns:isOccupied 'true'^^xsd:boolean)
]
[house_occupied_consistency:
        (?house rdf:type h_ns:Building) (?house h_ns:hasRoom ?room)
        (?room u_ns:containsDevice ?dev) (?dev rdf:type h_ns:MotionDetector)
        (?dev d_ns:MotionDetected ?state) equal(?state 'true'^^xsd:boolean)
        (?house h_ns:isOccupied 'false'^^xsd:boolean)
        ->
        remove(6)
]
```

Figure 15. Customized rules for reasoning.

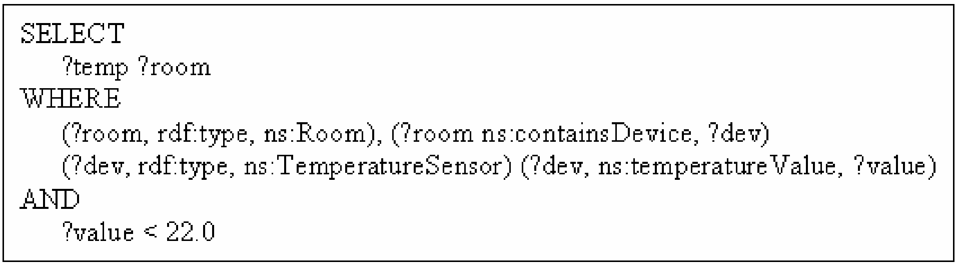
The first rule presented in Figure 15 is assigning a room's state to "occupied" if a motion detector located in the room observes motion. It sets the Reasoner to first find all the instances that have a type of Room class. Then the Reasoner seeks all instances in the model that have a containsDevice relationship to the instances of Room class and are instances of the MotionDetector class. If an instance of MotionDetector is located in the room, the room's occupancy status is set to the motion detector's status. It should be noted that, as presented in Figure 14 from Section 3.4.2, the MotionDetector class extends the Device class, thus the instances of MotionDetector can have the containsDevice relationship.

The other three rules are for determining whether or not the whole house is occupied. This is done by searching the occupancy statuses of all rooms and if one of them is occupied, the whole house is also occupied. A rule is also needed to assign the status back to "not occupied" if none of the rooms' motion detectors have detected motion. The sast rule is for keeping the context model consistent with the occupancy states.

3.6 Querying of the context model

One of the services that are provided for applications using the Context Monitoring Service is the possibility to query data from the model. Queries are made to the context model with RDQL language-defined query clauses. RDQL enables extensive and specific queries to the context model's data. The querying process itself is allocated for a query engine that is assigned to the context model. The Jena Semantic Web Framework provides an RDQL query engine that can be used in conjunction with the context model.

An application wanting to query the CMS's context model forms an RDQL query clause and provides it to the query engine. The engine performs the query over the context model's RDF graphs including the new data deduced by the Reasoner. As a result of the query, the application is given a result set that contains the entities that matched the query clause. An example RDQL query clause is illustrated in Figure 16.



```
SELECT
    ?temp ?room
WHERE
    (?room, rdf:type, ns:Room), (?room ns:containsDevice, ?dev)
    (?dev, rdf:type, ns:TemperatureSensor) (?dev, ns:temperatureValue, ?value)
AND
    ?value < 22.0
```

Figure 16. An example of an RDQL query clause.

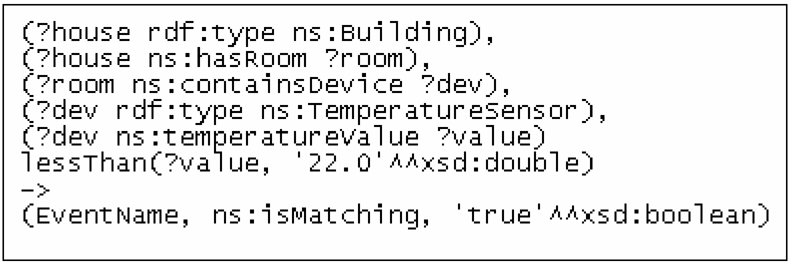
In this example the context model is queried for all rooms that have temperature sensors having readings below 22.0 degrees. The variables after the SELECT word are the entities that are wanted in the result set. In this case the results for temperatures and rooms are received after the query. The conditions after the WHERE word determine how the variables to be returned are selected and the constraints for the variables are after the AND word.

3.7 Conditional eventing

If applications were always required to query the model in order to enable context awareness, the performance would drop dramatically. Conditional eventing enables the applications to focus on more important tasks instead of regularly polling the context model. The Context Monitoring Service provides the applications with the possibility to register two types of conditional listeners to the model with arbitrary conditions. The first type of listener is a condition rule listener, where a rule is given during the registration, which then triggers the notification if the rule matches the state in the context model. The second type is an instance addition and removal listener, which notifies the listeners when an instance of a predefined ontology class is added to or removed from the CMS's context model.

Condition rule listener

The condition rule listener works in conjunction with the general-purpose rule engine attached to the context model. An application wanting to receive events registers a customized rule to the rule engine. Condition rules differ from the predefined rule set presented in Section 3.5.1 that they don't define any action when the rule triggers. That is done by the conditional eventing component of the CMS, which appends the triggering action to the end of the rule and thus forms the triggering condition. A better insight into a form of triggering condition and example condition rule with the added triggering action is given in Figure 17. This condition sets the Context Monitoring Service to notify an application when any of the room temperatures in a building is less than 22.0 degrees.



```
(?house rdf:type ns:Building),
(?house ns:hasRoom ?room),
(?room ns:containsDevice ?dev),
(?dev rdf:type ns:TemperatureSensor),
(?dev ns:temperatureValue ?value)
lessThan(?value, '22.0'^^xsd:double)
->
(EventName, ns:isMatching, 'true'^^xsd:boolean)
```

Figure 17. Example of a condition with condition rule and triggering action.

An instance of Listener class is created for the application when an application registers a condition rule listener to the context model. The triggering action for the condition rule is set to create a new instance of the MatchingEvent class to the Reasoner and the application instance is assigned to a listener for the event. Notification of the listening applications is done by querying all the instances of the MatchingEvent class and their listeners in the model. All found listeners are notified from all their matching conditions. Queries for the matching conditions are done after every change that occurs in the model.

Instance addition and removal listener

The conditional eventing mentioned previously can only notify the listener when the given rule matches the state of the context information, but a rule that triggers a notification when a predefined type of data is appended to the context model cannot be created. This is because the Jena generic rule engine only matches the rules against the current information in the context model. Another type of conditional eventing is needed to enable applications to listen to information additions and removals to the context model.

Instance addition and removal listening is a service where the application can specify a type of ontology class as a condition for the conditional eventing registration. The Context Monitoring Service keeps a record of all semantic instances that are added to or removed from the context model. If some of these instances are a type of ontology class that is listened to, the listener is notified of the addition or removal. The given ontology class condition is the URI of the class. An example of a condition for an instance addition and removal listener can be following: "*http://anso.vtt.fi/context/ANSO-HouseOnt.owl#Device*". This type of instance addition listener and removal condition triggers a notification every time a new instance of the Device class is added to or removed from the context model. This type of listener is useful when the application wants to listen when a new Device is added to the CMS somewhere in the deployment environment.

3.8 Model updater devices

The model requires constant updating in order to deploy a dynamic context model that is consistent with its deployment environment. The Context Monitoring Service's context model is updated with different model updater devices that include sensors and other CMS components that provide the context information in their context models by utilizing UPnP. The context model can be updated with environmental information from the surroundings, such as a person's context information like health conditions or his current location. Sensors that are used to update the model and keep it consistent will be referred as model updater sensors from now on, to distinguish them from the CMS components advertising their context information. The division of sensors and CMS components under the different definitions introduced here is illustrated in Figure 18.

Figure 18 shows that the CMS components are model updater devices because their context models can be retrieved. Sensors are defined to be model updater sensors after their raw data is converted to meaningful semantic statements. Both the CMS component and the model updater sensor have the same functionality to provide new context information, thus they are together called model updater devices. The physical locations relative to each other define the model updater devices as being either local or remote. Local model updater devices are deployed in the same OSGi framework and discovered by utilizing OSGi discovery. Remote model updater devices located in remote OSGi frameworks are discovered by utilizing UPnP discovery. To be more specific, sensors are distinguished from CMS components by referring to them as local or remote model updater sensors. When there's no practical need to distinguish them from each other they are referred to as local or remote model updater devices. However, in practice, CMS components are not deployed as local model updater devices because they are not used locally to provide new context information for other components of CMS.

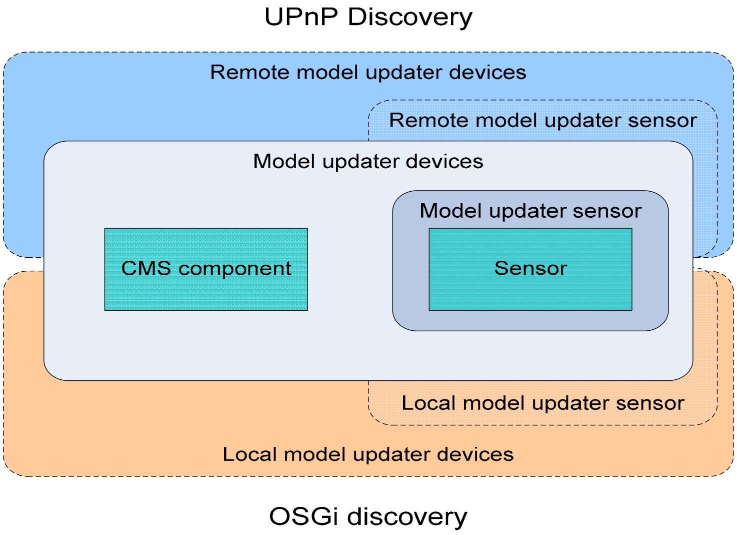

UPnP Discovery
Remote model updater devices
Remote model updater sensor
Model updater devices
Model updater sensor
CMS component
Sensor
Local model updater sensor
Local model updater devices
OSGi discovery

Figure 18. Definitions for sensors and CMS components.

The difference between a CMS component and a model updater sensor is that the sensor's semantic data model isn't as extensive as the context model. To define the semantic model for sensors, the lower-level ontologies represented in Section 3.4.2 are used for each type of sensor. A sensor-specific ontology defines the attributes that are updated with data the sensor gathers from the environment. When the CMS component registers to listen to the sensor it receives its semantic model, which is added as a submodel to the base context model. All changes that occur in the sensor's model are seen in the CMS component's context model.

3.9 Dynamic discovery and advertisement of model updater devices

As stated earlier, the model updater devices consist of the model updater sensors that are used to update the context information and the CMS components that advertise their context model for other CMS components via UPnP. This section explains how the local and remote model updater sensors are discovered and how the CMS components advertise their context model and discover the other CMS components.

Model updater sensors can be located in the same computational unit as the Context Monitoring Service or they can be used to update the model remotely. Both types of sensors must be discovered in order to use them to update the context model. To solve the discovery problem, the OSGi framework's service discovery was selected to find the locally connected updater sensors. For sensors connected to remote computational units and for the CMS component's context model, advertising UPnP technology was the selection.

3.9.1 Local model updater sensors

Local model updater sensors are installed in the same OSGi framework as the Context Monitoring Service and their controlling software is deployed in different OSGi bundles. When the CMS is installed and started in the OSGi framework all local sensors are discovered and registered to update the context model. The Context Monitoring Service contains a sensor discovery process that not only finds the sensors on start-up but also in run-time. If a new model updater sensor registers a service interface to the OSGi framework, the CMS's discovery process automatically receives the service and subscribes to the sensor's model change events.

3.9.2 Remote model updater devices

The model updater sensors that are located in the remote computational units and remote CMS components are discovered with UPnP technology. Remote model updater devices are deployed as UPnP devices that provide a service to receive the information from the device. The UPnP device's service contains an action to get the model updater device's semantic model, which is a state variable of the UPnP service. The model updater sensor's semantic model and the CMS component's semantic context models are shared by the UPnP service. The UPnP service also contains a possibility to subscribe to listening to the state variables that represent the semantic model and the latest data value that has changed in it. The semantic model's data is sent as an RDF/XML encoded string message and the most recently changed values are sent as string representations of the data.

UPnP sensor devices are deployed with a special component that locates all the sensors in the same OSGi framework and creates a UPnP device for each. Model updater sensors, which register to the framework after the UPnP sensor enabler component is run, are also discovered and UPnP devices are deployed for each. The CMS components create their own UPnP devices that are assigned to listen their context models. Both the model updater sensors and the CMS components deploy exactly the same type of UPnP service to provide better interoperability. Device descriptions can vary between the sensors and CMS components. For more detailed information, an example of the used UPnP device description is presented in Appendix 1 and its service description in Appendix 2.

The Context Monitoring Service contains a UPnP control point that is activated during the start-up. The service's UPnP control point discovers active remote updater devices and automatically receives their semantic model and starts listen to the changes in it. The semantic model of the UPnP devices is received by sending a HTTP GET request to the device. After receiving the model, the CMS adds it to the context model as a submodel and subscribes itself to listen to the UPnP service of the device. All changes in the device's data model are received through the UPnP eventing.

For the UPnP control point and devices, the selected subarchitecture is CyberLink for Java, which is an open source development package for UPnP developers. CyberLink for Java uses the protocols in UPnP automatically, hiding them from the programmer [54]. This provides the developers with a simple-to-use package for quickly creating UPnP solutions. CyberLink for Java is written in Java and, as it is an open source implementation, the source can be modified to suit the purpose better.

3.10 OSGi bundle configuration

To provide the service interface for the applications, the OSGi service framework is chosen for the platform on which the CMS is run. Due to its dynamic nature, additional components of the CMS are easy to integrate to extend and improve the CMS in run-time. The different deployment components of the CMS are deployed as OSGi bundles. However, the OSGi bundle configuration differs from the CMS's functional component configuration presented in Table 2. The differences are that the dynamic discovery and advertisement of model updater devices are divided between the CMS and UPNP model updater sensor device bundles.

The CMS's bundle division is designed so that the bundle contains the C-CMS or S-CMS component and its basic functionalities such as context model, reasoning engine, model updater device discovery, the CMS component's UPnP device advertisement and the service interface for the applications. The model updater sensors are each deployed in their own bundles and, if needed, are accompanied by sensor-specific driver bundles. The purpose of the UPnP model updater sensor device bundle is that it locates the model updater sensors from the same OSGi framework and establishes UPnP devices for each of them. The bundle configuration in the OSGi framework is illustrated in Figure 19.

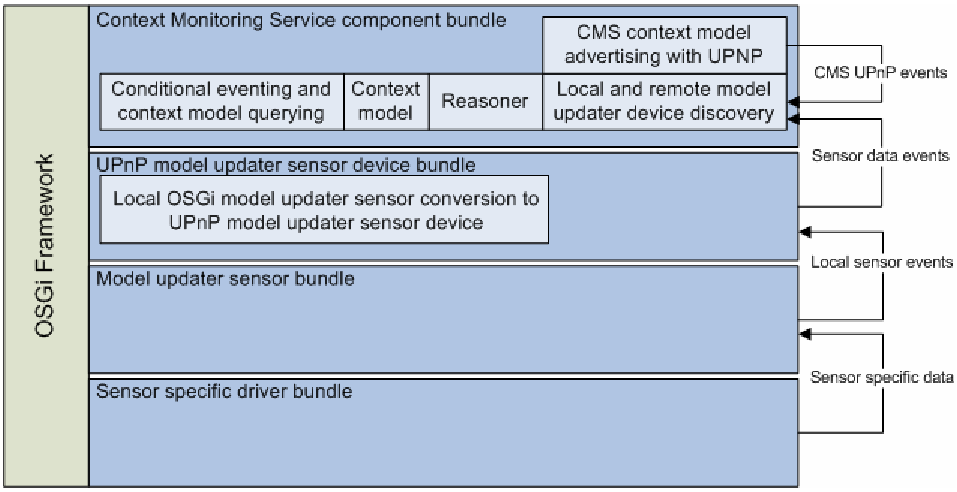

OSGi Framework
Context Monitoring Service component bundle
CMS context model advertising with UPNP
Conditional eventing and context model querying
Context model
Reasoner
Local and remote model updater device discovery
UPnP model updater sensor device bundle
Local OSGi model updater sensor conversion to UPnP model updater sensor device
Model updater sensor bundle
Sensor specific driver bundle
CMS UPnP events
Sensor data events
Local sensor events
Sensor specific data

Figure 19. Bundle configuration in the OSGi framework.

4. Architectural design

The Context Monitoring Service was decoupled into different functional components in Chapter 3. This chapter provides one possible architectural composition of these components that the Context Monitoring Service requires. The Context Monitoring Service is also decomposed into the different interfaces that it provides and uses in order to enable all the functionalities explained in Chapter 3. UML diagrams are used to give some insight to the interfaces of the service. Some of the UML class diagrams are simplified by leaving out some irrelevant methods and parameters to make them easier for the reader to understand.

4.1 Solution architecture

The solution architecture of the service describes one possible solution for how the components are integrated and how they communicate together. In this architectural design the different components are categorized into three different layers: service layer, model updater device advertisement and discovery layer, and model updater sensor layer. The layered architecture design of the whole system is illustrated in Figure 20.

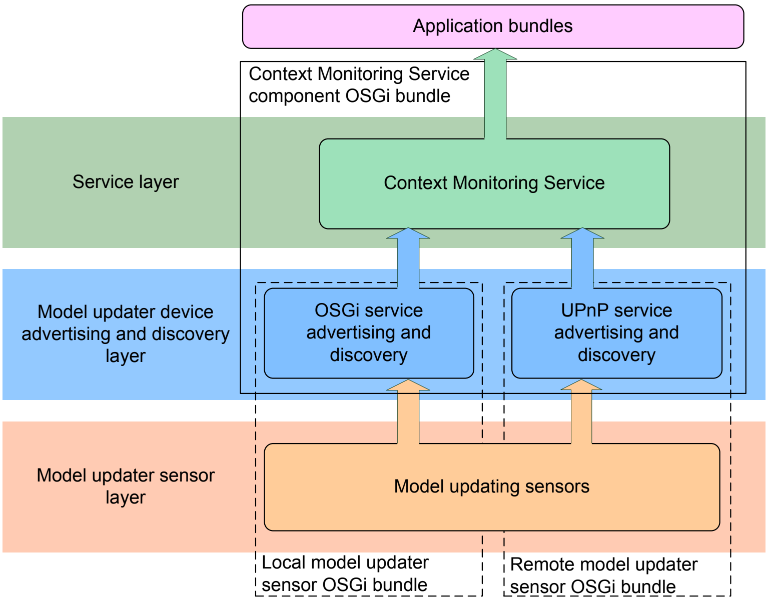

Application bundles
Context Monitoring Service component OSGi bundle
Service layer
Context Monitoring Service
Model updater device advertising and discovery layer
OSGi service advertising and discovery
UPnP service advertising and discovery
Model updater sensor layer
Model updating sensors
Local model updater sensor OSGi bundle
Remote model updater sensor OSGi bundle

Figure 20. Layered architecture of the service.

Layer division is done by including similar functionalities in the same layer. The service layer contains all the components that are used to provide the querying and conditional eventing service for the applications. The service layer receives the context model update data from the advertisement and discovery layer and updates the context model accordingly. The model updater device advertisement and discovery layer is responsible for locating the local and remote model updater devices and supplying their data to the service layer for updating the context model. The advertisement and discovery layer also creates UPnP devices from the model updater sensor's and Context Monitoring Service's semantic models and advertises them in the network. The model updater sensor layer is for all the model updater sensors, both local and remote.

The layered architecture presented in Figure 20 also shows how the layers appear in the OSGi bundles represented in Figure 19. The bundles are: the main bundle for the Context Monitoring Service deployment components, bundles for local model updater sensors and a bundle that deploys local sensors as UpnP-enabled sensors. Bundle division does not follow the layer division because the deployed bundles are designed to be executable without the other bundles.

The shown bundle division for the model updater sensors overlaps the model updater device advertisement and discovery layer. This is because the local and remote model updater sensors advertise their services using OSGi and UPnP technology respectively. The Context Monitoring Service uses the same layer for discovering the provided services, OSGi discovery for local model updater devices and UPnP discovery for remote updater devices. Furthermore, the CMS component uses the layer for advertising its context model as a UPnP device to enable context model discovery for other Context Monitoring Service components, as was described in the scenario explained in Section 3.2.

The layers are categorized into their own sections and are explained by describing the relationships between the components the layer contains. The first explained layer is the service layer, followed by the model updater device advertisement and discovery layer. The last of the layers is the model updater layer. All the layers are explained in detail and illustrated with Gane-Sarson-type data flow diagrams representing the interaction between the components.

4.1.1 Service layer

The service layer is responsible for providing the context model querying and conditional eventing services for the applications. The context model component defined with different ontologies is the main part of the layer. The RDQL query engine and Reasoner work by communicating with the context model. Conditional eventing works in conjunction with the Reasoner and query engine. The components in the service layer and their communication with the other components are shown as a Gane-Sarson data flow diagram in Figure 21.

As said earlier, the context model is the main component of the service layer. The context model is defined with the ontologies that form the semantic representation of the context information. It is updated with the sensor data received from the model updater device advertisement and discovery layer. The model updater is part of the Context Model and its purpose is to handle all the updates to the context model. The model updater device advertising and discovery layer receives the context model data through the interface for advertising the context model with a UPnP service.

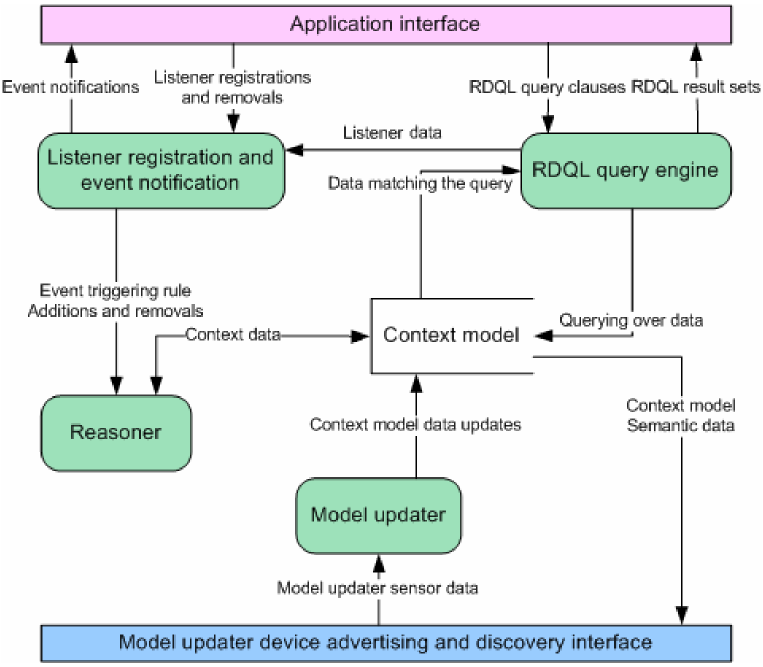

Application interface
Event notifications
Listener registrations and removals
RDQL query clauses
RDQL result sets
Listener data
Listener registration and event notification
Data matching the query
RDQL query engine
Event triggering rule Additions and removals
Context data
Context model
Querying over data
Reasoner
Context model data updates
Context model Semantic data
Model updater
Model updater sensor data
Model updater device advertising and discovery interface

Figure 21. Architecture of the service layer.

The Reasoner and RDQL query engine both operate in parallel with the context model by using its information for the operations they provide. The Reasoner uses the semantic data the context model contains and deduces new context data, which is stored back to the context model. The Reasoner listens to the context model and triggers the reasoning process when a change occurs in the model. The Reasoner also provides the functionality for the conditional eventing process to reason when the conditional events registered in the service match the current context. The RDQL query engine provides the model querying service for the applications. It processes the query clause against the context model statements and returns the result set of the query to the application. The Query engine is also used by the conditional eventing component, which queries the context model for all triggered events.

The listener registration and event notification component has dependencies on the RDQL query engine and the Reasoner and doesn't need a direct access to the context model. The application registers either a condition rule listener or an instance addition and removal listener through the application interface. The Reasoner is given the eventing rules, which it processes and infers whether the listener matches or not. Statements for matching listeners are appended to the context model, which is then queried for the matching events with the RDQL query engine. The listener registration and event notification component is responsible for handling the registrations of listeners and the listener notifications. It also keeps track of matching conditions in order to notify the applications when a listener's state changes.

4.1.2 Model updater device advertisement and discovery layer

The model updater device advertisement and discovery layer contains the components for discovering the local and remote model updater devices and supplying the update data as a semantic representation for updating the context model of the CMS component. The layer also provides functions for creating the UPnP devices for the model updater devices and advertises them in the network. The model updater device advertisement and discovery component presented in Section 3.9 is located in this layer and is decomposed here into the different processes. The architecture of the model updater device advertisement and discovery layer is presented as a Gane-Sarson data flow diagram in Figure 22.

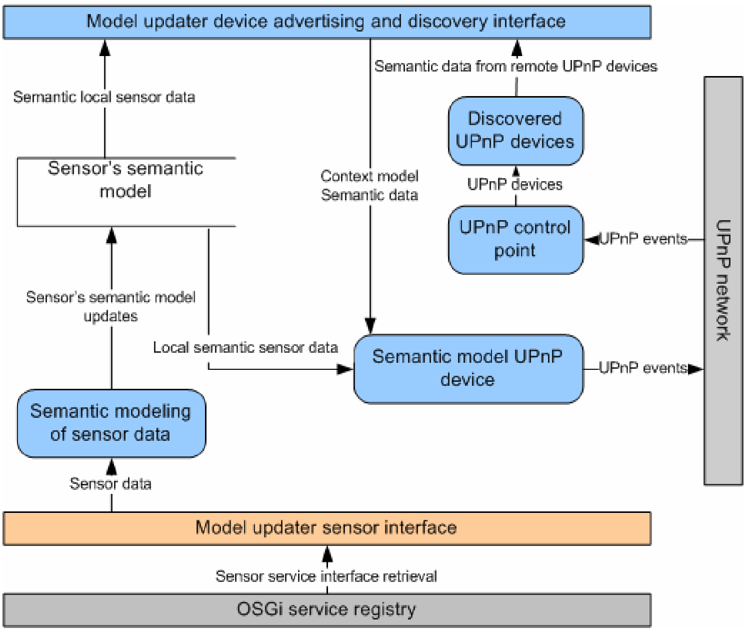

Model updater device advertising and discovery interface
Semantic local sensor data
Semantic data from remote UPnP devices
Discovered UPnP devices
Sensor's semantic model
Context model Semantic data
UPnP devices
UPnP control point
UPnP events
UPnP network
Sensor's semantic model updates
Local semantic sensor data
Semantic model UPnP device
UPnP events
Semantic modeling of sensor data
Sensor data
Model updater sensor interface
Sensor service interface retrieval
OSGi service registry

Figure 22. Architecture of the model updater device advertisement and discovery layer.

Local sensors that are registered in the same OSGi framework are discovered from the OSGi service registry, and semantic models are created for each. Semantic models for the sensors are defined with a specific lower-level ontology for each sensor. The process that creates the semantic model for the sensors registers to listen to the data events from the sensors and updates the model accordingly. The sensor data is received through the model updater sensor interface represented as a model and provided to the service layer through the interface for updating the CMS component's context model.

The same model updater sensors are used to provide the semantic data for the remote model updater sensors. UPnP devices are created for the model updater sensors and by setting the UPnP device to listen to the changes in the model updater sensor's semantic model. Changes occurring in the model are updated to the UPnP service's state variables. If the same OSGi framework contains a local sensor and a remote model updater sensor created from the local sensor, separate semantic models are created for both sensors. This enables creating stand-alone UPnP sensors without having to register a Context Monitoring Service component to the OSGi framework.

The remote model updater device for the CMS component's context model is also created in this layer. UPnP device deployment for the context model is the same as with the model updater sensors, but the semantic data is received from the service layer through the interface. The UPnP device for the context model registers to listen to the changes in the context model and updates the state variables accordingly.

The model updater device advertisement and discovery layer contains a UPnP control point for discovering all remote model updater devices in the network. The control point listens to the network for remote model updater device discoveries. The context information from the remote updater devices is forwarded to the service layer for context model update.

4.1.3 Model updater sensor layer

The last layer, the model updater sensor layer, contains the sensor configuration that is used to update the CMS component's context model. The required elements for a model updater sensor are presented as a Gane-Sarson data flow diagram in Figure 23.

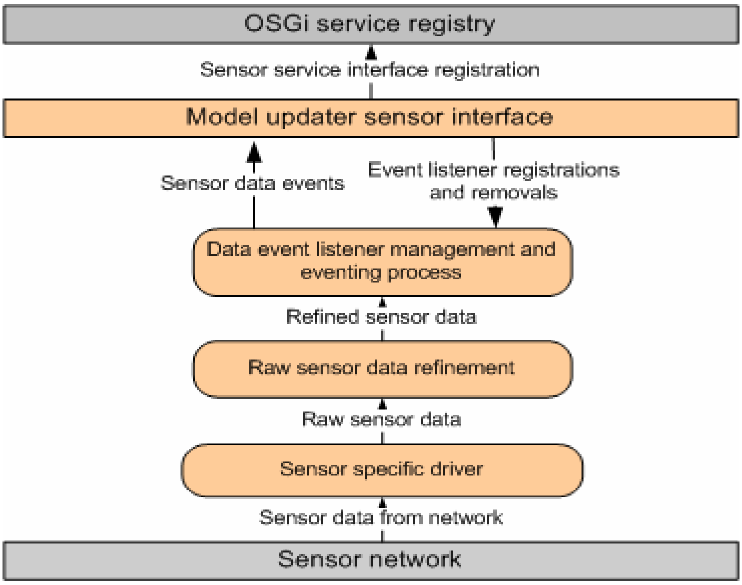

OSGi service registry
Sensor service interface registration
Model updater sensor interface
Sensor data events
Event listener registrations and removals
Data event listener management and eventing process
Refined sensor data
Raw sensor data refinement
Raw sensor data
Sensor specific driver
Sensor data from network
Sensor network

Figure 23. Architecture of the model updater sensor layer.

The architecture for the model updater sensors contains a driver for each type of sensor, the sensor's raw data refinement and sensor data listener registration, and an event notification manager.

Different sensors have specific communication protocols, so a driver is needed for each type of sensor. The sensor's driver uses the sensor-specific protocol to receive the raw data from the sensors. The raw data the sensor produces then needs to be converted to an understandable form, and a data refinement process should be designed for this purpose. For example, the data a sensor produces can be pure numerical values, so the data refiner converts the values to meaningful states. These states are used to represent the sensor data with ontologies as a model in the model updater device discovery and advertising layer.

The data event listener management and eventing process handles the registrations of listeners for sensor data events. A listener can listen to all sensors of a similar type or only some of them. The data events from the sensors are forwarded to the listeners that are interested in them.

Because the sensors are deployed in their own OSGi bundles they should provide an interface for applications to receive their data. In this solution the designed interface for sensors is the listener registration and notification interface. Interfaces to access the sensor data are registered to the OSGi service registry, from where the applications can retrieve it. The model updater device discovery and advertisement layer registers a listener to each sensor discovered in the OSGi service registry. The data from the sensors is received through the interface the sensors provide and is appended to the context model.

4.2 Architecture of the service

In this section the architecture of the Context Monitoring Service is explained in detail. It is categorized by dividing the service into different interfaces, which are further divided into local and remote interfaces. UML class diagrams are used to describe the design of the interfaces and their functionality. In addition, the main interface, the CMS's service interface, is explained in detail by providing a UML sequence diagram of how the application utilizes it. Brief descriptions of the identified interfaces and their deployment technologies are organized by interface types, local or remote, in Table 5.

The Context Monitoring Service contains two service interfaces, from which one provides the access to the monitoring service's context model and the other is the service interface of the model updater sensors. The Context Monitoring Service's service interface is used by the applications that are querying the context model or registering listeners for context information change notifications. The interface also includes the advertisement of the CMS component's UPnP device that allows other CMS components to receive the context model data and listen to the changes in it. The interface the model updater sensors provide is for listener registration to receive sensor data event notifications. The service interface of the model updater sensors is also provided locally and remotely.

The management interface of the Context Monitoring Service is used to update the service's context model with local and remote model updater sensors. The data received from the remote CMS component's UPnP devices is also appended to the CMS's own context model using this interface. In general, the local management interface provides the functionality to update the CMS's context model with the model updater sensor data. The remote management interface provides the functionality to discover and listen to the remote model updater sensors and Context Monitoring Service components. Model updater sensors do not provide a management interface for applications but their different drivers for the communication with the hardware could be seen as a management interface.

Table 5. Descriptions of the local and remote interfaces of the service.

Interface	**Interface type**	
	Local	**Remote**
Service interface	OSGi: – CMS's context model querying and conditional eventing. – Listener registration for model updater sensor data event notifications.	UPnP: – Context model advertisement in the network and listening between two Context Monitoring Service components. – Model updater sensor's UPnP device advertisement.
Management interface	OSGi: – Local sensor discovery and CMS's context model updating according to local sensor data events.	UPnP: – Remote sensor and Context Monitoring Service component discovery. – CMS's context model updating according to remote sensor and CMS component data.
Control interface	OSGi: – Local registration and start-up of the software bundles.	HTTP: – Remote registration and start-up of the software bundles.

The control interface provides the functionality for the user to install and execute the CMS component and model updater sensor software bundles, which compose the whole system. The designed deployment environment can contain several OSGi frameworks running the CMS's software components. If some of the framework's bundle configuration needs to be modified, for example shutting a sensor down, the modification can be done through the control interface either locally or remotely.

4.2.1 Local service interfaces

The local interfaces the Context Monitoring Service and the model updater sensors provide is used through the OSGi service discovery. Applications wanting to use the services discover the specific service from the OSGi service registry, where the service interfaces are registered. After receiving the interface the services can easily be used to register conditional listeners, query the context model or register listeners for the model updater sensor data.

Context Monitoring Service

The local service interface provided by the Context Monitoring Service is used by the applications installed in the same OSGi framework. As mentioned, the Context Monitoring Service provides services for conditional eventing and querying the context model. The conditional eventing consists of two types of listener services. The first service notifies the registered listeners when the given rule matches the state of the context model. The second listener service notifies the registered listeners when new ontology class instances of the type the application has defined are added to the context model. The architecture of the Context Monitoring Service's service interface is provided as a UML diagram in Figure 24.

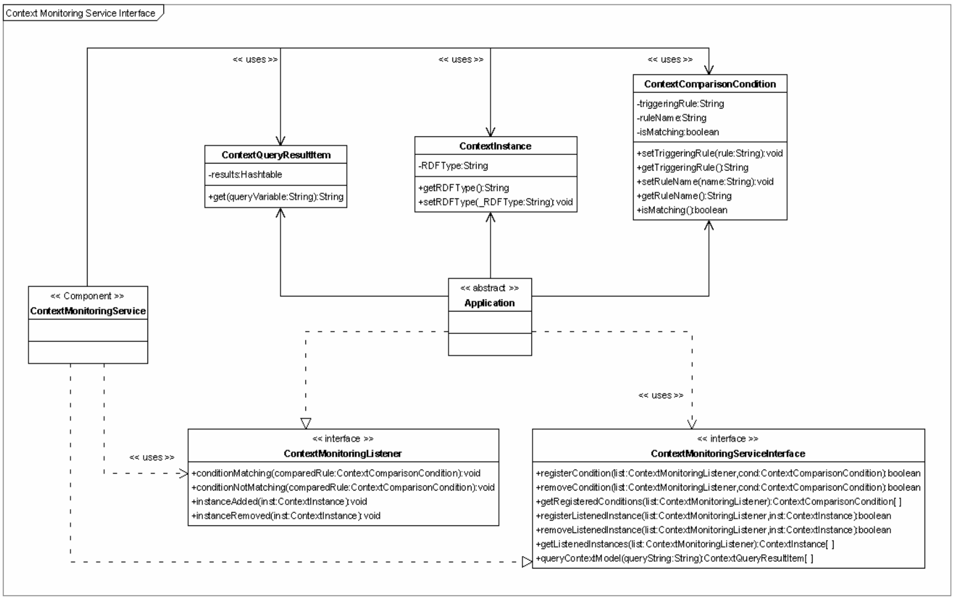

Context Monitoring Service Interface
<< uses >>
<< uses >>
<< uses >>
ContextComparisonCondition
-triggeringRule:String
-ruleName:String
-isMatching:boolean
+setTriggeringRule(rule:String):void
+getTriggeringRule():String
+setRuleName(name:String):void
+getRuleName():String
+isMatching():boolean
ContextQueryResultItem
-results:Hashtable
+get(queryVariable:String):String
ContextInstance
-RDFType:String
+getRDFType():String
+setRDFType(_RDFType:String):void
<< Component >>
ContextMonitoringService
<< abstract >>
Application
<< uses >>
<< uses >>
<< interface >>
ContextMonitoringListener
+conditionMatching(comparedRule:ContextComparisonCondition):void
+conditionNotMatching(comparedRule:ContextComparisonCondition):void
+instanceAdded(inst:ContextInstance):void
+instanceRemoved(inst:ContextInstance):void
<< interface >>
ContextMonitoringServiceInterface
+registerCondition(list:ContextMonitoringListener,cond:ContextComparisonCondition):boolean
+removeCondition(list:ContextMonitoringListener,cond:ContextComparisonCondition):boolean
+getRegisteredConditions(list:ContextMonitoringListener):ContextComparisonCondition[ ]
+registerListenedInstance(list:ContextMonitoringListener,inst:ContextInstance):boolean
+removeListenedInstance(list:ContextMonitoringListener,inst:ContextInstance):boolean
+getListenedInstances(list:ContextMonitoringListener):ContextInstance[ ]
+queryContextModel(queryString:String):ContextQueryResultItem[ ]

Figure 24. UML class diagram of the CMS's local service interface.

The ContextMonitoringServiceInterface interface class is the interface that provides all the service's functionalities and is registered to the OSGi framework. It is implemented by the ContextMonitoringService component that contains the underlying implementation for all the service functionalities.

An application using the conditional eventing should implement the ContextMatchListener class to receive the events of matching conditions and new ontology class instance additions to the model. To receive notification of the different states of the context model, applications create instances of the ContextComparisonCondition class where the rule for the matching condition is registered. An instance of the ContextComparisonCondition class should be created for each condition and registered with the reference of the application implementing the ContextMatchListener interface. To listen to new ontology class instance additions and removals in the context model, the applications create instances of the ContextInstance class that contains the class type definition. Instances of the ContextInstance class are registered to the service in a similar way as the ContextComparisonCondition instances.

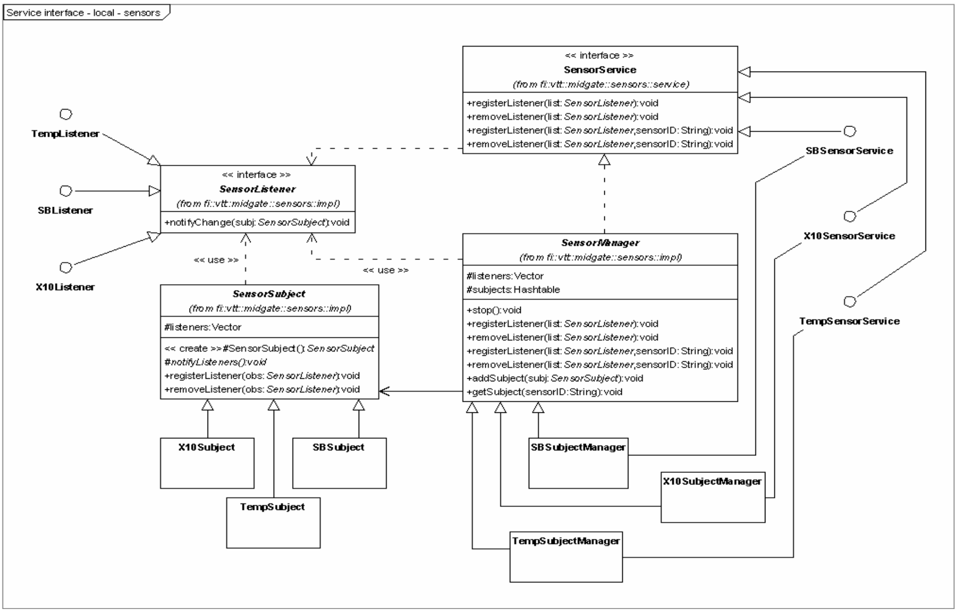
The registered listening conditions and application references are received by the eventing manager, which creates a separate notification process for each listener. All conditions and instance listeners that the application registers are stored to the notification process. The rule that indicates the context state of when to notify the listener is registered to the Reasoning engine's rule base. The Reasoner uses the rules in the reasoning process to create events of context state matches. The functionalities of the condition rule listener and the instance addition and removal listener are illustrated in more detail in Appendix 3 and Appendix 4 as UML sequence diagrams.

The event manager and notification processes are executed in their own threads and each application has its own notification process that is responsible for notifying the application of matching conditions. When the notification process is run in its own thread, it does not prevent other notifications from receiving notifications, even if a notified application goes into deadlock. The event manager is also run in a thread that blocks until a change in the Context Monitoring Service's context model occurs. The event process listens to the context model through a listening interface that is provided by the Jena semantic framework [35].

An application wanting to query the monitoring service's context model provides an RDQL query clause through the ContextMonitoringServiceInterface interface for the Context Monitoring Service. The query string is executed over the context model's semantic data and a result set of the query is returned. The result set is received as a table of ContextQueryResultItem class instances that contains each query result. Detailed information on querying the context model is presented in Appendix 5 as a UML sequence diagram.

Model updater sensors

Model updater sensors provide a local service interface for the Context Monitoring Service to enable context model updating with the sensor data. This type of service is listener registration for data event notifications. The architecture of the model updater sensors' service interface is shown in Figure 25.



Service interface - local - sensors
<< interface >>
SensorService
(from fi::vtt::midgate::sensors::service)
+registerListener(list: SensorListener):void
+removeListener(list: SensorListener):void
+registerListener(list: SensorListener,sensorID: String):void
+removeListener(list: SensorListener,sensorID: String):void
TempListener
SBListener
X10Listener
<< interface >>
SensorListener
(from fi::vtt::midgate::sensors::impl)
+notifyChange(subj: SensorSubject):void
SBSensorService
X10SensorService
TempSensorService
<< use >>
<< use >>
SensorManager
(from fi::vtt::midgate::sensors::impl)
#listeners:Vector
#subjects:Hashtable
+stop():void
+registerListener(list: SensorListener):void
+removeListener(list: SensorListener):void
+registerListener(list: SensorListener,sensorID: String):void
+removeListener(list: SensorListener,sensorID: String):void
+addSubject(subj: SensorSubject):void
+getSubject(sensorID:String):void
SensorSubject
(from fi::vtt::midgate::sensors::impl)
#listeners:Vector
<< create >>#SensorSubject(): SensorSubject
#notifyListeners():void
+registerListener(obs: SensorListener):void
+removeListener(obs: SensorListener):void
X10Subject
SBSubject
TempSubject
SBSubjectManager
X10SubjectManager
TempSubjectManager

Figure 25. UML class diagram of a sensor's local service interface.

The local service interface of the sensors is the SensorService interface class, which the SensorManager class implements. The SensorManager class provides the functionalities for listener registration to receive data events from all sensors or just one of them, and for removal of the listeners. An application wanting to receive sensor events should implement the SensorListener interface class in order to be able to register its reference to the SensorManager. The SensorManager class also contains the instances for the sensors in the system. The SensorManager can handle several similar types of sensors if there's a possibility to connect them to the computer more than once. An instance of the SensorSubject class is created for each sensor to hold the received sensor data.

When different types of sensors are connected to the system, each type of sensor should extend the SensorManager, SensorSubject, SensorListener and SensorService interfaces and classes. By extending them, every new sensor service provides a similar interface for the applications. This sensor specialization is shown in Figure 25 by showing the inheritances of the used sensors.

4.2.2 Remote service interfaces

The remote service interfaces of the Context Monitoring Service and the model updater sensors enable context model updating with remotely located context information. The interfaces are deployed as UPnP devices that have specific services for receiving the data as semantic models and statements. The UPnP device and service descriptions of the Context Monitoring Service and model updater sensors are exactly similar, except for each device's universal device name. For more detailed information, an example UPnP device description is shown in Appendix 1 and an example UPnP service description is shown in Appendix 2.

Context Monitoring Service

The Context Monitoring Service deploys a UPnP device that provides a service for the other CMS components to receive the context information in its context model as described in Section 3.2. The architecture of the UPnP device deployment is shown in Figure 26.

The main class of the service, the EMC class, contains the EMCUpnpDevice class, which is the implementation of the UPnP device. The EMCUpnpDevice class extends the Device class provided by CyberLink UPnP for the Java [54] development package. The Device class contains all the required functionalities for UPnP device deployment. The UPnP device contains the state variables *model* and *statement*, which are for the semantic model that can be requested and for the last changed statement in the model respectively. Applications can subscribe to receive events for changes in the *statement* state variable.

The EMCUpnpDevice class listens to the Context Monitoring Service's context model through the ModelChangedListener, which is a listener interface from the Jena Semantic Framework package. When a change occurs in the context model the EMCUpnpDEvice receives the changed statement, stores it to the state variable and sends a change event to the network.

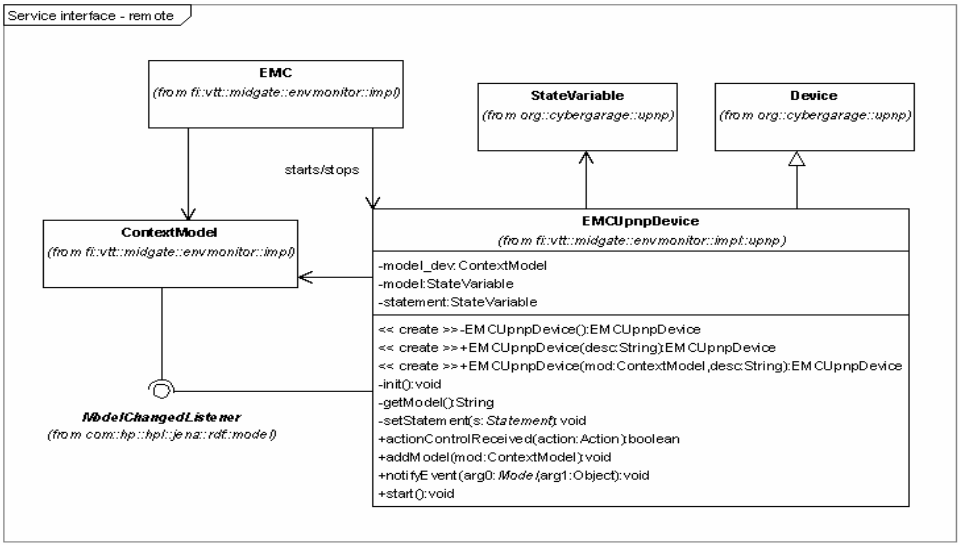

Service interface - remote
EMC
(from fi::vtt::midgate::envmonitor::impl)
StateVariable
(from org::cybergarage::upnp)
Device
(from org::cybergarage::upnp)
starts/stops
ContextModel
(from fi::vtt::midgate::envmonitor::impl)
EMCUpnpDevice
(from fi::vtt::midgate::envmonitor::impl::upnp)
-model_dev:ContextModel
-model:StateVariable
-statement:StateVariable
<< create >>-EMCUpnpDevice():EMCUpnpDevice
<< create >>+EMCUpnpDevice(desc:String):EMCUpnpDevice
<< create >>+EMCUpnpDevice(mod:ContextModel,desc:String):EMCUpnpDevice
-init():void
-getModel():String
-setStatement(s:Statement):void
+actionControlReceived(action:Action):boolean
+addModel(mod:ContextModel):void
+notifyEvent(arg0:Model,arg1:Object):void
+start():void
ModelChangedListener
(from com::hp::hpl::jena::rdf::model)

Figure 26. UML class diagram of the CMS's remote service interface.

Model updater sensors

The remote service interface for the model updater sensor works in a similar way. All local sensors are discovered from the same OSGi framework and UPnP devices are created for each found sensor. This interface is deployed as its own software bundle that can be executed anytime. It can also find all new sensors that are activated in the OSGi framework during the running of the interface bundle. The designed architecture for the remote service interface for sensors is presented in Figure 27.

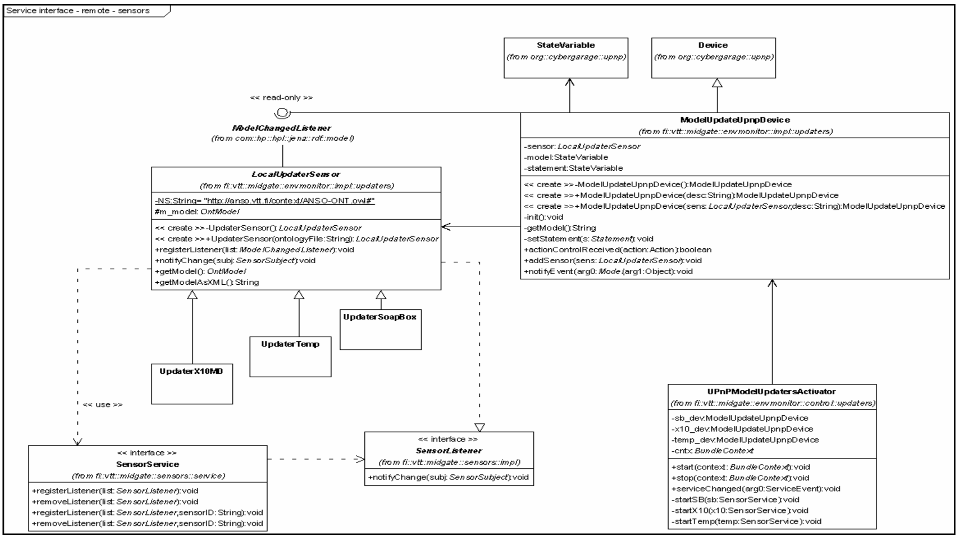

Service interface - remote - sensors
StateVariable
(from org::cybergarage::upnp)
Device
(from org::cybergarage::upnp)
<< read-only >>
ModelChangedListener
(from com::hp::hpl::jena::rdf::model)
ModelUpdateUpnpDevice
(from fi::vtt::midgate::envmonitor::impl::updaters)
-sensor: LocalUpdaterSensor
-model:StateVariable
-statement:StateVariable
<< create >>-ModelUpdateUpnpDevice():ModelUpdateUpnpDevice
<< create >>+ModelUpdateUpnpDevice(desc:String):ModelUpdateUpnpDevice
<< create >>+ModelUpdateUpnpDevice(sens: LocalUpdaterSensor,desc:String):ModelUpdateUpnpDevice
-init():void
-getModel():String
-setStatement(s: Statement):void
+actionControlReceived(action:Action):boolean
+addSensor(sens: LocalUpdaterSensor):void
+notifyEvent(arg0: Model,arg1:Object):void
LocalUpdaterSensor
(from fi::vtt::midgate::envmonitor::impl::updaters)
-NS:String= "http://anso.vtt.fi/context/ANSO-ONT.owl#"
#m_model: OntModel
<< create >>-UpdaterSensor(): LocalUpdaterSensor
<< create >>+UpdaterSensor(ontologyFile:String): LocalUpdaterSensor
+registerListener(list: ModelChangedListener):void
+notifyChange(subj: SensorSubject):void
+getModel(): OntModel
+getModelAsXML():String
UpdaterSoapBox
UpdaterTemp
UpdaterX10MD
<< use >>
UPnPModelUpdatersActivator
(from fi::vtt::midgate::envmonitor::control::updaters)
-sb_dev:ModelUpdateUpnpDevice
-x10_dev:ModelUpdateUpnpDevice
-temp_dev:ModelUpdateUpnpDevice
-cntx: BundleContext
+start(context: BundleContext):void
+stop(context: BundleContext):void
+serviceChanged(arg0:ServiceEvent):void
-startSB(sb:SensorService):void
-startX10(x10:SensorService):void
-startTemp(temp:SensorService):void
<< interface >>
SensorListener
(from fi::vtt::midgate::sensors::impl)
+notifyChange(subj: SensorSubject):void
<< interface >>
SensorService
(from fi::vtt::midgate::sensors::service)
+registerListener(list: SensorListener):void
+removeListener(list: SensorListener):void
+registerListener(list: SensorListener,sensorID: String):void
+removeListener(list: SensorListener,sensorID: String):void

Figure 27. UML class diagram of sensors' remote service interface.

Local sensors in the OSGi framework are found and their data events are registered to listen through the SensorService interface that the sensor service interfaces extend, as explained in Section 4.2.1. A special updater instance that extends the LocalUpdaterSensor class is created for each sensor. These instances contain a semantic model to hold the sensor data and a process to update the model according to the sensor data events.

A remote service interface is deployed for each sensor by creating an instance of the ModelUpdaterUpnpDevice class that listens to the semantic model of the sensor through the ModelChangedListener interface class. As changes occur in the model, the state variables similar to the UPnP device variables of the Context Monitoring Service are updated accordingly. Instances of the ModelUpdaterUpnpDevice class are created and held in the UPnPModelUpdaterActivator class, which is also the main class of the software OSGi bundle.

4.2.3 Local management interfaces

As stated in Section 4.2, the model updater sensors' management interface can be seen as their communication drivers for the hardware, and they are not explained in this section.

However, the Context Monitoring Service requires a management interface to enable updating of its context model with data provided by the local model updater sensors. The CMS's management interface receives the context information from the local sensors and updates the model accordingly. The architecture that is designed for the sensor discovery and context model updating is presented in Figure 28.

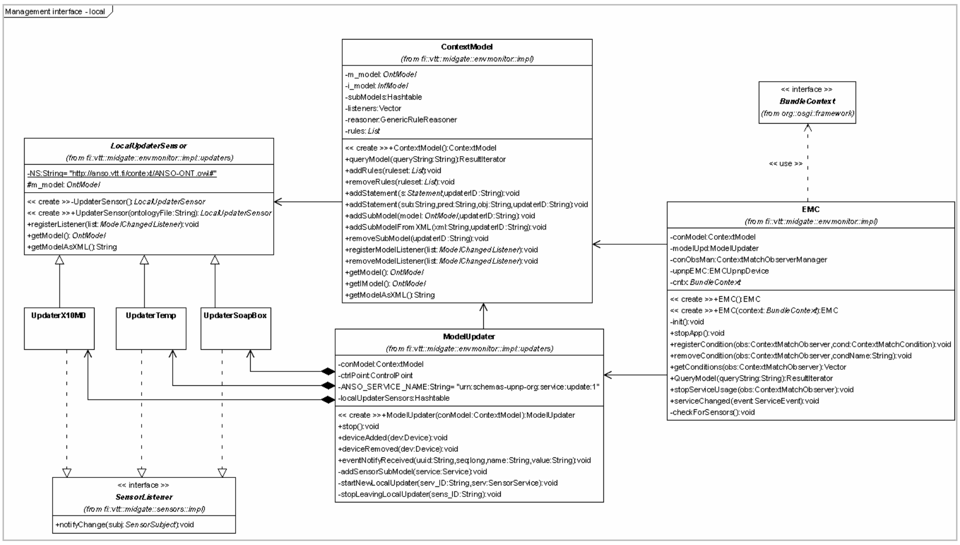

Management interface - local
ContextModel
(from fi::vtt::midgate::envmonitor::impl)
-m_model: OntModel
-i_model: InfModel
-subModels:Hashtable
-listeners:Vector
-reasoner:GenericRuleReasoner
-rules: List
<< create >>+ContextModel():ContextModel
+queryModel(queryString:String):ResultIterator
+addRules(ruleset: List):void
+removeRules(ruleset: List):void
+addStatement(s: Statement,updaterID:String):void
+addStatement(sub:String,pred:String,obj:String,updaterID:String):void
+addSubModel(model: OntModel,updaterID:String): void
+addSubModelFromXML(xml:String,updaterID:String):void
+removeSubModel(updaterID:String):void
+registerModelListener(list: ModelChangedListener):void
+removeModelListener(list: ModelChangedListener):void
+getModel(): OntModel
+getIModel(): OntModel
+getModelAsXML():String
<< interface >>
BundleContext
(from org::osgi::framework)
<< use >>
LocalUpdaterSensor
(from fi::vtt::midgate::envmonitor::impl::updaters)
-NS:String= "http://anso.vtt.fi/context/ANSO-ONT.owl#"
#m_model: OntModel
<< create >>-UpdaterSensor(): LocalUpdaterSensor
<< create >>+UpdaterSensor(ontologyFile:String): LocalUpdaterSensor
+registerListener(list: ModelChangedListener):void
+getModel(): OntModel
+getModelAsXML():String
EMC
(from fi::vtt::midgate::envmonitor::impl)
-conModel:ContextModel
-modelUpd:ModelUpdater
-conObsMan:ContextMatchObserverManager
-upnpEMC:EMCUpnpDevice
-cntx: BundleContext
<< create >>+EMC():EMC
<< create >>+EMC(context: BundleContext):EMC
-init():void
+stopApp():void
+registerCondition(obs:ContextMatchObserver,cond:ContextMatchCondition):void
+removeCondition(obs:ContextMatchObserver,condName:String):void
+getConditions(obs:ContextMatchObserver):Vector
+QueryModel(queryString:String):ResultIterator
+stopServiceUsage(obs:ContextMatchObserver):void
+serviceChanged(event ServiceEvent):void
-checkForSensors():void
UpdaterX10MD
UpdaterTemp
UpdaterSoapBox
ModelUpdater
(from fi::vtt::midgate::envmonitor::impl::updaters)
-conModel:ContextModel
-ctrlPoint:ControlPoint
-ANSO_SERVICE_NAME:String= "urn:schemas-upnp-org:service:update:1"
-localUpdaterSensors:Hashtable
<< create >>+ModelUpdater(conModel:ContextModel):ModelUpdater
+stop():void
+deviceAdded(dev:Device):void
+deviceRemoved(dev:Device):void
+eventNotifyReceived(uuid:String,seq:long,name:String,value:String):void
-addSensorSubModel(service:Service):void
-startNewLocalUpdater(serv_ID:String,serv:SensorService):void
-stopLeavingLocalUpdater(sens_ID:String):void
<< interface >>
SensorListener
(from fi::vtt::midgate::sensors::impl)
+notifyChange(subj: SensorSubject):void

Figure 28. UML class diagram of the CMS's local management interface.

The Context Monitoring Service's main class, EMC, contains the ModelUpdater class, which is responsible for updating the context model in the ContextModel class. Local sensors registered to the OSGi framework are discovered by the EMC class, which provides their service interfaces to the ModelUpdater class. The sensors registered to the OSGi framework during the running of the Context Monitoring Service are also discovered. OSGi provides listener interfaces to discover new service bundles that are registered to the OSGi framework.

An instance of their specific updater component class is created for each type of discovered sensor. The updater component classes are classes that extend the LocalUpdaterSensor class in order to create a specific updater component for each type of sensor. Each of these LocalUpdaterSensor extending classes listen to their specific type of sensor by implementing the sensor-specific extension class of SensorListener. Sensor-specific updater components update the context model in the ContextModel class according to the data event they receive from the sensors.

4.2.4 Remote management interfaces

The remote management interface is for discovering the remote sensors and Context Monitoring Service components in order to receive additional context information from other locations. An essential part of the interface is the UPnP control point used to discover the UPnP devices from the network and to provide functionalities to operate the devices.

The model updater sensors do not implement any kind of interface for remote management, thus the focus in this section is on the Context Monitoring Service. The Context Monitoring Service's architecture for remote context model updating is shown in Figure 29.

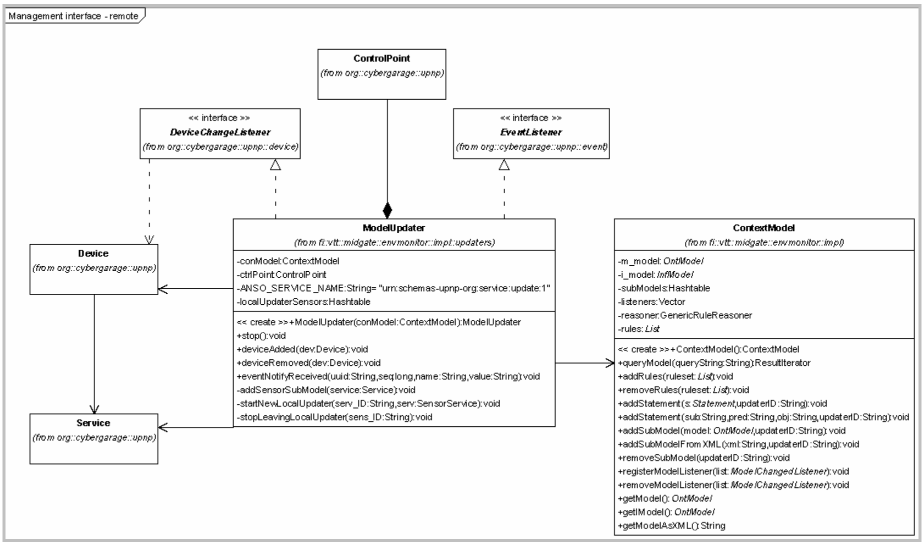

Management interface - remote
ControlPoint
(from org::cybergarage::upnp)
<< interface >>
DeviceChangeListener
(from org::cybergarage::upnp::device)
<< interface >>
EventListener
(from org::cybergarage::upnp::event)
Device
(from org::cybergarage::upnp)
Service
(from org::cybergarage::upnp)
ModelUpdater
(from fi::vtt::midgate::envmonitor::impl::updaters)
-conModel:ContextModel
-ctrlPoint:ControlPoint
-ANSO_SERVICE_NAME:String= "urn:schemas-upnp-org:service:update:1"
-localUpdaterSensors:Hashtable
<< create >>+ModelUpdater(conModel:ContextModel):ModelUpdater
+stop():void
+deviceAdded(dev:Device):void
+deviceRemoved(dev:Device):void
+eventNotifyReceived(uuid:String,seq:long,name:String,value:String):void
-addSensorSubModel(service:Service):void
-startNewLocalUpdater(serv_ID:String,serv:SensorService):void
-stopLeavingLocalUpdater(sens_ID:String):void
ContextModel
(from fi::vtt::midgate::envmonitor::impl)
-m_model: OntModel
-i_model: InfModel
-subModels:Hashtable
-listeners:Vector
-reasoner:GenericRuleReasoner
-rules: List
<< create >>+ContextModel():ContextModel
+queryModel(queryString:String):ResultIterator
+addRules(ruleset: List):void
+removeRules(ruleset: List):void
+addStatement(s: Statement,updaterID:String):void
+addStatement(sub:String,pred:String,obj:String,updaterID:String):void
+addSubModel(model: OntModel,updaterID:String): void
+addSubModelFromXML(xml:String,updaterID:String):void
+removeSubModel(updaterID:String):void
+registerModelListener(list: ModelChangedListener):void
+removeModelListener(list: ModelChangedListener):void
+getModel(): OntModel
+getIModel(): OntModel
+getModelAsXML():String

Figure 29. UML class diagram of the CMS's remote management interface.

The ModelUpdater class is used to update the remotely received data to the context model. The ModelUpdater class contains an instance of the ControlPoint class, which is provided by the CyberLink for Java development package. The UPnP control point of the ModelUpdater class discovers new UPnP devices in the network and receives their device and service descriptions. If the descriptions point to a device that is proper for model updating, it automatically requests the service's state variable containing the semantic model as XML. After that it subscribes to listen to a device's state variable for changed statements. The DeviceChangeListener interface class is used by the ModelUpdater class to receive new device descriptions in the network and the EventListener interface class is for listening to the received events of state variable changes.

4.2.5 Local control interfaces

Every software bundle in the whole service has a local control interface. The interface is used to start or stop the software bundles in the OSGi framework. The Context Monitoring Service component, local model updater sensor and remote model updater sensor deployment bundles' control interfaces are exactly the same as each other because they are defined by the OSGi Service Platform Release 3 Specification [55]. Figure 30 shows the control interfaces of the bundles the whole Context Monitoring Service contains.

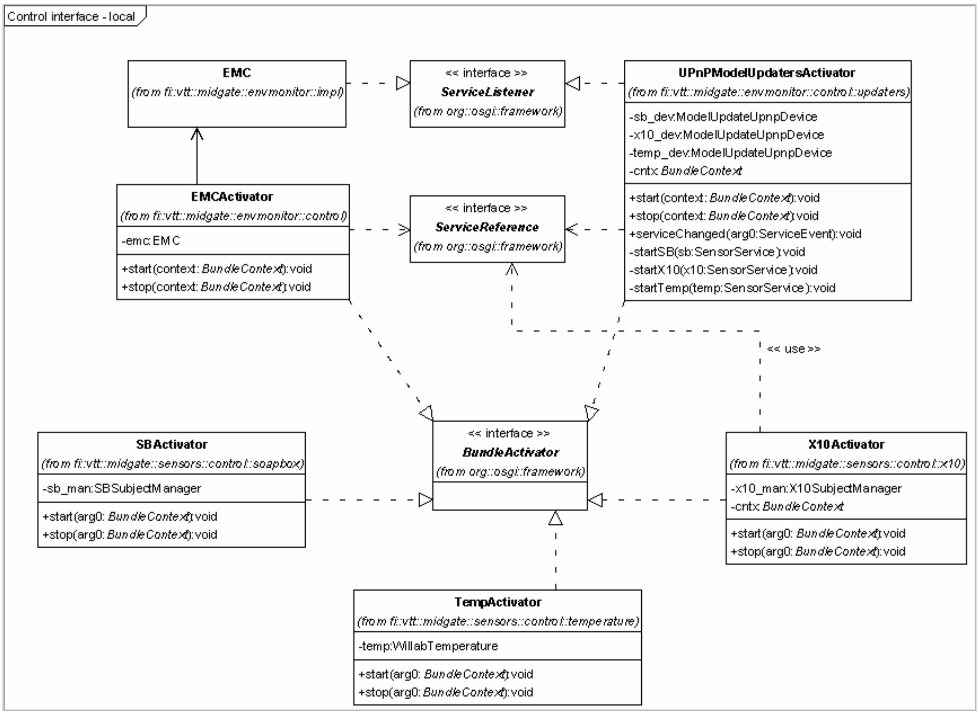

Control interface - local
EMC
(from fi::vtt::midgate::envmonitor::impl)
<< interface >>
ServiceListener
(from org::osgi::framework)
UPnPModelUpdatersActivator
(from fi::vtt::midgate::envmonitor::control::updaters)
-sb_dev:ModelUpdateUpnpDevice
-x10_dev:ModelUpdateUpnpDevice
-temp_dev:ModelUpdateUpnpDevice
-cntx:BundleContext
+start(context:BundleContext):void
+stop(context:BundleContext):void
+serviceChanged(arg0:ServiceEvent):void
-startSB(sb:SensorService):void
-startX10(x10:SensorService):void
-startTemp(temp:SensorService):void
EMCActivator
(from fi::vtt::midgate::envmonitor::control)
-emc:EMC
+start(context:BundleContext):void
+stop(context:BundleContext):void
<< interface >>
ServiceReference
(from org::osgi::framework)
<< use >>
SBActivator
(from fi::vtt::midgate::sensors::control::soapbox)
-sb_man:SBSubjectManager
+start(arg0:BundleContext):void
+stop(arg0:BundleContext):void
<< interface >>
BundleActivator
(from org::osgi::framework)
X10Activator
(from fi::vtt::midgate::sensors::control::x10)
-x10_man:X10SubjectManager
-cntx:BundleContext
+start(arg0:BundleContext):void
+stop(arg0:BundleContext):void
TempActivator
(from fi::vtt::midgate::sensors::control::temperature)
-temp:WillabTemperature
+start(arg0:BundleContext):void
+stop(arg0:BundleContext):void

Figure 30. UML class diagram of the local control interfaces.

In order to be able to register the service to the OSGi framework the bundle must contain an activator class that implements the BundleActivator interface class. The BundleActivator interface is provided by the OSGi framework and defines the methods for starting and stopping the bundle. The control interfaces of all bundles in the Context Monitoring Service are shown in Figure 30, and they all implement the BundleActivator interface.

The EMC and UPnPModelUpdaterActivator classes also implement the ServiceListener interface class. Through the ServiceListener interface, the classes are able to listen to new bundles that are installed, or to the OSGi framework, and operate accordingly. The removal of bundles is also listened for through the ServiceListener interface.

4.2.6 Remote control interfaces

Remotely starting and stopping the software bundles of the Context Monitoring Service system can be done over the network with the HTTP protocol. The OSGi framework provides an HTTP server bundle and a service for remote bundle handling over the HTTP protocol. These two bundles together can be seen as the remote control interfaces of the software bundles. The user can easily alter the OSGi frameworks' bundle configurations with an Internet browser. Access to the controlling software is password protected, so it can be safely taken advantage of for modifying the OSGi framework's bundle configurations in the deployment environment.

5. Prototype implementation and testing

The requirements and design of the Context Monitoring Service that evolves dynamically were brought up in Chapter 3 and the architecture for such a service was presented in Chapter 4. This chapter now provides a description of the developed CMS's prototype implementation and describes a demonstration application using the CMS. The implemented CMS with hardware and software configurations is presented first, followed by the developed test application to demonstrate, validate and test the designed service's proper functionality. The chapter also provides use cases of the demonstration scenario and an evaluation of the whole prototype system.

5.1 Prototype implementation

The purpose of the implemented prototype for a Context Monitoring Service was to validate the design and architecture presented in this thesis. To validate all aspects of the design, the prototype contains all the required functionalities presented in Chapter 3. The Context Monitoring Service has a context model with reasoning capabilities, which is defined with the designed ontologies. Dynamic discovery of the Context Monitoring Services and the model updater sensors is enabled by using the UPnP protocol. The Context Monitoring Service also provides conditional eventing and context model querying services for the applications.

The implementation was done by developing one OSGi bundle at a time following the designed bundle configuration presented in Figure 19. The bundles of the Context Monitoring Service prototype implementation were tested in many phases alongside the development process. The implementation was started by creating the sensor data analyzer software on top of each sensor communication driver. Data analyzers first convert the raw sensor data into different predefined states and then to semantic statements that can be added to the context model. The sensor bundles were deployed without the data conversion to semantic states to allow the data to be used by non-semantic applications as well. This also removed the need to include the Jena Semantic Framework in each bundle. The sensor bundles were tested by building a special application to use their service interfaces. The testing application was a simple GUI that showed the data the sensors produce on a screen.

After the sensor bundles were tested the context monitoring was implemented to use the sensor data. The functionalities to convert the sensor data to semantic statements were implemented into the Context Monitoring Service component bundle with the Jena framework. The context model and the Reasoner also use the same Jena framework as the data conversion processes. To provide the access to context model for the applications, the conditional eventing and model querying service were implemented. The sensor data converters and the service were tested simultaneously with a testing application that used both services.

UPnP advertisement and dynamic discovery were integrated into the system after the sensors and the Context Monitoring Service were tested properly so that they function together as expected. For the sensors, the UPnP device creator bundle was created and tested with generic UPnP control point software provided by the CyberLink UPnP package developers. The implemented bundle creates a UPnP device with a sensor-specific semantic model for each sensor. The UPnP sensor device bundle also needs the Jena Semantic Framework package for the semantic models of each sensor. However, in this way the bundle size is smaller than if each sensor had its own semantic models. The UPnP control point was implemented in the Context Monitoring Service to discover the UPnP devices. A UPnP device for the CMS was also implemented to enable discovery of other CMS components. All of these were also tested with the generic UPnP control point.

Finally, all the components of the system were integrated and tested with different bundle configurations and network topologies. The whole system was tested with the prototype application that exploits all the defined functionalities of the CMS system to see that it functioned properly. More detailed information on the prototype application is presented later on in this chapter

5.2 Configuration

The prototype implementation included different hardware and software entities to create a valid demonstration environment for the Context Monitoring Service.

The configuration was designed so that a minimal number of use cases were needed to cover the validation of all the CMS functionality. The used hardware and software configuration descriptions are presented in the following subsections, and both configurations are brought together in an overall view of the configuration.

5.2.1 Hardware

The hardware configuration of the prototype implementation includes four computational units used to run the OSGi frameworks, a VTT SoapBox (Sensing, Operating and Activating Peripheral Box) [56] for context sensing, a temperature sensor and X.10-enabled devices. This section introduces the sensors used in the prototype implementation. The computational units used are plain commercial computers and are not further described in this section.

The VTT SoapBox is a light matchbox-sized module with a processor, different sensors and wireless and wired communications. It has been developed by VTT to be utilized in prototype systems. The sensors of the SoapBox include 3-axis acceleration sensors, an illumination sensor and a proximity sensor. The Soapbox was utilized in this prototype to produce position data with the acceleration sensors. The data from the acceleration sensors was divided into six different positions states: front side up, back side up, left side up, right side up, nose up and bottom up. Wireless communication was not used in these prototypes and the SoapBox was connected to one of the computational units via a serial port [56].

X.10 [57] is a technology that allows different devices to communicate and control each other over electric wiring. The X.10 devices used in the prototype were an X.10 motion detector, an X.10 Radio Frequency (RF) receiver and an X.10 Programming Interface. The motion detector sends the detection data wirelessly to the RF receiver, which then forwards it to the electric network. The X.10 Programming Interface listens to the network and receives the motion detection data, forwarding it to the computational unit using serial communication. Controlling commands can also be sent to the X.10 network via the X.10 Programming Interface. The X.10 motion detector was used in the prototype system to receive information on inhabited spaces in the deployment environment.

The temperature sensor in the prototype system didn't include any hardware but the implementation was designed so that any type of temperature could only be utilized by changing the driver for each type of sensor. Sensor data analyzing ewas implemented as a generic interface to which sensors can easily be connected. The implemented sensor was an Internet weather station that received its data from a specific web page [58]. The web page for the weather station was developed in co-operation with VTT and Vaisala Oyj.

5.2.2 Software

The software configuration of the implemented prototype is divided to three different categories, as presented in Section 3.2.1: Central Context Monitoring Service component, Specific Context Monitoring Service component and stand-alone sensors. Computers running the Central Context Monitoring Service component contain the software OSGi bundle of the Context Monitoring Service in which the start-up setup loads the ontologies containing all the manually defined static information. In the prototype implementation the C-CMS is loaded with the static information on the environment such, as the room configurations and employee information.

The other category is the computers that are running the Specific Context Monitoring Service OSGi bundle that discovers the local model updater sensors connected to it. The purpose of the S-CMS components is to expand the whole system. The deployed S-CMS components are loaded at start-up with ontologies that define more specific information about the location in which they are deployed. By deploying the new S-CMS components the whole system's knowledge of environment expands after each deployment.

The last category is the stand-alone sensors that have an OSGi framework that only contains the software bundles for the model updater sensors and the software bundle that creates the UPnP devices for each model updater sensor. Both Context Monitoring Service components can also be directly connected to sensors, but deployment of the UPnP sensor device bundles in them is not encouraged because the data from the locally connected sensors can be acquired by other CMS components using the CMS component's model updater device discovery. The three different software configurations in their fullest extent and the possible communication paths between them are illustrated in Figure 31.

The previously introduced OSGi technology and its OSGi framework were utilized in the prototype implementation. The OSGi framework was used as an application execution framework for the components of the implementation. Every component is encapsulated in one service interface that is registered to the OSGi framework. The OSGi framework provides a controllable service execution framework and the service interfaces are easily deployed to the framework. Some of the services that OSGi provides – the HTTP server service and the service that allow remotes controlling of the bundle execution – were used in the implementation for controlling. The used implementation of the OSGi framework was the open source project OSCAR [59], which is compliant with a large portion of the OSGi service platform specification release 3 [55].

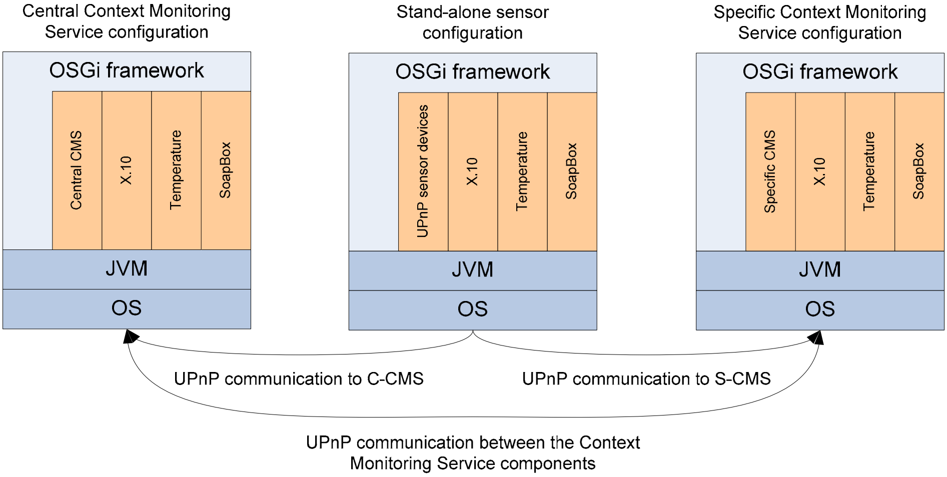

Central Context Monitoring Service configuration
Stand-alone sensor configuration
Specific Context Monitoring Service configuration
OSGi framework
Central CMS
X.10
Temperature
SoapBox
UPnP sensor devices
Specific CMS
JVM
OS
UPnP communication to C-CMS
UPnP communication to S-CMS
UPnP communication between the Context Monitoring Service components

Figure 31. The different software configurations of the prototype.

Both the Context Monitoring Service components and the UPnP sensor devices that contain the previously introduced Jena Semantic Web Framework [35] were utilized for the semantic models. The reasoning capabilities and query engine of the Jena framework were also used in the context models of both CMS components. The used version of the Jena was 2.3, which was the newest version available during the implementation process.

5.2.3 Overall view

In this section the hardware and software configurations are combined to create an overall view of the implemented prototype configuration. The distribution of the software bundles to the different computational units and the communication between them are explained. The overall view of the configuration is shown as a UML deployment diagram in Figure 32.

The deployment of the prototype consists of 5 different computers that have different CMS component OSGi bundle configurations. The solid line connections between the computers represent the UPnP communication between each of them. The blue solid line is bidirectional communication between two computers, meaning that both ends have a UPnP control point and both advertise a UPnP service in the network. The red lines represent unidirectional UPnP communication between the computers, which means that the other computer advertises the UPnP service and the other contains the UPnP control point to discover the service. Solid red lines between the OSGi bundles represent the one-way data flow from the sensors to the CMS components and the dashed line between two computers declares a dependency between them.

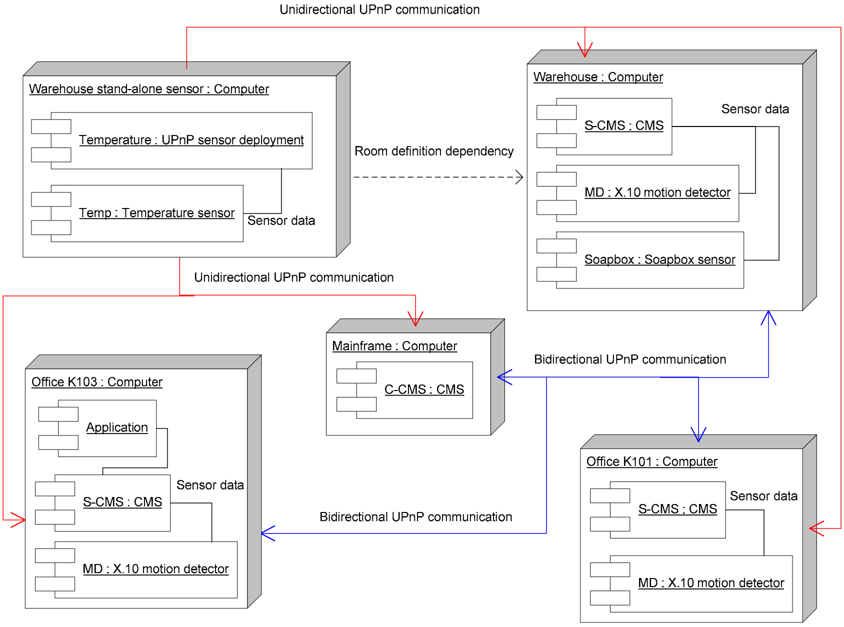

Unidirectional UPnP communication
Warehouse stand-alone sensor : Computer
Temperature : UPnP sensor deployment
Temp : Temperature sensor
Sensor data
Room definition dependency
Warehouse : Computer
S-CMS : CMS
Sensor data
MD : X.10 motion detector
Soapbox : Soapbox sensor
Unidirectional UPnP communication
Mainframe : Computer
C-CMS : CMS
Bidirectional UPnP communication
Office K103 : Computer
Application
S-CMS : CMS
Sensor data
MD : X.10 motion detector
Bidirectional UPnP communication
Office K101 : Computer
S-CMS : CMS
Sensor data
MD : X.10 motion detector

Figure 32. UML deployment diagram of the overall prototype configuration.

One of the computers is the mainframe containing only the Central Context Monitoring Service OSGi bundle, which contains the manually configured information on the deployment environment. Three other computers all have the Specific Context Monitoring Service for the specific information on the different districts in the deployment environment, which, in this case, are the different rooms of the building. These three computers also contain software bundles for different locally connected sensors. One of the computers is for the stand-alone temperature sensor and only contains the temperature software bundle and the UPnP device deployment bundle for the sensor. The prototype application that utilizes the CMS can be deployed to any of the computers because all the CMS components hold the same context information after the installation and context model synchronizations. In the prototype the application is installed in the computer in Office K103, as seen in Figure 32.

The computers that have the CMS software bundles installed communicate in both directions to receive the context information from each other. An exception is the UPnP-enabled temperature sensors, which only produce the context information for other CMS components. The declared dependency between the two components is that the stand-alone temperature sensor’s semantic data model has a mapping that it is located in the same district as the other computer. The mapping is a location reference to the room in which the computer containing the S-CMS is located.

5.3 Validation scenario and use cases

This section provides a scenario that validates the functionality of the implemented prototype. The design of the scenario enables use cases that take all the functionalities of the designed CMS system into account. The use cases have two different purposes from the validation point of view: to validate the architecture of the service and to validate the dynamic discovery of the CMS components and sensors. The architecture validation provides use cases for utilizing the service interface in the context model of the CMS. Dynamic discovery of the components is validated by designing the scenario so that the dynamic discovery is crucial for its functionality.

The overview of the whole scenario is provided to explain how the prototype is utilized. The execution of the scenario is divided into three separate use cases that validate the different functionalities of the prototype, and it is described as a short story divided according to the use cases. A test application was created to utilize the implemented CMS prototype and use the services it provides. Screenshots of the application's User Interface (UI) and UML activity diagrams are provided in each use case description to illustrate the explained functionality.

5.3.1 Scenario overview

The scenario designed to validate the functionality of the prototype utilizes the configuration described in Section 5.2.3. The scenario is built around a monitoring application that uses the Context Monitoring Service to decide when and how to trigger the alarm procedures. The deployment environment is a building complex of a company that contains offices and one warehouse. The warehouse is a large freezer used to preserve the company's highly temperature-sensitive products. The warehouse has a lever gear door, which must be kept closed in order to maintain a stable temperature. The warehouse door is equipped with a SoapBox sensor device to receive its position information, whether it is closed or open. The warehouse also contains a temperature sensor and an X.10 motion detector. The temperature sensor is used to monitor that the temperature inside the warehouse is below the maximum value and the X.10 motion detector determines whether or not the warehouse is occupied. Some of the offices in the building are also equipped with X.10 motion detectors to detect if there's an employee present in the room. The fact that not all the offices are equipped with motion detectors at the start of the scenario creates the need for dynamic discovery of different components of the CMS.

In the scenario prototype the Central Context Monitoring Service component system holds the static information on the building complex and receives the data from the deployed model updater sensors and additional S-CMS components. The C-CMS component is run on a server and each room, including the warehouse, contains an S-CMS component. The warehouse door's Soapbox and X.10 motion detector are directly connected to the computer running the S-CMS, but the temperature sensor is deployed as a stand-alone sensor.

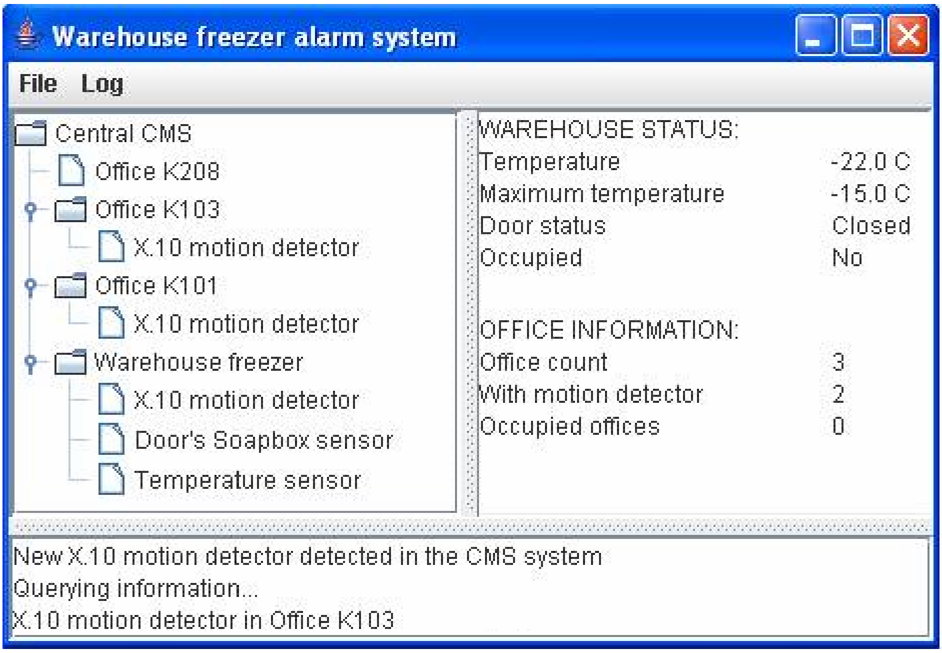
The monitoring application is installed in the OSGi framework in the warehouse's computer. The application monitors the state of the door, the presence of employees and the temperature in the warehouse. If the door is left open and there's no one present in the warehouse, the application checks to see if any of the employees are in their office rooms. The presence information for the application is provided by the X.10 motion detectors connected to the S-CMS components located in the offices. If a person is detected in an office room, the application queries the CMS's context model to receive information on the person. The room information is used to query the context model for the person located in that room and for his phone number. The application calls the person present in the building complex to go and close the door. If no one is present in the building complex, the application calls the security company for assistance. The same alarm process is executed if the temperature of the warehouse rises above the maximum value.

When a new room with a motion detector is discovered by the Context Monitoring Service component or an existing room is equipped with a motion detector the application adapts to the new situation and includes the rooms in the alarm process.

5.3.2 Dynamic discovery and advertisement

Dynamic discovery and advertisement of the Context Monitoring Service is utilized to extend the CMS system with additional CMS components or model updater sensors that provide new context information to be exploited by the applications. The new context information propagates through the CMS system using dynamic discovery and advertisement, and synchronizes the context models of the CMS components to be consistent with the information in the deployment environment. The progress of the dynamic discovery and advertisement when the CMS system is expanded is illustrated as a UML activity diagram in Appendix 6. Both progressions, utilizing the UPnP and the OSGi, are covered in the illustration.

The prototype application utilizes the CMS's dynamic discovery and advertisement of new CMS components to include more office rooms in its alerting process. Dynamic discovery of new X.10 sensor is illustrated in Figure 33.



```
WAREHOUSE STATUS:
Temperature                    -22.0 C
Maximum temperature            -15.0 C
Door status                    Closed
Occupied                       No

OFFICE INFORMATION:
Office count                   3
With motion detector           2
Occupied offices               0
```

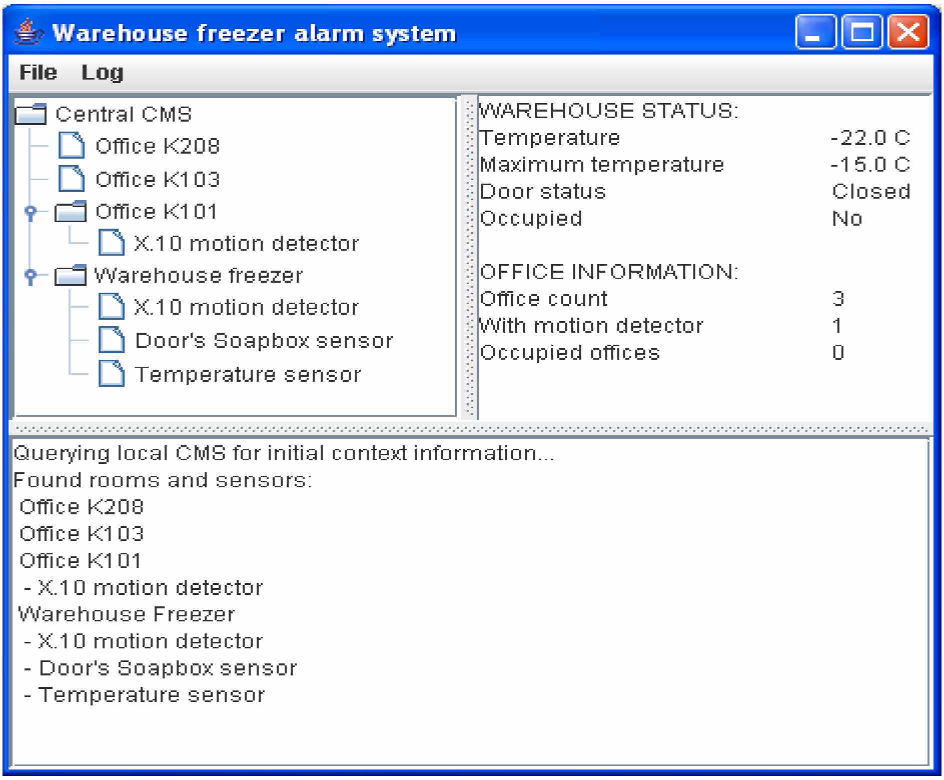
```
New X.10 motion detector detected in the CMS system
Querying information...
X.10 motion detector in Office K103
```

Figure 33. Discovery of a new X.10 motion detector.

If an office gains a motion detector, it can be used to detect whether a person is in the room and a call requesting a check of the warehouse can be made. The screenshot shows the application operation when a new S-CMS software bundle is deployed to an existing office along with an X.10 motion detector. Note that the application receives the notification about the addition by utilizing the instance addition and removal listener service that is covered in Section 5.3.4.

5.3.3 Context model querying

The monitoring application utilizes the Context Monitoring Service prototype's context model querying service for discovering all the required information during its start-up. The required information includes the offices and warehouse of the building and their sensor configurations. The query service is also utilized when the alarm processes are triggered to receive the occupancy statuses of the rooms and information on the employees occupying the rooms. An example of the employee information is the phone numbers that are used to call the employees to check the warehouse freezer. The effects of the context model querying and the received context information during the start-up are illustrated with a screenshot of the application GUI during the start-up in Figure 34.



```
Central CMS
    Office K208
    Office K103
    Office K101
        X.10 motion detector
    Warehouse freezer
        X.10 motion detector
        Door's Soapbox sensor
        Temperature sensor
```

```
WAREHOUSE STATUS:
Temperature                     -22.0 C
Maximum temperature             -15.0 C
Door status                     Closed
Occupied                        No

OFFICE INFORMATION:
Office count                    3
With motion detector            1
Occupied offices                0
```

```
Querying local CMS for initial context information...
Found rooms and sensors:
 Office K208
 Office K103
 Office K101
  - X.10 motion detector
 Warehouse Freezer
  - X.10 motion detector
  - Door's Soapbox sensor
  - Temperature sensor
```

Figure 34. Querying the context model for context information during the start-up.

When the monitoring application is installed and started in the OSGi framework it invokes a context model querying the CMS component installed in the same framework. At this stage all the CMS components in the deployment environment have synchronized their context models and all the CMS components contain the same context information. This enables the application to receive all the possible context information on the deployment environment by just querying the context model of the CMS component running in the same OSGi framework. The query is done by providing an RDQL query clause that defines what context information the application wants to receive as a result of the query. By invoking the queries the application receives the current state of the deployment environment's context and can start monitoring the changes in it.

To illustrate in detail how the context model querying is utilized by the application and how it progress from the application side to the CMS side, a UML activity diagram is provided in Appendix 7.

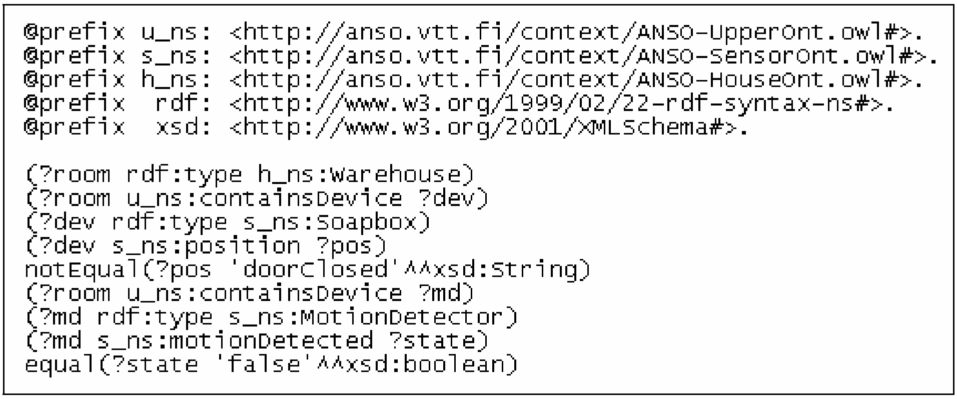
5.3.4 Conditional eventing

The monitoring application receives the context information changes through the conditional eventing service the Context Monitoring Service is providing. The conditional eventing includes the conditional rule listener and the instance addition and removal listener explained in Section 3.7. Both types of conditional eventing service are utilized by the application to validate their functionality.

The condition rule listener is used by the application to listen to the events of changes in the existing information in the context model and receive the new information to update its current state. Receipt of the changed temperature values, occupancy states in the rooms and door status in the warehouse is based on the conditional rule listener. However, utilization of the instance addition and removal listener is required because the office and sensor configurations can change due to the dynamic discoveries and advertisements of the CMS components. The application's correct and efficient functionality is dependent on the up-to-date information on these configurations. The application's room and sensor configuration information is preserved consistent by triggering the queries for new configuration information with instance addition and removal listener notification events. When the application receives the notification of a dynamic addition or removal of CMS components, the context model is queried for full information. More detailed information on the progression of both types of conditional eventing processes is illustrated as UML activity diagrams in Appendix 8.

The monitoring application determining when to trigger the alarm and call for assistance is based on a specific condition rule registered with the listener. The rule enables notification events to the application every time the warehouse is open, and whether the warehouse is or is not occupied, or the temperature is above the maximum value. An example of the condition rule that triggers the event when the warehouse door is left open and it is unoccupied is shown in Figure 35.

The rule in Figure 35 sets the Reasoner in the CMS's context model to find all instances of the ontology class type Warehouse. The found warehouse is then searched for the Soapbox device and the current value for its position is checked. If the Soapbox's position value is not "doorClosed", which means that the door is closed, the warehouse is searched for motion detectors, and if the presence state of the found motion detector is "false", the alarm is triggered. The rule for the temperature rise event is similar but the Soapbox sensor check is replaced with the temperature sensor. The temperature sensor's current value is compared with the defined maximum value for the warehouse and the alarm is triggered if it is greater.

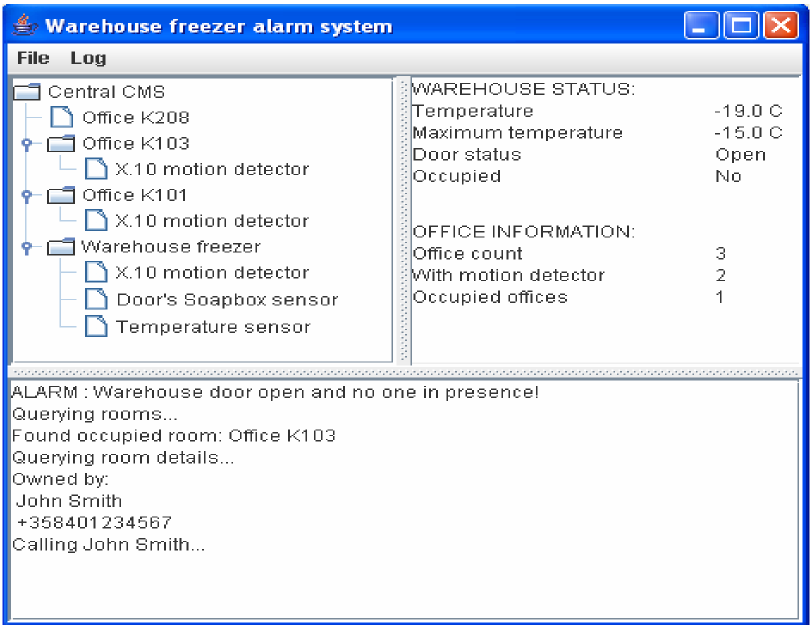


```
@prefix u_ns: <http://anso.vtt.fi/context/ANSO-UpperOnt.owl#>.
@prefix s_ns: <http://anso.vtt.fi/context/ANSO-SensorOnt.owl#>.
@prefix h_ns: <http://anso.vtt.fi/context/ANSO-HouseOnt.owl#>.
@prefix  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix  xsd: <http://www.w3.org/2001/XMLSchema#>.

(?room rdf:type h_ns:Warehouse)
(?room u_ns:containsDevice ?dev)
(?dev rdf:type s_ns:Soapbox)
(?dev s_ns:position ?pos)
notEqual(?pos 'doorClosed'^^xsd:String)
(?room u_ns:containsDevice ?md)
(?md rdf:type s_ns:MotionDetector)
(?md s_ns:motionDetected ?state)
equal(?state 'false'^^xsd:boolean)
```

Figure 35. Condition rule used to trigger the alarm process.

The application starts the alarm process when it receives notification from the S-CMS that the condition for the alarm matches. As described in the overview of the scenario in Section 5.3.1, if occupied office rooms are found, their owner is queried from the context model and a call for each is made until someone answers. If no employee is found in the office rooms, the application calls the security company for assistance. These cases are illustrated as application UI screenshots in Figure 36, where the alarm is triggered by the warehouse door that is left open and the rise in temperature is the triggering condition.

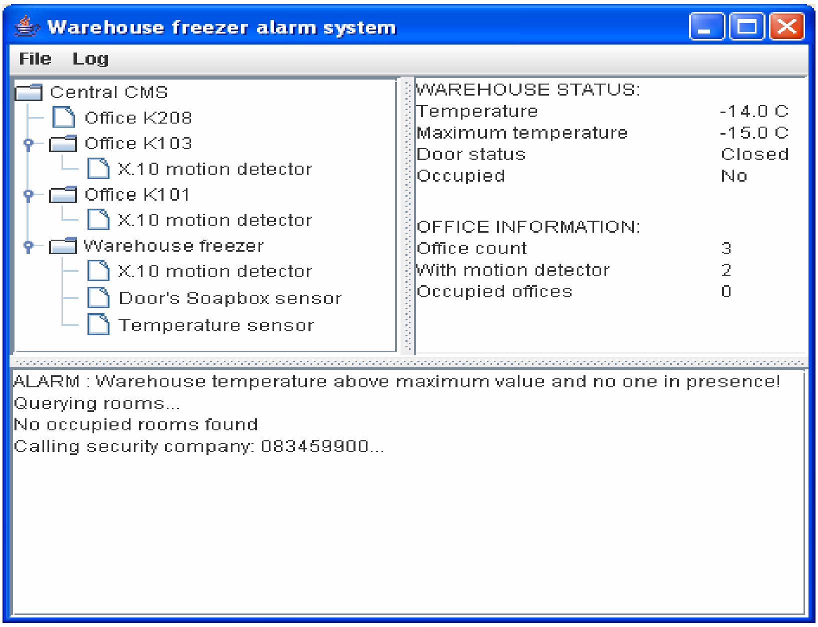


```
WAREHOUSE STATUS:
Temperature                -19.0 C
Maximum temperature        -15.0 C
Door status                Open
Occupied                   No

OFFICE INFORMATION:
Office count               3
With motion detector       2
Occupied offices           1
```

```
ALARM : Warehouse door open and no one in presence!
Querying rooms...
Found occupied room: Office K103
Querying room details...
Owned by:
 John Smith
 +358401234567
Calling John Smith...
```

a) Alarm process when one of the rooms is occupied.



```
WAREHOUSE STATUS:
Temperature                -14.0 C
Maximum temperature        -15.0 C
Door status                Closed
Occupied                   No

OFFICE INFORMATION:
Office count               3
With motion detector       2
Occupied offices           0
```

```
ALARM : Warehouse temperature above maximum value and no one in presence!
Querying rooms...
No occupied rooms found
Calling security company: 083459900...
```

b) Alarm process when none of the rooms are occupied.

Figure 36. Alarm processes triggered by conditional eventing.

5.4 Evaluation of prototype

The prototype implementation was created to validate and test the functional and architectural design of the service presented in Chapters 3 and 4 respectively. This section merely focuses on the evaluation of the implementation, not the design aspects. The functional and architectural designs are discussed in Chapter 6.

The prototype consisted of the Context Monitoring Service components and a test application that used the service to support its functionalities. The test application was based on a scenario that was designed to utilize all the functionalities that were identified for the Context Monitoring Service in Chapter 3. The evaluation of the utilized functionalities is discussed in the following subsections. The functionality division in the subsections is made according to the layered model of the system presented Section 4.1.

5.4.1 Service layer

The service layer contained the context model, the ontologies, the Reasoner, and the two services for the applications: context model querying and conditional eventing. In the prototype implementation the information in the context model was presented with ontologies identified from the design scenario. Use of the created ontologies provided the necessary semantic representation of the context information in the context model, enabling efficient deduction of new information with the Reasoner. However, the designed ontologies were simple and not comprehensive enough to be used in large-scale context-awareness systems.

The service interface the Context Monitoring Service provided for the applications functioned properly in the implementation. The queries to the context model were efficient enough and didn't stall the system, even when the efficiency was tested with several applications simultaneously querying the context model. Conditional eventing in the prototype also provided the designed functionality. Although some improvements in efficiency could have been made for both functionalities, the scope of the prototype was too wide to focus on small details. The Jena Semantic Framework was used for the context model, Reasoner and query engine implementation, and the current version of the framework was sufficient for the prototype.

5.4.2 Model updater device advertisement and discovery layer

The model updater device advertisement and discovery layer consisted of discovering the local sensors from the OSGi framework and the remote sensors via the UPnP protocol. The layer also contained the mechanism for the CMS components to discover each other.

Local sensor discovery proved to work fluently through the OSGi framework service discovery. Previously installed sensors were discovered during the start-ups of the Context Monitoring Service components bundle and the UPnP sensor device bundle, and during run-time the installed sensors were discovered and utilized by both bundles as soon as they were activated.

The implementation of the remote sensor discovery was enabled by deploying a UPnP control point to the Context Monitoring Service. The control point also automatically requested the context model from the remote model updater sensors and CMS components, and subscribed to listen to the changes. The CyberLink UPnP developer package was found to be very feasible for implementing the UPnP control point, although the delay parameters for updating the control point status had to be changed to guarantee efficient functioning of the prototype.

5.4.3 Model updater sensor layer

The local and remote sensors are located in the model updater sensor layer, together with the Context Monitoring Service's context model distribution using the UPnP protocol. In the prototype the utilized sensors were motion detector, SoapBox sensor device and temperature sensor. However, an actual physical temperature sensor was not used and the sensor was implemented as software that downloaded the data from the Internet.

The software for the sensors was designed so that the same software can be utilized with minimum changes for all different types of sensors. The implementation worked as expected with the three different sensors used in the prototype system. All the sensors have different driver components but the updater software is the same apart from small differences in data analyzing. The sensor software is same for the local and remote sensors, although the remote sensors are deployed by creating UPnP devices for each local sensor using a UPnP device software bundle.

The Central and Specific Context Monitoring Service components can be seen as model updater UPnP devices, as are the remote sensors. The UPnP device and service descriptions are exactly the same for both. To prevent the context information being multiplied in the context models by the CMS component discoveries, the received context models are appended to the context model as submodels. This allows the CMS components to extract the base model and send its unique information to other CMS components.

6. Discussion

The review of the literature on previous research and existing technologies in the domain area of this thesis was carried out in Chapter 2. This chapter draws together the implemented prototype of the Context Monitoring Service to compare its design and functionalities with the existing systems found in the literature review. Further in this chapter the prototype is dissected closely to reveal its strengths and weaknesses. The exposed weaknesses constitute the need for prototype's further development, and these development targets are discussed at the end of the chapter.

6.1 Comparison with existing architectures

Two interesting existing architectures that support context-awareness were presented in Chapter 2: the Semantic Space infrastructure and the CoBrA context broker architecture. In this section the Context Monitoring Service's architecture designed in this thesis is compared with both of these existing architectures and the features that are identical to each other, and the differences between the architectures are discussed.

There are several similarities in the functionalities and technologies used in these architectures and the Context Monitoring Service. The similarities consist of utilizing the Semantic Web technologies to create ontologies for representing the context information as a context model, using a Reasoning engine to infer high-level information from the sensed context, and providing a mechanism to make queries to the context information in a context model. However, these similarities are prevailing in systems that process semantic representations of context information to enable context-awareness. Despite the similarities between the architectures, CoBrA includes additional mechanisms for detecting and repairing inconsistencies in the context model, and policies for protecting the privacy of the users. These features are in the list of future research for the CMS.

The service interface towards applications in the CMS architecture aggregates the features of the service interfaces of the existing architectures. Both implement a querying service for context information which is based on utilization of the RDQL query language, but only CoBrA implements an event notification service to notify certain states within the context of the environment. A similar functionality to the CMS's event notification service can be seen in the Semantic Space architecture's high-level context information reception service, which provides the applications with a possibility to receive high-level context information by adding reasoning rules to the reasoning engine. However, the reception of the inferred information is based on the context querying service, thus the notification mechanism isn't included in the service.

All the architectures provide a mechanism to dynamically discover and advertise the devices that acquire new context information from different sources. Both CMS and Semantic Space utilize UPnP technology to provide a uniform interface for devices with different communication protocols, whereas CoBrA presents an approach that utilizes the proxy design pattern to create a proxy agent between the device and the CoBrA. However, this solution requires implementation of a proxy agent for every device utilizing a new communication protocol, but the UPnP technology is independent of the underlying physical media and transport so the interface is the same for all of the devices.

The CMS architecture exploits the use of UPnP even further. As the discovery and advertisement are only used in Semantic Space and CoBrA to discover and advertise the sensors that acquire new context information, CMS utilizes it to discover and advertise not only the sensors but new CMS components as well. New CMS components can be deployed to different locations and will dynamically share their information with each other. Context model synchronization with other CMS components facilitates deployment of context-aware applications in new physical locations. Further, if the CMS's service interface is extended to expose the context model for modification, the applications could produce new context information for the whole CMS system. Data produced by a single application would spread through the middleware to be utilized by all the applications in the CMS deployment environment.

Compared with Semantic Space and CoBrA, the utilization of the OSGi service framework is an advantage for the CMS architecture when using the integrability and reusability that OSGi provides as a point of view. The fact that all of the architectures utilize the Jena Semantic Framework – which has only been implemented in Java – heavily binds the implementation to being done in Java. The difference between the architectures is that instead of starting the Java

Virtual Machine (JVM) for every additional component installed in the same computational entity, the CMS architecture utilizes the OSGi service framework to allow the components to be started in the same JVM in run-time. This allows the CMS architecture to be more efficiently decoupled into separate independent components, which can be used to extend the system in run-time. In addition, as each component registers its own service interface to the OSGi service registry, the components can also be utilized by other applications and services not belonging to the Context Monitoring Service. For example, the Soapbox sensor's data can be received in its raw form by directly utilizing the service interface it has registered to OSGi.

6.2 Strengths and weaknesses

The designed architecture is dissected in this section in order to discuss the beneficial features and functionalities along with the restrictions that are in need of future development. The benefits that come from the ontology-driven processing of context information, such as reasoning over the context information and providing the ability to query the information with query languages, which are common to all context monitoring services, are only discussed briefly. First the Context Monitoring Service is dissected from a single CMS component's point of view, after which the dissertation focuses on the CMS system as a whole.

6.2.1 Single component

As stated previously, the architecture provides the common benefits of a context-awareness system that utilizes Semantic Web technologies to process the context information. Using ontologies to represent the context information enables better machine interpretation and a mutual understanding of the shared knowledge between different systems. These advantages are also exploited in the design of the CMS architecture.

The reasoning over the context information using logic that is provided by the ontology or some user-defined rules enables the deduction of new context information from the existing data. The Reasoning engine also provided a possibility to design a generic event notification service for the applications. The utilization of the forward-chaining rules with the Reasoning engine enables a simple interface to the application side, which provides an extensive possibility to define when to receive the notifications. Another exploited feature is the semantic relationships the ontology defines between the context information instances, which allows querying the information with conditional query clauses. The querying of the context model is utilized in the CMS architecture to provide the applications with a mechanism to receive the context information they desire. The querying service interface can also be designed to be simple but efficient, and extensive for receiving the context information.

Utilization of the ontologies also has disadvantages, which were found in the prototype. This is because the performance is always a problem in systems that utilize the semantic representation technologies for describing the context information. The problem particularly arises when the context model is substantially large. The processing of the context information represented with ontologies, such as the queries to the context information and the reasoning of the implicit information, are highly CPU consuming tasks. If the system produces a considerable number of context information change events, the Reasoning engine has to infer the context model several times in a small amount of time. Furthermore, if a number of applications are invoking several queries to the same context model at the same time, the querying engine requires a high amount of CPU time, which reduces the overall performance. High computing requirements create a challenge for the service architecture design to extend the CMS system to mobile terminals in the future.

6.2.2 System as a whole

The strength of the prototyped CMS architecture is that it utilizes the OSGi framework as a platform on which the components of the CMS are run. Utilization of the OSGi framework provides the benefit of being able to reconfigure the overall configuration in run-time. For example, if a new sensor is brought to the space and the model updater sensor software bundle for it is installed in the same OSGi framework, this can all be done without shutting down the existing software bundles running in the OSGi. Using the OSGi framework as platform prevents the configuration changes causing any harmful effects on the whole CMS system.

The feature enabling the CMS system to be extended by utilizing the UPnP advertisement and discovery to add new remote CMS components after the initial C-CMS component is deployed provides several benefits. Firstly, the ontologies that are to be used in the system do not need to be known before the system is deployed. The CMS system can be provided with context information represented by completely new ontologies in run-time. Any type of model updater device that senses and produces context information can easily be deployed without modifications to the system as long as its context information is presented with ontologies. As the CMS components use the same UPnP service interface for communication, the expansion of the CMS system's deployment environment with new districts containing new types of context information is also a trivial task.

The distributed CMS components also increase the performance of the whole CMS system and provide more efficient inferring of high-level context information with less expense of performance. As the context information is synchronized between all the CMS components, the context information is the same in every context model of each CMS component. The applications utilizing the information can be distributed to different computational units that have the CMS component installed with no loss in their context-awareness.

Inferring implicit high-level context information by utilizing the user-defined reasoning rules and semantic relationships of the ontologies is a demanding task for the computer. More efficient inferring is achieved by deploying the CMS system in several separate components and providing different rule sets for each CMS's Reasoning engine instead of a full rule set for all. This is particularly beneficial when the given reasoning rule set is extensive.

However, regardless of all the presented advantages of utilization of UPnP in the designed architecture, it also poses a security risk in the system. The different components in a CMS system use UPnP for the communication, which does not specify sufficient security mechanisms. This means that the context information of the deployment environment is available for other parties that should not receive the information. The context information can be, for example, the occupancy status of the house, which can be exploited for criminal purposes.

6.3 Development targets

The prototype has proven this kind of architecture to be very useful as the need for deployment of larger-scale context monitoring services arises. However, it is clear that further development is required if implementation of this architecture should be made available commercially. This section discusses possible solutions for the identified weaknesses and restraining features of the architecture and its future development.

The current implementation of the UPnP protocol in the prototype is not fully secure and security measures should be developed. One possible solution that can be applied to develop the system further is the secure UPnP platform [60] that is being developed at VTT Technical Research Centre of Finland. It provides authentication of hosts, data confidentiality and integrity, as well as key management, by employing well-known and proven security components, in particular the Secure Sockets Layer (SSL) and X.509 certificates [60].

It would be interesting to extend the Context Monitoring Service system to mobile terminals if the processing of semantic context information did not have high computing requirements. Using OWL Lite, the stripped-down version of OWL, could enable the deployment of the CMS to a mobile terminal, but it would require performance tests in the future. One possibility is to maintain the processing of semantic information in stationary units that have more computing power and deploy a lightweight application that provides the service interface of CMS to applications in a mobile terminal. Communication between full CMS and lightweight CMS clients could be arranged by extending the CMS architecture to correspond with the Message-Oriented Middleware (MOM) principle. The MOM architecture would utilize asynchronous message passing for communication between the two nodes. Another way could be that all the method invocations are mediated from the mobile terminal to a stationary unit using Java Remote Method Invocation (RMI).

The mobile terminal could also define its own lightweight context model as an RDF/XML coded text representation. The context model could contain its hardware attributes represented with an ontology that can be sent further to other CMS components. The limitations to this type of configuration are that the mobile terminal could not perform any operations in its context model without a loss in performance. Thus updating the context information, much less inferring new information from it, wouldn’t be reasonable.

The service interface provided for the applications can be also seen as a target for future development. The current design does not provide a possibility to alter the context information in the context model. A service interface that allows additions to and removals of semantic statements – or even a whole semantic model – from the context model would provide applications with a mechanism to configure the context model. The service interface could also provide a way to insert new reasoning rules in the Reasoner. However, if the modification of the context model information and Reasoner rules is enabled for the applications, an authorization system is required to separate reliable applications from unreliable. The system would only give the modification privileges to applications that are expected to modify the information without causing harm. This type of trusted application could be a tool for creating the static context information instances of the deployment environment, such as the building with room configuration and employee information.

In normal circumstances the sensors produce reliable data to the context model, but the Reasoner can be set to produce information that might not be consistent with the real situation. Another source of inconsistent information would be the service interface for applications that allows the modification of context information. Further development could be done by developing a mechanism for maintaining the consistency of the context model. For example, special consistency checking rules could be created for the Reasoner that would reason the context information isn’t consistent and would try to repair it or even send a notification about the inconsistent state.

The designed ontologies were minimal and were only developed to cover the scenario on which the design of the architecture is based. Research can be done to extend the ontologies to represent more detailed features and cover a wider area of context information. Extending the ontology to cover a more detailed representation of the user’s actions and preferences would be beneficial for the applications because in the end the applications are created to serve the user.

7. Conclusion

In this thesis the fundamental research problem was to find a solution for the dynamic acquisition and representation of distributed context information and its efficient provision for applications. To address this problem the research focused on a service architecture that enables the context-aware applications by compiling the context information from separate sources and providing the applications with access to it in a service-oriented manner. The problems and the motivation that led to the development of this kind of service architecture were introduced first. The goals for the work were also set and an approach to solve the research problem was presented. Preliminary research of possible solutions for the problem was done in the form of a literature review, which was presented to give an insight to the domain of the work. Through the literature review the best available technologies and the methods for solving the problems of such an architecture were selected.

The design of a dynamic context monitoring service was presented by first examining its requirements and operation principles from an example scenario in which such an architecture would be useful. The resulting design was decomposed into different features that provided the identified functionalities and formed the structure of the architecture. The deployment aspects of the features as OSGi software bundles to the OSGi framework were also presented.

Having gathered the requirements for a dynamic context monitoring service and identified the functional features to fulfil the requirements, an actual architecture was designed. The solution architecture, which integrated the functional features together, was presented by dividing the features into different functional layers. Each layer's internal operation was presented with Gane-Sarson data flow diagrams. The solution architecture for the internal functionalities was encapsulated with a service architecture that provides interfaces for the utilization and control of the service. The service architecture was presented as a division into service, management and control interfaces that were further divided into local and remote interfaces. The structure of these interfaces was illustrated with UML models.

The validation of the designed architecture was done by developing a prototype implementation and an example application that utilized the provided services. The application was designed to be dependable in all functions the context monitoring service was designed to provide.

To conclude, the research and work carried out in this thesis contributed a novel solution for a context monitoring service that is dynamically and easily extendable, and efficiently provides applications with the distributed context information to enhance their context-awareness. The novelty value derives from the utilization of a dynamically evolving semantic context model, which the context monitoring service provides for applications requiring context-awareness.

References

[1] Weiser, M. (1991). The Computer for the 21st Century. Scientific American, September, Vol. 26, Issue 3, pp. 94–104.

[2] Abowd, G. D. & Mynatt, E. D. (2000). Charting Past, Present and Future Research in Ubiquitous computing. ACM Transactions on Computer-Human Interaction, Vol. 7, pp. 29–58.

[3] Bagrodia, R., Chu, W. W., Kleinrock, L. & Popek, G. (1995). Vision, Issues and Architecture for Nomadic Computing. IEEE Personal Communications, Vol. 2, pp. 14–27.

[4] Gu, T., Pung, H. K. & Zhang, D. Q. (2004). Toward an OSGi-Based Infrastrcture for Context-Aware Applications. IEEE pervasive computing, Vol. 3, Issue 4, pp. 66–74.

[5] Satyanarayanan, M. (2001). Pervasive Computing: Vision and Challenges. IEEE Personal Communications, Vol. 8, pp. 10–17.

[6] Dey, A. K. & Abowd, G. D. (2000). Towards a Better Understanding of Context and Context-awareness. In: Proceedings of Computer-Human Interaction 2000 (CHI 2000), Workshop on The What, Who, Where, When and How of Context-Awareness, April 3, Hague, Netherlands. 12 p.

[7] Schilit, B. & Theimer, M. (1994). Disseminating Active Map Information to Mobile Hosts. IEEE Network, Vol. 8, Issue 5, pp. 22–32.

[8] Ward, A., Jones, A. & Hopper, A. (1997). A New Location Technique for the Active Office. IEEE Personal Communications, Vol. 4, Issue 5, pp. 42–47.

[9] Salber, D., Dey, A. K. & Abowd, G. D. (1999). The Context Toolkit: Aiding the Development of Context-Enabled Applications. In: Proceedings of Computer-Human Interaction 1999 (CHI 1999). Pp. 434–441.

[10] Schilit, B., Adams, N. & Want, R. (1994). Context-aware Computing Applications. In: International Workshop on Mobile Computing Systems and Applications. Pp. 85–90.

[11] Dey, A. K., Abowd, G. D. & Wood, A. (1998). Cyberdesk: A Framework for Providing Self-Integrating Context-Aware Service. In: Proceedings of Intelligent user interfaces, San Francisco, California, USA. Pp. 47–54.

[12] Pascoe, J. (1998). Adding Generic Contextual Capabilities to Wearable Computing. In: Proceedings of 2nd International Symposium on Wearable Computers. Pp. 146–153.

[13] Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, Vol. 5, Issue 2, pp. 199–220.

[14] Studer, R., Benjamins, V. & Fensel, D. (1998). Knowledge Engineer: Principle and Methods. IEEE Transaction in Data and Knowledge Engineering, Vol. 25, Issue 1 & 2, pp. 161–197.

[15] Perez, A. G. & Benjamins, V. (1999). Overview of Knowledge Sharing and Reuse Components. Ontologies and Problem-Solving Methods. In: Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5), August 2, Stockholm, Sweden. Pp. 1–15.

[16] Uschold, M. (1996). Building Ontologies: Towards a unified methodology. In: 16th Annual Conference of the British Computer Society Specialist Group on Expert Systems, December 16–18, Cambridge, United Kingdom.

[17] Pakkala, D., Koivukoski, A. & Latvakoski, J. (2005). MidGate: Middleware Platform for Service Gateway Based Distributed Systems. In: The 11th International Conference on Parallel and Distributed systems (ICPADS 2005), Fukuoka, Japan, July 20–22. 7 p.

[18] Pakkala, D. & Latvakoski, J. (2004). Distributed Service Platform for Adaptive Mobile Services. In: Proceedings of International Conference on Pervasive Computing and Communications (PCC-04), Las Vegas, USA. Pp. 771–777.

[19] World Wide Web Consortium. (25.7.2006). Home Page, URL: http://www.w3.org/.

[20] World Wide Web Consortium. (25.7.2006). W3C Semantic Web Activity, URL: http://www.w3.org/2001/sw/.

[21] Davies, J., Studer, R. & Warren, P. (2006). Semantic Web Techologies: Trend and Research in Ontology-based Systems. John Wiley & Sons, Ltd, Chichester. 312 p.

[22] World Wide Web Consortium. (26.7.2006). Extensible Mark-up Language, URL: http://www.w3.org/XML/.

[23] World Wide Web Consortium. (26.7.2006). XML Schema, URL: http://www.w3.org/XML/Schema.

[24] Fallside, D. (ed.) (2001). XML Schema Part 0: Primer, 2 May 2001, URL: http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[25] Manola, F. & Miller, E. (eds.) (2004). RDF Primer, W3C Proposed Recommendation 10, February 2004, URL: http://www.w3.org/TR/rdf-primer/.

[26] Brickley, D. & Guha, R. V. (eds.) (2004). RDF Vocabulary Description Language 1.0: RDF Schema, W3C Proposed Recommendation 10 February 2004. URL: http://www.w3.org/TR/rdf-schema/.

[27] McGuiness, D. & Harmelen, F. (eds.) (2004). OWL Web Ontology Language Overview, W3C Proposed Recommendation 10 February 2004. URL: http://www.w3.org/TR/owl-features/.

[28] Smith, M. K., Welty, C. & McGuinness, D. L. (eds.) (2003). OWL Web Ontology Language guide, W3C Proposed Recommendation 15 December 2003. URL: http://www.w3.org/TR/2003/PR-owl-guide-20031215.

[29] The Protégé Ontology Editor and Knowledge Acquisition System. (26.7.2006). Home Page, URL: http://protege.stanford.edu/.

[30] TopBraid Composer – The Complete Semantic Modelling Toolset. (22.9.2006). Home Page, URL: http://www.topbraidcomposer.com/.

[31] SWOOP – A Hypermedia-based Featherweight OWL Ontology Editor. (22.9.2006). Home Page, URL: http://www.mindswap.org/2004/SWOOP/.

[32] Seanborne, A. (ed.) (2004). RDQL – A Query Language for RDF, W3C Member Submission 9 January 2004,
URL: http://www.w3.org/Submission/RDQL/.

[33] Miller, L., Seanborne, A. & Reggiori, A. (2002). Three Implementations of SquishQL, A Simple RDF Query Language. In: Proceedings of 1st International Semantic Web Conference, June 9–12, Sardinia, Italy. Pp. 423–426.

[34] Guva, R. H., Lassila, O., Miller, E. & Brickley, D. (26.7.2006). Enabling Inference, URL: http://www.w3.org/TandS/QL/QL98/pp/enabling.html.

[35] Jena Semantic Web Framework. (27.7.2006). Home Page, URL: http://jena.sourceforge.net/.

[36] Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A. & Wilkinson, K. (2003). Jena: Implementing the Semantic Web recommendations, technical report HPL-2003-146, Hewlett Packard Laboratories Bristol. URL: http://www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf.

[37] Open Services Gateway Initiative (OSGi). (27.7.2006). Home Page, URL: http//www.osgi.org.

[38] Open Services Gateway Initiative (OSGi). (27.7.2006). OSGi Service Platform Release 4 CORE Specification, URL: http://www.osgi.org/ resources/spec_download.asp.

[39] Open Services Gateway Initiative (OSGi). (27.7.2006). OSGi Technology. URL: http://www.osgi.org/osgi_technology/index.asp?section=2.

[40] Flanagan, D. (2005). Java in a Nutshell: A Desktop Quick Reference, 5th edition, O'Reilly Media, Sebastopol. 1224 p.

[41] Helai, S. (2002). Standards for Service Discovery and Delivery. Pervasive Computing, IEEE, Vol. 1, Issue 3, pp. 95–100.

[42] UPnP Forum. (27.7.2006). Home Page, URL: http://www.upnp.org/.

[43] Miller, B. A., Nixon, T., Tai, C. & Wood, M. D. (2001). Home Networking with Universal Plug and Play. Communications Magazine, IEEE, Vol. 39, Issue 12, pp. 104–109.

[44] Understanding Universal Plug and Play: A White Paper. (27.7.2006). URL: http://www.upnp.org/download/UPnPDA10_20000613.htm.

[45] Droms, R. (1997). Dynamic Host Control Protocol. IETF Network Working Group, Request for comments (Standards Track) 2131. URL: http://www.ietf.org/rfc/rfc2131.txt.

[46] Goland, Y. Y., Cai, T., Leach, P., Gu, Y. & Albright, S. (1999). Simple Service Discovery Protocol, IETF Network Working Group, Draft. URL: http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt.

[47] Simple Object Access Protocol v. 1.2, W3C Recommendation (2003). (28.7.2006). URL: http://www.w3.org/TR/soap12-part1/.

[48] Cohen, J. M., Aggarwal, S. & Goland, Y. Y. (2000). General Event Notification Architecture Base: Client to Arbiter, IETF Working Group, Draft. URL: http://www.upnp.org/download/draft-cohen-gena-client-01.txt.

[49] UPnP Device Architecture. (27.7.2006). URL: http://www.upnp.org/download/UPnPDa10_20000613.htm.

[50] Chen, H., Tim, F. & Anupam, J. (2003). An Intelligent Broker for Context-Aware Systems. In: Adjunct Proceedings of Ubicomp 2003, Seattle, Washington, USA, October 12–15, 2003. Pp. 183–184.

[51] Chen, H., Finin, T. & Joshi, A. (2004). Semantic Web in the Context Broker Architecture. In: Proceedings of PerCom 2004, Orlando, Florida, USA, March 14–17, 2004. Pp. 277–286.

[52] Wang, X., Dong, J. S., Chin, C. Y., Hettiarachchi, S. R. & Zhang, A. (2004). Semantic Space: An Infrastructure for Smart Spaces. IEEE Pervasive Computing, Vol. 3, Issue 3, July–September, pp. 32–39.

[53] Matinlassi, M. & Niemelä, E. (2003). The Impact of Maintanability on Component-based Software Systems. In: Proceedings of 29th EUROMICRO Conference 2003, September 1–6, 2003. Pp. 25–32.

[54] CyberLink for Java. Development package for UPnP developers. (15.7.2006). URL: http://www.cybergarage.org/net/upnp/java.

[55] Open Services Gateway Initiative (OSGi). (27.7.2006). OSGi Service Platform Release 3 Specification, URL: http://www.osgi.org/resources/spec_download.asp

[56] Tuulari, E. & Ylisaukko-oja, A. (2002). SoapBox: A Platform for Ubiquitous Computing Research and Applications. In: Lecture Notes in Computer Science 2414: Pervasive Computing. Zürich, CH, August 26–28, Mattern, F. & NaghShineh, M. (eds.) Springer-Verlag. Pp. 125–138.

[57] Smarthome Inc. (10.7.2006). X.10 Overview, URL:
http://www.smarthome. com/aboutx10.html.

[58] VTT Technical Research Centre of Finland & Vaisala Oyj (27.6.2006). Internet Weather Station, URL: http://weather.willab.fi/weather.html.en.

[59] OSCAR – An OSGi framework implementation (12.7.2006). Home Page, URL: http://oscar.objectweb.org/.

[60] Keinänen, K. & Pennanen, M. (2005). Secure UPnP and Networked Healthcare. In: ERCIM News, No. 63, October 2005.
URL: http://www.ercim.org/ publication /Ercim_News/enw63/EN63.pdf.

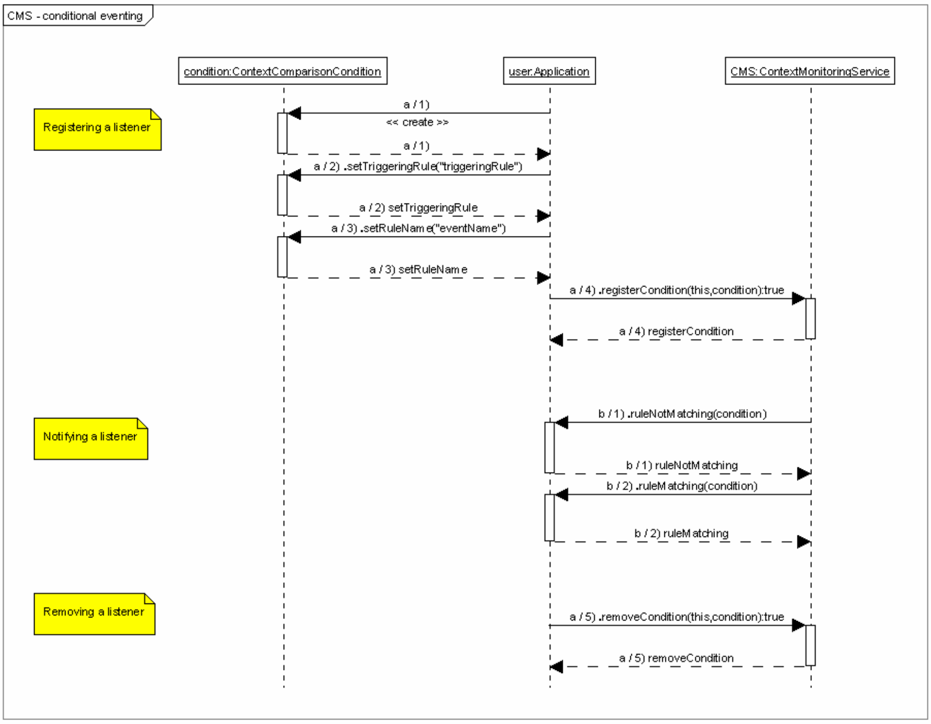
Appendix 1: Example of a model updater device UPnP device description

```
<?xml version="1.0" ?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
   <specVersion>
      <major>1</major>
      <minor>0</minor>
   </specVersion>
   <device>
      <deviceType>urn:schemas-upnp-org:device:cms:1</deviceType>
      <friendlyName>ANSO Central CMS</friendlyName>
      <manufacturer>ANSO</manufacturer>
      <manufacturerURL>http://anso.vtt.fi</manufacturerURL>
      <modelDescription>ANSO model updater
device</modelDescription>
      <modelName>Model</modelName>
      <modelNumber>1.0</modelNumber>
      <modelURL>http://anso.vtt.fi</modelURL>
      <serialNumber>1234567890</serialNumber>
      <UDN>uuid:ANSOCentralContextMonitoringComponent_1</UDN>
      <UPC>123456789012</UPC>
      <iconList>
         <icon>
            <mimetype>image/gif</mimetype>
            <width>48</width>
            <height>32</height>
            <depth>8</depth>
            <url>icon.gif</url>
         </icon>
      </iconList>
      <serviceList>
         <service>
            <serviceType>
               urn:schemas-upnp-org:service:update:1
            </serviceType>
            <serviceId>
               urn:schemas-upnp-org:serviceId:update:1
            </serviceId>
            <SCPDURL>/service/update/description.xml</SCPDURL>
            <controlURL>/service/update/control</controlURL>
            <eventSubURL>/service/update/eventSub</eventSubURL>
         </service>
      </serviceList>
      <presentationURL>http://anso.vtt.fi</presentationURL>
   </device>
</root>
```

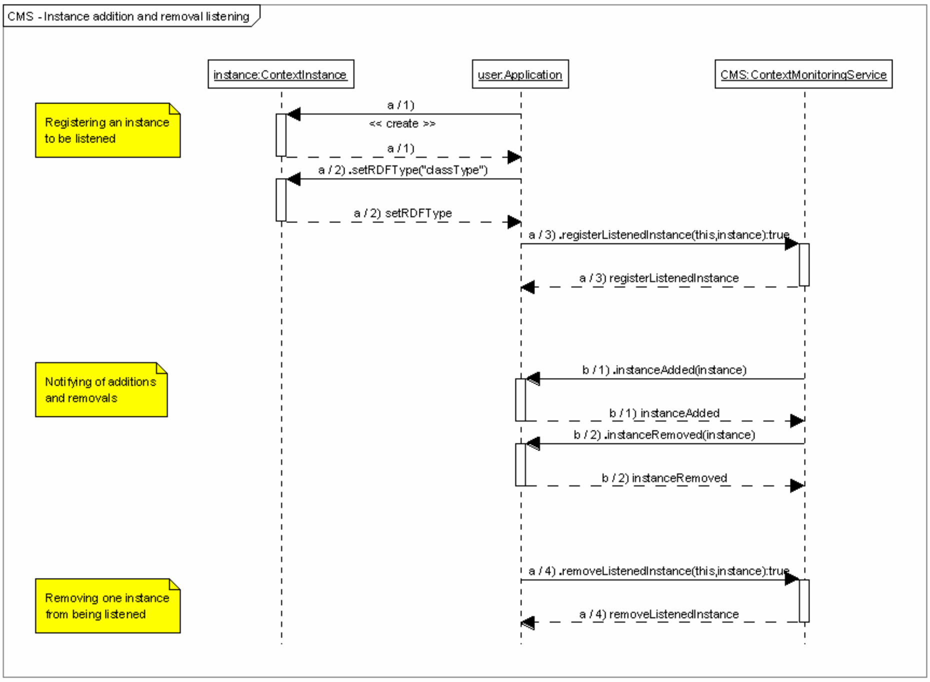
Appendix 2: Example of a model updater device UPnP service description

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0" >
   <specVersion>
      <major>1</major>
      <minor>0</minor>
   </specVersion>
   <actionList>
      <action>
         <name>GetModel</name>
         <argumentList>
            <argument>
               <name>CurrentModel</name>
               <relatedStateVariable>Model</relatedStateVariable>
               <direction>out</direction>
            </argument>
         </argumentList>
      </action>
   </actionList>
   <serviceStateTable>
      <stateVariable sendEvents="yes">
         <name>Statement</name>
         <dataType>string</dataType>
      </stateVariable>
      <stateVariable sendEvents="yes">
         <name>Model</name>
         <dataType>string</dataType>
      </stateVariable>
   </serviceStateTable>
</scpd>
```

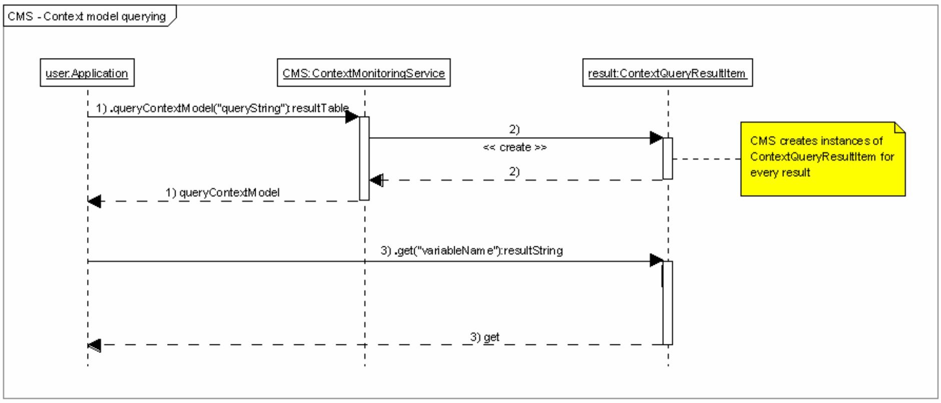
Appendix 3: UML sequence diagram of the conditional rule listener functionality


CMS - conditional eventing
condition:ContextComparisonCondition
user:Application
CMS:ContextMonitoringService
Registering a listener
a / 1)
<< create >>
a / 1)
a / 2) .setTriggeringRule("triggeringRule")
a / 2) setTriggeringRule
a / 3) .setRuleName("eventName")
a / 3) setRuleName
a / 4) .registerCondition(this,condition):true
a / 4) registerCondition
Notifying a listener
b / 1) .ruleNotMatching(condition)
b / 1) ruleNotMatching
b / 2) .ruleMatching(condition)
b / 2) ruleMatching
Removing a listener
a / 5) .removeCondition(this,condition):true
a / 5) removeCondition

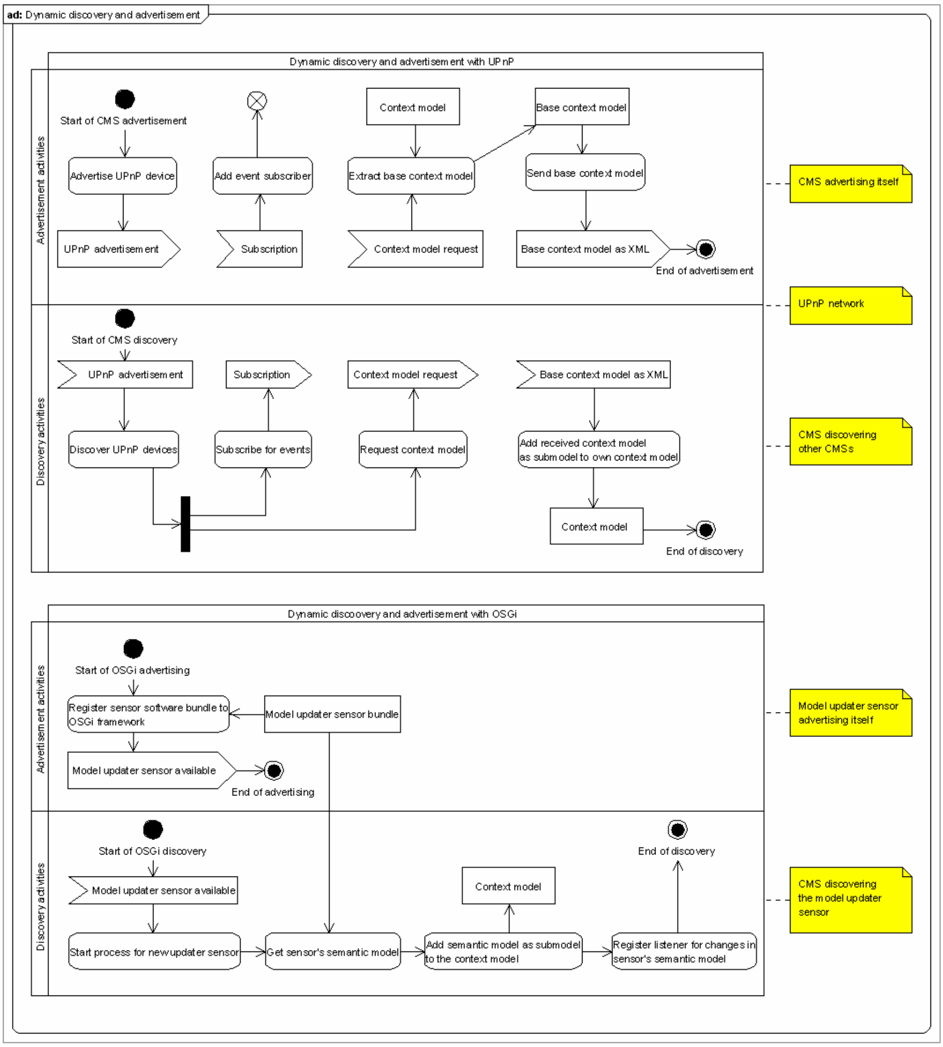
Appendix 4: UML sequence diagram of the instance listener functionality


CMS - Instance addition and removal listening
instance:ContextInstance
user:Application
CMS:ContextMonitoringService
Registering an instance to be listened
a / 1)
<< create >>
a / 1)
a / 2) .setRDFType("classType")
a / 2) setRDFType
a / 3) .registerListenedInstance(this,instance):true
a / 3) registerListenedInstance
Notifying of additions and removals
b / 1) .instanceAdded(instance)
b / 1) instanceAdded
b / 2) .instanceRemoved(instance)
b / 2) instanceRemoved
Removing one instance from being listened
a / 4) .removeListenedInstance(this,instance):true
a / 4) removeListenedInstance

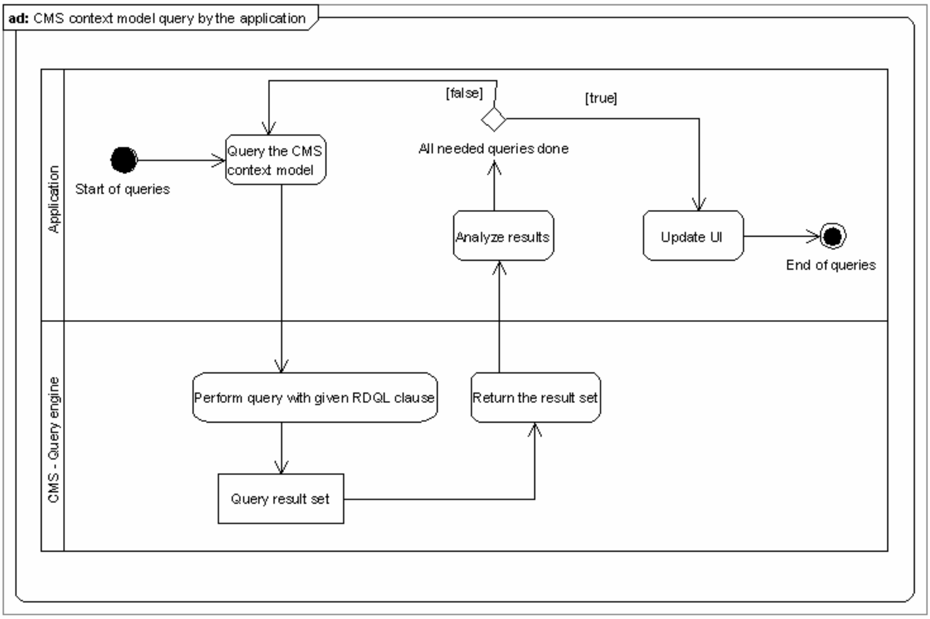
Appendix 5: UML sequence diagram of the context model query utilization


CMS - Context model querying
user:Application
CMS: ContextMonitoringService
result:ContextQueryResultItem
1) .queryContextModel("queryString"):resultTable
2)
<< create >>
2)
1) queryContextModel
CMS creates instances of ContextQueryResultItem for every result
3) .get("variableName e"):resultString
3) get

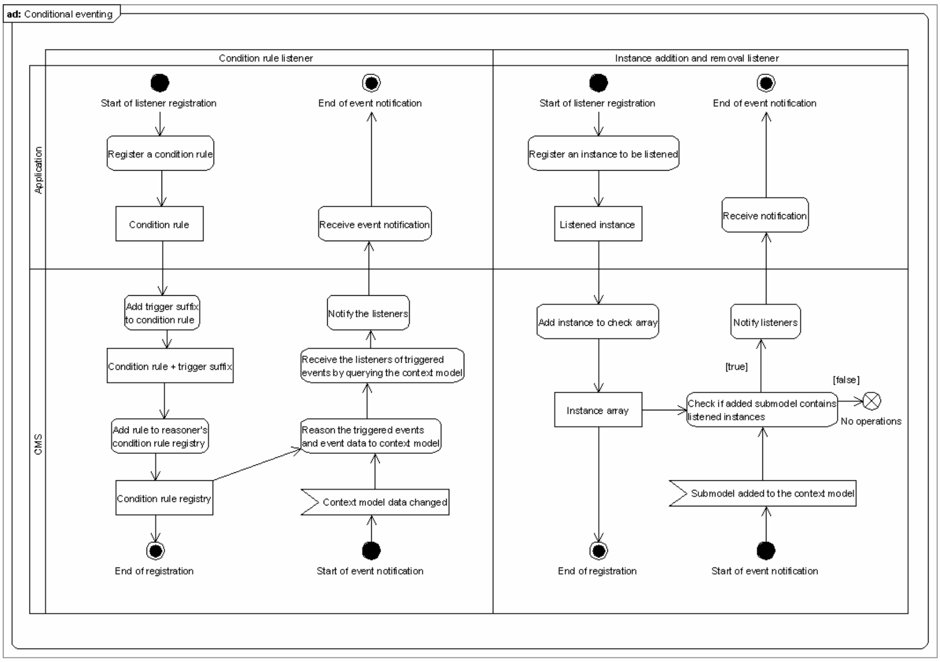
Appendix 6: UML activity diagrams of dynamic discovery and advertisement


ad: Dynamic discovery and advertisement
Dynamic discovery and advertisement with UPnP
Advertisement activities
Start of CMS advertisement
Advertise UPnP device
UPnP advertisement
Add event subscriber
Subscription
Context model
Extract base context model
Context model request
Base context model
Send base context model
Base context model as XML
End of advertisement
Discovery activities
Start of CMS discovery
UPnP advertisement
Discover UPnP devices
Subscription
Subscribe for events
Context model request
Request context model
Base context model as XML
Add received context model as submodel to own context model
Context model
End of discovery
CMS advertising itself
UPnP network
CMS discovering other CMSs
Dynamic discoovery and advertisement with OSGi
Advertisement activities
Start of OSGi advertising
Register sensor software bundle to OSGi framework
Model updater sensor bundle
Model updater sensor available
End of advertising
Discovery activities
Start of OSGi discovery
Model updater sensor available
Start process for new updater sensor
Get sensor's semantic model
Context model
Add semantic model as submodel to the context model
Register listener for changes in sensor's semantic model
End of discovery
Model updater sensor advertising itself
CMS discovering the model updater sensor

Appendix 7: UML activity diagram of context model query



Appendix 8: UML activity diagram of conditional eventing


ad: Conditional eventing
Condition rule listener
Instance addition and removal listener
Application
CMS
Start of listener registration
Register a condition rule
Condition rule
Add trigger suffix to condition rule
Condition rule + trigger suffix
Add rule to reasoner's condition rule registry
Condition rule registry
End of registration
End of event notification
Receive event notification
Notify the listeners
Receive the listeners of triggered events by querying the context model
Reason the triggered events and event data to context model
Context model data changed
Start of event notification
Start of listener registration
Register an instance to be listened
Listened instance
Add instance to check array
Instance array
End of registration
End of event notification
Receive notification
Notify listeners
[true]
[false]
Check if added submodel contains listened instances
No operations
Submodel added to the context model
Start of event notification

Series title, number and report code of publication

VTT Publications 651
VTT-PUBS-651

Author(s)
Laitakari, Juhani

Title

Dynamic context monitoring for adaptive and context-aware applications

Abstract

The field of ubiquitous computing has recently proliferated with a view to providing applications and services that are able to adapt to the rapidly changing situations in dynamic environments and act accordingly. The seamless adaptation to contexts and the alterations to behaviour require the applications to implement mechanisms for acquiring the context information. The required context information is usually diverse and scattered throughout the environment. On account of this, the processing of the context information and its compilation from separate sources is a requirement for the applications to reach adequate context-awareness for successful adaptation. To facilitate the development of context-aware applications, service-oriented architectures for supporting the context-awareness have emerged.

In this work the research problem was to find a solution for dynamic acquisition and representation of distributed context information and its efficient provisioning for ubiquitous applications. As a solution to the research problem this work provides a service architecture called Context Monitoring Service (CMS), which utilizes a dynamically evolving semantic model of context information that the applications can access. A requirement analysis for such architecture was carried out by a literature review in the field of context-awareness. The architecture of the CMS was designed according to the identified requirements and a prototype implementation was created for validation purposes. The prototype implementation successfully validated the architecture's functionality and also opened issues for future research and development in this field.

ISBN
978-951-38-7036-2 (soft back ed.)
978-951-38-7037-9 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN VTT Publications 1235-0621 (soft back ed.) 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	Project number 10886

Date August 2007	Language English, Finnish abstr.	Pages 111 p. + app. 8 p.

Name of project	Commissioned by
Keywords service architecture, context acquisition, Open Services Gateway initiative, Universal Plug and Play	Publisher VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374

Julkaisun sarja, numero ja raporttikoodi

VTT Publications 651
VTT-PUBS-651

Tekijä(t)
Laitakari, Juhani

Nimeke

Dynaaminen kontekstin monitorointipalvelu adaptiivisille ja kontekstitietoisille sovelluksille

Tiivistelmä

Kaikkialla läsnä olevan tietotekniikan aihealue on hiljattain kasvanut räjähdysmäisesti, näkemyksenään tuottaa sovelluksia ja palveluita, jotka pystyvät mukautumaan nopeasti muuttuviin olosuhteisiin ja toimimaan niiden mukaisesti. Saumaton mukautuminen olosuhteisiin ja käyttäytymisen muokkaaminen vaativat sovelluksilta mekanismeja kontekstitiedon keräämiseen ja prosessoimiseen. Vaadittu kontekstitieto on yleensä monimuotoista ja hajallaan ympäristössä, minkä vuoksi sovelluksilta vaaditaan tiedon prosessointia ja kokoamista useista lähteistä, jotta onnistunut mukautuminen saavutetaan. Helpottaakseen kontekstitietoisten sovellusten kehittämistä kehityssuuntana ovat olleet palvelulähtöiset arkkitehtuurit kontekstitietoisuuden tukemiseen.

Tässä työssä tutkimusongelmana on ollut löytää ratkaisu hajautetun kontekstitiedon dynaamiseen keräämiseen ja sen kuvaamiseen sekä tiedon tehokkaaseen välittämiseen mukautuville ja kontekstitietoisille sovelluksille. Tämä työ esittelee ratkaisuna tutkimusongelmaan palveluarkkitehtuurin nimeltään Context Monitoring Service (CMS). CMS hyödyntää dynaamisesti kehittyvää semanttista mallia kontekstitiedosta, joka tarjotaan sovellusten käyttöön palvelun kautta. Tällaisen arkkitehtuurin vaatimusmäärittely suoritettiin laajalla katsauksella kirjallisuuteen kontekstitietoisuuden saralla. CMS-arkkitehtuuri suunniteltiin vaatimusmäärittelyn mukaisesti ja arkkitehtuurista toteutettiin prototyyppi toiminnallisuuden vahvistamista varten. Prototyyppi vahvisti arkkitehtuurin toimivuuden onnistuneesti ja avasi myös uusia tutkimusaiheita tällä aihealueella.

ISBN
978-951-38-7036-2 (nid.)
978-951-38-7037-9 (URL: http://www.vtt.fi/publications/index.jsp)

Avainnimeke ja ISSN VTT Publications 1235-0621 (nid.) 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	Projektinumero 10886

Julkaisuaika Elokuu 2007	Kieli Englanti, suom. tiiv.	Sivuja 111 s. + liitt. 8 s.

Projektin nimi	Toimeksiantaja(t)
Avainsanat service architecture, context acquisition, Open Services Gateway initiative, Universal Plug and Play	Julkaisija VTT PL 1000, 02044 VTT Puh. 020 722 4404 Faksi 020 722 4374

VTT PUBLICATIONS

633 Oedewald, Pia & Reiman, Teemu. Special characteristics of safety critical organizations. Work psychological perspective. 2007. 114 p. + app. 9 p.

634 Tammi, Kari. Active control of radial rotor vibrations. Identification, feedback, feedforward, and repetitive control methods. 2007. 151 p. + app. 5 p.

635 Intelligent Products and Systems. Technology theme – Final report. Ventä, Olli (ed.). 2007. 304 p.

636 Evesti, Antti. Quality-oriented software architecture development. 2007. 79 p.

637 Paananen, Arja. On the interactions and interfacial behaviour of biopolymers. An AFM study. 2007. 107 p. + app. 66 p.

638 Alakomi, Hanna-Leena. Weakening of the Gram-negative bacterial outer membrane. A tool for increasing microbiological safety. 2007. 95 p. + app. 37 p.

639 Kotiluoto, Petri. Adaptive tree multigrids and simplified spherical harmonics approximation in deterministic neutral and charged particle transport. 2007. 106 p. + app. 46 p.

640 Leppänen, Jaakko. Development of a New Monte Carlo Reactor Physics Code. 2007. 228 p. + app. 8 p.

641 Toivari, Mervi. Engineering the pentose phosphate pathway of *Saccharomyces cerevisiae* for production of ethanol and xylitol. 2007. 74 p. + app. 69 p.

642 Lantto, Raija. Protein cross-linking with oxidative enzymes and transglutaminase. Effects in meat protein systems. 2007. 114 p. + app. 49 p.

643 Trends and Indicators for Monitoring the EU Thematic Strategy on Sustainable Development of Urban Environment. Final report summary and recommendations. Häkkinen, Tarja (ed.). 2007. 240 p. + app. 50 p.

644 Saijets, Jan. MOSFET RF Characterization Using Bulk and SOI CMOS Technologies. 2007. 171 p. + app. 4 p.

645 Laitila, Arja. Microbes in the tailoring of barley malt properties. 2007. 107 p. + app. 79 p.

646 Mäkinen, Iiro. To patent or not to patent? An innovation-level investigation of the propensity to patent. 2007. 95 p. + app. 13 p.

647 Mutanen, Teemu. Consumer Data and Privacy in Ubiquitous Computing. 2007. 82 p. + app. 3 p.

648 Vesikari, Erkki. Service life management system of concrete structures in nuclear power plants. 2007. 73 p.

649 Niskanen, Ilkka. An interactive ontology visualization approach for the domain of networked home environments. 2007. 112 p. + app. 19 p.

650 Wessberg, Nina. Teollisuuden häiriöpäästöjen hallinnan kehittämishaasteet. 2007. 195 s. + liitt. 4 s.

651 Laitakari, Juhani. Dynamic context monitoring for adaptive and context-aware applications. 2007. 111 p. + app. 8 p.

Julkaisu on saatavana

VTT
PL 1000
02044 VTT
Puh. 020 722 4404
Faksi 020 722 4374

Publikationen distribueras av

VTT
PB 1000
02044 VTT
Tel. 020 722 4404
Fax 020 722 4374

This publication is available from

VTT
P.O. Box 1000
FI-02044 VTT, Finland
Phone internat. + 358 20 722 4404
Fax + 358 20 722 4374

ISBN 978-951-38-7036-2 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 978-951-38-7037-9 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

VTT PUBLICATIONS 651 Dynamic context monitoring for adaptive and context-aware applications Laitakari