Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Catedra de Calculatoare

TEZĂ DE DOCTORAT

Sistem Multi-Agent de Gestiune a Contextului in Aplicatii de Inteligenta Ambientala

Conducători Științifici:
prof. Olivier Boissier
prof. Adina Florea
as. prof. Gauthier Picard

Autor:
Alexandru Sorici

București, 2015

University POLITEHNICA of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

PHD THESIS

Multi-Agent Based Context Management Middleware in Support of Ambient Intelligence Applications

Scientific Advisers:
prof. Olivier Boissier
prof. Adina Florea
as. prof. Gauthier Picard

Author:
Alexandru Sorici

Bucharest, 2015

Abstract

The complexity and magnitude of Ambient Intelligence scenarios imply that attributes such as modeling expressiveness, flexibility of representation and deployment, as well as ease of configuration and development become central features for context management systems.
However, existing works in the literature seem to explore these development-oriented attributes at a low degree.

Our goal is to create a flexible and well configurable context management middleware, able to respond to different scenarios. To this end, our solution is built on the basis of principles and techniques of the Semantic Web and Multi-Agent Systems.
We use the Semantic Web to provide a new context meta-model, allowing for an expressive and extensible modeling of content, meta-properties (e.g. temporal validity, quality parameters) and dependencies (e.g. integrity constraints).
In addition, we develop a middleware architecture that relies on Multi-Agent Systems and a service component based design. Each agent of the system encapsulates a functional aspect of the context of business processes (acquisition, coordination, distribution, use).
We introduce a new way to structure the deployment of agents depending on the multi-dimensionality aspects of the application's context model. Furthermore, we develop declarative policies governing the adaptation behavior of the agents managing the provisioning of context information.

Simulations of an intelligent university scenario show that appropriate tooling built around our middleware can provide significant advantages in the engineering of context-aware applications.

Contents

Introduction

Ambient Intelligence (AmI) is nowadays a well-known area of research, which has been made a priority of ICT development at European and world-wide level ever since the report of the ISTAG group in 2001 [Ducatel et al., 2001].
Since the initial vision, the AmI domain has matured enough to become the focus of industry-level projects. Recent European directives (Horizon 2020 funding programme) therefore encourage new research that supports the trend towards technological innovation in Ambient Intelligence.

Research into AmI is a full-stack effort (i.e. from network and sensing technologies to novel human-computer interaction interfaces), but one of the foundational (enabling) activity domains is the subject of *context management*. Context-awareness is a key element of an Ambient Intelligence application and the research field concerns itself with the development of support systems that enable an application to service the user with *the right information*, *at the right time* and *in the right way*.

Context management itself requires the coverage of many research aspects, from knowledge representation and reasoning issues to information management system specific concerns such as appropriate system architectures for the multitude and diversity of producer and consumer services that can be encountered in ubiquitous and pervasive applications.
This work focuses on the creation of a context management middleware solution presenting strong features that *alleviate context-aware application development effort*, thereby coming in direct support of the trend towards technological innovation in AmI.

The thesis is a joint-coordination (co-tutelle) effort between University Politehnica of Bucharest and Ecole Nationale Supérieure des Mines of Saint-Etienne. It has been funded by the French Foreign Ministry through the *"Doctorat en co-tutelle"* scholarship program and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

Motivation

Research into context-aware application development actually pre-dates the vision of Ambient Intelligence (e.g. [Schilit et al., 1994]) and, since its introduction as a particular domain of information systems, many advances into modeling and management of contextual information have been put forth by the research community.
As noted in the above discussion, besides the existing academic interest, there is an observable growth of industry engagement into the Ambient Intelligence application development scene. The most notable examples of this trend are in activity areas such as home monitoring and automation, smart city sensing and monitoring infrastructures or Internet-of-Things related scenarios.

However, as we will explore in our review of the state-of-the-art, few of the context management

solutions proposed in the literature have been packaged in a form capable of addressing aspects related to flexible and easy context-aware application development and deployment.
This is why in this work we choose to focus on an approach that aims to elaborate and improve upon existing context modeling and management methods, while expressly maintaining an overarching objective of providing novel mechanisms for flexible design and runtime system configuration and deployment. We thereby wish to obtain a context management middleware solution that is able to address the development requirements for Ambient Intelligence scenarios of various degrees of complexity and scale.

Objectives

The main research question which this thesis addresses is "**What is a suitable design for a Context Management Middleware offering strong *flexibility* and *ease of usage* features in support of application development?**".
Flexibility refers to the ability of the middleware to accommodate large variability in the modeling of information, as well as to offer support for changing the processing of context information according to dynamic application usage needs.
Ease of usage on the other hand refers to the perceived facility with which an application developer can use / control the flexibility options offered by the middleware.
Consequently, the main objectives we define to answer our research question are the following:

1. Development of a *flexible meta-model* for the uniform representation of context information content, meta-properties and constraint dependencies.
2. Development of a *reasoning component*, which exploits the defined context meta-model and uses semantic event processing principles to perform expressive context inference and consistency maintenance operations.
3. Design and implementation of a *Context Management Middleware architecture* based on Semantic Web and Multi-Agent Systems (MAS) principles, allowing for an adaptable context processing cycle, flexibility of configuration and deployment, as well as easier application development.

The fields of Semantic Web and Multi-Agent Systems are chosen as *good engineering fits* for the goal we are targeting. Technologies of the Semantic Web bring the advantages of strong information representation and reasoning capabilities under uniform and standardized means. On the other hand, as we will see in Chapter 5, it makes *engineering* sense to conceive the architecture of a context management system in terms of the notion of agency. Using MAS principles brings advantages over simple distributed service design, as will be pointed out in Chapter 5 as well as in the perspectives for future work.

While exploring and commenting on related work, the set of concrete sub-tasks related to the main goals described above will be presented to the reader.

Thesis Structure

This thesis begins with a *state of the art part*, where we present the research problem we are targeting in more detail and where we identify the most relevant related works that try to address it from modeling and architectural view points.
An analysis of shortcomings in related work leads to our own objectives and the means to achieve them are presented in the chapters describing the *contributions of the thesis*.
Finally, since we focus on aspects related to application engineering, the *implementation and evaluation part* of the thesis contains the chapters that detail the internals of our proposed

middleware, as well as a qualitative and quantitative analysis of its usage for the development of a reference scenario.

The chapter-by-chapter based outline of this work is described in what follows.

In Chapter 1 we define the research and engineering problem we are addressing in a clear way. The exact positioning of the work within the domain of Ambient Intelligence research is explained and the main challenges of managing context information are presented on hand of selected application examples. The description of the scenario used as reference throughout the rest of the thesis ends the chapter.

Chapter 2 marks the beginning of our state-of-the-art exploration. We examine work done in finding requirements and approaches for the most adequate means of representing and reasoning about context information.
An analysis of the related work with respect to identified requirements leads to the presentation of our own objectives concerning context information modeling.

In Chapter 3 we complete the related work overview by analysing different system architecture proposals that enable *context provisioning* within an application. The main provisioning life-cycle steps, as well as complementary functional and non-functional requirements are identified and the benefits and downsides of each proposed context management approach are discussed. The analysis at the end of the chapter reflects about issues of AmI scenario diversity and positioning of reviewed context management systems with respect to the overarching goals defined for this thesis. From the ensuing discussion, our concrete objectives for the design and implementation of our context management middleware are determined.

The analysis of state of the art performed in this first part of the thesis, reveals the important requirements for context management and the challenges that are insufficiently addressed in related work (especially from an application engineering perspective). The contributions of the thesis, presented in the chapters that follow, shows how we employ semantic web technologies and MAS to explicitly tackle the issues unexplored in existing approaches.

Chapter 4 begins the presentation of the contributions of this thesis. We present a formalization of the context meta-model constructs we envision and detail the implementation of the formal representation and reasoning meta-model using semantic web technologies (ontology-based modeling and SPARQL query based inferences). We conclude the chapter with a discussion on how our contributions (e.g. predicates of arbitrary arity, modeling of meta-properties and explicit dependencies, implicit temporal reasoning) provide a more suitable approach to context modeling requirements than those existing in related work.

The architecture and functionality of the context management middleware (CMM) that we propose are presented in Chapter 5. We argue for the use of a multi-agent based system design and a declarative policy-driven mechanism for context management process control/adaptation. We then detail how these principles are applied within the envisioned CMM. Finally, we analyze how policy-driven agents and the context provisioning protocols they carry out address the context management requirements described in Chapter 3.

In Chapter 6 we present the methods by which the deployment of our proposed CMM is tied and structured in total correspondence with the context model of an application domain. We describe the available deployment schemes, the policy-based configuration options and the complex context producer/consumer behavior that these options enable. The chapter concludes with a discussion on how explicit, context model-related deployment structuring concepts that we introduce (*ContextDimension* and *ContextDomain*) allow developers to more easily and dynamically switch between active and inactive context management units currently required by their application.

Chapter 7 covers the implementation of our context management middleware. The semantic web and multi-agent development frameworks used in the implementation are presented, their

choice and concrete usage in the middleware are explained. An aspect of great importance is the focus on a service component based design of key CMM elements, with benefits in terms of runtime life cycle management.

The evaluation of our CMM is detailed in Chapter 8. It is based on an implementation of the reference scenario. The physical environment of the scenario is simulated, but the application that uses it is developed over a real instance of the proposed context middleware. Qualitative assessments of middleware functionality with respect to scenario requirements are discussed and results to quantitative performance analyses are presented. Furthermore, an account of the experience of using our CMM to develop the application from the reference scenario is given, from which aspects of future improvements are collected.

The thesis concludes with a review of important contributions and a pertinent description of short and long term directions for future work, in Chapter 9.

Chapter 1

Problem Definition

Our objective in this chapter is to *put the problem of context management into context.* To understand what the challenges for the main topic of this thesis are, one first has to understand the broader research field of which it is part. Furthermore, it is important to look at the history of this field and comprehend the direction to which it is moving and the current endeavours and factors (both academic and industry-related) that influence its development.
Not least, examples are the best method to provide adequate insights into the issues and complexity of any given subject matter. Consequently, we aim at presenting scenarios which will put the problem of context management into perspective, providing show cases for all the aspects that will be discussed throughout this thesis.

In Section 1.1 we provide an overview of the research field of Ambient Intelligence (AmI), the overarching activity domain of which the problem of context management is part. We present a definition of the field and introduce several well-known AmI scenarios, which are used later on to illustrate the challenges of managing context information.
Section 1.2 goes into more details about particular aspects of context management which will be addressed in the thesis. We explore challenges in deployment of context management solutions, controlling/adapting the functionality of such systems and down to the necessity of having the right information modeling capabilities.
Lastly, Section 1.3 introduces the scenario which will serve as a reference for explaining problem positioning and exemplifying contributions brought all throughout this thesis.

1.1 What is Ambient Intelligence

Ambient Intelligence and its cognates (e.g. Pervasive Computing, Internet-of-Things) provide the vision and the technological setting into which the issues of context management play out. For this reason, it is important to understand the objectives of this research field, to investigate its current trends and directions, as well as to examine the nature of scenarios that have been thus far envisioned by the AmI research community.

1.1.1 Defining the Field

The name *Ambient Intelligence* has been coined in 1998 by a group of people from Palo Alto Ventures and Philips (Eli Zelkha, Brian Epstein and Simon Birrell) who were commissioned by the Philips board of management to provide a series of internal presentations on scenarios that would present the vision of a world in 2020 where *user-friendly* devices support *ubiquitous*

information, communication and entertainment.
In the over 15 years that have since then passed, Ambient Intelligence has received a lot of backing from academic, governmental and private enterprise sources, making it a noteworthy and active research field. At European level, for example, advice from the Information Society and Technology Advisory Group (ISTAG) was used to include research into Ambient Intelligence enabling technologies into the pillars of the FP6[1] and FP7[2] funding processes.

Characterization From a research perspective, the field of Ambient Intelligence has been characterized in several ways. [Cook et al., 2009a] provide a summarization of various definitions that have been given to the field of AmI in the literature. The authors then go on to extract the following *highlight* features that are expected to be found in AmI technologies: *sensitive, responsive, adaptive, transparent, ubiquitous* and *intelligent*.
The first three features (sensitive, responsive and adaptive) highlight the fact that *context-awareness* is an integral and key part of AmI-related research. Ambient Intelligence is designed to proactively support people in their daily lives. Therefore, getting a sense of the *situation* of a user or a group of users and responding to it in a timely and adequate manner become important objectives.
The features of transparency and ubiquitousness are aligned with the vision introduced by Mark Weiser [Weiser, 1991], stating that

> "*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it*".

Lastly, the feature of intelligence hints towards the fact that, besides incorporating aspects of pervasive/ubiquitous computing, the proactiveness demanded from AmI applications requires the usage of artificial intelligence techniques to address the specific needs of each user. Therefore, it is nowadays common to see advances in fields such as machine learning, agent-based software engineering, computer vision or natural language processing being incorporated into the development of AmI applications.

Subdomains The amount of support received by the Ambient Intelligence vision along the years has led to it becoming a full-stack research and innovation domain. This means that AmI spurred advances in technology areas ranging from sensors and wireless communication to new means of human computer interaction. In what follows we intend to briefly present the different sub-fields of Ambient Intelligence with the purpose of showing the breadth of research efforts conducted in the past 15 years.
Hong et al. perform an ample review of the works related to context-aware system development that were published in journals between the years 2000 and 2007. Their objective was to obtain a classification framework of the different kinds of work into context-awareness and track the evolution of the research focus (obtained from article review) across the given timespan [Hong et al., 2009].

Their results suggest a partitioning of research into Ambient Intelligence in four categories: network infrastructure, middleware layer, application and user interface layer. This categorization is reflected also in Figure 1.1, where we show the positioning of context management as a subdomain of AmI.
Exploring the analysis from [Hong et al., 2009] we note the following:

- Network Infrastructure: includes research into sensing equipment, internet-based protocols (e.g. design of the session initiation protocol - SIP, mobile IPv6), self configuring mobile ad hoc networks (MANET), handoff management mechanisms (e.g. for seamless transfer of ongoing call or data sessions) or network requirements and implementation

[1] http://cordis.europa.eu/fp6/
[2] http://cordis.europa.eu/fp7/

(e.g. frameworks for integration of user services out to mobile devices and heterogeneous network infrastructures)

- Middleware Layer: covers the research effort done in the attempt to regroup necessary and frequently occurring AmI functionality aspects into frameworks readily usable by upper application levels. Typically this involves functionality aspects such as communication, service discovery or, as we show later, *context management*. Hong et al. identify several subtypes of middleware development efforts based on the predominantly used technology: agent-based, reflective, metadata based, tuple space based, adaptive and objective based, OSGi[1] based

- Application and Service Layer: includes the research done into defining and implementing the actual envisioned AmI application scenarios and service functionality. Based on the nature of scenarios and objectives, the authors of [Hong et al., 2009] further distinguish a set of AmI application domains: smart environments (e.g. home, workspace, hospital, classroom), tourist applications, adaptive information and communication systems, mobile commerce and smart web services.

- Human-Computer Interfaces: comprises the developments of new user interfaces (e.g. mobile devices, gesture and voice based controls, elastic displays, intelligent lighting) and the usability studies that attempt to evaluate the suitability and usefulness of these interfaces. While the volume of research in this sub-field of AmI ranks behind the other three ones, it is certainly the one that attracts attention the quickest, when new achievements are presented.

This brief overview of research areas within the Ambient Intelligence field shows the extent of the problem space. As noted in the list above, research into context management falls in the category of middleware development. In Section 1.2 we will show that this sub-field itself presents numerous challenges and corresponding research aspects.

Trends and Directions In previous paragraphs we have tried to define the research sphere of the Ambient Intelligence field. However, recent developments show that, as was the original vision, the field is beginning to gain traction in the industry. Enterprises are starting to embrace scenarios and ideas from the AmI domain, most notably in activity areas such as home monitoring and automation, smart city sensing and monitoring infrastructures. There is a growing number of start-ups that are active in the mentioned areas (e.g. Ninja Sphere[2], Nest[3], SmartThings[4]) and increasingly more cities are offering their support for installing prototype smart environment infrastructures.
The possibilities for AmI application development increase even further given the emerging industry enterprises that offer entire development platforms for creating application and business logic in the Internet-of-Things (IoT) and Machine-to-Machine (M2M) domains (e.g. Xively[5], ThingWorx[6]). Such initiatives open up a trend that leads towards systems which promote anonymous social experiences and focus on models of group activity rather than just individual ones. It raises an AmI that is centered on enhancing human interaction apart from intelligently supporting individual needs and preferences.
This trend that focuses on using the vision of Ambient Intelligence as a promoter of *technological innovation* is even reflected in the continuation of the European Commission's *Framework Programmes* (FP). The "Horizon 2020" (or FP8) funding programme which encompasses the 2014 - 2020 time window focuses on innovation and faster delivery of solutions to end clients.

[1] http://www.osgi.org/
[2] http://ninjablocks.com/
[3] https://nest.com/
[4] http://www.smartthings.com/
[5] https://xively.com/
[6] http://www.thingworx.com/platform/#how-it-works

This is in contrast with the previous initiatives (FP6 and FP7) which still concentrated on technological research.
The above observations have an obvious impact on the way in which developments of new technology in any of the AmI-related subfields (including context management) will take shape. As we will see throughout this thesis, providing the means for *easy development* and *deployment* of ambient intelligence applications becomes an important object of study, because it is directly related to the speed with which the resulting research work can be picked up by the industry.

1.1.2 Scenarios

To see the extent of the Ambient Intelligence vision, in what follows we will comment on a selection of scenarios from the literature. These descriptions are meant to give the reader a sense of the breadth of some of the proposals in this research field but, more importantly, they are intended as means to further identify an important list of *challenges* for the context management side of things.

The Maria Scenario In a report dating back from 2001, the ISTAG advisory board developed a set of scenarios for the vision of *Ambient Intelligence in 2010* [Ducatel et al., 2001]. One of these is the "Maria" scenario.

> *Maria is a sales representative who is travelling to a foreign country to perform a sales pitch. The scenario focuses on the extent of her mobility. It also presents the seamless interaction between her smart watch (in today's terminology) which holds information about digital identification keys and Maria's personal preferences and which allows Maria to interact with many different services either from home or from the new country she is visiting. For example, she is allowed to walk quickly through customs because the visa for this trips was self-arranged and the ambient intelligence infrastructure at the airport of the foreign country clears her on the fly. A car has been already reserved for her at the exit of the airport. The car opens automatically because it recognizes Maria by communicating with the smart watch on her hand and verifying the stored digital keys. While she is driving, the car's on board computer interacts with the smart watch again. Maria's daughter, who is at home, has detected that her mother is in a place that supports direct voice contact and she wants to talk to her mother. The request is dispatched to the smart watch which forwards it to the on-board computer.*
> *At the hotel, Maria's personal preferences are automatically detected by the AmI service of the room in which she is staying, such that the room adjusts for Maria to feel as home as possible.*
> *When Maria gives the sales pitch the next morning, her smart watch automatically gives the computer connected to the projector access to her slides, but only for the duration of the actual presentation. When she starts her presentation, the smart watch sets her availability status to busy and will hold off all non-urgent calls to her phone.*

The most important aspects of this scenario regard the multitude of *just-in-time*, *short-lived*, *context-aware* servicing that Maria's smart watch is able to perform. Important questions regarding the architecture of the infrastructure supporting this services and the way in which they can be automatically discovered by Maria's smart watch are raised. We will shine some more light on these issues from a context management perspective in Section 1.2.2.

The Mobile University Scenario A scenario similar to the previous one is the "Mobile University" proposal that was part of the Daidalos project[1] [Aguiar et al., 2007]. In this case though, the ambient intelligence services are restricted to usages and activities encountered in a university environment.

> *The main vision of the scenario is that of helping students studying abroad to have access to their personal set of services and to dynamically discover local services and devices. Key functionality of the proposed scenarios involves things like organizing the daily life at the university (e.g. friend contacts, appointments, reservations, classes, projects), locating people and devices, moving sessions and content between devices or working and playing while on and off campus.*
> *An example episode involves Dani who arrives at the university to join his friends in a project group and work on an assignment. When he arrives at the desk where his friends are present, the project group meeting is automatically marked as active in his calendar. The project group decides to use local computers to work on the assignment, so all information prepared in advance by Dani is transferred from his mobile device to the local computer. During the work session, Dani is marked as busy.*
> *When the group decides to take a lunch break, the mobile terminals allow them to search for an available table at the cafeteria and reserve it for the whole group.*

As in the case of "Maria", the "Mobile University" scenario relies on individual contextual interactions with services and devices discovered on the fly, in different environments of a university (e.g. on the hallway, in an office, in the cafeteria). While the scale of the mobility aspects is less ample than that of the "Maria" scenario, it does raise many of the same context management challenges.

Care for the Elderly In a change of perspective, the next scenario is proposed by Olaru [Olaru, 2011] and is part of the Ambient Assisted Living[2] vision. The scenario talks about the personal care services offered to an individual senior person.

> *A senior person walks on the street towards her house. In the pocket she has a mobile phone with an AmI software agent installed which communicates with a multipurpose sensor that monitors vital signs. The person lives in a small basement apartment. She climbs down the stairs, misses one of the last steps and falls. She loses consciousness for a few moments. By means of the vital signs sensor, the AmI agent determines that the situation is not life threatening and that no major injury has occurred. There is no need for an ambulance, but care may still be needed, so the personal medical assistant for the senior person should be called immediately.*

As opposed to previous scenarios, in this case the contextual interactions of the considered services (vital sign sensing and processing, contacting care facilitators) all have a single purpose, that of assisting the elderly person. However, the main challenges come exactly from the "intelligence" and decision making part of the scenario. Especially in the cases involving sensors that are worn by an individual, uncertainty of the readings (due to false positives) or incompleteness of the information (due to the elder forgetting to put on one of the vital sign sensors or due to a sensor failure) have to be expected. This means that the AmI agent has to act based on imperfect context information which becomes a reasoning challenge in itself.

Sensor Control in Body Area Networks The following analysis is not performed for a particular scenario, but rather for the issues encountered in a category of scenarios that involve utilization of sensors that are worn by the user, must rely only on battery supplied energy and

[1] http://www.ist-daidalos.org/
[2] http://www.aal-europe.eu/

which may be shared between multiple end applications. Example scenarios and experiments that refer to such settings are reported in works such as [Riboni and Bettini, 2009; Kang et al., 2008, 2010].
The main challenge in such environments is to enable multiple context-aware applications that require continuous context monitoring to simultaneously run and share highly scarce and dynamic resources. Therefore, it becomes important to find means to search and select the most appropriate sensors for a given task and to use context information usage patterns to control when and which body worn sensors need to actually be active.

Scenario	Focus Points
Maria	Variety of context services, shifting context interest
Care for the Elderly	Information uncertainty and incompleteness, reasoning capabilities
Mobile University	Variety of context services, shifting context interest
Body Area Networks	Dynamic scheduling of context services, task based sensor selection

Table 1.1: Synthesis of main issues and focus points for the presented scenarios.

What is clearly observable from the brief list of Ambient Intelligence scenarios presented above is that the envisioned applications (and their contextual management requirements, implicitly) cover very wide scale and complexity ranges. Furthermore, the focus point of an application is also highly variable, as highlighted in Table 1.1.
From an application development point of view, this again shows the need to focus on the *flexibility* (in terms of modeling and architectural design) of the technical solutions that enable context-aware programming. In what follows, we introduce context management as a research field and present the challenges of this domain which stem from the above mentioned scenarios and which we will try to address in this thesis.

1.2 What is Context Management

As shown in the list of Ambient Intelligence subdomains, Context Management is an integral and important objective of study, constituting a foundational layer for modern AmI application development. In order to understand the issues regarding the management of information in context-aware applications, we must first provide a definition of the term *context*. We will then use the AmI scenarios presented earlier to give the reader a better sense of the problems faced by solutions attempting to tackle the management of context information.

1.2.1 What is Context

The term *context-aware computing* actually predates the notion of Ambient Intelligence and was first introduced for the area of computing and information systems by Shilit et al. in 1994 [Schilit et al., 1994]. Since then, a large number of definitions for the terms *context* and *context-awareness* have been proposed in the literature.
Zimmermann et al. make the observation that most of the existing definitions of these terms can be partitioned into definition by synonyms and definition by example. Works such as [Brown, 1995; Hull et al., 1997] equate the meaning of context to that of the *application's environment* or the *situation* of the user.
Then again, other authors [Gross and Specht, 2001; Chen et al., 2003] use enumerations of elements such as location, time, identity, temperature, noise or beliefs and intentions of a human to define the notion of context.

Attempting to address the limitations of early context definitions, Dey provided a more general and comprehensive meaning of the notion, which has become the most widely accepted one within the research community. The definition goes [Dey, 2001]:

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

As we will also see in an analysis in chapter 4, the notion of *context* provided by Dey includes *any kind* of information that is relevant to the interaction between a user and an application, as well as to the interaction in between applications or in between users themselves.

It is arguable that, while comprehensive, Dey's context definition tends to be overly general. In the attempt to constrain the universality of the notion conveyed by Dey, Zimmermann et al. [Zimmermann et al., 2007] propose a formalization of the elements describing context information into five categories: *individuality*, *activity*, *location*, *time* and *relations* (cf. Section 4.1.3 for more details). The authors furthermore extend this general terms definition with an operational side which characterizes the *use* of context and its *dynamic behavior*. Specifically, based on review of work from the literature, [Zimmermann et al., 2007] determines that the *activity* predominantly influences the relevancy of context elements in specific situations, while the *location* and *time* elements mainly drive the creation of relations between entities (as a consequence of shared context) and enable context information exchange among those entities. This latter operational aspect of context is highly relevant from an application development point of view because it raises the question of *managing* context information: how is a context model created, how and when is contextual information acquired, processed or disseminated only to the interested parties at the optimal time? What are the means for controlling the flow of context information based on the operational characteristics mentioned above?
A graphical representation of the context management breakdown and its placement as an object of study within the Ambient Intelligence research field can be seen in Figure 1.1.

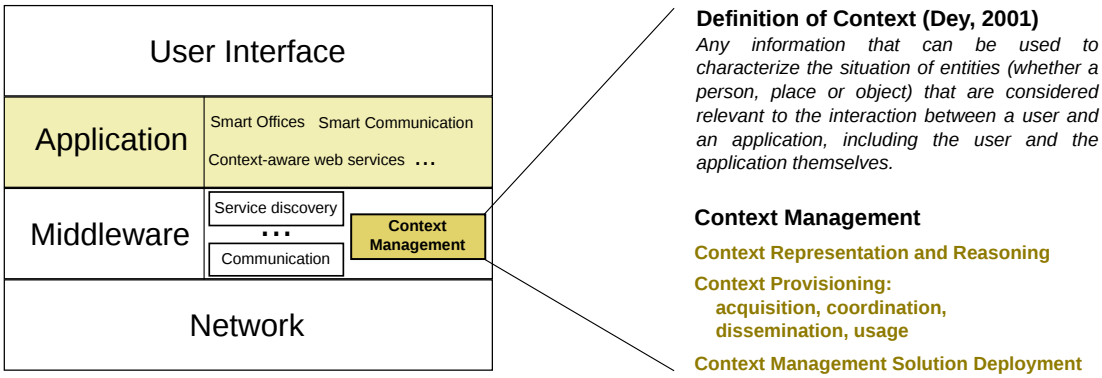

Figure 1.1: Aspects of Context Management and their place within the Ambient Intelligence research field.

In essence, these represent the challenges posed to any information management system, but applied to the particular characteristics of context information. We provide an introduction to these issues in the subsections that follow.

1.2.2 Challenges in Deploying Context-Aware Applications

We propose to start our inquiry into the challenges of managing context information in a top-down manner. Consequently, the first problem we identify is that of deploying a context-aware

application itself.
Deployment of a context management system refers to the process of configuration, installation and runtime administration of the system. In essence, it deals with the concrete means in which the context management support infrastructure (both in terms of physical machines as well as software services) is setup within an AmI application.

The biggest challenge in this regard comes from one of the desired characteristics of Ambient Intelligence, namely *ubiquitousness*. The "Maria" and "Mobile University" scenarios (but more notably the "Maria" one) make this issue very visible. In Maria's case, her smart watch is expected to have contextual interactions with different services (e.g. the customs office at the airport, the on-board car computer while driving to the hotel, the hotel room) on the fly and for a short, non-repeating time interval (we can safely assume that Maria does not travel to the foreign country every week). Several questions present themselves in this case:

- Is Maria's smart watch running a single AmI application that interacts with all these services?
    - If yes, how is the support for all the totally unrelated contextual interactions foreseen (understand programmed in advance)? Is it realistic to expect this?
    - If not, are the different applications/services already present on Maria's smart watch or are they deployed just-in-time?
- In accordance with Zimmermann et al.'s observation that activity determines the relevancy of context elements, does the service managing Maria's contextual interactions with the hotel room need to persist on her smart watch, after she checks out of the hotel?
- From a development point of view, is it easily expectable that the team implementing the AmI infrastructure in the car or the hotel uses the same context information model and communication standards as the developers of Maria's smart watch?

Considering the trends and directions discussion from Section 1.1.1, an adequate response to these questions is required in order to identify the most suitable approach for deploying context-aware applications.
Note furthermore that, in contrast to the "Maria" or "Mobile University" scenario, the "Care for the Elderly" one has a much less stringent requirement for ubiquitousness and a less diverse set of services. The main focus of the AmI application in the scenario of [Olaru, 2011] is the well-being and care of a senior person and all context interactions revolve around this objective. From the perspective of wanting to develop a context management system capable of addressing as many AmI scenarios as possible, the concerns listed above suggest the need for a highly flexible approach. As we will see in chapter 6, our solution to this issue is based on the idea of configuration and deployment flexibility. We propose a *modular* design based on *control units* and their dynamic life cycle management as the key to answering the deployment specific questions discussed here.

1.2.3 Challenges in Controlling Context-Aware Applications

Assuming that a context-aware application is deployed, the next challenge lies in determining how to control its *information provisioning process*.
The notion of *context provisioning* is discussed in detail in Section 3.1, but essentially refers to the entire set of mechanisms and interactions used by a context management system to let AmI applications built on top of them access the desired and needed context information when and how they require it.

The need for control and adaptation of this process is immediately observable from Zimmermann et al.'s operational views on context. The attention and the focus on specific context information varies as aspects of current activity, time or location change.

In our presented scenario list, this issue becomes apparent in the discussion on judicious usage of body worn sensors, particularly because of the energy consumption constraints of such sensors. In the "Care for the elderly" scenario, we notice that the AmI agent needs to have access to the most complete and accurate information as possible in order to evaluate the nature of the senior person's injury after her fall. Thus, in a critical situation all vital sign sensors should be employed in assessing the condition. However, it might well be the case that during normal and routine activities of the elder, a much smaller amount of sensors is required to track his vitals, possibly at a lower update rate (since no real danger is anticipated).

Taking again the perspective of support for easy application development discussed in Section 1.1.1, the question becomes centered on the concrete means by which context provisioning adaptation and control is included in the application. Is the control logic embedded within the functionality of context management solution or can the application layer on top influence/-modify it in any way? In the latter case, is the application layer required to assume full control of the adaptation decisions or does it act by altering the value of some modifiable parameters that govern the adaptation logic?
Furthermore, if parameters exist that influence the actions of different provisioning subsystems (e.g. acquisition, coordination, dissemination of information) what is an adequate design for the architecture of a context management system where these subsystems can be controlled both independently and in connection to one another?
Our proposed solution for these issues is discussed at length in Chapter 5.

1.2.4 Challenges in Modeling Context Information

If we can consider deployment and provisioning control of a context management system as established, a final question we can ask is related to the *form* of information provided to the end AmI application, as well as to the type of reasoning that the context management system can apply to the information it handles on behalf of the application. Collectively, these aspects are referred to as a *model* for context information.

The challenges for this concern can be best observed in the "Care for the elderly" scenario. As we already mentioned when discussing the story, the AmI agent works in conditions subject to uncertainty and incompleteness, given that the agent must expect the sensors worn by the elder to be inaccurate or faulty. Furthermore, sensor inaccuracy might even lead to detection of contradictory/ambiguous situations.
The question then becomes whether the chosen *information representation* formalism can adequately support the context management system in dealing with such issues. Can the representation capture the value of confidence in sensed information? Can it represent the fact that one context element constrains or depends on the value of another?
Additionally, returning again to aspects of application development, the issue of modeling flexibility arises. In the case of the "Care for the elderly" scenario is the AmI agent forced to retrieve only key-value like information from each vital sign sensor or does the underlying context management system allow a richer representation where the model designer is, for example, able to aggregate several inter-dependent context elements (e.g. x,y,z acceleration on an accelerometer, heart rate and blood pressure from a heart monitor) into a single statement?
Furthermore, from the perspective of reasoning about the collected context information (e.g. to infer the degree of the injury of the senior person after her fall) what is the most suitable formalism for performing inference which combines adequate deductive capability with ease of development and understandability of the deduction process (i.e. be able to explain why a given result was derived)? Can the particular reasoning method also perform time-related operations (e.g. determine for how long the blood pressure of the senior was below a given threshold)?

In Chapter 2 we explore this set of desirable context modeling characteristics in further detail and then analyze the multitude of different approaches that have been proposed for the problem

of context modeling. Given the results of that analysis and our explained goal of ease of development, we present our own proposal for modeling context information based on principles of flexibility and expressiveness in Chapter 4.

1.3 Reference Scenario

In previous sections we defined the general field of Ambient Intelligence and its Context Management subdomain, which is the topic of this thesis. We then provided an overview of the challenges that context management systems *are* and *will be* expected to handle.
To put these things into a clearer perspective and be able to exemplify the conceptual solutions we will bring to each challenge throughout this work, in the following we introduce a simple, yet complete, scenario that showcases all the issues discussed above. Subsequent chapters of this thesis will use it as a reference.
The scenario falls into a category that is well known in the AmI literature, namely the interactions of students and professors within the premises of a smart university campus.

> *Alice is a student of Computer Science (CS) at University Politehnica of Bucharest. It is currently 11:50 and Alice is attending a lecture in the Ambient Intelligence Laboratory (AmI Lab) of the CS Building.*
>
> *The AmI laboratory is a smart multi-functional room of the university in which different activities can take place. It is used by both faculty members and students as a lecture room, research facility and meeting room. The laboratory is equipped with sensors for detection of various environmental and user related information. Temperature and luminosity sensors collect data which is used by air conditioning and slide projector units throughout the day. Each desk in the room has sensors that can detect presence of bluetooth enabled devices, noise level readings and body postures (via Kinect cameras) of users in their vicinity.*
> *However, to save energy and bandwidth the updates of these sensors are tightly controlled by a management server which knows when and how to use the data coming from each sensor type. Policies set up by the faculty administration as well as the dynamic usage of context information determine allow the server to manage both sensors and its internal reasoning processes.*
>
> *Students and faculty staff have installed a context-aware application on their smartphones to help them on the university premises. The application contains both personalized modules (i.e. preferences and deductions that are specific to each user), as well as modules that allow it to interact with all the individual smart environments (e.g. the CS Building hallway, the AmI laboratory, different offices, the university outdoor campus).*
>
> *Alice's lecture finishes early, but she remains in the AmI laboratory since she wants to meet with two friends, Bob and Cecile, to talk about their Ambient Intelligence project. Normally, the smart application on Alice's phone knows that she is attending a lecture from 10:00 to 12:00 and is therefore busy. In this case all incoming and non-urgent calls will be directed to voice mail. However, the application interacts with the AmI-Lab management server and determines that Alice is alone in the room, meaning that the lecture situation is no longer valid, even though it is before 12:00 o'clock.*
> *Bob calls Alice to find out where to meet and the application accepts the call since it deemed Alice as not being busy. While Alice waits for her friends to arrive, the only sensors which are maintained active by the AmI-Lab server are the presence sensors and the temperature sensors since it is mid May and there is a person in the room (therefore, custom AC settings may be required).*

When Bob and Cecile arrive, the three friends sit down at a desk and begin talking about their project. Their applications detect that there are more than two people at the same desk, such that a collective activity might be in place. The smartphones subscribe to the AmI-Lab server to find out if their users are in an ad hoc discussion. The server contains rules that help derive the type of activity in the smart room. One such rule detects ad-hoc discussions and mentions that when the noise level near a desk is higher than 60 dB and two or more people are perceived as sitting near the same desk for over two minutes, then that corner of the room is hosting an ad-hoc discussion. However, the sensors required for detecting this situation (microphones and Kinect cameras which can detect the posture of a person) are not active. The system therefore actively searches for providers of the mentioned context information, indicating it requires updates every 20 seconds.
Alice, Bob and Cecile keep talking and the application on their smartphones is soon informed they are in an ad-hoc discussion, so they are deemed as busy. Alice's application is configured to share this information at room level so that friends of hers in the CS building may access it. During the discussion, Bob uses the projector to show some slides. The projector unit subscribes for information on luminosity levels in the room, so the AmI-Lab server enables the updates from light sensors. When Bob closes the projector, the AmI-Lab management policies tell it that if no request for luminosity information arrive for more than 1 minute, light level updates can be deactivated.
Meanwhile, Dan, who is a friend of Alice's and a PhD student at the university is working in the EF301 office. He knows that Alice is in the building and wants to be notified when she is free so he can call her to meet for lunch. However, he does not know where Alice is exactly. He therefore uses his application to place a subscription for Alice's availability status at CS Building level. The query will thus get routed to all rooms within the building.

When the meeting stops, Alice, Bob and Cecile leave the AmI-Lab and move into the hallway. After 5 minutes, a closing policy of the AmI-Lab server is triggered, stating that if no query for the type of hosted activity is received for this amount of time, the sensors providing information required to infer this situation can stop sending their data. Meanwhile, Alice's smartphone application infers that her user is no longer in a meeting and therefore not busy. Dan is thus notified that Alice is free and he calls her to see if they can meet for lunch.

The presented scenario highlights many of the challenges discussed previously.
In terms of deployment, the scenario features a distributed deployment, with management modules installed in the smart environments of the university and mobile client modules installed on user smartphones. Furthermore, the smartphone application is required to interact with multiple modules, either in sequence or at the same time (e.g. when solving the contradictory information of Alice being free or busy while waiting for her friends).
The scenario shows requirements for adaptation and control of context provisioning, since the sensors updates and inferences carried out in the Ambient Intelligence Laboratory are directly dependent on how those pieces of context information are used.
Lastly, from a representation and reasoning point of view, the scenario requires some complex modeling capabilities of the situations in the AmI-Lab. Temporal validity reasoning and aggregation of information is required to detect the ad hoc discussion situation. Furthermore, the ability to solve contradictory situations (such as inferring that a user is busy and free at the same time, as in the case of Alice) is demanded.
As we discussed in Section 1.1.1, on top of these functionality requirements, we add the overarching goal of developing solutions to these requirements that offer extensive support for *flexibility of modeling*, *easy development* and *flexible deployment*. These non-functional requirements have

the express purpose of alleviating the time and effort spent developing context-aware applications.

Chapter 2

A State of the Art in Context Modeling

Now that we have seen the multiple dimensions of research objectives existing within the field of context management, in this chapter we begin the examination of work related to our own that has addressed all or parts of the previously defined objectives.

While defining our research problem, we provided a top-down incursion into the different challenges with which a context management system is faced. This served the purpose of showing the depth of the context management problem and that there exists a certain inclusion relation (from an architectural point of view) between its aspects (system deployment, information provisioning, representation and reasoning). We will approach the presentation of the state of the art and contribution chapters in the opposite direction, from foundational aspects to engineering related ones, allowing the reader to gain a gradually broader picture of the subject matter.

In this first state of the art chapter we explore the aspect of context modeling, specifically the different means to represent and reason about context information. At the end of each section we provide a summary presenting the section's main take-away points and the issues that we consider remain insufficiently addressed.
Section 2.1 looks at commonly desired attributes of context representation and at how different representation methods are able to address such requirements. During our analysis, we place greater accent on ontology-based approaches which have seen the greatest uptake in recent years.
In Section 2.2 we examine methods to perform reasoning about context information. We look at the different aspects of reasoning (e.g. making deductions, maintaining consistency, ensuring integrity constraints) and how existing work approaches them.
Finally, Section 2.3 presents the detailed objectives of our own approach with respect to context modeling, in light of issues that are insufficiently addressed in the reviewed state-of-the-art, but are required by the reference scenario given in Section 1.3.

2.1 Representing Context Information

We start the review on context information representation by listing general consensus attributes which should ideally characterize a representation method. We then provide an overview of various context modeling efforts that have been proposed throughout the literature. Further on, we look more closely at work that relied on two types of approaches that prove to be more

expressive: ontology-based representation and meta-model oriented approaches. A summary presenting the main take-away points concludes the section.

2.1.1 Context Representation Requirements

It is clear from the example scenarios presented in the introduction, that the complexity for context-aware applications stands at a high level. Moreover, the industry uptake of development of Ambient Intelligence and Pervasive Computing applications, which is expected in the current decade, means that the different layers of engineering (e.g. sensor communication protocols, networking, context management, user interfacing) of such applications *must* be supported by their proper development tools. This is an important requirement, as it facilitates the design and ensures the maintainability and evolvability of the application.
Context Management itself is a complex engineering layer for Ambient Intelligence applications and within it special attention is given to methods for representing context information as the foundation on which further processing depends. In the attempt to prepare for an as large as possible list of possible uses of context information, the ambient intelligence community has put an emphasis on well-designed context models and has established a set of requirements which characterize the ideal context representation method [Bettini et al., 2010; Bolchini et al., 2007a]. We present this list of attributes in what follows.

A first concern is **model flexibility**, which refers to the ability of a representation model to be suitable to different contexts. That is, this attribute relates to model generality: is it more suited to specific application domains or is it general purpose? Is it possible to capture any and all kind of context information and if so, how easy is it. This requirement is commonly set for data modeling in the information system domain [Hirschheim et al., 1995], because it means that the model offers the users the ability to reach consensus regarding conceptual schemas that capture the sense of the domain of discourse and how these schemas may be adapted further down the life time of the application.

Heterogeneity examines how easy a context model can deal with a large variety of context information sources that differ in their update rate (e.g. a video camera stream versus a temperature sensor that only sends updates when the value changes) and their semantic level (e.g. GPS raw coordinates versus a logical localization on a street). Information can be static (e.g. taken from a database), sensed, or provided by the user. The interpretation of such data is usually application specific. Does the chosen representation allow a differentiation of further processing based on such criteria? This attribute relates more to the operational mechanisms (e.g. reasoning, provisioning) that are built around the context model and is a key issue in AmI and smart environment applications where heterogeneity of the sensing apparatus is inherent [Augusto et al., 2010].

Context relationship and dependency specification refers to the ability of a representation method to support explicit definition of different semantic relations between context information instances. This includes, for example, the ability to specify information integrity constraints (e.g. value restrictions, uniqueness conditions) or the ability to express implications from one context property to an other (e.g. the a change in network bandwidth may affect the remaining battery power of a device). This requirement is again commonly found in the information systems literature, whether focused on databases and entity-relation models [Brodie, 1984] or knowledge bases and description logic based models [Calvanese et al., 1998].

Timeliness (dealing with context history): context-aware applications may need to have access to past states (for mining and statistical purposes) and future states (prognosis). However, in cases with high sensor update rates it may be impractical to store every value. This attribute thus relates to the ability of the representation method to support aggregation or summarization mechanisms. It is furthermore of importance to many applications that involve

temporal reasoning or the training of statistical models for activity recognition [Cook et al., 2009b].

Management of imperfect/ambiguous or incomplete information: The dynamic and heterogeneous nature of context information can lead to variations in the quality of context information (e.g. due to inherent sensor inaccuracies). Accumulated errors or malfunctions can also lead to incorrect or incomplete statements (e.g. a presence sensor that fails to detect a person may cause to report an empty room). The context representation method must therefore take such problems into account and provide adequate support for reasoning procedures (e.g. constraint checking, information fusion) that can mitigate the issue. As with heterogeneity of context sources, ambiguity of information is an inherent setting of ambient intelligence applications [Schmidt, 2006] and one that has yet to find standard means of management.

Reasoning support relates to how amenable the representation method is to application of various reasoning approaches that can ensure context consistency and derive further higher-level information from more basic one (e.g. deduce current activity of a user combining data from presence, audio and posture detection sensors). It complements the dependency specification attribute in the sense that modeling semantic relations (e.g. the type of activity in a room depends on the number of people in a room) of context information can be exploited to infer new knowledge [Bikakis and Patkos, 2008].

Usability of modeling formalism measures how easy it is for designers of context-aware applications to translate real world concepts into the constructs of the chosen context representation method. Furthermore, it analyses how easy it is for the application-level to use and manipulate the represented context information at runtime. While there exist several views as to how usability should be measured [Bevan, 2009], in this case we refer to definitions of usability in terms of the mental effort and attitude of the context model designer, as well as the product-oriented view of model ergonomics (i.e. how does it facilitate context-aware application development).

2.1.2 Context Representation Methods

Within the past 15 years, the research community on context management has made use of several methods for information representation. We want to perform an overview of these approaches, explaining their specifics, giving example of some systems that use them and, most importantly, analyse how they fare against the modeling requirements introduced previously. As a result of this analysis, we motivate why in following sections we place additional focus on two modeling techniques that are most promising: ontology-based modeling and context meta-model approaches.

The simplest context representation technique among the reviewed ones is *key-value modeling*. Context information is modeled as key-value pairs either in a textual or in a binary format. Models defined in this way are easy to build and manage, provided the number of keys (attributes) is small. The resulting models, however, end up being strongly coupled with the particular application that uses them, thus showing low flexibility. Furthermore, since no actual structure or semantics is explicitly defined, requirements such as dependency or imperfection management, timeliness or reasoning support cannot be addressed by the model and have to be added by the application level in an alternative way. Still, context management solutions which rely on event processing and operator composition to drive their context processing mechanism employ such representations. SOLAR [Chen et al., 2008] for example, uses a key-value context model in its solution for an infrastructure for context-aware applications, showcasing its usage in a smart meeting application.

Markup Scheme Models are based on using markup languages (especially XML) to represent context information. Common to all such modeling approaches is a structure consisting of

markup tags with attributes and content. Markup based modeling is more structured than the simple key-value based approach, as the used markup language is usually accompanied by a schema. This allows for certain kinds of validation, such as type and range checks for numeric values, and leads to better context retrieval capabilities. However, more complex types of relations (dependencies, constraints) are hard to model, as reported by early works [Indulska et al., 2003], and the representation does not readily offer support for reasoning. This lack of explicit semantics means that markup based context models become closely tied to the application, making sharing of context information difficult. The approach is mainly used to build *profiles* of either user or device preferences and configurations in works such as [Buchholz et al., 2004; Indulska et al., 2003] which extended the Composite Capability/Preference Profiles (CC/PP)[1] vocabulary proposed by the W3C. ContextML [Knappmeyer et al., 2010] was used in a similar manner to represent context information within the C-CAST project[2], which sought to provide context-aware multicasting of content and services to mobile device consumers.
While a purely markup-based context representation fails to provide adequate support for information consistency checking and higher-order inference, due to its lack of semantics, it can actually be used successfully as a means to transfer or temporarily store context information. XCML [Robinson et al., 2007] is an XML-based serialization of context information constructed using CML [Henricksen et al., 2005b], while RDF[3] is the default approach for serializing ontology-based context models. Both ontology-based representation and CML will be discussed in more detail in Sections 2.1.3 and 2.1.4 respectively.

Graphical Models use techniques such as UML[4] or ORM[5] to model context information. An important advance over markup based models is the ability to explicitly indicate relationships that hold between context elements. The actual underlying representation of a graphical context model can vary and techniques such as SQL and noSQL databases, XML or even graph-based storage can be used as concrete implementations.
Olaru et al. introduce a graph-based representation format for both context information itself, as well as for the situation patterns that need to be recognized [Olaru et al., 2011]. The modeling resembles concept maps, in that the edges of the graph are predicates and nodes have semantic labels attached to them. An interesting feature is the ability of the proposed reasoning algorithm to perform a partial matching of a situation pattern to the existing knowledge graph and identify the missing information. However, the model cannot currently handle aspects of timeliness or reasoning over uncertain situations (quality of information cannot be captured).
Henricksen et al. propose an extension of ORM called CML (Context Modeling Language) to model context information and use an SQL database as a support [Henricksen et al., 2005b]. Such an approach holds benefits in terms of providing validation (integrity and constraint checks are possible) and timeliness. Databases also allow for efficient storage and retrieval of information.
Graphical models in general provide strong support for usability (e.g. tools allowing a visual representation of all relationships facilitate development). However, the lack of model semantics means that reasoning and generalization support are limited. Furthermore, depending on the chosen concrete representation technology, alterations of an initial context model may be harder to address.

Object-Oriented Models use concepts from the field of Object-Oriented Programming (OOP) for context modeling. Notions such as class inheritance and object composition are used to model context information and its relationships. Being as close to programming languages as possible, object-oriented context models are easy to use for application developers and promote code reusability. However, the lack of an explicit schema or semantics of the model makes it very tied to a specific application and hinders flexibility. Furthermore, means for model validation

[1] http://www.w3.org/TR/CCPP-struct-vocab/
[2] http://cordis.europa.eu/project/rcn/85341_en.html
[3] http://www.w3.org/RDF/
[4] http://www.uml.org/
[5] http://www.orm.net/

(consistency and constraint checking) or timeliness and reasoning support have to be added in a customized manner, as they are not inherently supported by the model. Still, such context representation approaches have been well used within context management solutions that target more specific domains, such as context-awareness on a device level (e.g. COSMOS [Conan et al., 2007]) or centralized control of a smart house environment (e.g. COPAL [Sehic and Dustdar, 2010]). These works define domain specific languages (DSL) to more easily specify a context model schema which then creates the corresponding context model objects at runtime.

Logic-Based Models use the constructs of various logical frameworks to model context information in terms of predicates, expressions and terms. One important advantage is the ability to define inference rules exploiting the reasoning capabilities of the chosen logical framework to ensure context consistency and integrity, and to provide higher-level context deduction capabilities. Bikakis and Antoniou [Bikakis and Antoniou, 2010], for example, extend the Multi-Context System formalism [Giunchiglia and Serafini, 1994] with non-monotonic logic features (defeasible local rules, defeasible mapping rules) and a preference ordering mechanism in order to handle unknown, uncertain and conflicting context information. The authors showcase the approach in scenarios such as an ambient intelligence home care system or an application for context-awareness in a smart classroom [Bikakis et al., 2011]. One of the few downsides of logic-based models is that terms in usual logic formalisms lack an explicit semantics, making the resulting models tightly coupled to a specific application and thus less reusable. Nonetheless, as we discuss in Section 2.2, rule-based reasoning grounded in a logical formalism either drives or accompanies the inference mechanisms of various context management systems.

Ontology-Based Models use the expressive power of description logics to represent context information. Recent review works [Baldauf et al., 2007; Bettini et al., 2010; Perera et al., 2014a] show that ontologies have become the preferred choice of modeling in the context-aware computing community. There are two main reasons for this choice: the inherent expressiveness and the large number of standards-based tools and technologies of the semantic web community which provide good support for development, reasoning and query of ontology-based context models. In terms of expressiveness, ontologies rely on description logics to define an explicit semantics of context elements, define relations and restrictions between them and offer extensive support for consistency checks. Apart from these benefits, ontologies allow context-aware application developers to perform a clear separation of concerns between domain knowledge definition and its runtime usage, by creating explicit and reusable models of context information.

	Key-Value	**Markup Scheme**	**Graphical**	**Object Oriented**	**Logic-based**	**Ontology-based**
Model Flexibility	∼	∼	+	∼	+	+
Heterogeneity	∼	∼	∼	∼	∼	∼
Dependencies	-	-	+	-	+	+
Timeliness	∼	∼	∼	∼	∼	∼
Imperfect Information	-	-	∼	-	∼	∼
Reasoning Support	-	-	-	-	+	+
Usability of Formalism	∼	∼	∼	+	+	+

Table 2.1: Analysis of support for modeling requirements for each representation method. Meaning of notations: - means no support is given, ∼ means that no inherent support exists, but it can be partly addressed through clever design, + means support is provided inherently.

In Table 2.1 we provide an overview of the degree to which the discussed representation meth-

ods address the modeling requirements listed in the previous section. The evaluation is based on the analysis performed in this section. We distinguish between three levels of gradation: no support (-), support that can be added on an application-specific basis ($\sim$) and support that is offered inherently by the given modeling formalism (+). For instance, real reasoning support is only available in the representation methods relying on a form of logic (e.g. defeasible logic or description logic in case of ontologies). On the other hand, timestamps are not inherently captured by any representation method, but the timeliness property can be ensured if, for example, the application models every situation as a generic event for which a timestamp attribute can be attached.
From the contents of our table it becomes clear that logic and ontology-based representation means satisfy most of the requirements. For this reason, in the next section we will focus more closely on works from the literature which have used ontologies as their support for context representation. However, as we see from Table 2.1, ontology modeling alone turns out to be insufficient to address requirements such as timeliness and imperfect/ambiguous information management. Section 2.1.4 explores works that have taken the more general approach of constructing context meta-models to face these challenges.

2.1.3 Ontology-based Context Representation

Expressive power as well as development and optimized reasoning tool support are key motivation elements for the use of ontologies as the preferred context modeling method in the ambient intelligence community. However, concrete ontology proposals from the literature vary significantly when it comes to the actual context representation concepts and relations that are captured by the approaches. This is mainly a consequence of the degree of genericity and extensibility given to the ontology vocabulary. While some works focus on more specific AmI domains such as modeling Activities of Daily Living (ADL) in smart home scenarios [Chen and Nugent, 2009; Riboni and Bettini, 2011], others try to build ontologies that would cover as many context domains as possible, from information about devices and hardware platforms [Preuveneers et al., 2004], to user profiles, general location and activity data [Gu et al., 2004; Chen et al., 2005].

General Context Ontologies

We start our review of ontology-based context modeling with CONON [Gu et al., 2004] (CONtext ONtology) which defines a vocabulary for indoor environments. The core (upper) ontology contains 14 classes that model different kinds of *ContextEntities*. These include person, location, activity and computational entities such as a device, network or service. This upper ontology is meant to be extended with an application specific vocabulary, as in the case of smart home scenario used in [Gu et al., 2004]. The authors also introduce specially constructed OWL properties to convey additional information about captured context information. The `owl:classifiedAs` property performs a classification of context based on its acquisition method (sensed, defined, aggregated or deduced), while the `rdfs:dependsOn` property indicates that a property of a *ContextEntity* depends on another one. CONON also defines a vocabulary to express quality constraints, which the authors claim can be attached to properties of *ContextEntities*. However, the work in [Gu et al., 2004] and subsequent follow-ups fail to explain how quality constraints as well as the `owl:classifiedAs` and `rdfs:dependsOn` properties are created and leveraged at runtime.

One of the most well known and reused context ontologies is the Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA) [Chen et al., 2005]. It achieves great genericity and reusability by creating its core vocabulary in a modular way, based on upper-level consensus

ontologies that cover aspects including person (FOAF[1]), time (OWL-Time[2]), space (spatial ontologies in OpenCyc[3]) or even security and privacy policies (REI policy ontology[4]). Overall, SOUPA provides a broader core vocabulary for context modeling than CONON and has been extended with elements specific to smart meeting applications (covering device properties, schedule and meeting definitions, etc) [Chen et al., 2004a]. In the smart meeting scenario, one of the most interesting uses of SOUPA is the definition and enforcing of context access policies using an OWL reasoner. However, in contrast to CONON, SOUPA does not provide any constructs that can characterize the quality of perceived context information or provide further classification criteria (e.g. type of acquisition).

Strang et al.[Strang et al., 2003] provide an even more general approach than the ones previously presented with their proposed Context Ontology Language (CoOL). The main purpose of CoOL, according to [Strang et al., 2003], is to enable context-awareness and contextual interoperability during service discovery and execution in a distributed architecture.
As opposed to the previous works, CoOL is based on an abstract meta-model called ASC (Aspect Scale Model). It is named after its core concepts of *aspect* and *scale* and *context information*. An aspect is a *dimension of the situations space* (e.g. spatial, activity-related, quality-related) that describes a piece of context information. The authors define a scale as an unordered set of objects defining the range of valid context information. An aspect can have one or more scales that give its value (e.g. a spatial distance aspect may have a "MeterScale" and a "KilometerScale"). The definition of a piece of context information then becomes: *a characterization of an entity relevant for a specific task in its relevant aspects (along the defined scales).*

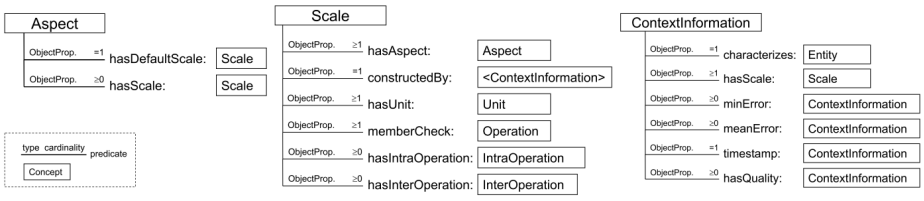

Figure 2.1: Aspect-Scale-Context (ASC) Model [Strang et al., 2003].

Figure 2.1 shows an overview of the ASC model and the relevant auxiliary properties (e.g. quality information or intra- and inter-operations) that can augment runtime processing capabilities. Interestingly, the authors argue that the concepts of CoOL can be used as a transfer model to convert the knowledge expressed in other context models. As an example, they show how a context model expressed using CML (which we discuss in the next section) can be mapped to CoOL using the ASC constructs.

Domain-Centric Context Ontologies

Though restricted to specific domains in their scenario examples, previously discussed ontologies have tried to create generic vocabularies for context-aware applications. More recent works have focused on providing more detailed modeling of *specific* ambient intelligence domains such as human activity recognition and Ambient Assisted Living (AAL).

In [Chen and Nugent, 2009], Chen and Nugent focus on providing an extensive vocabulary for Activities of Daily Living (ADL) describing a human activity in terms of it location, actors,

[1]http://www.foaf-project.org/
[2]http://www.w3.org/TR/owl-time/
[3]http://www.cyc.com/opencyc
[4]http://rei.umbc.edu/

required resources, effects, goals and more. The authors also propose a reasoning algorithm, based on equivalence and subsumption computations of description logics, to determine the current activity based of a person. The inputs to the algorithm are domain knowledge that describes various activities in terms of the proposed ontology as well as sensory information that notifies about usage of every-day objects. The reasoning algorithm can work in an incremental fashion, where with every step an increasingly precise realization of the activity of the person can be built.
The authors note however, that they leave all modeling and reasoning related to temporal and meta-properties (e.g. quality of information) to future work.

Riboni and Bettini [Riboni and Bettini, 2011] examine the benefits of using the OWL 2 ontology language to build a vocabulary for human activity recognition. The authors analyze the way in which newer constructs available in OWL 2 (e.g. qualified cardinality restrictions, property composition) help cover modeling efforts which had previously used a combination of OWL 1 and predicate logic, thereby reducing hybrid reasoning mechanisms to a single well-defined one. The work illustrates this by presenting an OWL 2 based model of ADLs for a smart home and smart workspace scenario. Still, the authors observe that their model cannot currently support the definition of context information quality metrics or easy handling of conflicting and incomplete information, while the *tree model property* condition [Grosof et al., 2003] of OWL 2 limits the expressiveness of the language.

All of the works presented so far have put the *entities* of an application domain at the center of their model and tried to provide vocabularies that would cover as many context domain dimensions as possible. Only some of them ([Strang et al., 2003], [Gu et al., 2004]) offer support to characterize meta-properties of context information (e.g., quality information), However, they do not detail how these annotations are further used during runtime processing.

2.1.4 Representation using Context Meta-Models

As explained at the end of Section 2.1.2, there exist approaches that go further in terms of provided expressiveness than ontology-based modeling. They focus on creating meta-models which are specifically designed for handling the challenges posed by effective context modeling. In contrast to the above mentioned approaches, these context meta-models usually try to provide a first-class construct for the *events* and *situations* that need to be observed. This translates into a focus on the *predicates* of the context model, the ones that describe the relations that exist between entities of an application domain. Such models can than apply any number of additional annotations (e.g., quality of information) so as to characterize entire context statements, rather than just entities.
In general, modeling support of these approaches is further increased by distinguishing between a base component (a meta-model realization), that provides a vocabulary for working with the different model elements and their properties, and an upper-component that when extended captures the different domain dimensions of a particular application field.

mySAM [Bucur et al., 2006] introduces an ontology model able to define arbitrary context predicates. The principal model element is the *ContextAttribute* which is a general construct that can express arbitrary statements of a context domain. The *ContextAttribute* has properties stating its arity, the list of entities over which it applies and the value(s) it returns. For example, a *ContextAttribute* such as "DevicesAvailableInRoom" applies to entities of type room and returns a list of devices found therein. The model distinguishes between a context ontology and a domain ontology. The domain ontology is used to define the actual concepts that pertain to an application domain. The context ontology contains all the *ContextAttributes* that apply over the domain concepts. The approach is flexible in terms of its domain modeling expressiveness but the work does not attempt to model quality (meta-properties) of the *ContextAttributes*, nor does it specify how reasoning is performed with the given constructs.

Fuchs et al. [Fuchs et al., 2005] propose a Context Meta Model capturing semantics of entities, properties and quality classes that characterize the properties. The model overview is shown in Figure 2.2. Similar to ASC presented earlier [Strang et al., 2003], the meta model defines

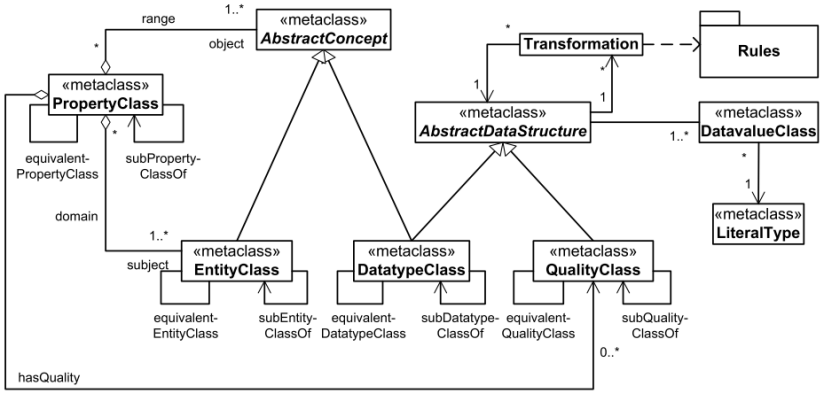

Figure 2.2: Overview of the Context Meta Model proposed by Fuchs et al.[Fuchs et al., 2005]

DataStructure classes and transformation rules that can convert from one DataStructure to another. Predicate dependencies and derivation rules are also specified in order to perform inferencing. We note also that the OWL-DL instantiation that the authors give to their Context Meta Model deals only with binary predicates and uses rules defined in SWRL[1] to accomplish derivation of higher level context information.

The work in [Fuchs et al., 2005] is similar to another proposal, which has been mentioned before and is of particular interest, the Context Modeling Language (CML[Henricksen et al., 2005b]). CML builds upon the Object-Role Model (ORM) conceptual language for data modeling used in the Relational Database domain and extends it with constructs specific to the area of context representation. The basic representational unit in CML is the *fact*, a relationship holding between one or more entities, categorized into static, sensed, profiled or derived depending on the acquisition type. It allows expression of uniqueness constraints and fact dependencies as well as annotation of facts with quality indicators. CML also introduces a form of first-order predicate logic used to derive higher-level information (called situations). The model has been used in demo applications involving context-aware communication [Henricksen and Henricksen, 2006] (i.e. selection of the communication channel based on user preferences and current availability status) and adaptation of media streaming to a mobile user according to current context [Henricksen et al., 2005a]. Though the modeling constructs of CML cover many of the requirements outlined in Section 2.1.1, one disadvantage of the approach is its concrete implementation based on Relational Database schemas, thereby missing out on the important aspect of an explicit semantic dimension for facts, entities and situations of a given application domain.

2.1.5 Context Representation Summary

We have seen that the task of effective modeling of context information poses numerous challenges. We argue that in order to meet the requirements listed in Section 2.1.1 a context modeling approach has to consider three important aspects:

(i) represent context content using an expressive method with an explicit semantics

[1] http://www.w3.org/Submission/SWRL/

(ii) defining context annotations (meta-properties) which can capture timestamps, quality information and properties that can further categorize context statements (e.g. their acquisition type)

(iii) provide a means to express context dependency relations (e.g. consistency, constraint integrity)

The first condition relates to fulfilment of requirements such as flexibility, heterogeneity and (re)usability of a context model. The explicit semantics allows for increased flexibility through the separation of concerns between knowledge engineering and application usage, while expressive modeling capabilities (e.g. n-ary predicates in CML) improve usability of an approach.
The ability to express context annotations addresses concerns such as timeliness and ambiguity/imperfect information management. Timestamps and temporal validity meta-data offer support for time-related queries and reasoning, while quality of context (QoC) metrics improve the ability to handle ambiguous or uncertain data.
Lastly, the third condition is meant to address the need to maintain consistent and constraint free context knowledge bases, in face of very frequent and possibly erroneous updates of sensed or derived information. Moreover, explicit dependencies can help improve the capabilities associated with reasoning engines that leverage the context model at runtime.

Examining the works we have reviewed previously, we note that some of the issues listed above remain insufficiently addressed. The majority of approaches mentioned in Section 2.1.2, other than ontologies and context meta-models, suffer from a lack of sufficient expressiveness and support for more complex reasoning, which stems from the fact that they lack an explicit semantics of their model constructs and cannot readily capture information dependencies or constraints (e.g. as is the case for key-value, object-oriented or markup-scheme based models). While this may be sufficient with regard to certain application domains (e.g. low-level device resource monitorization, smart homes), our objective is to create an approach that is usable in a wider set of scenarios.
We therefore chose to focus more closely on ontology-based context modeling and context meta-modeling. However, as we observed in Section 2.1.3, presented works have several shortcomings and a trade-off between expressiveness and usability exists, as shown in Figure 2.3.

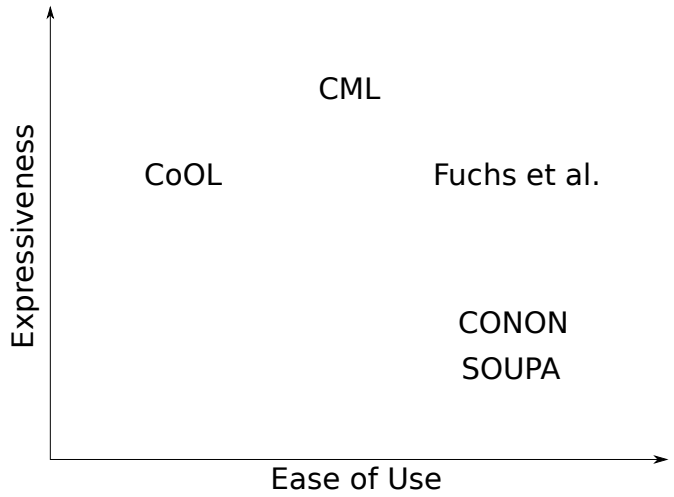

Figure 2.3: Analysis of trade-off between expressiveness and usability requirements for ontology and context meta-model representation approaches. Expressiveness is an attribute covering model flexibility, dependency and ambiguity management. In the figure, only generic context ontologies are shown (domain-specific ones such as [Chen and Nugent, 2009] maintain virtually the same characteristics as generic domain ones, with the exception that they are less flexible).

SOUPA cannot capture context quality information and other meta-properties, while CONON offers an annotation vocabulary but fails to explain the way it is used. CoOL is more expressive than the previous approaches but does not consider integrity constraints and the ASC constructs on which it is based seem to be unwieldy in terms of usability. The domain-centric ontologies on the other hand ([Chen and Nugent, 2009],[Riboni and Bettini, 2011]) can only be used for applications from the domain for which they are meant.
Amongst the context meta-model approaches, CML and the solution of Fuchs et al.[Fuchs et al., 2005] seem to provide strong modeling capabilities. However, CML loses in usability and reasoning support by adopting a relational database support for concrete model implementation. Fuchs et al. make use of a semantic realization (via OWL-DL classes and properties) of their meta-model, but unlike CML, they can only address binary predicates and the chosen inference method relies on SWRL rules which limits their expressiveness. Additionally, though both works provide support for modeling context annotations, neither of them specifies how the values of such meta-properties are combined during inference rules that derive higher level information from existing context statements.
These and other considerations motivate objectives and design choices (explained in Section 2.3) of our own context representation method.

2.2 Reasoning about Context Information

Context information acquired from sensors and devices or directly provided by users can often be of low-level character (e.g. battery level, received signal strength of an RFID, noise level in the room, calendar information). This data can furthermore have a high change rate and be simultaneously provided by several different entities with varying degrees of resolution, accuracy or confidence.
However, context-aware systems are interested in making decisions based on higher-level features (e.g. user presence, existence of a meeting, user availability status) that are more closely tied to the actual application logic. Furthermore, decision making must be performed on hand of unambiguous and highly accurate information. For this reason, all proposals for context management in the literature provision their systems with a means for reasoning about acquired context information. The objective of reasoning approaches is to ensure information consistency and to provide the ability to derive higher-level meaning out of raw data.
In this section, we firstly look at the typical aspects and factors that concern reasoning about context information and present a brief overview of the different kinds of inference approaches that have been proposed in the literature. We then explore more closely two of the methods that find most support amongst recent research efforts, ontology-based reasoning and rule-based reasoning. Lastly, we give examples of other possible means to accomplish this task before summarizing our discussion.

2.2.1 Reasoning Concerns

In the discussion about methods to represent context information we explained that aspects such as heterogeneity of sources, ambiguity/uncertainty management and dependency relations play a very important role in context modeling, since they are inherent to AmI applications. We argued that an effective representation method must provide constructs that offer adequate assistance to reasoning mechanisms charged with addressing these challenges.
It follows from the above characteristics that the reasoning layer in a context management solution has to perform two main tasks:

(i) ensure consistency of the knowledge base.

(ii) filter, aggregate or otherwise manipulate raw (primary) context information so as to derive higher-level knowledge based on known data dependencies and correlations.

Before delving into describing means for performing reasoning, let us first consider some of the factors that can make context information inconsistent [Henricksen and Indulska, 2004b] and the way in which they can be handled.

A first factor for uncertainty of acquired context can be the inherent accuracy limitations of the sensing hardware. For example, the certainty with which an RFID base station detects the presence of an RFID badge depends on the capabilities of the base station and the distance of the badge from the base station. Similarly, a Kinect[1] camera assigns a degree of confidence to the coordinates (relative to its view angle) for the joints of a person's "skeleton" detected based on its depth sensor. Thus, a representation method which provides QoC annotations such as accuracy, certainty or resolution can help the reasoning layer to overcome this type of uncertainty.

Information ambiguity can also appear when several sensors or inference rules provide information content that is contradictory on a semantic level. For example, consider a smart laboratory where there are several bluetooth beacons[2], one for each desk of the room, which are used to infer the location of a person. It may happen that, given the movement of the person in the laboratory, two or more beacons will assert having detected the user's smartphone (though with varying certainty). In this case, it is necessary to enforce context *integrity constraints* specifying that a person cannot be located in two places at the same time. The actual resolution of constraint violations in the example given previously can again be based on certainty annotations.

Imperfect or contradictory information can also be caused by sensors that stop sending updates or by data that becomes stale (i.e. its validity expires). In the bluetooth-based person location example given above, the inference mechanism that derives that a person is located near a certain desk must operate only with the freshest data from each beacon, to avoid reasoning over past locations. Therefore, context statement meta-properties such as timestamps and validity information are required in order to ensure valid and uptodate inferences. In dealing with timestamps, the context management system must also ensure that all sensors and reasoners use the same mechanism of time calculation. This issue can become complex in distributed settings, where context information can be produced in one location and consumed in an other.

Lastly, inconsistency of a knowledge base can be a consequence of context data that breaks the semantics of the underlying model (e.g. due to erroneous derivations or human errors). For example, in a smart meeting context, an application which erroneously asserts that the current activity in the room is both a board meeting as well as a brainstorming session will render the knowledge base inconsistent, given that the two semantic classes of board meeting and brainstorming session are disjoint. Such errors can generally be detected using consistency check reasoning algorithms, such as those of ontology based models, where the description logics based context model allows for this kind of inference. Resolution of a detected inconsistency in such cases is generally a harder task and is in many cases strongly application dependent.

2.2.2 Categories of Context Reasoning

There is a large number of different decision models that have been used in the context reasoning literature. However, it is noteworthy that no decision model in itself is specific to context management. It is rather the case that context management solutions have adapted general reasoning techniques for the purpose of inference. Broadly viewed, these reasoning models can

[1] http://www.xbox.com/en-US/xbox-one/accessories/kinect-for-xbox-one
[2] http://estimote.com/

be classified into the following categories: machine learning methods (supervised and unsupervised), rule-based models (including complex event processing [Buchmann and Koldehofe, 2009]), ontology-based reasoning and probabilistic methods (e.g. Naive Bayes, Hidden Markov Models).

Machine Learning (ML) and probabilistic reasoning approaches are often used in tasks such as activity recognition tasks, indoor localization, low-level device operation management (e.g. routing in sensor networks based on expected network congestion). The reasoning happens usually very close to the sensing layer, that is, via making direct use of raw context data from which sufficient features can be extracted to train the ML models. The advantage of such approaches is that in many situations they can achieve a high degree of accuracy in their recognition. This, however, comes at the price of needing great amounts of data to train the recognition/classification models to an adequate level, which if often a challenging endeavour and also renders the resulting models strongly application dependent.

Ontology and rule-based systems, on the other hand, operate on a higher semantic level in reasoning tasks such as event processing, preference-based adaptation / personalization or high-level activity recognition (e.g. activity of a group of persons instead of just the individual). The advantage of such reasoning approaches relies on their explicit use of semantics which allows for existing domain knowledge to be easily introduced. However, situations with high dynamics and uncertainty of sensed context can pose difficulties.

It is worth noting, however, that many works in the literature employ a combination of the above described methods when performing reasoning. As we discuss in the sections that follow, ontology-based context models often use rule-based derivations to augment their inference capabilities. An ontology for domain knowledge modeling can be combined with probabilistic reasoning at a lower sensing level to increase the accuracy of single-person activity recognition applications such as in [Riboni and Bettini, 2009]. Furthermore, works exist that use various forms of probabilistic logic (e.g. Dempster-Shafer evidence theory [Yager et al., 1994]) to perform reasoning about uncertain context information, especially in the sensor fusion domain. [Dargie, 2007] provides a good overview of such approaches.

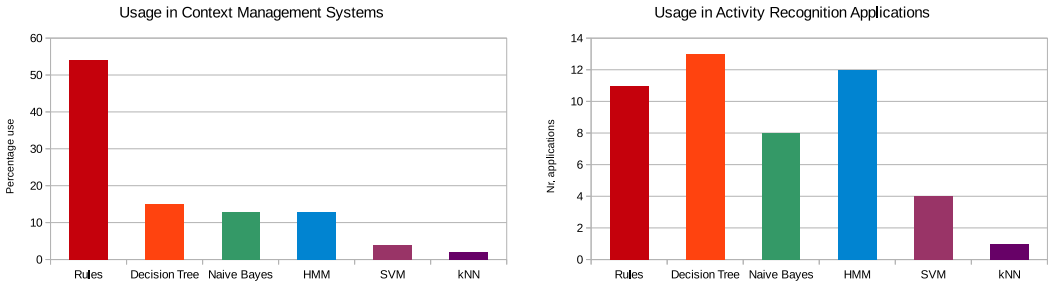

Figure 2.4: (a) Counts of model types used in 109 of 114 reviewed context-aware applications. (b) Counts for 50 recognition applications; classifiers are used most often for applications that do recognition [Lim and Dey, 2010].

Lim and Dey analyze the chosen reasoning method in over 100 ambient intelligence and pervasive computing applications [Lim and Dey, 2010]. The results they report are summarized in Figure 2.4.
It follows from these results that rule-based reasoning approaches (which include event processing) are used in the majority of systems because of their advantage of being simple to define and to extend, as well as using less resource (in terms of storage or processing power).

In the introduction to this thesis, we mentioned that one of the objectives of our work is to alleviate the development effort for engineering a context-aware application. As noted in Section

2.1.5, we believe that the support for explicit semantics in both representation and reasoning is a key issue in this regard. Consequently, in what follows we consider taking a closer look at knowledge oriented reasoning approaches, investigating the key features and downsides.

2.2.3 Ontology-based Reasoning

Ontologies are modeling formalisms by which an *explicit specification of a shared conceptualization* [Studer et al., 1998] can be achieved. Nowadays, the de facto language used to create ontologies is OWL-DL [Horrocks et al., 2003], or some of its variations, as it is becoming a standard choice in various application domains and it is supported by a number of reasoning services (e.g. Pellet[1], Racer[2], Fact++[3]).
By means of OWL-DL, ontologies can model a particular context domain in terms of *classes*, *individuals*, relations between individuals (*object properties*) and characteristics of individuals (*datatype properties*). Furthermore, more complex descriptions of concepts and properties can be built by using operators provided by the language to combine (e.g. union, intersection operators) or further characterize elementary descriptions (e.g. via property role restrictions, subclass or subproperty relations).

Ontology knowledge bases have two components, a TBox and an ABox. The TBox contains all the terminology in the form of elementary and complex concept descriptions that the application domain defines. The ABox includes all assertions about individuals that represent instances of the concepts defined in the TBox.
With respect to reasoning, the key operation that ontology reasoners perform is that of *subsumption determination*, i.e. checking if one concept is more general than an other. Based on this, ontology reasoning with respect to context information makes use of the following inference procedures:

(i) *TBox classification*: compute a complete concept hierarchy based on the existing concept descriptions and defined subsumption relations

(ii) *ABox realization*: determine all the concepts instantiated by a given individual

(iii) *Knowledge base consistency*: check whether there is a contradiction with respect to the chain of concept definitions and their instantiation by individuals

Subsumption determination is very useful in context models where the hierarchy of concepts is very deep. Take the following simple example of relation types between people.

```
ex:Alice rel:isCloseFriendOf ex:Mary.
rel:isCloseFriendOf rdfs:subPropertyOf rel:knows.
```

In this case, the ABox realization inference will determine also that *ex:Alice rel:knows ex:Mary*.

Several works use ontology reasoning with the purpose of ensuring consistency of the knowledge base under its semantic restrictions. In [Chen et al., 2004a] this is used in the context of smart meetings, while in [Gu et al., 2005a] it is used in smart home scenarios (e.g. home energy saving or happy dining room services). Additionally, Chen et al.[Chen et al., 2004a] use ABox realization and instance check queries to enforce context access control policies defined using the SOUPA[Chen et al., 2004b] ontology.
Then again, other works use ontological inferences as the primary method to derive new context information. Turhan et al. [Turhan et al., 2006] use OWL-DL reasoning to implement the logic of a context-aware door lock within a smart home.
In the COSAR [Riboni and Bettini, 2009] application, ontology-based reasoning (specifically

[1] http://clarkparsia.com/pellet/
[2] http://franz.com/agraph/racer/
[3] https://code.google.com/p/factplusplus/

TBox classification and ABox consistency) is used to provide domain knowledge referring to what type of human activity (e.g. brushing teeth, hiking up, jogging) can be performed in what kind of environment (e.g. rest room, woods, urban area).
Chen and Nugent use an iterative reasoning process [Chen and Nugent, 2009] involving subsumption and ABox instance realization checks to infer the current activity of a person in a smart home (e.g. MakeDrink, DoHousework). The algorithm uses the ontology for describing ADLs mentioned in Section 2.1.3 to define detailed characteristics of each activity type. Sensors that detect presence in a given room and usage of a given household device or object (e.g. cups, milk carton) provide the instantiations (individuals) of the ontology concepts. An interesting feature of the proposed activity recognition algorithm is that the subsumption relation (e.g. the subclass relation between MakeDrink, MakeHotDrink and MakeTea) can be used to provide increasingly accurate descriptions of the performed activity. In case of incomplete information, such as when the sensor for temperature on a cup is broken and the system cannot determine if it is hot or cold, the reasoning procedure can still infer the fact that a person is making a drink, based on the description of the activity and the sensors that are still active.
Riboni and Bettini [Riboni and Bettini, 2011] perform an extensive analysis of OWL 2 DL[1] modeling and reasoning capabilities in context-aware applications for activity recognition in smart homes and offices. The main point of their analysis is the measure of increase in modeling and reasoning capabilities given by language constructs that were introduced in OWL 2 (e.g. qualified cardinality restrictions, property composition). They put this in contrast with the way in which the same information content would be modeled as a combination of OWL 1 axioms and FOL (first-order logic) predicates. For instance, they give the example of the following rule for inferring co-location of two individuals.

$$Actor(?x) \wedge Actor(?y) \wedge SymbolicSpace(?z) \wedge located(?x, ?z) \wedge$$
$$located(?y, ?z) \rightarrow collocated\_with(?x, ?y).$$

The authors then show how this rule can be expressed using a single OWL 2 axiom:

$$located \circ hosts \dot{\sqsubseteq} collocatedWith$$

where the property *hosts* is the inverse of propery *located*.
Interestingly, the work in [Riboni and Bettini, 2011] goes on to present issues regarding the coexistence of open-world assumption (OWA) of OWL and the closes-world assumption (CWA) of rule-based reasoning, which can lead to inconsistent reasoning outcomes. The authors argue in favor of using a single formalism (i.e. just ontology-based inferences), but point out that because of limitations of the OWL reasoning model (e.g. the tree model property [Grosof et al., 2003]) certain expressions, which are easily captured by a rule system, cannot be modeled as ontology axioms. Additionally, while the OWA can handle inferences under incomplete information, reasoning with uncertainty and imperfection can much harder be addressed using ontology modeling alone.

In summary, ontology-based reasoning is well suited to enforce knowledge base consistency given the explicit semantics of a context model, as well as to perform inference based on explicit domain knowledge. These features are important in the attempt to build a context management system that promotes easy of development and usage. However, limitations of expressiveness and reasoning under uncertainty/imperfection must be overcome by a complementary mechanism.

2.2.4 Rule-based Reasoning

We mentioned in Section 2.2.2 that rule-based reasoning is the most popular approach in recently developed context management systems. The ability to easily encode domain knowledge

[1] http://www.w3.org/TR/owl2-overview/

into a rule form is the main reason for this choice. Additionally, rule-based systems can more easily exploit modeling constructs that capture Quality of Context aspects (e.g. accuracy, resolution, freshness) in case the underlying context representation method supports them.
Rule-based reasoning approaches differ according to the formalism they use when performing inferences.

Logic-Based Approaches

The majority of works use a logical framework to support the construction and execution of their rules. The Context Broker Architecture (CoBrA [Chen et al., 2004a]) and the Service-Oriented Context-Aware Middleware (SOCAM [Gu et al., 2005a]), which have already been introduced, complement ontology-based reasoning with user supplied rules utilizing First-Order Predicate Logic (FOPL) in a forward-chaining deduction cycle (e.g. CoBrA uses the JESS[1] production rule system).

Toninelli et al. [Toninelli et al., 2006] develop an access control policy model that exploits context-awareness for the specification and evaluation of the defined policies. Context-awareness in their approach is implemented by a combination of ontology modeling and FOPL to overcome limitations of pure ontology reasoning.

The authors of SAGE [Broda et al., 2009] propose a system using both forward chaining deductive reasoning and abductive reasoning. They use it to create an agent-based environment monitoring and control system. In particular, SAGE uses forward chaining to interpret direct sensor data (e.g. deduce movement speed from two different collocated motion detection sensors). On the other hand, abductive reasoning (on hand of the DARE system [Ma et al., 2008]) is utilized to generate possible explanations of events (e.g. movement is detected because a person is on their way to the elevator of a building).

Bikakis and Antoniou [Bikakis and Antoniou, 2010] place more focus on reasoning with uncertainty and turn to the support of defeasible inferences. In the proposed system they extend the Multi-Context Systems paradigm[Giunchiglia and Serafini, 1994] with non-monotonic logic features (defeasible local rules, defeasible mapping rules) and a preference ordering mechanism in order to handle unknown, uncertain and conflicting context information. Furthermore, they provide a distributed query evaluation protocol that implements an argumentation framework capturing the semantics of the proposed reasoning approach.

Operator Composition Approaches

In the discussion about rule-based reasoning we also include the works that take an approach based on the composition of different types of processing units (operators). These operators can perform various tasks such as aggregation, summarization or filtering (which involves if/else like statements and are therefore similar to applying simple rules).

SOLAR [Chen et al., 2008] proposes a reasoning mechanism based on reusable, distributed operators that use the filter and pipe paradigm to form DAG-like flows of context information processing. The authors claim the genericity and extensibility of operators (i.e. they are implemented by developers), but the ones used in experimentation provide simple match and filter functions.

COPAL [Sehic and Dustdar, 2010] is a context-aware middleware solution intended for smart home applications. In a manner similar to SOLAR, it defines 5 types of processing patterns inspired by work in complex event processing [Luckham, 2008] and event processing networks [Sharon and Etzion, 2008]. Processing units in COPAL can use filter, abstraction

[1] http://www.jessrules.com/jess/index.shtml

(e.g. summarization, aggregation), differentiation, enrichment (add additional information to incoming events, e.g. timestamp, quality metrics) and peeling (drop context attributes which are no longer required in subsequent processing flows).

Semantic Web Based Approaches

Some more recent reasoning approaches try to make use of semantic web technologies to achieve a combination of both forward-chaining deduction systems and event processing capabilities.

Meditskos [Meditskos et al., 2013] propose an ontology for modeling complex activities. Though restricted to the domain of activity definition, the authors utilize SPARQL [1] through its CONSTRUCT queries as a rule language that helps reason about composition of simpler activities into more complex ones. The inherent expressiveness of the SPARQL query syntax is the main advantage of this approach, while an additional benefit is its reliance on a standard of the semantic web community, receiving substantial engineering support.

Further works (e.g. EP-SPARQL [Anicic et al., 2011]) considered enhancing the SPARQL standard with the ability to perform temporal reasoning tasks commonly found in event-processing systems. Others [Teymourian et al., 2012] made it an integral part of semantic event-driven systems, which combine static background knowledge modeled using ontologies with the complex dynamics of event processing systems.

A possible advantage of implementing rule systems using approaches like the ones above is that of uniformity. The SPARQL language was built to allow complex query expressions over RDF knowledge bases. Ontology based context representation, on the other hand, can also be easily stored in RDF syntax, such that no further information representation format has to occur in order to use external rule engines (e.g. JESS), thereby eliminating an engineering overhead.

2.2.5 Other Approaches

In Section 2.2.2 we listed four broad categories for context reasoning approaches and in the previous two Sections we analyzed the ones that make use of explicit domain knowledge to drive inferences (which are our focus). For the sake of completeness, we provide a brief overview of reasoning algorithms belonging to the remaining two categories (machine learning and probabilistic methods).

Within the machine learning category there are two distinct types of approaches.
Supervised machine learning models use algorithms such as Decision Trees [Quinlan, 1986], Artificial Neural Networks (ANN [Yegnanarayana, 2009]) or Support Vector Machines (SVM [Hearst et al., 1998]). They have been overwhelmingly used in activity recognition applications in domains such as smart homes, ambient assisted living or body sensor networks. In such applications a high volume of raw context data is required to train the respective models. The trained model is then used at runtime to perform prediction or classification of sensory information into higher-level situations. Example works where these reasoning methods have been used in a context-awareness setting can be found in [Huang et al., 2008; Korel et al., 2010; Doukas et al., 2007; Brdiczka et al., 2009].
Unsupervised machine learning models refer to algorithms which do not / cannot benefit from a training phase and must find hidden patterns within unlabelled data. These algorithms employ what are called clustering techniques such as k-Nearest Neighbours (kNN [Zahid et al., 2001]), k-means [Kanungo et al., 2002] or self-organizing maps (SOM [Kohonen, 2001]). Clustering has been used in building indoor-location systems [Youssef et al., 2003], while in [Van Laerhoven, 2001] SOMs are used to perform online (real time) classification of accelerometer sensor data

[1] http://www.w3.org/TR/rdf-sparql-query/

into simple activities such as sitting, standing, walking, running and bicycling. Korel and Koo [Korel et al., 2010] provide a survey of unsupervised learning methods for body sensor network applications.

The probabilistic reasoning algorithms make decisions based on probabilities assigned to facts and events of the application domain. They are often used to perform sensor fusion (combination of data from different sources, with varying degree of certainty) or classification of occurring events. Various methods which are commonly used in statistical processing can be employed for context reasoning related tasks.
Naive Bayes is a probabilistic classifier that applies Bayes' theorem to model the probability of the output of a system given the inputs. It has been used, for instance, to recognize physical or domestic activities [Chang et al., 2007; Tapia et al., 2004].
Hidden Markov Models (HMM [Rabiner, 1989]) are Bayesian classifiers modeling the probability of a sequence of hidden states given the observed events that depend directly on the current state. Naturally, this characteristics renders them useful in various activity recognition tasks such as for physical [Chang et al., 2007] or domestic [Van Kasteren et al., 2008; Brdiczka et al., 2009] ones.
Lastly, the Dempster-Shafer [Yager et al., 1994] theory of evidence can be seen as a method of probabilistic logic, allowing to compute degrees of belief in a rule-based combination of observed sensor evidence. As a consequence, it is commonly used in sensor data fusion for activity recognition. The works in [Peizhi and Jian, 2008; Zhang et al., 2009; Lyu et al., 2010] provide examples of systems and context-aware applications that use this method.

2.2.6 Context Reasoning Summary

We saw that reasoning over context information is a challenging task which focuses around two main aspects: keeping a consistent knowledge base and higher-level derivations under conditions of uncertainty and imperfection.

	Machine Learning	Ontology-Based	Logic-Based			Probabilistic	
			Logic	Operators	Semantic Web	Statistical	Prob. Logic
Reasoning Expressiveness	+	+++	+++	+	++	+	++
Accuracy	+++	+	++	++	++	+++	++
Development Ease	+	+++	++	+++	++	+	++
Resource Consumption	+++	+++	++	+	++	+++	++
Comprehensibility	+	+++	+++	++	+++	+	++
Extensibility	–	+++	+++	+++	+++	–	++

Table 2.2: Analysis of context reasoning methods. Meaning of notations: – means no support is given, while the + signs represent the degree to which an approach can address the specified attribute (from week to strong support).

Table 2.2 shows an overview analysis of the discussed reasoning methods and the way they cater to a selection of approach characteristics. These attributes were distilled based on the discussion of pros and cons for each reasoning mechanism performed by [Perera et al., 2014a]. By *reasoning expressiveness* we refer to the ability of the reasoning approach to capture and use semantic dependencies that exist between one or more pieces of context information, as well as the complexity of inference expressions. The attribute *accuracy* refers to the quality of the inference result in the face of unknown, ambiguous or unforeseen information. *Ease of development*

denotes the support for engineering and deploying systems that use the given reasoning method. *Resource consumption* refers to a cumulative characterization of the design-time (e.g. data needed for training a machine learning model), runtime storage and runtime processing (i.e. CPU) effort required to successfully use the given approach. By *comprehensibility* we mean the degree of naturalness and transparency of both internal reasoning (i.e. how close are the internal mechanics of the reasoning method to the domain on which inference must be carried out), as well as final output (e.g. a numeric versus a semantic result). Finally, *extensibility* refers to the ability of the reasoning method to support easy design-time or runtime modifications (addition of new input conditions, internal operations, changing the output type, etc) of the respective reasoning model.

Given the analysis in Table 2.2 we observe that, though ML and probabilistic methods can achieve high accuracy in face of uncertain sensor information, gathering the amount of required training is a big challenge and the constructed recognition models are often strongly application dependent. The fact that no semantics is attached to them means that it is difficult to easily include existing domain knowledge in the reasoning process.
This lies in contrast with one of our goals, namely the construction of a context management middleware supporting openness and ease of context-aware application development.

We therefore decided to focus on higher-level reasoning approaches such as ontology and rule-based reasoning. However, Section 2.2.3 showed us that ontology-based inferences alone are currently insufficient to offer required expressiveness or the ability to reason about uncertainty. This is why many context management solutions which use an ontology to represent context information usually employ additional rule-based inference mechanisms.

From the works in Section 2.2.4 we note that operator composition based approaches are easy to engineer and deploy, but lack the ability to aggregate and reason about complex situation definitions or to enforce information consistency (e.g. integrity constraints). In contrast, the various logic-based approaches can cope well with expressive derivations of higher-level knowledge and ensure context knowledge base consistency. If based on underlying representations that support expression of meta-properties and quality of context metrics, logic-based rules can also perform temporal reasoning and inferences under uncertainty. However, not many works presented in Section 2.2.4 have this capability. Furthermore, those that do are not specific about how annotations of the derived context statement are obtained during inference from those of the premises.

Semantic web approaches which use SPARQL as a rule language have the advantage of expressiveness (e.g. SPARQL has the ability to express conditions over aggregate expressions) and uniformity if used together with an ontology based representation. Nonetheless, the works reviewed in Section 2.2.4 lack means to model relations that exist between context information and can therefore not detect consistency and constraint violations on a semantic level.

2.3 Our Context Modeling Objectives

The task of modeling context information faces several challenges in order to obtain an effective context management system. On hand of examples from related work we have seen that the research community has defined and focused on addressing several requirements for both context representation and context reasoning. Based on our discussions in this chapter and the summarizing of our analysis in Tables 2.1 and 2.2, we explained the shortcomings and trade-offs of existing context representation and reasoning approaches.
Furthermore, adding our own general objectives expressed in the introduction, we advocate for the definition of a context modeling approach catering for needs such as openness, flexibility of design, ease of development and ease of usage. We consider these aspects to be of great importance in the attempt to move context-aware application programming from the research

to the industry domain.
In what follows, we lay out a set of objectives which are meant to define guidelines for the way in which we attempt to mitigate the discussed downsides and trade-offs. The resulting context modeling approach we are proposing will be detailed in Chapter 4.

1. Create a context meta-model providing uniform representation support for the main context modeling concerns: content, annotation, dependencies. A meta-model satisfies our requirements for flexible design and model expressiveness. However, to address concerns of reasoning support and usability of the modeling formalism, we set further objectives which also focus on *uniformity* and *extensibility* of the approach.

2. Use semantic web technologies to implement the proposed context meta-model. We opt for an ontology-based definition of all context modeling elements and the use of semantic web techniques such as SPARQL to express both higher-level context derivation rules as well as context integrity constraints. The choice of using semantic web standards throughout the entire context modeling approach leads to uniformity of development effort for context-aware application designers.

3. Use of semantic-web based reasoning methods increases our reasoning expressiveness, comprehensibility and extensibility (cf. Table 2.2). However, to increase the accuracy of the reasoning approach in face of ambiguity, we strive to provide an extensible, but also structured, approach for the definition of context annotations. That is, offer the flexibility of defining different *types* of annotations, but also the ability to specify annotation-specific operators with a well-defined usage semantics, which govern the way in which annotations are combined during inferences.

4. Propose a reasoning mechanism that relies on ontology reasoning to ensure knowledge-base consistency and SPARQL-encoded rules to derive higher level context information in an expressive manner.
   This means that we focus on a combination of ontology and rule-based reasoning in the attempt to address ease of development and intelligibility of the reasoning approach.

5. Define a reasoning cycle which includes automatic computation of *temporal continuity of events*, leading to semantically distinguishable situations. This facilitates more complex situation definitions such as those based on temporal validity reasoning while also attempting to reduce storage resource consumption (e.g. just update the validity of an existing event, instead of storing a new one with the same content).

6. Address the problem of knowledge base consistency by building a reasoning engine able to detect context integrity constraint violations expressed using SPARQL queries as explained previously. Additionally, create the support for customizable constraint resolution services.

This chapter has covered a review of state-of-the-art approaches in context modeling and discussed the requirements that have been determined by existing work to be of great importance. The above listed objectives are meant to provide solid options for addressing all the mentioned context modeling needs.
In the next chapter we continue our state of the art review by expanding our viewpoint to the system architectures that are built around a given context model.

Chapter 3

Advances in Context Management Systems

Previously we have explored state-of-the-art approaches to representation and reasoning over context information. This subject lies at the heart of a context management solution. However, as we have explained in the chapter discussing our problem statement, effective and efficient programming of a context-aware application requires the existence of an extensive architecture that helps create the so called *context provisioning process*, that is, the set of tasks involved in acquiring, modeling/reasoning and disseminating context information within the application. Furthermore, this process has to be accompanied by a set of auxiliary operations such as mobility management, access control or preference management. All the while, the context management systems that provide these facilities must strive to ensure non-functional characteristics such as scalability, ease of configuration and usage or fault tolerance.

Consequently, this chapter continues the presentation of the state-of-the-art by looking at the architectures of context management solutions and the capabilities they provide.
In Section 3.1 we begin by detailing the operational aspects involved in the context provisioning process and then continue to discuss about non-functional concerns and their influence over these operations. We then provide an overview of different provisioning architectures, detailing the functionality of their architectural units and analyzing them with respect to the aspects introduced previously.
Section 3.2 widens our analysis framework by looking at how context management solutions are typically deployed in a context-aware application. We discuss deployment concerns, a categorization of application scenarios that impacts deployment requirements and the design time and run time options that different context management systems provide with regard to the deployment of their context provisioning units.
Finally, Section 3.3 presents the detailed objectives of our own approach to building a context management middleware solution, in light of issues that are insufficiently addressed in the reviewed state-of-the-art and those that are important with respect to our own goals stated in the introduction.

3.1 Provisioning Context Information

Context provisioning refers to the entire set of mechanisms and interactions used by a context management system to let applications built on top of them access their desired context information *when* and *how* they need it. Several main steps can be distinguished within the provisioning process as well as a set of transverse functionalities which accompany and augment

the processing capabilities. Throughout the provisioning process context management systems try to address a set of non-functional requirements which have been deemed important and necessary by the context-aware application development community.
We start this section by presenting the aspects of context provisioning before presenting various proposals for context provisioning architectures and the way they include the context modeling approaches explored in the previous chapter.

3.1.1 Operational Aspects

Several existing survey works [Baldauf et al., 2007; Perera et al., 2014a] propose describing context management architectures in terms of: A) the operational cycle and B) transverse functionality blocks. These are listed below.

A) Context Management Life Cycle

The Context Management Life Cycle refers to the set of steps (operations) taken by a context management system to deliver (provision) information from the producer of context to the entities (context-aware services or end applications) that consume it.

Context Acquisition is the first stage in the provisioning process and refers to the collection of information from sensors using different sensor access and interaction methods. The task of acquiring context information can be characterized along several dimensions [Perera et al., 2014a].
First, we can distinguish between different types of sensors. *Physical sensors* are hardware devices and represent the most common sensor type. They usually provide low-level context data (e.g. temperature, light-level, noise level) about an environment which they are set to monitor. *Virtual sensors* retrieve data from many sources and publish it as sensor data (e.g. calendar entries, twitter feeds, facebook status). This type of sensors do not have a physical presence and are most commonly accessed through web services. *Logical sensors* (or software sensors) provide higher-level sensory information by combining data from physical devices or virtual sensors. Examples of logical sensing would be an indoor location system which can provide semantic positioning (i.e. the name of the room, street or city) instead of geographic coordinates, or a weather service that predicts tomorrow's forecast based on current and historical temperature, pressure and wind speed data. Logical sensors are usually also accessed as web services.
We can then observe different technical means to communicate with a given type of sensor. *Direct sensor access* implies communication with the sensor hardware via related APIs. This requires the installation of software drivers and libraries and works in cases where the sensors are found on the same physical machine as the application that retrieves their data. *Acquisition through a middleware infrastructure* is a method where the sensor information is retrieved through an intermediary layer that has the objective of hiding low-level sensing details (e.g. hardware communication). This technique allows easier extensibility since sensing logic is separated from client retrieval logic. *Acquisition based on a Context Server* is an approach where access to sensed data is performed through a specialized component which permits existence of multiple concurrent clients. The server handles all communication with sensors and can even perform more complex aggregation operations. Systems that use this kind of client-server access to context information must then consider aspects such as appropriate communication protocols, network performance, quality of context parameters, etc.
Lastly, we can consider the responsibility of performing the acquisition. In *pull mode*, the software component responsible for acquisition has to issue an explicit request to the sensing hardware to acquire data. In *push mode*, the physical or virtual sensor pushes information

to the component responsible with acquisition. In this case, it becomes important to have a method to control what kind of data to push and under what conditions (frequency of updates or updates sent only when changes from a previous value occur).

Context Modeling refers to the subjects discussed in the previous chapter, giving a representation to acquired context information and using means to ensure consistency and derivation of higher-level context.
In this layer, we will also include the operations which are typically referred to as *pre-processing* steps, that is methods to transform coarse-grained raw data into more useful information. Examples could include averaging of temperature readings from sensors spread throughout a room or transformation of signal strength measurements from several indoor location beacons into the values of a coordinate system relative to a specific room.
Furthermore, since the Context Modeling step lies at the heart of the provisioning process, it is closely linked to the operations that implement *provisioning coordination*. This means that it is at this level that the context management system, or the application using it, can make decisions about: what type of information to receive and distribute, which sensors should be active in order to reduce network traffic, what reasoning operations should be active or how they should be scheduled.

Context Dissemination is the provisioning operational block which provides methods to deliver context to the consumers. From the consumer point of view, two options are commonly used to access context information. *Queries* are direct and one-time requests for which the context management system has to provide an answer. *Subscriptions* are the other option, which allows consumers to express a longer-lasting interest in specific information, whereby they receive answers either periodically or when the event they are subscribing for is actually observed (i.e. the conditions of the consumer query are satisfied).
However, from the management system point of view, the tasks related to dissemination may involve more complex issues depending on the chosen system architecture. Thus, for example, in systems using a P2P or hierarchical distribution architecture, a query or subscription may have to be routed to the appropriate management node. Complex query federation mechanisms must exist internally when the application level is *aware* of the distributed management architecture and can express queries which require the collection of context information from several distribution units. Furthermore, in case of a consumer overload, the dissemination layer can benefit from methods that perform query prioritization or that can change the communication protocol in use to lower network traffic (a feature useful especially in vehicular and mobile ad-hoc network domains).

B) Transverse Context Management Functionality

Apart from the operational life cycle, a number of complementary blocks are usually included within the functionality requirements of a context management solution.

Context Producer Discovery refers to the ability of the context management system to detect new sensing resources that are dynamically coming online as well as to know when sensors become offline (either by choice or by failure). One method to achieve this feature is by means of creating an explicit yellow pages registration service (e.g. as in [Román et al., 2002]), which is known a-priori by new producers who can publish their capabilities in the lookup directory. Other mechanisms (e.g. [Gu et al., 2003]) involve employing protocols and frameworks usually used in software service discovery (e.g. jSLP[1], UPnP[2]).

[1] http://jslp.sourceforge.net/
[2] http://www.upnp.org/

Mobility Management seeks to provide support for frequent sensor and user registration and disconnection and offer handover capabilities for users or sensors on the move. This feature is important in order to ensure an optimal and continuous context consumer experience. Usually this support is enabled via the existence of a well defined proxy mechanism (such as the one used in SOLAR [Chen et al., 2008]) as well as by employing buffering and caching features for existing queries and subscriptions either on the management or the consumer client side.

Provisioning Adaptation/Control refers to the ability to perform structural (e.g. load balancing) or functional (e.g. activate/deactivate a reasoning component, change dissemination protocol) changes at runtime. This feature involves the oversight of the entire provisioning life cycle and requires the existence of a mechanism by which *current* and *intended* usage of context information can be monitored. Furthermore, such changes can be triggered based on quality of context (QoC) conditions (e.g. information freshness, accuracy) imposed by the context consumer. These requirements directly influence the sensors and reasoning algorithms that must be active, the rate at which sensors have to provide updates, etc. This is a challenging task which has received some attention (e.g. [Corradi et al., 2010; Juszczyk et al., 2009]) but which still requires exploring, especially from an ease of development point of view.

Whilst offering the functionality of the above listed blocks, a context management system should ideally exhibit the set of properties detailed in what follows.

3.1.2 Non-Functional Aspects

Henricksen et al. [Henricksen et al., 2005a] list a set of requirements for context management middleware solutions that enable context-aware application development. We do note however that these properties are meant for systems that strive to provide a general and more holistic approach to context management. Solutions that focus only on particular dimensions of context information (e.g. spatial awareness, device hardware monitorization) will naturally not exhibit the same management needs. The list of requirements is presented below.

Support for heterogeneity expresses the need to support interoperability of producers and consumers with different capabilities and different requirements (e.g. from resource-poor sensors to high-performance servers), as well as to try and expose standard interfaces exploitable through several application programming languages.

Support for mobility refers to the mobility management and context resource discovery functionality blocks that were detailed in the previous section.

Scalability specifies the ability of the context management architecture to support high volumes of produced information and incoming queries. This feature translates into engineering concerns that target both how the system can be deployed over multiple administrative domains, as well as how the internal reasoning and communication protocols are built to perform adequately in scenarios with high load.

Support for privacy and security establishes the need to allow for both system and user defined control settings that regulate the access to specific context information. Moreover, these settings themselves may be allowed to vary according to the current situation.

Traceability and Control expresses the requirement of being able to maintain transparency of the context management process. This means that the system should provide a reflection or monitorization mechanism by which it can maintain a history of produced context information and how/why it was acquired/derived. Furthermore, the decision making process and the information flow generating interactions should be controllable/configurable by the application level.

Fault tolerance and robustness refers to the requirement of maintaining adequate functionality in face of failures and disconnections of sensors or other components.

Ease of deployment/configuration states the need to alleviate application design and deployment effort by providing mechanisms and tools that allow developers to more easily (e.g. through a declarative approach) configure a context management system to meet user and environmental requirements.

As we have mentioned previously, these requirements normally apply to general context management solutions. However, Bolchini et al. [Bolchini et al., 2007a], being interested in the *data tailoring* problem [Bolchini et al., 2007b] (i.e. how to create the relevant data views depending on current context), argue that there is currently no "silver bullet" context management system suitable for all application scenarios. They put forth an analysis framework that distinguishes between five usages of context information as a matter of: channel-device-presentation, location and environment, user activity, agreement and sharing (among groups of peers) and selecting relevant data, functionalities and services.
From their analysis of existing context management systems it results that they believe that existing systems are either specialized for a certain category from the ones above, or are too general in their approach to be effective in several categories.
Nonetheless, we believe that setting out to build a more holistic context management middleware is beneficial as it can positively impact more application developers. Still, Bolchini et al. bring arguments that constitute additional support for one of our own main objectives: ease the development effort of context-aware applications through flexibility in configuration and deployment.
Indeed, we argue that in order for a general context management solution to be applicable to as many application scenarios as possible it has to be built in a modular way and give the developer the ability to customize the functionality and deployment of system components so as to be effective for the problem at hand.

In the following section, we begin our analysis of several of the existing context management systems, inspecting the way in which they implement the context provisioning process and how they address the accompanying functionality blocks. In Section 3.2 we further examine the various deployment and configuration options offered by these systems. To motivate the design choices taken for our own architectural approach, we will provide our own application scenario analysis framework, starting from that of Bolchini et al., to explain the need for flexibility in deployment options.

3.1.3 Context Provisioning Architectures

We start out by specifying the *class* of context management systems for which we will perform the analysis. Specifically, Bellavista et al. [Bellavista and Corradi, 2012] perform a review in which they analyze context management solutions from the point of view of their approach in context data distribution. For the dissemination layer they distinguish systems based on the considered type of network communication support. Thus, systems which use flooding or gossip-based communication protocols (e.g. in vehicular or mobile ad-hoc networks - VANETs and MANETs) are distinct from ones that benefit from an existing and stable network infrastructure (e.g. wireless or ethernet).
In this analysis we focus only on the latter type of approaches, because they address a larger number of possible applications and because our own approach assumes the existence of a stable communication network in which interactions can take place. We present the systems in an increasing order of the scale of scenarios they target and the architecture they use.

Centralized Approaches

COSMOS [Conan et al., 2007] targets managing the context of applications running on a single user device, such as tourist computer-based guides with contextual navigation or multi-player gaming applications that provide contextual annotations. The system collects device data (e.g. battery, processor, memory, open files) using the SAJE framework [Courtrai et al., 2003], thereby having a middleware-based access to context information. It represents the information using the object oriented paradigm and its most interesting contribution is the component-based development it offers. The solution employs an architecture description language (ADL) to specify different aggregation patterns for its context processing components, promoting reusability and sharing of components. Given the targeted applications, COSMOS does not address aspects of mobility, context discovery or access policies. Its greatest attribute comes from the ease of deployment and configuration being offered to engineers.

COPAL [Sehic and Dustdar, 2010] is a context management middleware (CMM) solution for smart home environments and its design principles are similar to those of COSMOS. The solution proposes device wrappers that expose data as web services to the COPAL processing units (middleware access). Data is represented as attribute-value pairs and aggregated using units that operate according to the complex event processing paradigm[1]. Consumers use the EPL (Event Processing Language) to access required information. The strong suit of COPAL lies again in its support for configuration and deployment, as the middleware proposes its own DSL (Domain Specific Language) to describe the functionality of processing units, queries as well as the deployment of a COPAL instance.

CoBrA [Chen et al., 2004a] is a context management middleware intended for use in scenarios like smart meetings and smart spaces in general. Acquisition and dissemination occur through use of web service interfaces, while representation and reasoning are a strong suit. CoBrA uses the SOUPA ontology [Chen et al., 2005] to model context information and a combined ontology and rule-based reasoning mechanism to support consistency and higher-level derivation of context. Another interesting feature is the ability offered to consumers to express context access policies using a subset of the SOUPA ontology.

SOCAM [Gu et al., 2005a] is a system that resembles CoBrA in terms of modeling and reasoning support. However, it offers much better assistance for sensor discovery by creating a distributed *service location service* [Gu et al., 2005b]. This service can also help SOCAM context interpreters to advertise their presence to context-aware consumers.

The CARE Middleware [Bettini et al., 2007] is intended to support adaptation of *continuous* services, such as those that perform media streaming, navigation support or publish/subscribe based services (e.g. a location based tourist application). Rather than general context management, CARE is focused on performing an aggregation and merging of *profiles* (set of context parameters and preference policy rules) coming from the users and operators of a particular continuous service. The merged profiles are then fed back to the service provider application, such that it may take a context-aware decision about how to adapt its servicing given the aggregated and conflict-free multi-consumer requirements.

While the above systems support the existence of several producers and consumers of context information, so far they perform reasoning and management in a single centralized component of their architecture. In what follows, we widen our view to approaches supporting multiple context administration domains, via a federated or decentralized context management architecture.

[1] http://esper.codehaus.org

Decentralized Approaches

The ACAI [Khedr and Karmouch, 2005] system proposes an agent-based context-aware infrastructure. The individual agents map to the different services that the system provides (e.g. system administration, entity coordination, reasoning, knowledge base access). Context information is maintained on a per entity basis in units called capsules (i.e. per user or consumer application) instead of globally. Ontology and rule-based reasoning ensure consistency of context information and derivation of higher level context within a capsule, while a collaboration protocol between the entity coordinator agent and the system administration agent ensure inter-capsule consistency. ACAI allows a spatial federation of context administration domains and uses a custom wrapper over the SIP[1] protocol to provide discovery of remote ACAI nodes and support intra- and inter-domain mobility of consumers.

SOLAR [Chen et al., 2008] defines a pervasive computing infrastructure built around a P2P network of processing nodes called Planets. Its strength lies in the extensive number of services provided by Planets (e.g. resource discovery, fault-tolerance, mobility management) as well as in its focus on reusable, distributed operators that use the filter and pipe paradigm to form DAG-like processing flows. Solar also provides configuration options for internal communication services that underpin its functionality, as well as XML-based definitions of operator compositions. Applications can also specify policies that control operator functionality (e.g. to prevent request buffer overflows) in order to adapt context data distribution.

CoCA [Ejigu et al., 2008] is a CMM solution used in a smart university campus scenario, attempting to support professors and students with their everyday work, like managing scheduled and spontaneously occurring meetings. CoCA uses a hybrid representation and reasoning mechanism. It uses a DBMS to handle efficient storing and querying of context data and ontology tools to reason over context knowledge (semantics of data). Queries to the system can then be submitted using the SPARQL query language. CoCA uses Collaboration Managers to form peer-to-peer networks, albeit with the purpose of offloading heavy computations from resource constrained devices (PDAs, smartphones) to more capable ones that are in the same peer group. The strongest feature, however, is that of runtime adaptability to required context information. CoCA uses a heuristic mechanism to engage the semantic reasoner only with those pieces of context that are relevant (from the whole domain), given the identity, location or activity of a user, leading to improved scalability and efficiency.

A different and larger scale scenario is considered in [Perera et al., 2012]. The work addresses the idea of managing context for the IoT related vision of sensing-as-a-service, like for instance in an application of monitoring the health of agricultural crops in all of Australia. CA4IOT is intended to solve the problem of finding and selecting the most appropriate (based on quality criteria) internet connected sensors to answer a user specified query. The solution does not pretend to offer a complete context management middleware functionality but offers a cloud-based architecture model that shows how CA4IOT might be used in existing middleware to handle such large scale context management scenarios. Its most important features are that of a scalable design and runtime adaptability with regard to user specified quality of sensing data demands.

3.1.4 Context Provisioning Summary

The presented systems each tackle scenarios of different complexity and focus on specific features to address these scenarios. As expected, all systems provide a corresponding method for the context management operational life cycle. However, depending on the targeted applications, not all systems insist on the transverse functionality blocks or try to address the non-functional requirements set we introduced in Section 3.1.2.

[1] http://www.voip-info.org/wiki/view/SIP

	COPAL	COSMOS	CA4IOT	CoBrA	SOLAR	ACAI	CoCA
Context Acquisition	(DA) Device Wrappers	(MA) SAJE Framework	(SA) GSN Wrappers	(SA) Web Service	(DA) Proxy in Planet	(MA) Context Provider Agents	(MA) Capture Tool Interface
Context Distribution	EPL lang.	-	Publish/ Subscribe	Query Web Service	INS/Twine	Context-Sensitive Comm. Protocol	Query SPARQL
Context Modeling/Reasoning	Attr.-Val+ CEP	OO+ Composition Patterns	Ontology Data Fusion Operator	Ontology + Rule Based	Attr-Val.+ Operator Composition	Ontology + Rule Based	Ontology + Rule Based
Context Consistency	-	-	-	Ontology Reasoning	-	Ontology Reasoning + Entity coordination	Ontology Reasoning
Mobility Mgmt.	-	-	-	-	P2P Proxy Mechanism + INS/Twine Discovery Mechanism	Modified SIP protocol	Collaboration Mgr. + JXTA P2P framework
Resource Discovery	-	-	Semantic Discoverer Layer	-	INS/Twine	Context Provider Agents	-
Middleware Adaptability	-	-	-	-	Buffering Policies + P2P overlay	CLA negotiation	P2P overlay + Heuristic Reasoner Usage

Table 3.1: Overview of context provisioning for reviewed systems: **-** (not addressed/mentioned), **DA** (direct sensor access), **MA** (middleware access), **SA** (context server access)

Table 3.1 shows a summary view of the way in which most of the works discussed previously approach the context provisioning tasks. What is immediately noticeable corresponds to the observation made at the end of Section 3.1.2: systems that are focused on *specific*, small or large scale scenarios (e.g. COSMOS, COPAL, CA4IOT) address only the provisioning concerns relevant for that scenario. On the other hand, systems that attempt a more general approach (SOLAR, ACAI, CoCA) end up building infrastructures that are too rigid to be applied to multiple application scenarios. Indeed, regarding adaptability, SOLAR allows applications to specify policies that control operator functionality, but the system focuses mostly on providing proper routing of context requests directly to the best provider, thus neglecting representation expressiveness and aggregation of context events into more complex situations such as the ad-hoc meeting example in our reference scenario.
ACAI has the benefit of agent-based negotiations and wrappers over the SIP protocol to ensure context provisioning control, but the work offers no configuration support and, as in the case of SOLAR, is very rigid in its infrastructure-focused deployment, being unsuitable for application scenarios that are lighter-weight or that require a different kind of context administration partitioning besides a spatial one.
CoCA uses heuristic methods to ensure loading of only relevant context data into an ontology-based reasoner. However, the heuristics cannot be controlled and the middleware offers no mean to adapt information flow.

The analysis made above is reflected also in Table 3.2 which provides an overview of how the reviewed works address the non-functional context management requirements. Thus, the main shortcomings of the above approaches is either their specialization for certain application types, either their lack of modularization and support for configuration and flexible deployment. We discuss these issues in more detail in the sections that follow.
In summary, it is important to note that while the subject of context provisioning has been addressed in many ways and with various auxiliary support, the engineering-related problems of ease of development and flexibility of deployment remain insufficiently explored.

	COPAL	COSMOS	CA4IOT	CoBrA	SOLAR	ACAI	CoCA
Heterogeneity	∼	-	+	+	∼	+	+
Mobility Support	-	-	-	-	+	+	∼
Scalability	-	-	+	-	+	∼	∼
History + Traceability	-	-	-	∼	-	-	+
Fault tolerance	-	-	∼	-	+	-	∼
Privacy + Security	-	-	-	+	-	-	-
Easy Deployment/ Configuration	+	+	+	-	∼	-	-

Table 3.2: Overview of requirement addressing for reviewed systems: **-** (not addressed/mentioned), ∼ (adequate support), + (strong support)

3.2 Deploying Context Management Solutions

In the previous section we discussed the basic context provisioning process and explored the means by which several works from the context management literature compose this process. We continue broadening our view of the context management problem by now considering the way in which the architectural units that compose the context provisioning life cycle of a context management system can be deployed within an application space (i.e. available hardware and software infrastructure). As we will see throughout the section, these aspects are closely related to the perceived level of support given to context-aware application engineers.
We start by listing the concerns we investigate in relation to context management system deployment. Next, we look at how these issues are addressed in the works we have reviewed previously. We then summarize our findings, analyzing the strong suits and shortcomings of inspected solutions, which motivate our own objectives.

3.2.1 Deployment Concerns

Each context management system enables context provisioning by defining a number of functionality modules and the interactions between them. The issue of context deployment relates to how the application designer or the runtime application itself can configure these modules and indicate how they are to be assigned to physical machines (nodes) that constitute the application space. Consequently, deployment concerns come into focus at two development moments, resulting in design-time and run-time related issues.

The first design-time issue regards the deployment architecture itself, meaning how the context provisioning modules envisioned by a context management system are distributed among different computing nodes and how the communication between them is implemented. We can distinguish between centralized, decentralized and peer-to-peer (P2P) architectures.
Within the centralized architectures we can differentiate between systems where all the provisioning modules are on the same machine (e.g. sensor wrappers, modeling/reasoning engine, query handlers) and those where context management (e.g. reasoning, access control) is implemented on a central machine, while the sensors and consumers may run on other machines, connecting to the management node via pre-specified communication protocols and service interfaces.
Among decentralized architectures we can again identify systems that provide a federation of context management nodes (i.e. a flat connectivity network) and systems that adopt a hierarchical approach, building tree or graph like connections between management or query nodes which can be exploited for information routing.
We categorize P2P systems apart from the decentralized ones because in addition to providing

a distribution of context provisioning elements, they usually also exploit known peer-to-peer mechanisms to distribute the context model. This allows such architectures to provide better scalability and faster information retrieval times at the expense of losing *locality* of context (i.e. events are no longer kept or used only in the spatio-temporal domain where they were produced).

A second deployment issue concerns both design-time and run-time moments. To support development of applications, context management systems must offer configuration options that specify where and how different context provisioning modules are deployed. Furthermore, the application should have runtime control over the life cycle of deployed modules (e.g. what sensor wrappers or reasoning mechanisms are active, whether adaptation mechanisms are running). As we will see in our analysis in Section 3.2.2, it is this aspect that is missing the most in related work in the literature.

To see why a modular design and deployment flexibility are important, let us consider a categorization of application scenarios based on the intended use of context information, as well as on the extent and "locality" of context producer, manager and consumer nodes. The categorization starts from Bolchini et al.'s [Bolchini et al., 2007a] analysis and enables us to see that different scenarios exhibit different context provisioning requirements, such that design-time configuration and runtime control of deployed modules become a necessity for systems attempting a holistic approach to context management. In what follows we introduce the considered scenario categories.

T1. Context Management for Devices of Personal Use This scenario groups together applications that monitor the context of usage of a personal computing device (smartphone, tablet, laptop) in terms of available resources (memory, CPU, WiFi connection, etc), open files, currently running processes and others. Applications like context-aware tourist tablets, or context-aware multi-player games on a laptop handled by COSMOS represent examples. Concerns such as scalability and mobility are not an issue in this case. However, ease of deployment and configuration is an important feature, since in such scenarios the application is usually interested in low-level context information and as such, facilitation of engineering the system and reusable design are required.

T2. Context Management for Personal Activities This category is closely related to the previous one. It deals with applications that gather the context data given by things like body-worn sensors, wearable electronics or sensors attached to a personal vehicle (e.g. bike, car). The purpose of these applications is to monitor the collected data and derive a personal health status or the current activity and mood of a user (e.g. COSAR [Riboni and Bettini, 2009]). Like in the previous scenario type, there is only one consumer of this information (the user), the application runs on the same physical device (e.g. smartphone) that does the collection and the type of sensors are likely to be known in advance, such that the earlier analysis applies here too.

T3. Spatial Domain Specific Context Management This third scenario type sees a change in scale, since the context data is no longer describing just a single user and, likewise, there may be more than one consumers of this information. The category includes the applications that showcase systems such as COPAL, CoBrA or CoCA. Depending on the size of the spatial domain, mobility support for decentralized architectures may be required. The kinds of situations that need detection in this type of scenarios are more complex, such that an emphasis on context reasoning and consistency is placed. Furthermore, the variability and number of sensor may be increased, such that aspects of context discovery and openness become important.

T4. Context Management in Large Scale Monitorization This scenario type includes applications from the smart city domain, with a specific focus on system-centred context management (monitorization of an environment is not used to determine the context of a user entity). An example is the crop health monitorization application presented in [Perera et al., 2012]. Variety and number of sensors is high, leading to required emphasis on expressive context modeling and reasoning. Given the large physical distribution of sensors, context discovery also plays an important role. However, there are usually fewer mobile context producers and consumers to account for, such that mobility and structural adaptability are not strictly necessary. Since the purpose is that of monitorization, the support for traceability and history of context information, as well as fault tolerance, are valued.

T5. Context Management for Wide Mobility The last scenario category addresses the applications that include context management for entities (users or devices) that have a large degree of mobility. It includes the user-centric applications from the smart city domain, like the one oriented towards intelligent tourism presented in [Da Rocha and Endler, 2012]. The large physical scale of this type of scenarios means the analysis carried out earlier above is valid here too. In addition, however, support for mobility management, context discovery or access policies become highly valued, because of the multi-producer, multi-consumer nature of the applications.

	T1	T2	T3	T4	T5
Heterogeneity	-	∼	+	+	∼
Mobility Support	-	-	+	∼	+
Scalability	-	-	∼	+	+
History + Traceability	∼	+	∼	+	∼
Fault tolerance	-	-	∼	+	+
Privacy + Security	∼	∼	+	+	+
Easy Deployment/ Configuration	+	+	+	+	+

Table 3.3: Requirements Analysis for Scenario Types: - (no obligation), ∼ (nice to have), + (must have)

Table 3.3 summarizes the above discussion by showing the typical non-functional requirements for each type of scenario. We constructed the table based on our own perception as well as based on a reverse engineering process which considered the manifested characteristics of the reviewed systems that were showcased using scenarios from a given category.
The table makes intuitive sense, but the most important idea worth noting from it is the one we have been emphasizing earlier: addressing scenarios of various scales requires modular architecture design and flexible deployment support.

3.2.2 Deployment Approaches

Following our scenario analysis framework, we review the configuration and deployment options offered by works introduced in Section 3.1 in the increasing order of scenario scale.

COSMOS [Conan et al., 2007] targets context management for applications running on a single user device, thereby overseeing only local production and consumption of context information. The strong suit of COSMOS is the support for declaratively specifying the computation patterns of its processing units (called context nodes). Using the FRACTAL [Bruneton et al., 2006]

architecture description language and related tools, a developer can specify different context node combination or sharing patterns inspired from object-oriented software design patterns (e.g. composite, factory method, singleton). The code for most context node components specified by a developer can thus be automatically generated, which eases application development.

COPAL [Sehic and Dustdar, 2010] has a similar component-based design philosophy. COPAL is targeted for use in smart home applications and, consequently, allows for actual sensing and consumption to occur on different machines (e.g. temperature sensors and user smartphones). However, all middleware components (e.g. device wrappers, context processors, context query listeners) are found on a single machine, leading to a centralized deployment. Nonetheless, as in the case of COSMOS, the most important feature of COPAL is support for configuration. COPAL defines its own domain specific language (DSL) to declaratively indicate all context management concerns: define types of context information, specify processor type, input and output, define queries and listeners for query answers. Specifications written by a developer using the COPAL-DSL are automatically deployed as an OSGi[1] bundle. The use of this service-component software engineering framework adds clear benefits from the ease of development point of view.

CoBrA [Chen et al., 2004a] is a typical exponent of a centralized solution for the management of context information in a single smart space environment. The context knowledge base, context reasoning engine and context acquisition modules of the broker architecture are all deployed on the same node. Interaction with both sensor and consumer layers occurs through SOAP[2]-based web service interfaces. Thus, deployment of CoBrA within an application occurs as a service provider for context information. However, no configuration means or development tools are detailed by the authors, such that practical use of the broker service becomes strongly tied to the application development cycle.

Previous works featured a centralized deployment. The following ones provide support for applications that need to access context information in a decentralized, federated or full peer-to-peer fashion.

As mentioned previously, the ACAI [Khedr and Karmouch, 2005] system targets the creation of a context management support infrastructure for spontaneous (i.e. on demand, ad-hoc usage) application, device and service interactions. ACAI supports a view of spatial federation of context administration domains wherein applications can, for instance, distinguish between a *home site* (e.g. the domain where the profile information of a user is always stored), a *current site* (e.g. context coming from the devices and services in the current user location) and a *remote site* (e.g. context coming from services located in a remote location but which involve the current user, such as a remote video conference). Each context domain is administrated by a Context Management Agent (CMA) and CMAs help different agents from within a domain perform inter-domain context provider lookups and communication via a modified version of the SIP protocol.
However, the authors of [Khedr and Karmouch, 2005] fail to mention any support for the configuration of domain deployments. Thus applications that want to use the ACAI context management infrastructure must firstly be compatible with the envisioned spatial administrative domain distribution and secondly must perform their integration with the system at coding level, which slows down development.

The Feel@Home project [Guo and Zhang, 2010], which was not discussed in Section 3.1.3, provides a similar approach to ACAI. It offers a context management infrastructure based on the same administrative domain idea, proposing the existence of a *home*, *office* and *outdoor* (mobile) domain. Like in ACAI, the Domain Context Managers (DCM) keep context information about entities (e.g. users, devices). Queries addressed to entities from remote domains are mediated

[1] http://www.osgi.org/
[2] http://www.w3.org/TR/soap/

by the Global Administration Server (GAS) which provides a heuristic entity lookup method to find the DCM with which the sought for entity is currently registered and which can respond to the requested context query. As in the case of ACAI, applications are not meant to configure or influence a Feel@Home deployment and must be compatible with the administration domain-based division of context management it proposes.

SOLAR [Chen et al., 2008] is also focused on providing a pervasive computing infrastructure, but attempts to do so in a pure P2P fashion. SOLAR builds a peer-to-peer network of context management service nodes called Planets. The system uses the INS/Twine [Balazinska et al., 2002] resource discovery framework to register both context providers as well as client queries. Using a defined naming scheme, the framework manages how information about which producer can provide which type of context data is distributed among the Planets. Operators that use the filter and pipe paradigm to form DAG-like processing flows can also be distributed among the Planets. Unlike the previous two systems, SOLAR offers deployment support by providing configuration options for internal communication services that underpin the functionality of a Planet, as well as XML-based definitions of operator compositions.

CA4IOT [Perera et al., 2012] is targeted at large scale context monitorization in Internet-of-Things (IoT) applications and the system relies on a cloud-based deployment. While the architecture modules performing context reasoning and dissemination cannot be influenced, CA4IOT provides extensive configuration support for sensor discovery and communication using *sensor device definition* (SDD) files. Furthermore, each type of user request, requiring the use of maybe thousands of sensors, receives its own *context and semantic discoverer* (CSD), a component that is customly generated at runtime to satisfy the specific user requirement. The main responsibility of a CSD is to collect sensor data, by communicating with the appropriate sensor wrappers generated based on the SDDs and bundling their response together to satisfy the user query.

The last system we explore differs from the ones analyzed above, in that it is not explicitly conceived as a context management system. Rather, the tAtAmI platform [Olaru et al., 2013] is a development framework for multi-agent systems as a general middleware for AmI applications. However, the platform proposes an interesting hierarchical mechanism for deploying agents whose execution cycle and interactions depend on the context of the user and the application. In [Olaru, 2011], the author explains that the topology of the system should be induced by context, i.e. apart from the physical distribution over a machine network, the hierarchy of agent relations is established depending on whether two agents share context. The author uses a categorization of context information (e.g. spatial, computational, activity, social) to specify the possible relations (e.g. is-in, executes-on, part-of) that define the neighbourhood relationship of agents.
Inspired by the work in [Olaru, 2011], in Section 4.1.3 and throughout Chapter 6 we will show how the introduction of explicit, model related concepts for the structuring of context management unit deployment provides assistance in terms of dynamically using the needed context processing functionality.

3.2.3 Deployment Summary

Deployment options directly influence application development support in terms of type and scale of achievable scenarios. As mentioned, our scenario requirements analysis from Section 3.2.1 was based on an inductive process which used the features available in context management systems used for a specific application to determine the necessary characteristics for that application type. Table 3.3 showed that different requirements arise as the scale and intended usage of application context change.
Given that one of our main goals is to build a context management middleware able to support development of multiple application scenarios, the above issue translates into a problem

of configuration assistance and flexible context provisioning module deployment.

In decentralized and distributed architectures, some of the important problems relate to how context provisioning units are connected (in which ways can information flow) and how is addition/activation of context provisioning units controlled. Yet, as our literature review from Section 3.2.2 shows, only centralized approaches like COSMOS or COPAL offer the ability to declaratively specify such configurations. Little support for this is offered in works adopting decentralized architectures, even though that is the case where the need is greater.

Furthermore, another aspect that can be observed in the presented works is that they lack a means to tie the distribution of their provisioning units to the context model used by the applications running on top of them. Indeed, approaches like ACAI or Feel@Home propose constructing context management domains based on an abstract definition of administration units (e.g. home, office domains), mostly guided by a spatial distinction. This means that application developers can not tailor the usage of various context provisioning units based on other context information dimensions such as activity or organizational relations.

These shortcomings influenced the objectives that were set for our own proposal of a context management middleware architecture, which are presented in the following section.

3.3 Our Context Management Objectives

Throughout this chapter we have discussed the meaning of context provisioning, the operational and non-functional challenges this process poses, as well as the different approaches to deploying a context management system, given the architectural choice.
We have also seen that the main shortcomings discussed in the sections on both context provisioning and deployment summed up to insufficient means for alleviating application development (e.g. binding deployment to context model, declarative configuration support for provisioning support and provisioning adaptation).
Thus, given these issues and our own goals, we lay out the following set of objectives as guidelines for the context management middleware we are proposing:

1. Propose a middleware architecture based on the agentification of context provisioning units (sensing, coordination, dissemination, usage). Use of design principles and tools from the multi-agent system domain allow for a better encapsulation and the *potential* for increased autonomy of individual provisioning units. This addresses the problem of modularization of context provisioning concerns.

2. Provide a declarative means for specification of application-level control over the context provisioning process. This amounts to creation of policies that govern the behavior of the agent-based provisioning units, leading to a reduced development effort.

3. Using the agent-based architecture and a component-oriented software model, provide two deployment schemes: a centralized one and a decentralized one supporting a tree-like hierarchical relation structure. This addresses usability of the middleware for multiple scenario types.

4. Provide a configuration vocabulary declaring the desired structure of context provisioning deployment. The aim is to feature additional development support, since all provisioning and deployment related issues can be declaratively specified by a programmer.

5. Define concepts that use the dimensionality of a context model (i.e. the types of modeled context information) to induce the *structure* of context provisioning units. The idea is to provide a means to create context *domains* starting from the domain of discourse considered by the application and to then create a mapping between a context domain and

the set of context provisioning units that are required to service the context management needs of that domain.

6. Exploit a component-based software engineering method to create support for runtime management of context provisioning unit life cycles. This again targets the idea of flexibility in deployment and provisioning by adding an additional mechanism for the application level to control these aspects during execution.

The above listed objectives are meant to provide adequate support for addressing all the context management needs mentioned in this chapter.
Having finished our overview of related works from the literature and the lessons we can learn from these solutions, with the next chapter we begin a new part of this thesis. In it, we will again take views of increasing wideness that go through the concepts underpinning our context modeling and system architecture approaches.

Chapter 4

Representing and Reasoning About Context

We begin the incursion into our contributions with the foundational aspect of context modeling. This chapter presents our approach to representing and reasoning about context information. In Section 4.1 we formalize the model in terms of representation concepts, reasoning methods related elements. They will enable us to provide a design structure for a context-aware application.
Section 4.2 presents the implementation of our model using an ontology-based vocabulary.
In Section 4.3 we outline our context inference approach as a combination of ontology and rule-based reasoning to ensure consistency of information and derivation of higher-level knowledge. The architecture and execution cycle of the reasoning engine that leverages our representation and reasoning methods is covered in Section 4.4.
We conclude this chapter with a summary in which the modeling contributions are analyzed with respect to the requirements introduced in Sections 2.1.1 and 2.2.1, as well as with regard to our own general objectives stated in the introduction.

4.1 CONSERT Context Formal Model

In chapter 2 we argued that context *meta-models* were the most flexible approach with respect to the ability to address all the necessary context modeling requirements (cf. Section 2.1.1). We therefore choose to follow a similar path and improve upon existing approaches by focusing on uniformity and extensibility of our context meta-model.
In this section we formalize the CONSERT (an abbreviation from CONtext asSERTion) meta-model, by defining all model concepts and how they are used in the rule-based reasoning approach we propose. Along the way, we exemplify the introduced notions on hand of the reference scenario given in Section 1.3.

4.1.1 Representation Concepts

The CONSERT meta-model comprises different elements that are related to each other as shown in Figure 4.1. Before giving a formal definition as well as properties of each element, let us give a global overview of this model.

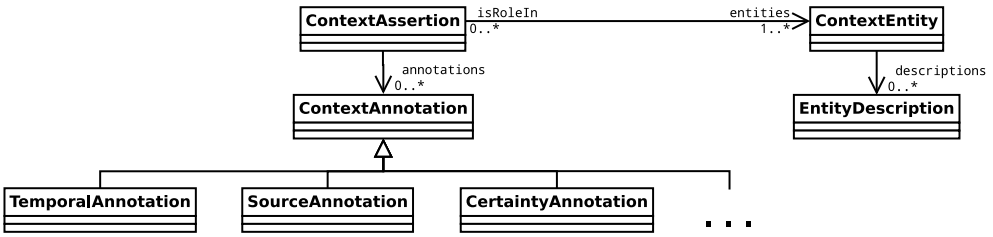

Figure 4.1: The defining concepts of our proposed context meta-model.

Overview

A *ContextAssertion* represents the basic construct used to describe the situation of entities (e.g. a "person", a "place", an "object") "which are considered relevant for the interaction between a user and an application" (in the sense of Dey's definition of context [Dey, 2001]). Each assertion involves one or more *ContextEntities*. Consider this example from our reference scenario.

Example 4.1.1. The two *ContextEntities*: *Person*(`alice`) (i.e., Alice is a person) and *LaboratoryRoom*(`ami_lab`) (i.e., ami_lab is a laboratory) may be related together through a *ContextAssertion* describing the situation of being located in some university room by the following expression: `locatedAt(alice, ami_lab)`.

ContextAssertion and a *ContextEntity* may be further characterized by *ContextAnnotation* and *EntityDescription* respectively.

A *ContextAnnotation* is a meta-property of a *ContextAssertion* and relates to information such as the source (author of the statement), the timestamp of its generation, the validity of the statement (time intervals for which the assertion is considered to be true) or the certainty with which the assertion is affirmed. Our model does not impose a limit on the type of possible *ContextAnnotations*. Other more complex properties can be imagined such as ownership (one or more entities which "hold control" of an assertion), access control (who is allowed to query or access the value of the assertion) or others.

An *EntityDescription* represents *static* information (i.e. does not vary with time) that provides additional characterization of a *ContextEntity* (e.g. spatial inclusion and distance relations, temporal relations, descriptive properties). It therefore holds between a *ContextEntity* and a literal value.

Example 4.1.2. Consider as before modeling the location of a user as `locatedAt(alice, ami_lab)`. We further know that `includedIn(ami_lab, cs_building)` and `hasCoordinates(cs_building, geoCoordinates)`, where *Person*(`alice`), *SpatialStructure*(`ami_lab`, `cs_building`) are *ContextEntities* and `geoCoordinates` is a literal. The statement `locatedAt` represents a *ContextAssertion*, having dynamic value changes for each individual person, whereas `includedIn` and `hasCoordinates` are modeled as *EntityDescriptions* since they only provide additional static descriptions of a *SpatialStructure* instance.

Notations and Definitions

In order to properly introduce the model formalization we consider the following notations. First, we call E the set of all *ContextEntities* that are considered within a model instance. We then define V as an infinite set of variables, disjoint from E, and L as a set of literals, disjoint from $E \cup V$. Further, to introduce the *values* for *ContextAnnotations* we give the following definitions for *Annotation Domains*.

Definition 4.1.3 (Annotation Domain)**.** An *annotation domain* d is an idempotent, commutative semi-ring $\langle A_d, \oplus, \otimes, \bot, \top \rangle$ such that:

- A_d is a set (of *annotation values*);
- $\oplus$ is idempotent, commutative, associative;
- $\otimes$ is commutative and associative;
- $\bot \oplus \lambda = \lambda$, $\top \otimes \lambda = \lambda$, $\bot \otimes \lambda = \bot$, and $\top \oplus \lambda = \top$;
- $\otimes$ is distributive over $\oplus$, *i.e.*, $\lambda_1 \otimes (\lambda_2 \oplus \lambda_3 = (\lambda_1 \otimes \lambda_2) \oplus (\lambda_1 \otimes \lambda_3)$.

This definition of a *ContextAnnotation domain* A_d is inspired from work on annotated RDF [Zimmermann et al., 2012]. As we have mentioned in Section 2.2.6, one of the shortcomings of existing rule-based context reasoning approaches is that they do not provide a clear mechanism for obtaining annotations of a *derived* context statement from a combination of the annotations of the rule premises. Our choice for the above given form relies on the ability to obtain this structured way of combining *ContextAssertions* during rule based inferencing by making use of the $\oplus$ and $\otimes$ operators.

In what follows, we briefly present the form of the annotation domains we chose for the common settings (timestamp, time validity, certainty) which we discussed earlier in this section (partly adopted from [Zimmermann et al., 2012]):

Annotation	Value set	Algebraic form
timestamp	set of time points $\cup \{-\infty, +\infty\}$	$\langle A_{timestamp}, max, min, -\infty, +\infty \rangle$
validity	set of sets of pairwise disjoint time intervals	$\langle A_{validity}, \cup, \cap, \emptyset, [-\infty, +\infty] \rangle$
certainty	$[0, 1]$	$\langle A_{certainty}, max, min, 0, 1 \rangle$

In the current version, the source annotation is kept simple (a URI identifying a service or actor that produces a *ContextAssertion* instance, or one that performs a derivation using our reasoning engine) and not modeled as a structured annotation ($\oplus$ and $\otimes$ are not defined). In section 4.2.2 we show it to be a subclass of a BasicAnnotation.
The other provided annotation domain definitions have an intuitive interpretation of their respective $\oplus$ and $\otimes$ operators.

The timestamp domain considers time points as its vocabulary. The additional $-\infty$ and $+\infty$ are used to complete the formal definition of the timestamp vocabulary. Based on the natural ordering of timestamps, *max* and *min* play the roles of $\oplus$ and $\otimes$.

The temporal validity domain vocabulary consists of consecutive disjoint time intervals, where each end of an interval is a time point or $-\infty$ or $+\infty$. As noted in the examples, the roles of $\oplus$ and $\otimes$ are played by $\cup$ and $\cap$ respectively.

Lastly, our definition for the certainty annotation domain uses decimal values from the interval $[0, 1]$ as vocabulary for expressing certainty values. Using the natural order of real numbers, we can use *max* and *min* as the $\oplus$ and $\otimes$ operators. Note however that this choice is not unique, as another option for $\otimes$ could be the multiplication operator $\times$ for real numbers.

As mentioned previously, the purpose of this algebraic form for annotation domains is to be able to define an *usage semantics* for the combination of *ContextAnnotation* values during inference. Following the observation in [Zimmermann et al., 2012], we use $\oplus$ to combine annotation information for the same *ContextAssertion instance*, whereas $\otimes$ models the conjunction of annotation statements from *ContextAssertions* of different *types*. To exemplify, consider the following sim-

ple demonstration from the domain of temporal validity:

Example 4.1.4. Let $f : \{[12{:}00,\ 12{:}05]\}$ and $f : \{[12{:}03,\ 12{:}08]\}$ be two instances of the same *ContextAssertion* with different temporal validity.
We infer $f : \{[12{:}00,\ 12{:}05] \cup [12{:}03,\ 12{:}08]\} = f : \{[12{:}00,\ 12{:}08]\}$, where $\cup$ plays the role of $\oplus$.

On the other hand, when dealing with a conjunction of statements with different type, a similar example to the previous one gives:

Example 4.1.5. Let $f_1 : \{[12{:}00,\ 12{:}05]\}$ and $f_2 : \{[12{:}03,\ 12{:}08]\}$ be two different *ContextAssertions*.
We can infer $f_1 \wedge f_2 : \{[12{:}00,\ 12{:}05] \cap [12{:}03,\ 12{:}08]\} = f_1 \wedge f_2 : \{[12{:}03,\ 12{:}05]\}$, where in this case $\cap$ plays the role of $\otimes$.

While the above definition and examples give us the semantics and operational properties of a *ContextAnnotation domain*, in a concrete implementation, the values of a *ContextAnnotation domain* will be based either on existing (e.g. RDF datatypes) or customly created datatypes (more on this in Section 4.2.2). For this reason, we introduce below the definition of *concrete ContextAnnotation domain*.

Definition 4.1.6 (Concrete annotation domain)**.** A *concrete annotation domain* is a pair $\langle d_1, d_2 \rangle$ such that d_1 is a datatype and d_2 is an annotation domain, and the value space of d_1 is equal to the set of annotation values of d_2.

Formalization

With the previously explained definitions and notations we are now ready to provide the formalization for all the concepts introduced in the model overview. We start with the definition of a *ContextEntity*, which is based on the one of Dey [Dey, 2001], and with that of a *ContextAnnotation*.

Definition 4.1.7 (ContextEntity)**.** *A ContextEntity is any physical, virtual or conceptual element that is considered relevant to the interaction between a user and an application, including the user and the application themselves.*

Definition 4.1.8 (ContextAnnotation)**.** A *ContextAnnotation* is a pair $\langle lf, cad \rangle$, where lf and *cad* are abbreviations for *lexical form* and *concrete annotation domain*. We denote by $\mathcal{A}$ the set of all annotations.

As discussed in the model overview, *ContextEntities* play roles in a *ContextAssertion* and can be characterized by *EntityDescriptions*. These properties can be seen as *predicates* defining relations between entities. We note with P the set of all predicates, disjoint from E, V, L and $\mathcal{A}$, and characterize each predicate $p \in P$ by its arity $\text{ar}(p) \in \mathbb{N}$. Following this, the definitions of *EntityDescription* and *ContextAssertion* are given below.

Definition 4.1.9 (EntityDescription)**.** An *EntityDescription* is a formula of the form $D(x, y)$ where $D \in P$ is a binary predicate, $x \in E \cup V$ and $y \in E \cup V \cup L$.

Definition 4.1.10 (ContextAssertion)**.** A *ContextAssertion* is a formula of the form $p(x_1, \ldots, x_n) : \{\lambda_1, \ldots, \lambda_m\}$ where $n, m \in \mathbb{N}$, $p \in P$ with $\text{az}(p) = n$, $x_i \in E \cup V \cup L$ for $[1, n]$ and $\lambda_j \in \mathcal{A} \cup V$ for $j \in [1, m]$. We denote by $\mathcal{F}$ the set of all context assertions.

Finally, to help us explain the reasoning formalism in the next section, we introduce the following functions:

$$\begin{array}{rccc} \textit{entities} & : & \mathcal{F} & \rightarrow & 2^{E \cup V \cup L} \\ & & p(x_1, \ldots, x_n) : \{\lambda_1, \ldots, \lambda_m\} & \mapsto & \{x_1, \ldots, x_n\} \end{array}$$

$$\begin{array}{rccc} \textit{annotations} & : & \mathcal{F} & \rightarrow & 2^{\mathcal{A} \cup V} \\ & & p(x_1, \ldots, x_n) : \{\lambda_1, \ldots, \lambda_m\} & \mapsto & \{\lambda_1, \ldots, \lambda_m\} \end{array}$$

The function *entities* retrieves the set of all *ContextEntity* or *Literal* instances that play a role in a given *ContextAssertion* p. The *annotations* functions works similarly and gets the set of *ContextAnnotation* instances that characterize the *ContextAssertion* p.

4.1.2 Reasoning Formalism

In Section 2.2.2 we explained that rule-based reasoning approaches are most widely used given their advantage of being simple to define and to extend. Since this aligns also with our objective of alleviating application development effort, we choose to employ a similar reasoning method. In what follows we present the formal model for a rule-based inference approach which uses the representation concepts defined previously to create expressive conditioning over both content and meta-properties of context information.

The CONSERT model uses *ContextDerivationRules* as a *deduction* method that expresses conditions over *EntityDescriptions*, *ContextAssertions* and *ContextAnnotations* in order to obtain higher-level *ContextAssertions*. Each *ContextDerivationRule* is made up of a *head* (the deduced *ContextAssertion*) and a *body* which contains the condition statements required for the rule head to be deduced.
The head of a derivation rule ρ is a *ContextAssertion* $P(x_1, \ldots, x_k) : \{\lambda_1, \ldots, \lambda_l\}$ where $x_i \in E \cup V \cup L$ and $\lambda_j \in \mathcal{A} \cup V$. Notice that $\textit{entities}(P)$ and $\textit{annotations}(P)$ can include variables which will be bound during the reasoning process.
The body consists of so called *ConditionExpressions* (detailed later in this section) and constrained forms of universal and existential quantification.
We next introduce three auxiliary functions that help us to better present the formal definition of a *ContextDerivationRule* ρ. Let $\mathcal{R}$ be the set of *ContextDerivationRules*. The function $\textit{head} : \mathcal{R} \rightarrow \mathcal{F}$ retrieves the head of a rule, i.e. the *ContextAssertion* that is inferred. Similarly, the function $\textit{body} : \mathcal{R} \rightarrow 2^{\mathcal{F}}$ retrieves the set of all *ContextAssertions* contained in the body of a *ContextDerivationRule*. Lastly, the function $\textit{constraint} : \mathcal{R} \rightarrow \mathcal{F}$ gets the *ContextAssertion* that provides the expression for the constrained universal or existential quantification.

With these notations, the definition of a *ContextDerivationRule* is the following.

Definition 4.1.11 (Context Derivation Rule)**.**

$$\begin{array}{lll} \rho : P(x_1, x_2, \ldots, x_k) : \{\lambda_1 \lambda_2, \ldots, \lambda_l\} \leftarrow \textit{body} & & \text{where body may be:} \\ \quad \textit{ConditionExpr} & & \\ \textit{or} \quad \exists y_1, \ldots y_r \bullet P_c(z_1, \ldots, z_m) : \{\lambda_1, \ldots \lambda_p\} \bullet \textit{ConditionExpr} & & (EQC) \\ \textit{or} \quad \forall y_1, \ldots y_r \bullet P_c(z_1, \ldots, z_m) : \{\lambda_1, \ldots \lambda_p\} \bullet \textit{ConditionExpr} & & (UQC) \end{array}$$

where $y_i \in V$, $Y_\rho = \{y_1, \ldots y_r\} \subseteq \textit{entities}(\textit{constraint}(\rho)) \cup \textit{annotations}(\textit{constraint}(\rho))$ and $\textit{entities}(\textit{head}(\rho)) \cap \textit{entities}(\textit{constraint}(\rho)) \neq \emptyset$.

In the above rule, EQC (resp. UQC) refers to a *constrained* existential (resp. universal) quantification. The constraint comes from the fact that the quantification does not refer to all possible variable values, but only to those which make the constraining *ContextAssertion* P_c true. That is, at runtime, only those values for y_i are selected for which instances of P_c

(instantiated with those values) *can be found in the knowledge base* of the system running the rule. From this it follows that:

- In the existential case, at least one value assignment for each y_i has to also observe the conditions set in $ConditionExpr$.
- In the universal case, all possible value assignments have to do so.

Additionally, Y_ρ and all the variables that appear in the rule head ($entities(head(\rho))$, $annotations(head(\rho))$) must also appear in $ConditionExpr$, which we discuss next.

Definition 4.1.12 ($ConditionExpr$)**.** A *ConditionExpression* contains a domain expression ($DomExpr$) and an annotation expression ($AnnExpr$) as follows:

$$
\begin{aligned}
ConditionExpr &::= DomExpr \wedge AnnExpr \\
DomExpr &::= ComExpr \mid DomExpr \wedge ComExpr \\
ComExpr &::= SimExpr \mid AggExpr \\
SimExpr &::= AssertExpr \mid \neg AssertExpr \mid DescExpr \mid \neg DescExpr \mid TermExpr \\
AggExpr &::= aggregate(FuncExpr, FilterExpr, ResExpr) \\
FilterExpr &::= SimExpr \mid FilterExpr \wedge SimExpr \\
AssertExpr &::= P(x_1, \ldots, x_n) : \{\lambda_1, \ldots, \lambda_m\},\ x_i \in E \cup V \cup L,\ \lambda_j \in A_{d_j} \cup V \\
DescExpr &::= D(x, y),\ x \in E \cup V,\ y \in E \cup L \cup V
\end{aligned}
$$

Definition 4.1.13 ($DomExpr$)**.** A $DomExpr$ is a conjunction of positive or negated *ContextAssertions* ($AssertExpr$) and *EntityDescriptions* ($DescExpr$), term expressions ($TermExpr$) and aggregations ($AggExpr$).

Definition 4.1.14 ($TermExpr$)**.** Term expressions contain terms $t \in E \cup L \cup V \cup \mathcal{A}$ which are entities, literals, variables or annotations. Terms can be related by boolean comparators ($>$, $<$, $\geqslant$, $\leqslant$, $=$, $\neq$), logical connectors ($\wedge$, $\vee$, $\neg$) and system or user-defined functions $func(t_1, \ldots, t_n)$. Functions in term expressions act as predicates which return a truth value when all their arguments are bound. If the arguments contain free variables, the function call binds them to values that make the function true.

Definition 4.1.15 ($AggExpr$)**.** An *aggregation expression* contains three subexpressions: $FuncExpr$, $FilterExpr$ and $ResExpr$. The $FuncExpr$ is a list of one or more aggregation functions that take a single variable as their argument $[aggFunc_1(z_1), \ldots, aggFunc_k(z_k)]$. We employ the typical aggregation functions: $aggFunc \in \{count, sum, avg, min, max\}$. $FilterExpr$ is the expression used to condition the values of the variables z_i over which we perform the aggregation. It takes the form of a conjunction of $SimExpr$. Therefore, all variables z_i must occur in *ContextAssertions*, *EntityDescriptions* or *ContextAnnotations* contained in $FilterExpr$. Finally, $ResExpr$ is a list of variables $[aggRes_1, \ldots, aggRes_k]$ which will store the result of the k $aggFunc_i(z_i)$ functions.

Definition 4.1.16 ($AnnExpr$)**.** An *annotation expression* is a conjunction of functions of the form $f_j(\lambda_{j1}, \ldots, \lambda_{jq})$ where each function f_j binds a free variable $\lambda_j^{head} \in annotations(head(\rho))$ (the annotations of the *ContextAssertion* in the rule head). All λ_{jk} and λ_j^{head} belong to the same annotation domain A_{d_j} and we additionally know that λ_{jk} belongs to the annotations of some *ContextAssertion* in $ConditionExpr$. The functions f_j are user-defined and can either directly bind λ_j^{head} to a value from the annotation domain A_{d_j}, or they can determine their output by computations using the $\oplus$ and $\otimes$ operators specific to annotation domain A_{d_j}.

These definitions conclude the formalization of the reasoning approach. To get a better sense of the presented concepts, let us illustrate them along an example of a *ContextDerivationRule* used in the reference scenario to derive the occurrence of an ad hoc discussion in the AmI Laboratory.

Example 4.1.17.

```
1  hostsAdhocMeeting(RL):{λ_src, λ_t, λ_valid, λ_acc}:
2    isA(K, camera) ∧ isA(Mic, microphone) ∧
3    deviceLocatedAt(K, RL) ∧ deviceLocatedAt(Mic, RL) ∧
4    makeInterval(now()-5,now(),λ_interv) ∧
5
6    aggregate([count(S),avg(λ_accS)],
7      sensesSkelInPos(K,S,sit):{λ_validS, λ_accS}
8      ∧includes(λ_validS, λ_interv),[Ct,avgAccS]
9    ) ∧
10   Ct⩾3 ∧ λ_avgAccS ⩾0.75 ∧
11
12   hasNoiseLevel(Mic, NL):{λ_validMic, λ_accMic} ∧
13   includes(λ_validMic, λ_interv) ∧ NL⩾60 ∧ λ_accN ⩾ 0.75 ∧
14
15   assignAcc(λ_acc, λ_AvgAccS ⊗ λ_accN) ∧
16   assignSrc(λ_src, currentAgent) ∧
17   assignTimestamp(λ_t, now()) ∧
18   assignValid(λ_valid, {[now()-5, now()]})
19
```

In the above example, lines 3 - 14 constitute instances of *DomExpr*, while lines 16 - 19 are instances of *AnnExpr*.
Among the *DomExpr* instances, one can observe examples of both *EntityDescriptions* (lines 3 and 4), as well as *ContextAssertions* (line 13). Furthermore, one can notice different types of *TermExpr*, including boolean comparisons (line 11) and function calls which can both define values for new variables (e.g. in line 5) or return a truth value for a computed evaluation (e.g. line 14).
Lastly, there is also an example of an *AggExpr* (lines 7 through 10), which is used to compute the number of skeletons which are detected as being in the `sit` posture, as well as to get an estimate of the average accuracy of the detection.

With regard to the annotation expressions (*AnnExpr*), the reader can observe both types of possibilities discussed in Definition 4.1.16. Line 16 contains an instance of an annotation assignment that makes use of the corresponding $\otimes$ operator to compute the resulting value (in our case the resulting annotation based on the certainty degree for observing sitting postures and the given noise level).
Lines 17 - 19, on the other hand, show examples of the ability to directly set the value of the *ContextAnnotations* for the derived *ContextAssertion* instance.

Remember from the objectives outlined in Section 2.3 that we mentioned the intention to use techniques of the semantic web to implement our rule-based reasoning approach, on account of achieving uniformity and expressiveness. In Section 4.3 we show how the form of *ContextDerivationRules* defined here can be implemented using the SPARQL[1] query language.

4.1.3 Context Dimensions and Context Domains

Before going over to detail the implementation of the CONSERT context meta-model using semantic web technologies we want to introduce two additional concepts which will be later used

[1] http://www.w3.org/TR/rdf-sparql-query/

in Chapter 6 to obtain the different deployment schemes of our proposed context management middleware.
Specifically, we want to introduce concepts that can help a developer design his/her context-aware application by exploiting the natural dimensionality of the context model that he defines, thus addressing one of the shortcomings of related work discussed in Section 3.2.3. To do this, we take inspiration from the work of Zimmermann et al. [Zimmermann et al., 2007], who in the attempt to provide an operation-oriented view of context modeling, present a categorization of context information. Starting from Dey's view, that context defines situations of an entity [Dey, 2001], they consider the following five categories:

- Individuality (I) - describe an entity itself
  - Natural Entities
  - Human Entities
  - Artificial Entities (e.g. sensors and the information they sense)
- Space (S)
- Time (T)
- Activity (A) - what tasks an entity might be involved in
- Relations (R) - any kind of relation that can be established between two entities
  - Functional Relations
  - Compositional Relations
  - Social Relations

As given in [Dey, 2001], context is "Any information that can be used to characterize the situation of entities (i.e. a person, a place or an object) that are considered relevant to the interaction between a **user** and an **application**, **including the user and the application themselves**".
Notice that in Dey's definition we have bolded the words "user" and "application" as being the most representative *entities* of the context model of an application. Indeed, from the above definition we may consider that context-aware applications are either:

- **user-centric**: the majority of context statements describe and analyze the situations of one or more physical or logical users (e.g. an agent, an organization)
- **application-centric**: the majority of context statements describe situations of the internal runtime environment (i.e. application-level introspection) of an application

This means that either user-related entities (e.g. user, groups, organizations) or application-related entities (e.g. device, platform, service) will be the ones that ultimately relate all context information.

Having considered all the above and using the modeling concepts defined in the previous sections, let us now provide a simple formalization of the notion of a *context model* C of a particular application. This formalization will then allow us to express how a context model can exhibit a logical partitioning into *usage domains*.
For a particular application, let us consider CE as the set of all *ContextEntities*, ED as the set of all *EntityDescriptions* and CA as the set of all *ContextAssertions* defined by a developer for use in that specific application. Furthermore, let us denote by U and App the sets of *ContextEntities* included in CE which are respectively user and application-related.

We may now express the *context model* C of an application as the following set-theoretic union:

$$C = U \cup App \cup \bigcup_{cat \in \{I,S,T,A,R\}} (CE_{cat} \cup CA_{cat} \cup ED_{cat})$$
$$CE = U \cup App \cup CE_I \cup CE_S \cup CE_T \cup CE_A \cup CE_R$$

In the above, the indexes I, S, T, A and R correspond to the five context categories described previously (individuality, space, time, activity and relations).

With the idea of context categories we are now ready to define the concepts that establish the mentioned logical structuring of context model provisioning: *ContextDimension* and *ContextDomain*.

Definition 4.1.18 (ContextDimension)**.** A *ContextDimension* is a *ContextAssertion* P belonging to the category *cat* that defines the dimension: $P \in CA_{cat}$ where $cat \in \{I, S, T, A, R\}$. Properties of P:

- $arity(P) = 2$
- P of the form: $P(E_{subj}, E_{obj})$, or otherwise $entities(P) = E_{subj} \cup E_{obj}$, where $E_{subj} \in U \cup App$ is the subject entity and $E_{obj} \in CE_{cat}$ is the object entity with values belonging to the category defining the dimension (cat).

In general, in context-aware applications the *ContextAssertion* that plays the role of a *ContextDimension* can be understood as a *privileged direction* (e.g. spatial location, user activity, organizational relation) along which the application will structure its context provisioning process. The subject part of a *ContextDimension* is a *ContextEntity* which can generally be regarded as a *consumer* of context information (e.g. a human user). The object parts, on the other hand, are instances of *ContextEntities* which define *values* along the *ContextDimension*, representing what we call *ContextDomains*, which we introduce next.

Definition 4.1.19 (ContextDomain)**.** A *ContextDomain* establishes a *logical partition* of the global application context model, along the chosen *ContextDimension*. The values of a *ContextDomain* come from those of the *ContextEntity* that plays the object role in a *ContextDimension*. A formal way to define this partitioning is based on the following.
Let $\overset{*}{\mapsto}$ be a function from $\mathcal{F} \to 2^{\mathcal{F}}$ with the following definition:

$$a \in \mathcal{F}, a \overset{*}{\mapsto} \{b | b \in \mathcal{F}, entities(a) \cap entities(b) \neq \emptyset\}$$

Let $P_{dim}(E_{subj}, E_{obj})$ be the *ContextAssertion* that gives the *ContextDimension*. Let *cat* be the context category from which the *ContextDimension* was chosen and *dimval* the current value of E_{obj}, $dimval \in CE_{cat}$ (the object value of the *ContextAssertion* has a fixed value). Then the **ContextDomain** CD is defined as the ***closure under*** $\overset{*}{\mapsto}$ of $\{P_{dim}(E_{subj}, dimval)\}$.

Example 4.1.20. Let us turn to our reference scenario to give some examples that clarify the above defined notions. Alice, who is sitting in the AmI laboratory is currently attending the CS lecture. Her smartphone is consuming context information regarding the current room, as well as her current activity. The context-aware application on her smartphone receives updates about context in the AmI-Lab (e.g. her logical position in the room: in the presenter area, near a desk) from the context management system installed in the lab, while the activity-related information is obtained from a server of the CS building of the university. Furthermore, Alice is only engaged with the context data mentioned above, when she is in the AmI laboratory, or when she attends the CS lecture respectively. Likewise, her friends Bob and Cecille are only subscribed to notifications to ad-hoc meetings for the duration of their discussion in the AmI laboratory.

Thus, we can see that Alice currently has two distinct *usages* of context information: one from an `ami-lab` *ContextDomain* and one from a `cs_lecture` *ContextDomain*. `ami-lab` is a value of the `UniversitySpace` *ContextEntity* ($\in CE_S$) and `cs_lecture` is a value of the `TeachingActivity` *ContextEntity* ($\in CE_A$).
These entities play the object role in a spatial (`locatedAt(Person, UniversitySpace)`) and an activity-based *ContextDimension* (`engagedIn(Person, TeachingActivity)`), such that the above two *ContextDomains* are defined along these dimensions. Within the application context model, Alice herself is an instance of the `Person` *ContextEntity* ($\in U$) playing the subject role in these *ContextDimensions*. The domains make use of subsets of the global context model and have specific management needs.
The latter part regarding management of context information within a *ContextDomain* is essential, because it allows an application to consider custom context provisioning (e.g. ensuring consistency, controlling acquisition, inference and dissemination) for each *ContextDomain*, thereby substantially increasing design and runtime flexibility. We will discuss these issues in more detail in chapter 5.

One last thing to note regards the organization of *ContextDomains*. Left only with the previously given definitions, the default result is that of a flat network of domains. However, our formal meta-model can be exploited further to allow the possibility for *ContextDomain hierarchies*. This follows from the capability of *ContextDomain* types (i.e. the type of the *ContextEntities* that play the object role in the *ContextDimension defining the domain*) to be characterized by *inclusion-like EntityDescription* relations. Formally we define this as follows:

Definition 4.1.21 (Context Domain Hierarchy)**.** A *ContextDimension* chosen from a *context category cat* is said to be *hierarchical* if the *ContextAssertion* $P(E_{subj}, E_{obj})$, with $E_{obj} \in CE_{cat}$ is such that there exists an *EntityDescription* $D \in ED_{cat}$ with the property that D defines an *inclusion relation* between the instances of E_{obj}.

To clarify this again with examples from our reference scenario, consider the `locatedAt(Person, UniversitySpace)` *ContextDimension* and the `includedIn(UniversitySpace, UniversitySpace)` *EntityDescription*. Both `ami-lab` and `cs_building` are instances of the `UniversitySpace` *ContextEntity* and it further holds that `includedIn(ami-lab, cs_building)`. Thus we obtain a domain hierarchy based on a spatial dimension.
As we will further see in Chapter 6, *ContextDomain* hierarchies are an additional means for application structuring, help in establishing routing protocols and allow an application to perform remote provisioning of context information in a manner that maintains *locality* (i.e. dissemination and consumption of context occurs in accordance to the logical partition that produced it).

4.2 Ontology-based Meta-Model

Having introduced the formalization of the proposed CONSERT context meta-model we now continue by presenting the CONSERT Ontology, which gives an ontological form to all the key context model elements introduced in the previous sections. As we will see later one, the choice of implementing our formal model using an ontology brings benefits in terms of expressiveness (e.g. the ability to define subclass relations between *ContextEntities* or subproperty relations between *ContextAssertions*) as well as reasoning (e.g. consistency check inferences for static knowledge - *ContextEntities* and *EntityDescriptions*).
Furthermore, besides providing an implementation for the constructs of the CONSERT meta-model, the proposed ontology vocabulary introduces properties that offer auxiliary information. They concern aspects such as information acquisition method or functions to validate temporal

continuity of a situation. These properties influence the runtime reasoning and provisioning control behavior.

The vocabulary of the CONSERT Ontology is divided in three modules, which are depicted in Figure 4.2. The *core* module contains the vocabulary which allows expressing the *con-*

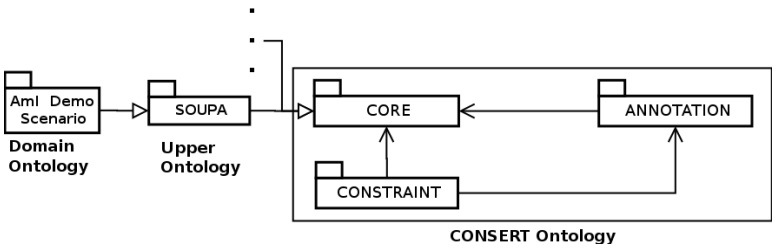

Figure 4.2: Components and their associations in the CONSERT ontology

tent of context statements (i.e. *ContextEntities*, *EntityDescriptions* and *ContextAssertions*). The *annotation* module defines the different types of *ContextAnnotations* which characterize a *ContextAssertion*. Lastly, the constraint module represents our solution to the challenge of modeling context integrity. In particular, it provides the vocabulary that is used to express the rules that detect integrity, uniqueness or value constraint violations.

Figure 4.2 also highlights the fact that the CONSERT Ontology defines a context meta-model. To build the context model of an application our meta-model must be extended by an upper-ontology that lies on top. This can either be done from scratch or, as we show in Figure 4.2, by coupling an existing context modeling ontology (such as SOUPA [Chen et al., 2004b]) and building on top of that. In this way, existing classes (which become *ContextEntities*) and properties (which become binary *ContextAssertions*) can be reused when building the desired domain ontology.
In what follows we present each CONSERT Ontology module in more detail.

4.2.1 Content Representation

Figure 4.3 shows a class-like diagram representation of the CONSERT core vocabulary[1]. The ontology defines the generic class ContextEntity that becomes the new root for all classes of a domain ontology.

EntityDescriptions and *binary ContextAssertions* of a context model can be readily defined in the CONSERT ontology by means of two OWL object properties (entityRelationAssertion, entityRelationDescription) and two datatype properties (entityDataAssertion, entityDataDescription). These properties help to "classify" the object and datatype properties as either a *ContextAssertion* with arity $n = 2$ (subproperties of entityRelationAssertion and entityDataAssertion) or *EntityDescription* (subproperties of entityRelationDescription and entityDataAssertion).

To express *ContextAssertions* with arities $n = 1$ or $n \geqslant 3$ we introduce two new classes within the CONSERT ontology: UnaryContextAssertion ($n = 1$) and NaryContextAssertion ($n \geqslant 3$). For both these cases we use a mechanism which is similar to reification of RDF statements[2]. In the CONSERT ontology we define the assertionRole property relating an instance of a UnaryContextAssertion or NaryContextAssertion to a ContextEntity or Literal which plays a role in the assertion.

[1] http://purl.org/net/consert-core-ont
[2] http://www.w3.org/TR/rdf-schema/#ch_reificationvocab

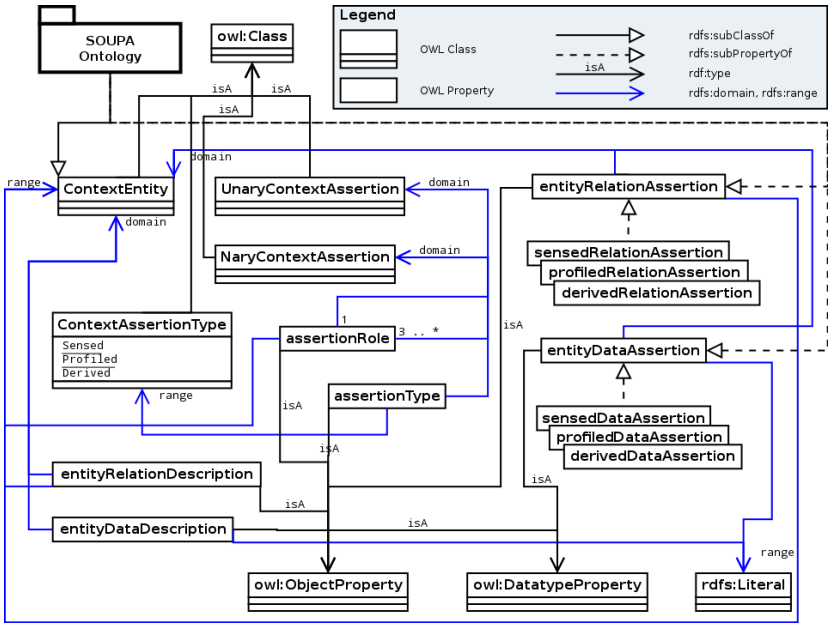

Figure 4.3: CONSERT Ontology core vocabulary

Example 4.2.1 (UnaryContextAssertion example)**.** For the unary case, a *ContextAssertion* like inAdHocDiscussion(*alice*) entails the creation of the inAdHocDiscussion subclass of UnaryContextAssertion and the assertion of the statements (in Turtle syntax):

```
{
    []  a :inAdHocDiscussion;
        :assertionRole :alice.
}
```

where `[]` represents a blank node.

Example 4.2.2 (NaryContextAssertion example)**.** Taking a possible example from the reference scenario, in order to express a *ContextAssertion* like sensesSkeletonInPosition(camera, skeleton, sitting) (e.g. as a Kinect camera based posture sensor would do in the AmI-Lab), where *KinectCamera*(camera) and *PostureSkeleton*(skeleton) are *ContextEntities* and sitting is an instance from an enumeration (e.g. {sitting, standing}), we first create the sensesSkeletonInPosition subclass of NaryContextAssertion together with subproperties of assertionRole specifying its roles (cameraRole, skeletonRole, postureRole). The *ContextAssertion* in our example would be then expressed as the following group of statements:

```
{
    []  a :sensesSkeletonInPosition;
        :cameraRole :camera;
        :skeletonRole  :skeleton;
        :postureRole :sitting.
}
```

In order to express information about the method of acquisition of a *ContextAssertion* instance, we take inspiration from work in [Henricksen, 2003].
In the binary case, we extend entityRelationAssertion and entityDataAssertion into properties

that specify whether an assertion has been acquired from physical or virtual sensors (sensed), directly specified by the system or the user (profiled) or deduced by an inference (derived), as can be seen in Figure 4.3. This classification will be later exploited at runtime, influencing the processing of each *ContextAssertion* based on the acquisition type (more details in chapters 5 and 7).
For instances of UnaryContextAssertion and NaryContextAssertion, the CONSERT ontology defines the assertionType property (whose range is an owl:unionOf class called ContextAssertionType) which states if they are sensed, profiled or derived.

4.2.2 Annotation Representation

We move forward to detailing the concrete means of representing *ContextAnnotations* with a focus on the definitions for the annotation domains discussed throughout the chapter (source, timestamp, time validity and certainty). These are the ones most commonly used in the literature. They allow inspection of the temporal relationships between detected situations and support inference and query time decision making, by identifying the origin of the information and its quality metrics. Figure 4.4 presents the annotation vocabulary[1] of the CONSERT On-

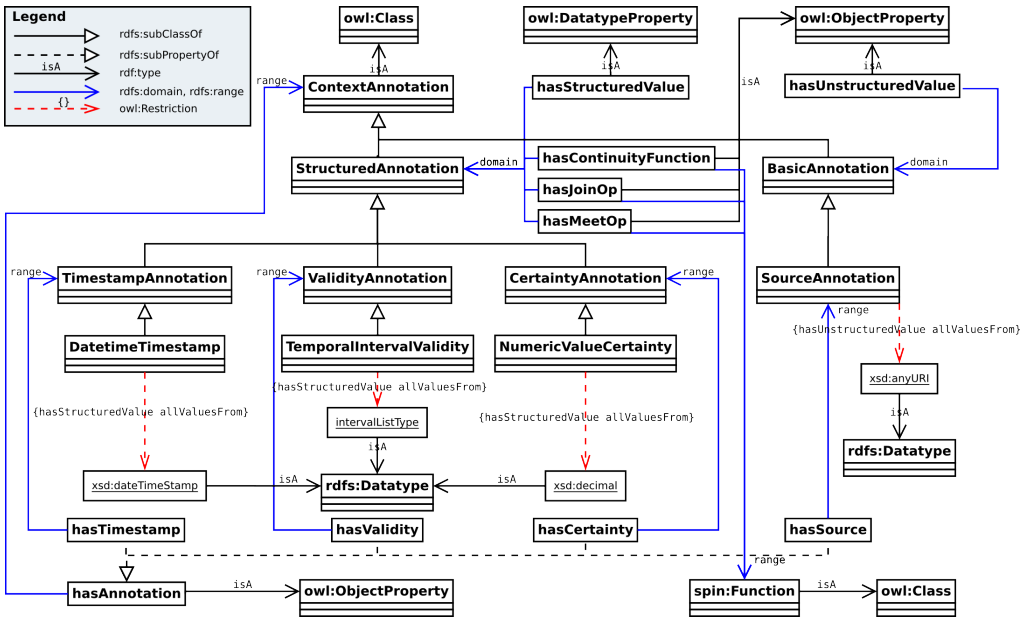

Figure 4.4: CONSERT ontology annotation vocabulary

tology. The classes are defined with extensibility in mind, such that context model designers may have the ability to append new *ContextAnnotation* definitions according to their need. Two distinct types of annotations are considered: BasicAnnotation and StructuredAnnotation.

BasicAnnotation is the class meant to describe *ContextAnnotations* which do not have (or need) a structured manipulation during inference. In Section 4.1.1 we mentioned that the source annotation is an example of a BasicAnnotation (as can be seen also in Figure 4.4).
The hasUnstructuredValue functional property provides the actual value of a BasicAnnotation. The context model designer is required to specify the corresponding ontology class representing the range of this property for each defined subclass of BasicAnnotation. Further, a custom implementation must be provided for the annotation expression function (cf. Definition 4.1.16)

[1] http://purl.org/net/consert-annotation-ont

that is used to specify the value of this type of annotation for a *ContextAssertion* derived during inference.

StructuredAnnotation is the class describing *ContextAnnotations* for which a specific inference usage semantics is defined (i.e. a concrete form of the $\oplus$ and $\otimes$ operators introduced in Definition 4.1.3). Its direct subclasses (called base structured annotation classes) represent a given annotation domain (cf. Definition 4.1.8).
Derivatives of a base annotation class (e.g. DatetimeTimestamp, NumericValueCertainty, TemporalIntervalValditiy describe means of concretely expressing the value set of an annotation domain. For example, the default value set for the timestamp annotation domain is made up of xsd:dateTimeStamp instances. It may be however that for a particular application scenario the timestamps are much more suitably modeled as simple integer values, providing just relative order of events instead of an explicit time measurement. In this case, an application designer could extend the TimestampAnnotation base annotation class with one called IntegerTimestamp. This new class would define a restriction over the hasStructuredValue property, stating that all its values must be integers.

A *ContextAssertion* URI is bound to a *ContextAnnotation* of a particular domain by means of the corresponding subproperty of hasAnnotation (e.g. hasValidity for the validity domain, hasCertainty for the assertion certainty domain).
The hasStructuredValue property gives the actual value of a *StructuredAnnotation* instance. Each derivative of a base annotation class comprises in its definition an OWL allValuesFrom restriction which states the corresponding (standard or customly defined) rdfs:Datatype instances (e.g. xsd:datetime, ctx:intervalListType) that denote the value set of the given annotation domain. The restriction definition can be used at runtime to ensure that all the *ContextAnnotation* instances of a newly inserted *ContextAssertion* provide a correct value.

Next, we introduce the rdfs:Datatype instance corresponding to each of the discussed annotation domains (i.e. the *concrete annotation domains* according to Definition 4.1.6). The value for the *source* basic annotation consists in a URI (rdfs:Literal of type xsd:anyURI) identifying authors of *ContextAssertions* (services, actors). For the timestamp annotation domain, the vocabulary $A_{timestamp}$ consists of the set of date timestamp strings. The default CONSERT ontology specification defines xsd:dateTimeStamp as the datatype for timestamp literals.
In the case of the time validity annotation domain, an element of the vocabulary $A_{validity}$ is a set of pairwise disjoint time intervals. Though not shown in the annotation vocabulary figure, the CONSERT representation and reasoning engine defines a custom rdfs:Datatype called intervalListType which becomes the default for literals expressing values of the validity domain. Lastly, for the accuracy annotation domain the vocabulary consists of decimal values in the interval $[0, 1]$. They are expressed using the xsd:decimal datatype.

One last noteworthy aspect is that StructuredAnnotation subclasses provide values for three important properties: hasJoinOp, hasMeetOp and hasContinuityFunction. The ranges of these properties are instances of functions. The value of the first two properties represents the implementation of the $\oplus$ and $\otimes$ operators respectively, for the given StructuredAnnotation subclass. The value of the hasContinuityFunction property is used during one of the steps performed by the CONSERT Engine when inserting a new *ContextAssertion* instance. It is further detailed in Section 4.4.2. The concrete implementation of $\oplus$ and $\otimes$ operators will be more closely explained in Section 7.1.2.

4.2.3 Constraint Representation

The modeling uniformity we try to achieve with the CONSERT Ontology extends to the way in which context integrity, more specifically the violations thereof, are expressed. The CONSERT Ontology defines a module that contains explicit vocabulary used to describe the nature of a context constraint violation. The means by which the violation is detected are detailed in Section 4.3.2. Figure 4.5 shows the vocabulary[1] used to define constraints. The UniquenessConstraintTemplate, ValueConstraintTemplate and IntegrityConstraintTemplate (and corresponding constraint violation classes) are the ones that realize the concept of context information constraints in the CONSERT ontology. The hasSource property is used on ContextConstraintViolation instances to refer to the ConstraintTemplate instance that triggered the conflict signalization. The conflicting *ContextAssertions* (one in the case of value constraints and two for uniqueness and integrity constraints) are given by the value of the hasConflictingAssertion property. The range of this property is an instance of the ConflictingAssertion class. Instances of this class have two properties. hasAssertionType specifies the ontology resource denoting the *ContextAssertion* on which the constraint is placed. hasAssertionInstance indicates identifier (the URI of the named graph that wraps its contents) of the conflicting *ContextAssertion* instance. The difference between uniqueness and integrity constraints is that the former are always specified for *ContextAssertions* of the same type (i.e. same value for the hasAssertionType property), while the latter capture unsatisfied dependencies between instances of two different *ContextAssertion* types. For value constraint violations, the CONSERT ontology also defines the possibility to express the annotation or assertion value that triggered the violation.

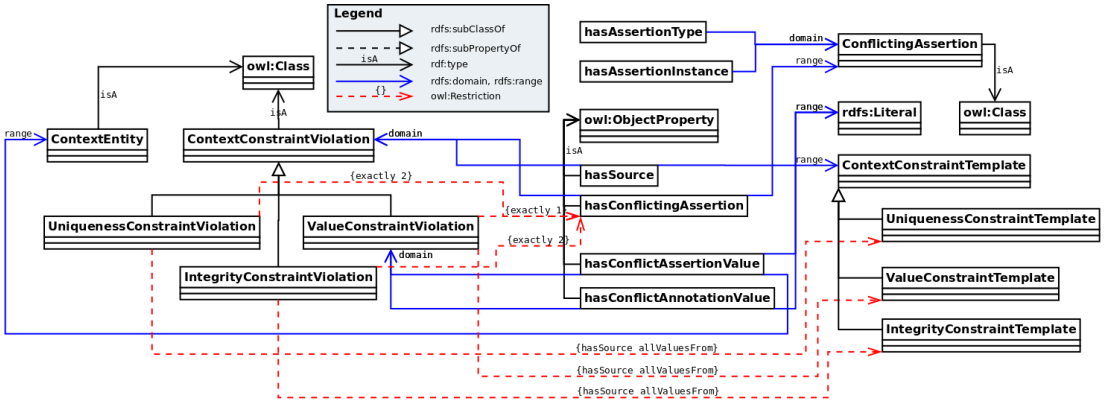

Figure 4.5: CONSERT ontology constraint vocabulary

In Section 4.3.2 we detail how we attach constraints to a *ContextAssertion* class by using the introduced vocabulary.

4.3 Rule-based Context Inference

We have seen previously how the CONSERT Ontology is defined from the proposed meta-model, achieving uniformity and expressiveness of representation. We now focus on showing how we continue to use semantic web technologies to exploit the CONSERT Ontology with regard to reasoning over context information.

In Section 2.2.1 we mentioned that two main concerns must be regarded: maintaining consistency of the knowledge base and inference of higher-level knowledge based on existing primary

[1] http://purl.org/net/consert-constraint-ont

information. Consequently, in what follows we present how the formal model of the inference rules described in Section 4.1.2 is implemented using SPARQL, as well as how this inference mechanism is complemented by ontology-based reasoning for both higher-level knowledge derivation and consistency maintenance.

4.3.1 Context Derivation Rules

The first concern we treat is the derivation of new knowledge from existing information, be it static (i.e. descriptive information), sensed or directly provided by a user or service (i.e. profiled). We establish two processes that are at work in this case: one for the dynamically changing one (i.e. sensed or profiled) and one for inferences on the *static* part of the knowledge base (i.e. EntityDescriptions).

Dynamic Inference

For this type of deduction we consider a rule-based inference approach. This is because the expected frequency of changes in sensed or profiled information can lead to ontology-based reasoning processes (e.g. *realization*) becoming computationally expensive.
Remember from Section 4.1.2 that a *ContextDerivationRule* ρ is composed of a head and a body section and that the head of the rule is in fact an instance of the newly derived *ContextAssertion* type: $P_{head}(x_1, \ldots, x_k) : \{\lambda_1, \ldots, \lambda_l\}$, $Pin\mathcal{F}$. To create the inference rule we use the SPARQL CONSTRUCT[1] query in which the CONSTRUCT clause implements the head of the rule (i.e. they *build* the instance of the derived *ContextAssertion*) and the WHERE clause implements the body of the rule (i.e. the *ConditionExpr*).
The statements in the CONSTRUCT clause perform an *instantiation* of a *ContextAssertion* by using the corresponding vocabulary statements of the CONSERT Ontology. The statements will usually refer to both content (e.g. an NaryContextAssertion with statements for the subproperties of assertionRole defining the *ContextEntities* that play a role in the assertion) as well as annotations of the *ContextAssertion* instance, using the vocabulary introduced in the *ContextAnnotation* module of the CONSERT Ontology. The statements in the CONSTRUCT clause can also use variables that are bound in the WHERE clause. The exact mechanism by which a new *ContextAssertion* is constructed at runtime as part of a *DerivationRule* execution is out of the scope of this chapter (since it is an implementation detail), but will be discussed at more length in Section 7.1.

In what follows, we want to focus on explaining how the different formal expressions introduced in Section 4.1.2, which compose the body of a *DerivationRule* can be mapped to SPARQL syntax.

We begin with the elements composing a *ConditionExpr* as presented in Definition 4.1.12.

- *AssertionExpr*: a *ContextAssertion* and its *ContextAnnotation* instances are expressed using RDF statements wrapped in named graphs as will be further explained in Section 7.1.1. These form SPARQL basic graph patterns (BGP).
- *AggExpr*: are expressed using SPARQL aggregates[2]
- *TermExpr*: *EntityDescriptions* are expressed as RDF triples that reside within the entityStore named graph as will be detailed in Section 7.1. Boolean operations, logical connectives and functions on terms are implemented using the equivalent SPARQL syntax and are contained within SPARQL FILTER expressions.

[1] http://www.w3.org/TR/rdf-sparql-query/#construct
[2] http://www.w3.org/TR/sparql11-query/#aggregates

- *AnnExpr*: the annotation assignment functions are user-defined. They are implemented based on custom code that runs during query evaluation in the software engine that we detail Sections 4.4 and 7.2. The value they compute is bound to the corresponding λ_j variable in the rule head ($annotations(head(\rho))$) using a SPARQL BIND statement.

Next, let us consider the existentially and universally constrained quantifications. For the existential case support is already provided in SPARQL by the EXISTS filter expression (see Figure 4.6). For the universal case the intuition behind the SPARQL query shown in Figure 4.6 is the following: consider a substitution $\sigma = \{y_1/t_1, \ldots, y_r/t_r\}$ which binds each variable $y_i \in Y_\rho$ to a *ContextEntity* or literal. Let us then denote by $\Sigma_{F_c \downarrow Y_\rho}$ and $\Sigma_{DerivExpr \downarrow Y_\rho}$ the sets of all substitutions σ binding variables in Y_ρ for which the constraining assertion F_c and the assertions in *ConditionExpr* are true respectively. The interpretation of the universal constrained quantification rule then implies that $\Sigma_{F_c \downarrow Y_\rho} \subseteq \Sigma_{DerivExpr \downarrow Y_\rho} \Leftrightarrow \Sigma_{F_c \downarrow Y_\rho} \setminus \Sigma_{DerivExpr \downarrow Y_\rho} = \emptyset$. The SPARQL MINUS[1] filter expression used in Figure 4.6 provides this exact semantics.

```
CREATE GRAPH <gURI>;
INSERT{
  GRAPH <gURI> {new assertion}
  GRAPH <newAssertionStore> {annotations}
}
WHERE {
 { constraining assertion } .
 FILTER (
  EXISTS { ConditionExpr }
 )
}
```

```
CREATE GRAPH <gURI>;
INSERT{ assertion and its annotations }
WHERE {
 { SELECT (COUNT(*) AS ?count)
    WHERE {
     {constraining assertion}
     MINUS
     {ConditionExpr}
    }
 } . FILTER (?count = 0)
}
```

Figure 4.6: SPARQL expressions for existentially (left) and universally (right) constrained quantifications

In Section 8.2.1 we provide an exemplification of SPARQL-encoded *ContextDerivationRules* as part of the modeling and experimentation on hand of the reference scenario.

Static Inference

We mentioned that static context information is modeled as declarations of *ContextEntity* instances and *EntityDescriptions* in the CONSERT meta-model. While entities and their descriptions are not inferred using the previously presented rule approach, it is nonetheless useful to have the ability to reason over *ContextEntity* (following rdfs:subClassOf relations) or *EntityDescription* hierarchies (following rdfs:subPropertyOf relations), as well as *ContextEntities* whose definition is simply given as OWL class construction axioms involving several *EntityDescriptions* and *ContextAssertions*. For this, an ontology-based reasoning mechanism is used.

In Sections 4.4.2 and 7.2 we talk about the implementation and execution cycle of our proposed reasoning engine. There we show that new *ContextEntity* declarations and insertions of *EntityDescriptions* result in statements that are stored and processed in a specific way. Every time an update containing references to such elements of static context information is perceived, the inserted information is subjected to *RDFS entailment*[2]. In this way it is possible to infer information that exploits subclass and subproperty relations. Let us consider our scenario and a modeling of information such as the following:

[1] http://www.w3.org/TR/sparql11-query/#neg-minus
[2] http://www.w3.org/TR/rdf11-mt/#rdfs-entailment

```
ex : ami − lab        rdf : type              ex : LaboratoryRoom .
ex : alice            ex : friendOf           ex : cecille .

ex : LaboratoryRoom   rdfs : subClassOf       ex : UniversitySpace .
ex : friendOf         rdfs : subPropertyOf    ex : acquaintanceOf .
```

where `ex : LaboratoryRoom` and `ex : UniversitySpace` are examples of *ContextEntity* types and `ex : friendOf` and `ex : acquaintanceOf` are *EntityDescriptions*. By RDFS entailment, the following additional information would be derived:

```
ex : ami − lab   rdf : type            ex : UniversitySpace .
ex : alice       ex : acquiantanceOf   ex : cecille .
```

This constitutes a substantial benefit from a reasoning perspective because in this way the body of *ContextDerivationRules* can contain conditions not *explicitly* asserted (but derived through entailments such as the one above) in the knowledge base.

However, the above type of inferences only exploits the subclass and subproperty relations of static context information. We mentioned that another reasoning use case may be the one in which *ContextEntities* are defined using OWL axioms. For instance, consider the following example of defining the concept of a busy person entity:

$$\texttt{BusyPerson} \equiv \texttt{Person} \sqcap \exists \texttt{hasTimedActivity}$$

where `Person` and `BusyPerson` are *ContextEntity* types and `hasTimedActivity` is a profiled *ContextAssertion*. Considering our scenario, an instance of *hasTimedActivity* could be derived upon the insertion of information such as:

```
ex : alice                   ex : hasTeachingActivity   ex : cs_lecture .
ex : hasTeachingActivity     rdfs : subPropertyOf       hasTimedActivity .
```

In this example, first the information that `ex : Alice ex : hasTimedActivity ex : CS_Lecture` would be inferred based on the previously explained RDFS entailment. Then, another ontology-specific reasoning procedure, the ABox realization mentioned in Section 2.2.3, would have to be employed to infer the information that `ex : Alice rdf : type ex : BusyPerson` (i.e. Alice is a busy person).
However, as we have pointed out in the beginning of this discussion, applying ontology reasoning such as ABox realization in relation to information that changes more frequently (such as the sensed or profiled *ContextAssertions*) soon proves to be computationally difficult, especially under high update rate and a large knowledge base. For this reason, in Section 5.2.2 where we talk about options for coordinating the provisioning of context information (which includes control over how inferences are executed by our reasoning engine), we point out a mechanism by which ontology-based reasoning procedures such as the one described above can be scheduled based on the *type* of *ContextAssertions* they involve.

4.3.2 Context Consistency

The second aspect of reasoning about context information relates to maintenance of its consistency. As in the previous case of knowledge derivation, two reasoning approaches can be used with respect to this goal.
For dynamic context information (i.e. sensed, profiled or derived *ContextAssertions*) the *dependencies* between assertion instances resolve to the definition of *integrity constraint detection rules* using the constraint module of the CONSERT Ontology. For static *ContextEntities* (defined either as simple OWL classes or using OWL axioms) and *EntityDescriptions* consistency amounts to ensuring that instances of entities and descriptions do not break the semantics associated to the OWL class and property definitions which are used to implement them.

To start with the consistency of static information, a simple source of inconsistency can come, for example, from asserting that a entity *instance* represents two *disjoint ContextEntity* types at the same time. Continuing the example introduced just earlier (the `BusyPerson` *ContextEntity* definition), consider the following additional modeling:

`ex:alice`	`rdf:type`	`ex:FreePerson`
`ex:FreePerson`	`owl:disjointWith`	`ex:BusyPerson`

The above declares that the *ContextEntity* types `FreePerson` and `BusyPerson` are disjoint and that, initially, Alice is free. If now at runtime, the conditions for deriving that Alice is a busy person become true and the assertion that Alice is also an instance of a `FreePerson` *ContextEntity* is not removed, then the knowledge base would become inconsistent.
To detect such cases, remember from Section 2.2.3 that ontology-based reasoning offers a procedure to determine if the ABox (i.e. the set of instances in a knowledge base) are consistent with respect to the defined semantics of the TBox (i.e. the *ContextEntity* and *EntityDescription* modeling using OWL classes, properties and axioms).
As in the previous case for higher-level knowledge derivation, running such an ontology-based inference procedure every time a *ContextAssertion* update occurs (which, as in the presented example, could bring about the assertion of a new *ContexEntities* that are defined using axiomatic expressions) is computationally intensive. As explained previously, in Section 5.2.2 we present the means by which a context-aware application can control when such ontology-based consistency verifications of the static knowledge base must take place.

By contrast, the integrity constraints that are defined for dynamically updated *ContextAssertions* can and must be executed on each insertion. *Value* and *Uniqueness* constraints are defined for a type of *ContextAssertion*. Further, general integrity constraints are typically defined between two interdependent *ContextAssertion* types. Consequently, when executing a constraint detection rule the reasoning will be performed only over the concerned assertion instances, as opposed to the entire knowledge base, as would be the case for the ontology-based consistency check algorithms. This is why the rule-based constraint detection mechanism employed for dynamic context information is performed upon every update of a *ContextAssertion* type for which such constraints are defined. Section 4.4.2 presents details of how these verification fit in the general execution cycle of our proposed reasoning engine.
To implement constraint detection rules, whilst having the same goals of uniformity and expressiveness in mind, we opt again to use SPARQL CONSTRUCT queries to declare the triggering conditions and produce the resulting violation notices. Let us examine the case of defining an Uniqueness Constraint Violation.

Example 4.3.1 (locatedAt uniqueness constraint)**.**

```
CONSTRUCT {
  _:b0 a ctx:UniquenessConstraintViolation .
  _:b0 ctx:onContextAssertion person:locatedAt .
  _:b0 ctx:hasConflictingAssertion ?g1 .
  _:b0 ctx:hasConflictingAssertion ?g2 .
}
WHERE {
  GRAPH ?g1 {
    ?this person:locatedAt ?Loc1 .
  } .
  GRAPH ?g2 {
    ?this person:locatedAt ?Loc2 .
  } .
  GRAPH <http://pervasive.semanticweb.org/ont/2004/06/person/locatedAtStore> {
    ?g1 ctx:hasValidity ?valAnn1 . ?valAnn1 ctx:hasValue ?validity1 .
    ?g1 ctx:hasCertainty ?certAnn1 . ?certAnn1 ctx:hasValue ?cert1 .
    ?g2 ctx:hasValidity ?valAnn2 . ?valAnn2 ctx:hasValue ?validity2 .
    ?g2 ctx:hasCertainty ?certAnn2 . ?certAnn2 ctx:hasValue ?cert2 .
```

```
  } .
  FILTER (
    NOT EXISTS {?Loc1 (spc:spatiallySubsumedBy)+ ?Loc2 .} &&
    (?Loc1 != ?Loc2) && (?cert1 >= 0.75) && (?cert2 >= 0.75) &&
    cfn:validityIntervalsOverlap(?validity1, ?validity2)).
}
```

It expresses the fact that a person cannot be deemed as finding herself in two places at the same time (overlapping validity intervals) with high certainty in both affirmations. While certain details about the form of the above query will be discussed more closely in Section 7.1, where we talk about the implementation of the CONSERT Middleware, we draw the attention to how the constraint violation definition vocabulary discussed in Section 4.2.3 helps us capture all the required information about the contradicting *ContextAssertion* instances. Notice how in the CONSTRUCT clause of the query, the type of *ContextAssertion* for which the constraint is defined (`person:locatedAt`) as well as the two instances (marked by the `?g1` and `?g2` variables) that lie in conflict are represented.
Furthermore, notice that one advantage of using SPARQL as a constraint definition language is that it allows us to compose expressive constraint statements. In this example, not only can we condition the triggering of a violation based on values of the annotations of a *ContextAssertion*, but the domain knowledge check includes a call to a SPARQL 1.1 Property Path (spc:spatiallySubsumedBy+) which states that physical spaces that lie in a spatial subsumption relation are excluded from the set of conflicting ones (e.g. if a user is in a laboratory, it is ok to have another *ContextAssertion* that says the user is also in the university building containing the laboratory).
In Section 7.1.2 we discuss how a context integrity constraint query such as the one above is serialized and *attached* to the corresponding *ContextAssertion* definition in a context model.

4.4 Reasoning Engine

To complete the overview of our contributions to the context representation and reasoning issue, let us introduce hear the architecture and execution cycle of the CONSERT Engine, the component of the CONSERT Middleware in charge of handling context updates, higher-level inferences, constraint and consistency checks, as well as query answering. While the implementation of the engine and the services it provides within the context provisioning process will be discussed in later chapters (7 and 5, respectively), in this section we wish to explore how the life cycle of the CONSERT engine accommodates all the representation and reasoning aspects described previously. We first present the engine architecture (without entering into implementation details) and then discuss the execution cycle which exploits all the representation capabilities offered by the CONSERT Ontology to obtain semantically distinguishable context situations (as defined in the objectives in Section 2.3).

4.4.1 Architecture

Figure 4.7 presents an architectural overview of the CONSERT engine. Its most important building blocks and internal data structures can be observed on the right side of the figure, while on the top of the figure we can observe a set of 4 external services which influence the execution cycle discussed in the next section. The engine defines three indexes (ContextAssertionIndex, ContextAnnotationIndex, ContextConstraintIndex) which create an internal representation of the context model built using the ontology modules described in Section 4.2. The indexes create wrappers over the modeled constructs, facilitating access to required information at runtime (e.g. annotation statements for instances of a given *ContextAssertion* type, records of the hasJoinOp, hasMeetOp and hasContinuityFunction properties for each *ContextAnnotation* type,

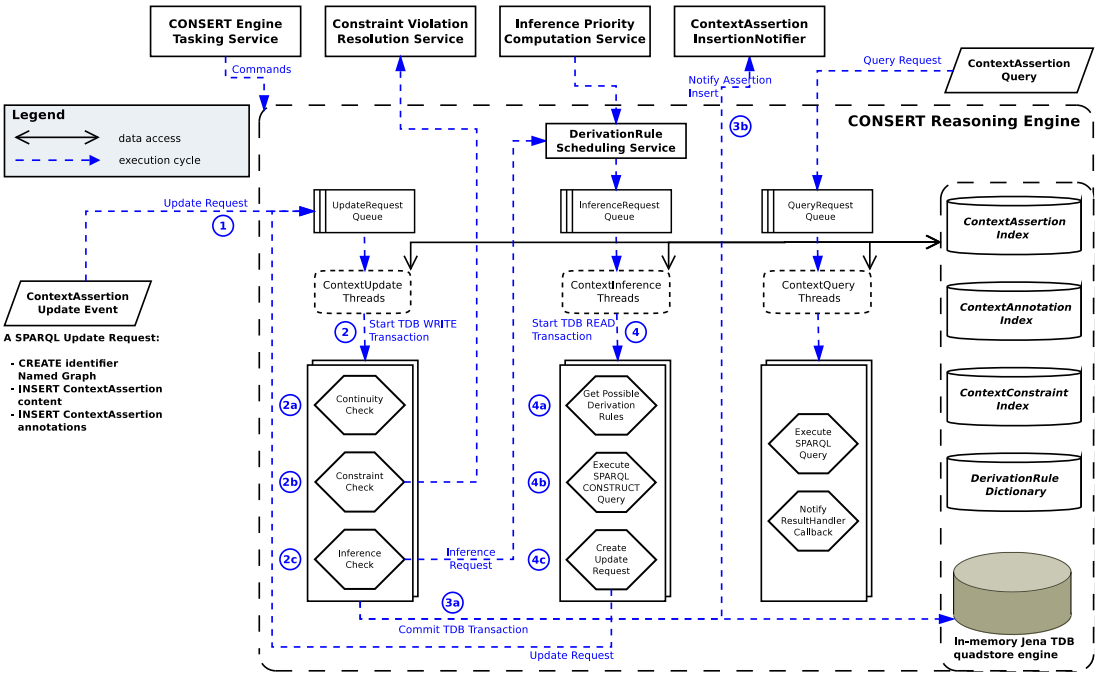

Figure 4.7: The CONSERT Engine architecture and main activity cycle

as described in Section 4.2.2).
The Derivation Rule Dictionary provides a mapping from every *ContextAssertion* to the list of *DerivationRules* in the body of which it appears. This dictionary is used during inference checks to determine if the update request for a given *ContextAssertion* can trigger the execution of a deduction process. Further aspects regarding the semantic web concepts and frameworks (e.g. Named Graphs, Jena TDB) that help to concretely store and work with a context model built using the CONSERT Ontology are detailed in chapter 7.

The CONSERT Engine works with additional application-specific services and in Figure 4.7 we observe a list of four such services depicted above the CONSERT engine container delimiter.
The engine interacts with the *Constraint Violation Resolution Service* when an integrity constraint violation is detected during a *ContextAssertion* update. The service is supplied by the application developer and it effectively implements a policy for deciding which of the two conflicting assertion instances must be kept. The statements generated using the constraint vocabulary of the CONSERT ontology described in Section 4.2.3 provide the service with all the information required to retrieve both the content and the annotations of conflicting *ContextAssertion* instances, so as to make an informed decision. The CONSERT engine supplies two default implementations of the service which may be used to discriminate based on the timestamp (PreferNewest) or certainty(PreferAccurate) annotations.
The *Inference Priority Computation Service* is used by the *DerivationRule* scheduler of the CONSERT Engine when new inference requests get enqueued. The engine provides a default first-come first-served implementation of this service, but application developers may provide their own, which can assign, for example, a priority for each type of *DerivationRule*. The priority value can be based on the inference usage and success statistics collected periodically by the CONSERT Engine. Details of how and what statistics are collected are further provided in Section 5.3.1.

The *ContextAssertion Insertion Notifier* is a service notified by the CONSERT Engine whenever a *ContextAssertion* update is successfully stored in the knowledge base. In chapter 5 we talk about the functionality of our context provisioning units that are part of the proposed CON-

SERT Middleware. There we explain how the provisioning unit responsible for dissemination uses this *InsertionNotifier* service to handle context subscription requests.

Finally, the *CONSERT Engine Tasking Service* is the means by which the *coordinating* context provisioning unit of the our proposed middleware (discussed further in Section 5.1.2) can control the different aspects of the CONSERT engine runtime execution. The service allows the external management unit to *(i)* request that a *DerivationRule* be enabled or disabled, *(ii)* trigger an ontology reasoning process or *(iii)* clear the in-memory storage of *ContextAssertion* instances that exceed a certain time-to-live threshold. As we can see, these options (especially the first two ones) address aspects that were hinted towards in previous sections (cf. Sections 4.3.1, 4.3.2). Options *(i)* and *(ii)* regulate the inference capabilities of the CONSERT Engine. In particular, the *Derivation Rule* enable/disable switch controls the dynamic event processing side of the inference process. The ontology reasoning trigger, on the other hand, is targeted towards inference over the static background knowledge consisting of *ContextEntity* and *EntityDescription* definitions. Remember from previous observations that performing an ontology reasoning effort whenever *ContextAssertions* that influence *ContextEntities* or *EntityDescriptions* are updated incurs a significant overhead. Therefore, the *CONSERT Engine Tasking Service* externalizes the logic of requesting ontological reasoning to the broader context provisioning functionality of which it is part. As we will see more closely in Section 5.2, policies can be defined to specify the timing for periodic invocations of the ontology reasoning process, so that the period is correlated with the perceived update frequency of each type of *ContextAssertion*.
Option *(iii)* (in-memory storage clearing) is a way to free up memory from the statements of *ContextAssertions* which have outlived their possible usage during runtime inference processing. Through the same policy mechanism mentioned above, an application can set a time-to-live threshold for each *ContextAssertion* type and request that assertion instances with a timestamp annotation that is older than the specified threshold be moved from memory to a persistent storage for possible offline processing.

4.4.2 Execution Cycle

The CONSERT engine is initialized based on a set of OWL files that define the context model using the CONSERT Ontology. To make things easier for a developer, he or she can specify a file that defines application specific concepts based on each module of the CONSERT Ontology (i.e. core, annotation, constraint). The engine uses the files to build the auxiliary data structures discussed above. Once they are initialized the engine can start operating.
We begin our execution cycle description by considering the arrival of a *ContextAssertion* insertion/update request, as shown in Figure 4.7. As mentioned several times during this section, the specific contents of the update request are an implementation detail which we discuss more thoroughly in Section 7.1. The update request is put into the waiting queue of the insertion handling thread pool. Once it is picked up, the handler thread proceeds to Step 2 of the execution process and performs a sequence of three verifications:

Step 2a) The first one is called the *continuity check*, where the system checks if the content of the new *ContextAssertion* matches any of the already existing ones. If a content-based match is found, the check proceeds by looking at each *StructuredAnnotations* attached to the two continuity-merge candidate assertions. As hinted towards in earlier sections, the procedure accesses the *ContextAnnotationIndex* to retrieve the value of hasContinuityFunction property for the given *StructuredAnnotation*. The continuity function examines if the two *ContextAssertion* instances can be merged from the viewpoint of the annotation. For example, in the case of the certainty annotation, only assertions for which their certainty annotations are close to one another (e.g. within 0.1) may be allowed to merge. That is, a situation described by a *ContextAssertion* with a high certainty value is different from one where the same situation is asserted with a much smaller degree of confidence.
If after checking every continuity function, the two *ContextAssertion* instances are allowed

to merge, the engine will access the corresponding $\oplus$ operator of each *StructuredAnnotation* to update the annotation values of the existing *ContextAssertion* (an action which represents a use case of combining annotation information for statements that have the same content, as explained in Section 4.1.1). The values for the *BasicAnnotations* are simply taken from the newly inserted *ContextAssertion* and attached to the resulting merger.

Step 2b) The class of the *ContextAssertion* to be inserted is checked against the ConstraintIndex to determine if it has any attached integrity, uniqueness or value constraints. If found, the associated SPARQL queries are used to execute the constraint detection. While assertions violating value constraints are simply rejected, for any uniqueness or integrity violations, the *Constraint Violation Resolution Service* is accessed as explained earlier above.

Step 2c) (not depicted in figure) If both continuity and constraint verifications are successful, the engine checks to see if the *ContextAssertion* type for which an update is being made is defined in the context model as a subclass (in case of UnaryContextAssertion and NaryContextAssertion) or as a sub-property (in case of a binary *ContextAssertion*) of a parent *ContextAssertion*. In the affirmative case, the engine takes the necessary steps to create an instance of the parent assertion using the same *ContextEntity* instances that play a role in the updated *ContextAssertion* instances. The exact steps depend again on the implementation-centric aspect of how *ContextAssertion* of each arity-type are stored internally in the CONSERT Engine and are thus left for further examination in Section 7.2. After creating the parent assertion content, the *ContextAnnotations* of the updated assertion are copied over to the parent instance. This inheritance procedure differs from the RDFS entailment applied in case of static knowledge, precisely because the instantiation of a parent assertion depends on its arity and also has to preserve existing annotations, a process which is specific to our context meta-model, rather than RDFS inferencing.

Step 2d) The final check is made against the Derivation Rule Dictionary which the system uses to determine if the class of the updated *ContextAssertion* appears in any *Context Derivation Rules*. On success, the insertion handler thread will enqueue an inference request triggered by the current *ContextAssertion*.

After all verifications are completed, Step 3a of the execution cycle stores the validated update in the runtime knowledge base, while Step 3b sends a notification to the *ContextAssertion Insertion Notifier* such that it may inform any registered listeners.
When an inference request is received, the corresponding handler threads start a processing sequence in which it executes three actions as follows:

Step 4a) The handler thread uses the Derivation Rule Dictionary to retrieve the list of all rules in which the *ContextAssertion* in the inference request plays a role.

Step 4b) For each *DerivationRule* in turn it executes the associated `SPARQL CONSTRUCT` query. If the rule could be successfully applied the thread proceeds to the last step in the inference process.

Step 4c) The thread takes the statements constructed by the inference query and transforms them into `SPARQL UPDATE` statements similar to the format used when inserting a new *ContextAssertion*. The newly inferred *ContextAssertion* is thus enqueued in the insertion waiting queue, which completes the deduction functionality cycle.

The execution cycle detailed above represents the typical processing sequence for a *sensed ContextAssertion*, that is, one updated frequently by a sensor source. As explained in Section 4.2.1, the CONSERT Ontology supports modeling constructs that can inform of a *design-time expected* or *runtime observed* type of acquisition for *ContextAssertion* instances. This modeling feature has of course an influence on the above presented processing steps.
Since the *derived* (i.e. inferred) *ContextAssertions* are created largely based on aggregation

of multiple sensed assertions, their update dynamics closely depends on that of the sensed *ContextAssertions* and, therefore, the CONSERT Engine applies the same processing steps (as mentioned as well in Step 4c).
A *profiled ContextAssertion* however is one which is not sensed periodically, but rather explicitly given by a user or service whenever a relevant change needs to be transmitted. Therefore, the source user or service already take care of the step of determining *semantically distinguishable situations* such that Step 2a (situation continuity check) can be omitted.
Lastly, a similar line of reasoning applies for *static* information updates (i.e. *ContextEntities* or *EntityDescriptions*). Furthermore, as explained in Section 4.3, both consistency and derivation of static knowledge follows is based on ontology reasoning procedures, rather than the rule based approach that is part of the typical execution cycle, such that the constraint and inference steps (2b and 2d) do not apply. We mentioned when we talked about the *CONSERT Engine Tasking Service* that the ontology reasoning procedures are controlled by the context provisioning coordination unit which manages a CONSERT Engine instance in our proposed middleware.

The design specifications listed above present the sequence of steps that the CONSERT Engine undertakes while handling updates of *ContextAssertions* that it may receive from external applications or sensing services. These steps help to build and maintain the contextual situation knowledge about current states and activities in the environment that other services can then exploit through querying. What is important to notice is that the steps are built around the idea of making use of all the supporting benefits introduced by the CONSERT ontology discussed in the previous section: from using the structured operators of *ContextAnnotations* in the continuity check, to applying *Context Constraints* and collecting the possible constraint violation according to the properties of the ConstraintViolation class defined in the CONSERT ontology and down to using SPARQL-encoded *ContextDerivationRules* that infer derived *ContextAssertions*.

4.5 Discussion

In this chapter we described our approach with respect to representation and reasoning about context information. We have opted to develop a context meta-model which he have presented formally and implemented using semantic web technologies. The choice of creating a meta-model and its ontology-based implementation are motivated by the requirements for expressiveness and uniformity. These same objectives justify our approach to consistency maintenance and knowledge inference based on a SPARQL-encoded rule-based reasoning mechanism. In what follows, we perform an analysis of the proposed contributions by seeing how they address the representation and reasoning requirements outlined in Sections 2.1.1 and 2.2.1 and seeing how they compare against some of the reviewed state of the art work.

4.5.1 Analysis of Modeling Contributions

Let us begin by considering the support for context modeling. We first provide a complete description for each mentioned requirement and then present a summary table for a concise overview.

CONSERT addresses **model flexibility** by proposing a meta-model approach to context modeling. That is, the proposed constructs do not model a *application domain* in itself, but rather provide representation concepts on top of which several domains can be built. However, the fact that we use an ontology-based implementation allows our meta-model to be easily connected to existing general-domain ontology-based models such as SOUPA [Chen et al., 2004b] or CONON [Gu et al., 2004], which can become upper ontologies (as shown in Figure 4.2).

One of the flexibility advantages of our approach is the ability to define arbitrary arity *ContextAssertions* as well as to specify inheritance relations between *ContextAssertions* (given our ontology based implementation). This represents an improvement upon both CML [Henricksen et al., 2005b] and the work of [Fuchs et al., 2005] which can only account for one of these two aspects.
It might be argued that arbitrary arity modeling is not of the utmost significant importance since in the majority of cases it can be replaced by a semantically equivalent set of binary statements. For instance, our example of the `sensesSkeletonInPosition` *ContextAssertion* can be expressed by the combination of two binary assertions such as `sensesSkeleton(Camera, Skeleton)` and `hasPosture(Skeleton, Posture)`. However, the relevance of the ability to capture n-ary *ContextAssertion* becomes apparent when we consider the annotations or constraints for instances of such assertion types. Indeed, with such an ability the context model designer can express the fact that meta-properties such as temporal validity or certainty apply to the the whole n-ary statement and not its individual role statements. In our above example, a model designer would have to duplicate such annotations or constraint definitions for each individual binary *ContextAssertion*, when in fact semantically they characterize the group of statements (i.e. the `sensesSkeletonInPosition` *ContextAssertion*).

Our modeling approach accounts for **heterogeneity** of the context information sources by considering properties (e.g. the subproperties of entityRelationAssertion for binary assertions or the assertionType property for UnaryContextAssertion and NaryContextAssertion) that allow a developer to state the *design-time expected* or *runtime observed* acquisition type of a *ContextAssertion* instance. As shown in the previous section, these acquisition types influence the processing steps undergone by the corresponding assertion instance. Furthermore, as we will explore in chapter 5, the update rates of various sensors can be controlled via provisioning policies and will reflect in the timestamp and temporal validity annotations of those *ContextAssertion* instances.

Context relationship and dependency specification is enabled in the CONSERT metamodel by means of *ContextDerivationRules* and *Context Constraints*. The first one specifies dependencies that result in the inference of new knowledge (derived *ContextAssertions*) from existing information, while the other one specifies value, uniqueness or general integrity constraints that keep the knowledge base consistent. In addition, the ontology-based implementation of the meta-model allows us to exploit ontology-specific constructs and reasoning procedures applicable for static context information, as explained in Section 4.3.

Timeliness and **management of imperfect/ambiguous information** are both handled based on the ability to capture the relevant *ContextAnnotations* and to access their value during inference. Currently, the proposed model does not explicitly deal with *incomplete* knowledge, as was the case for instance in Chen and Nugent's [Chen and Nugent, 2009] usage of OWL to model and reason about activities of daily living. This is because, as opposed to OWL reasoners which adopt an open world assumption, our rule-based approach uses a closed world assumption which only deals with explicitly stated *ContextAssertions*. The problem is partially handled for static context information and could be further mitigated by considering other future reasoning procedures (e.g. abductive inference) for the CONSERT Engine besides deductive inference.
On the other hand, our support for handling of uncertain/ambiguous information is more elaborate then that of other semantics-based works reviewed in Sections 2.1 and 2.2. Not only do we offer means to represent and reason about annotations such as certainty and temporal validity (which are useful when dealing with uncertain information), but by defining a specific structure and inference *usage semantics* for this type of *ContextAnnotations* we can make sure they are properly propagated to the derived *ContextAssertions*.

Table 4.1 summarizes our previous discussion. As concluding remarks for our modeling contributions we wish to point out that the resulting meta-model also fulfils our objectives for uniformity and expressiveness. Uniformity is achieved by the fact that many relevant aspects

	Flexibility	Heterogeneity	Dependencies	Timeliness	Ambiguity
CONSERT	Meta-Model + Arbitrary arity	Modeling acquisition type	*ContextDerivation Rules* + *ContextConstraints*	*ContextAnnotations*	Structured *ContextAnnotations*

Table 4.1: Overview of how context modeling requirements are addressed by the CONSERT Meta-Model.

of a context model (e.g. content of information, meta-properties, integrity dependencies) are captured using the same ontology-based vocabulary. Furthermore, uniformity of reasoning is ensured because SPARQL-based rules are used to perform knowledge derivation and constraint detection. The expressiveness of the approach results from the available modeling constructs and the use of ontologies to ensure support for concept hierarchies and easy coupling with existing context domain representation approaches.
While the resulting CONSERT Meta-Model might be more complex and verbose than other similar approaches studied in Chapter 2, from an engineering and development perspective the overhead is acceptable compared to the above outlined benefits. Furthermore, model verbosity can be easily overcome by supplying the designer with adequate modeling tools, which, given our ontology based implementation, should be a straightforward aspect of future work.

4.5.2 Analysis of Reasoning Contributions

In Section 2.2.1 we explained that reasoning about context information targets the aspects of knowledge derivation and consistency maintenance. The CONSERT Meta-Model accounts for both concerns as explained in Section 4.3. In the following we want nonetheless to analyze the aspects that distinguish the CONSERT Engine execution cycle from other reasoning approaches.

The first aspect we mention regards the *continuity check*. Thanks to timestamp and temporal validity *ContextAnnotations* the CONSERT Engine performs an automatic computation of the temporal continuity of context events. This leads to what we call semantically distinguishable situations, i.e. situations which are well defined both from a temporal point of view (i.e. start and finish moment) as well as from their characterizing meta-properties (e.g. certainty). It is in this continuity check that the hasContinuityFunction property of a structured *ContextAnnotation* becomes relevant, since it allows to establish which updates of a *ContextAssertion* types can be considered a continuation of a previously inserted event from the point of view of its annotation information.
The difference with other reasoning engines in the reviewed literature is that the mechanism for computing such temporal situation continuity is *implicit* rather than requested on demand. Our argument in favour of this choice is the idea that current AmI applications require recognition of situations with increasingly complicated detection conditions, which often involve reasoning over time. Therefore, having the ability to inspect the temporal validity of each currently detected contextual situation during inference is an obvious benefit. Furthermore, this mechanism is useful with respect to the *quality of context dissemination*. Besides being readily able to answer to queries that explicitly state the validity period for which they want to retrieve context information, a query-handling context provisioning unit using the CONSERT Engine can ensure that its answers (which may take time to be routed back to the original requester) are not based on information which becomes stale by the time they reach the requester.

The above mentioned continuity check also has implications with respect to acquisition of context information. Thus, temporal validity annotations and the *CONSERT Engine Tasking Service*, which allows the application level to perform a removal of *ContextAssertion* instances having surpassed their time to live, eliminate the need for a non-monotonic reasoning approach (i.e. perform explicit assertion and retraction of statements according to their truth value). The end of the validity of a situation is marked by the corresponding *ContextAnnotation* and

the *ContextDerivationRules* run by the CONSERT Engine can readily exploit this information when performing inference.

Our consistency handling mechanism also differs from other solutions because it uses a combination of ontology and rule-based mechanisms. The context integrity constraint approach defined for dynamically updated *ContextAssertions* is furthermore functionally equivalent to the defeasible logic reasoning approach proposed by Bikakis et al. [Bikakis and Antoniou, 2010] to deal with ambiguous information, having the added benefit that constraint detection rules can express conditions over both assertion content and annotations.
As explained, ontology reasoning is performed in a controlled manner (configuration of which is further discussed in Section 5.2.2). Furthermore, we show in Section 7.1 that the specific implementation of the storage of *ContextAssertion* and *ContextAnnotation* information within the CONSERT Engine can lead to a more efficient way to perform ontology-based *realization* and *knowledge base consistency* inference procedures, by loading only necessary subsets of the runtime knowledge base into the reasoner.

Chapter 5

Adaptable Context Provisioning

Once an approach for representing, storing and reasoning about context information is established, the next step in designing a context management middleware (CMM) involves determining how to create a *context provisioning* process around the chosen modeling and inference methods. In chapter 3 we have seen that one of our main objectives, given the shortcomings of the reviewed related work, is to propose a CMM architecture where the components (units) that are responsible for different steps of the provisioning process can be individually and flexibly deployed on different machines, started or stopped according to the application needs. Furthermore, we mentioned that we want to set out and empower the application developer with the ability to configure the functionality of each provisioning unit and specify *when* and *how* they should react and *adapt* the provisioning process depending on the perceived *usage* of the managed context information.

In this chapter we describe the multi-agent architecture of our CMM and present the policy-based configuration/control means used to adapt the provisioning process managed by the agents. In Section 5.1 we start by explaining why multi-agent technologies are a perfect fit for our intended middleware design. We then continue detailing this design by introducing the different agent types that make up our context provisioning units and presenting the *environment* in which they execute.
Section 5.2 presents the agent functionality in more detail. We introduce the notion of *context provisioning policies* and detail the type of specification they contain regarding control and adaptation of the context provisioning process.
In Section 5.3 we then present how the agents *integrate* the policy specifications in their behavior in order to control context provisioning.
The interaction protocols taking place between the agents given their provisioning responsibilities and assigned policies are presented in Section 5.4.
The chapter concludes with Section 5.5, where we discuss how the context management requirements mentioned in Sections 3.1.1 and 3.1.2 are addressed.

5.1 Multi-Agent Based Architecture

One of the defining characteristics of the CONSERT Middleware is its architectural design that is based on techniques and tools from the Multi-Agent System (MAS) domain. The choice for this approach is motivated by the potential to achieve flexible management of middleware deployment and functionality. We provide a rationale for the use of MAS techniques and then introduce the agents and their environment.

5.1.1 Rationale

Multi-Agent Systems are a field of research with a contribution activity that spans a time line of more than 25 years and domain space comprising research areas such as distributed communication, control and communication, planning, constraint satisfaction, game theory, argumentation and more. However, on top of these research directions particular paradigms of development were born, namely *agent-oriented programming* (AOP) [Shoham, 1993] and *multi-agent oriented programming* (MAOP) [Bordini et al., 2005]. These paradigms argue for using the notion of *agency* and everything related to it (e.g. environment programming, organizational programming) as an application design methodology. Multi-agent systems and technologies are supported by a vigorous research community, but it may be argued whether MAOP has had the same *impact* on software application development as other technologies which had started from academia (e.g. semantic web, machine learning, big data). However, the MAS research community is starting to put increasingly strong emphasis on the practicality of developed frameworks and tools and a recent survey [Müller and Fischer, 2014] has found that MAS technologies have been successfully deployed in a significant number of applications coming from niche markets. Though currently not as much in the spotlight as other software engineering domains, multi-agent systems are highly useful in specific sectors.
It is our opinion that context management middleware development is one of these sectors, especially since topics such as flexible control, distributed deployment or coordination are all relevant within a context management system and were intensely studied in the MAS literature. In what follows we provide a short overview of the notion of agency and then present further arguments why the agent-based design method is suitable for defining our middleware architecture.

The term *agent* has a broad sense and there is currently no real commonly agreed upon definition of the notion within the agent and multi-agent research community. However, by examining the literature, two clear usages of the term (expressed based on attributed requirements) can be identified: the *weak* notion of agency and the *strong* notion of agency [Wooldridge and Jennings, 1995].
The weak notion of agency denotes a software-based computer system with the following properties [Wooldridge and Jennings, 1995]:

- Autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.
- Social ability: agents interact with other agents (and possibly humans) via some kind of agent communication language.
- Reactivity: agents perceive their environment and respond in a timely fashion to changes occurring therein.
- Pro-activeness: in addition to acting in response to their environment, agents are able to exhibit goal-directed behaviour by taking the initiative.

Wooldridge and Jennings [Wooldridge and Jennings, 1995] remark that the stronger notion of agency is most notably used by researchers coming from the field of Artificial Intelligence and is characterized by additional properties (with regard to the ones above) that conceptualise the notion using concepts that generally apply to humans. Thus, an agent could be characterized as:

- Displaying and functioning based on *mentalistic* states: belief, desire, intention, obligation (the BDI agency model) [Shoham, 1993].
- Further extending the BDI model with an *emotional* component [Bates et al., 1994; Pereira et al., 2008].

- Showing *intelligence* by having the ability to *learn* about the environment in which they are placed, as well as about the behavior of other agents in the environment [Alonso et al., 2001]. The purpose of the learning processes is the adaptation to the conditions of the environment and better achievement of goals entrusted to them.

As explained previously, our intended use of multi-agent systems is as an elegant and well-fitted engineering solution for the design and development of a context management middleware architecture. Considering the main context provisioning operational blocks and transverse functionality identified in Section 3.1.1, we adopt the *weak* notion of agency in terms of describing the behavior of our agents. Furthermore, given that the analysis made by [Müller and Fischer, 2014] shows that mature agent-based applications are built using existing agent development frameworks, we dwell on this insight and choose to use a well known agent development framework to help with the engineering of the behavior and communication of our context management agents (more details on this in Section 7.3).
These design choices bring the following benefits to our approach:

- Encapsulation of each aspect of the context provisioning life cycle (sensing, coordination, dissemination, usage). This reflects not only in decision making, but also in interaction planning, error handling, adaptation management. In short, it opens up the possibility for achieving autonomy of each provisioning step (this would be an instrumental step in the vision of Sensing-as-a-Service [Perera et al., 2014b] oriented applications).
- Easily create interaction protocols by using existing agent communication language standards (e.g. FIPA ACL[1]) and communication infrastructure support (e.g. those offered by the JADE [2] framework). The advantage of these standards is that they contain a communication intention semantics and encapsulate both success and failure interaction sequences, such that both successful invocation and error handling are uniformly treated and do *not* amount to a set of ad-hoc callback procedures which need to be correlated in a custom way by the application developer. This represents a noticeable facilitation in the development of our context management middleware.
- Each provisioning step can be described in terms of *behaviors* which the corresponding provisioning agents use to either *react* to received messages (a *ContextAssertion* update, a query, a request to modify the sensing update rate), or to actively pursue *maintenance goals* expressed by the context-aware application developer as *policies* that govern the functionality of the context provisioning process (refer to Section 5.2). Therefore, we see that the reactivity and pro-activity attributes of an agent fit well with the requirements of flexible context provisioning.

Given the above arguments, the next section introduces the multi-agent based architecture that operates the context provisioning process in the CONSERT Middleware.

5.1.2 Context Provisioning Agents

The CONSERT Middleware defines a set of five agent types. Each agent is responsible for a part of the context provisioning process, as we explore in what follows, and their combined functionality constitutes what we call a *CONSERT Management Unit* (CMU). In chapter 6 we will see that the agent *composition* of a CMU can differ according to the provisioning aspect required to be enacted on a physical machine. That is, not all agents have to be present on every deployment, but rather the CONSERT Middleware offers the flexibility of selecting which agents need to run on which machine (e.g. a dedicated sensing machine, a management/coordination machine, a consumption/client machine). Furthermore, for a given application, multiple CMUs can be deployed and organized in different ways for managing and provisioning context to the

[1] http://www.fipa.org/repository/aclspecs.html
[2] jade.tilab.com

applications.
However, in this section we wish to introduce the main functionality of CMM agents and the *environment* in which they operate *within* a Context Management Unit. The flexible configuration and deployment of one or more CMUs within an application is the subject of the next chapter.

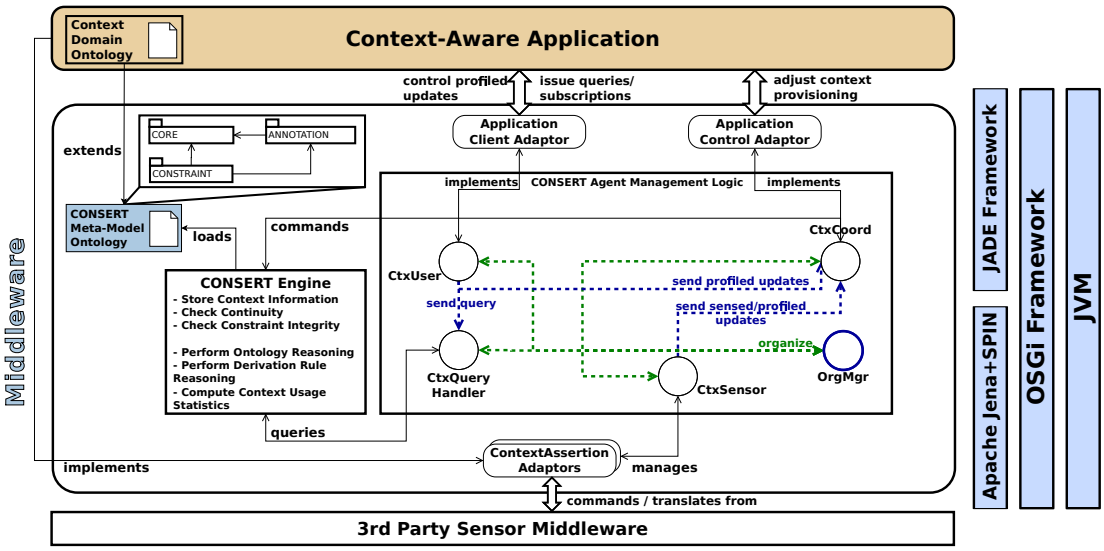

Figure 5.1: CONSERT Middleware: multi-agent architecture and interactions

Figure 5.1 shows an overview of the typical setup of a CMU. It presents the CMM agents and the main provisioning interactions that take place between them, as well as the service component based environment in which they operate (more details on the component-based implementation in Section 7.3). The services in the environment help the agents perform reasoning tasks, extract data from third-party sensor middleware and respond to application demands.

The CONSERT Middleware defines the following agent types: `CtxCoord`, `CtxQueryHandler`, `CtxSensor`, `CtxUser` and `OrgMgr`. We review their main functionality in what follows.

`CtxSensor` agent : is responsible for managing interactions with sensors and with the `CtxCoord` agent of its own CMU to handle *provisioning commands* (e.g. start/stop sending updates, change update rate - cf. Section 5.2.2 for more details). Communication with physical sensors is achieved by using `Context Assertion Adaptors` from the environment.

`CtxCoord` agent : is in charge of the coordination of the main life cycle of a CMU. It creates and manages a CONSERT Engine instance (described in the previous chapter) to control context derivation and consistency. As we explore more in Section 5.2.2, the `CtxCoord` establishes *what* context information needs to be acquired/derived and *how/when* to do so. He can further more interface with the application level to receive instructions on how to set/alter parameters that control the CMU provisioning life cycle.

`CtxQueryHandler` agent : is in charge of the dissemination of context information. The `CtxQueryHandler` maintains access to the instance of the CONSERT Engine managed by the `CtxCoord` deployed in the same CMU. By default, it uses this local knowledge base to answer to queries. In case of more complex settings however, involving multiple agents, it can

participate in a *federation* protocol which involves interacting with `CtxQueryHandler` agents deployed in other CMUs (in case of a decentralized deployment detailed in the next chapter).

`CtxUser` agent : is in charge of interfacing with the application and acting as a *prosumer* (i.e. producer and consumer - see more in Section 6.4.4) of context information. It receives instructions from the applications regarding what context information to receive/send and how to do so.

`OrgMgr` agent : is responsible for controlling the deployment of a CMU and of the entire middleware in cooperation with other `OrgMgr` agents. That is, this agent launches and controls the state of CMM agents (started / stopped / uninstalled) within the CMU it manages, but can also hold the overview of other deployed CMUs in case of a decentralized deployment scheme (as discussed in chapter 6). It furthermore acts as a yellow pages agent for the agents in its CMU and is in charge of managing mobility aspects. Given that the purpose of the `OrgMgr` agent is to manage the deployment and lifecycle of a CMU, the details of its functionality will be discussed in Section 6.3 of the chapter on middleware deployment. Having briefly introduced the agent types and their responsibilities, let us now provide an overview of the environment in which they execute.

5.1.3 Context Provisioning Agent Environment

The environment of a CMU is constituted as a set of *services* that enable the agents within the CMU to communicate with sensing and application levels, as well as to perform actions for storing and reasoning about the context information they manage. The concrete means in which the agents acquire or expose the services in their environment is an implementation specific aspect and will be detailed in Section 7.3. In what follows, we present what these services are and how the CMM agents interact with them.

CONSERT Engine. The CONSERT Engine is the service component of the agent environment that handles storage, inference and consistency of the context information managed by the agents of a CMU. This means that reasoning about context happens within the agent environment. However, as visible in Figure 5.1, it is the `CtxCoord` agent that creates and commands an engine instance, controlling its functionality. Remember from Section 4.4.1, that the CONSERT Engine exposed several interfaces that allowed an external management system to affect its workings. The `CtxCoord` agent uses the *CONSERT Engine Tasking Service* to control enabled *DerivationRules*, trigger ontology reasoning and manage the lifetime of stored *ContextAssertion* instances.
At the same time, the `CtxQueryHandler` uses the query interface and *ContextAssertion Insertion Notifier* of the CONSERT Engine to execute received queries and check for answers to subscriptions every time a *ContextAssertion* referenced by those subscriptions is updated.

ContextAssertion Adaptors. These services are implemented by the application developer and provide an interface with the given sensor infrastructure that is part of the application. The adaptor allows the `CtxSensor` to translate retrieved sensor data into the corresponding *ContextAssertion*, *EntityDescription* and *ContextAnnotation* statements of the application context model created using the CONSERT Ontology. In a CMU, one such adaptor is created for each type of *ContextAssertion* that the `CtxSensor` agents within the CMU have to manage.

Application Client/Control Adaptors. Whereas the `ContextAssertion Adaptors` pre-exist in the agent environment, the `Application Client` and `Application Control` adaptors are exposed by the `CtxUser` and `CtxCoord` agents respectively. The client adaptor presents the service interface which allows the application to instruct the `CtxUser` on launching queries and subscriptions, as well as sending profiled *ContextAssertions* (i.e. act as a sensor) or static *EntityDescriptions*.
The `Application Control Adaptor`, on the other hand, allows the application level to set/modify some of the parameters that control the context provisioning (more on this in Section 5.2.2). For example, the application can use the service to instruct the `CtxCoord` agent to set the current resolution service for a given *ContextAssertion* integrity constraint.

5.2 Context Provisioning Agent Policies

Having introduced the agent types and their environment, we now want to more closely present the agent provisioning functionality. In our state of the art overview of context management systems we mentioned that an important complementary aspect is that of adaptation/control of the context provisioning process (cf. Section 3.1.1). Furthermore, in Section 3.3 we explained that in order to address the non-functional requirement of ease of configuration and development, we wish to focus on a declarative means for specifying *what*, *when* and *how* should be changed in the context provisioning process enacted by CONSERT Middleware agents. Consequently, our approach to context provisioning management is based on *policies* that guide the behavior of the CMU agents. This choice relies on our belief that declarative policy-based control specifications provide the best balance between application engineering flexibility and development effort.

Because the notion of a *policy* has different meanings in different research domains, we start off by providing our definition of this concept.

Definition 5.2.1 (Context Provisioning Policy)**.** A *Context Provisioning Policy* is a collection of ***parameters*** that control aspects of context provisioning and ***rules*** that set conditions for setting/modifying the value of these parameters.

In the CONSERT Middleware, the *parameters* that are part of a provisioning policy govern both the flow of context information within a CMU and the type of inferences carried out by the CONSERT Engine. That is, they affect both interactions between CMM agents as well as the functionality of the agent environment. Designers can specify *what* and *how* information is transmitted and how inference and query handling are prioritized depending on current provisioning requirements (e.g. number of subscriptions, frequency of queries). In this section we will present the type and role of existing provisioning parameters and rules on a conceptual level. Two aspects of the context provisioning process, *sensing* and *coordination* are currently impacted by context provisioning policies. The specifics of each one are described next.

5.2.1 Sensing Policies

Sensing policies contain parameters that control *how* updates of sensed information are to be forwarded. Within such a policy, for each type of *ContextAssertion*, the paremeters listed in Table 5.1 can be specified.

The *update mode* specifies the nature of the forwarding mechanism. In a *time-based* mode updates of the particular *ContextAssertion* are send at specified time intervals. In a *change-based* mode, on the other hand, updates are sent only upon change from a previous instance. The detection of a change is dictated by the specific `ContextAssertion Adaptor` and is based on the values of both content and annotations of a *ContextAssertion*. For *time-based* modes,

Parameter	Values	Role
update mode	time-based, change-based	Specify condition for making a new update.
update rate	number in seconds	Specify the refresh rate for time-based updates.

Table 5.1: List of parameters available in a sensing policy.

the *update rate* (in seconds) can also be expressed.
It is worthy to note that these parameters govern the way in which updates for a given `CtxSensor` are sent to the corresponding `CtxCoord` agent. Notice that with respect to agent communication this is a controlled *push* update scheme. The communication between the `CtxSensor` agent and the physical sensor from which he extracts the *ContextAssertion* information can vary (i.e. push, pull mode, fixed update rate) depending on the capabilities of the specific sensor. It is here that the `Context Assertion Adaptor` plays an important part, namely to accustom the fixed communication means between `CtxSensor` and physical sensor, to the dynamic provisioning requirements set by the sensing policies and possible commands received from the `CtxCoord`.

Below is a small pseudo-configuration excerpt that sets the update mode and rate for the presence and luminosity related *ContextAssertions* from the AmI-Lab ad hoc discussion part of our reference scenario (details in Section 8.2.3).

Presence
`update mode` := *time-based*
`update rate` := 5 s

Luminosity
`update mode` := *change-based*
`update rate` := 0 s

This configuration tells the `CtxSensor` agents which are in charge of managing the updates for the above mentioned *ContextAssertions* that presence updates (i.e. detections of the bluetooth address of smartphones in the laboratory) are to be sent every 5 seconds. On the other hand, updates for luminosity must only be sent when the value of the *ContextAssertion* instance changes from the previous one.

5.2.2 Coordination Policies

Sensing policies specify the desired default context update modes. *Provisioning coordination policies* define settings (via *control parameters*) and specify actions (via *control rules*) that address such dynamics. They are defined in a file which is read by the `CtxCoord` agent upon initialization. In what follows, we give some perspective over the provisioning control parameters and provisioning control rules from a conceptual point of view.

Control parameters govern relevant settings of the context provisioning process. They affect both context *information transmission* and *inference processes*. As we can see in Tables 5.2 and 5.3, control parameters can be divided into two categories: *general* parameters and *assertion-specific* parameters. The former provide a default (general) setting, while the latter can override the general setting with a value specific to a *ContextAssertion* type. In terms of transmission, control parameters specify *which ContextAssertions* currently need to be enabled and how long a particular instance of a *ContextAssertion* must be kept in the CONSERT Engine working memory (its TTL). For these parameters there is both a general as well as a *ContextAssertion*-specific value.
With respect to inference, the *assertion enabling* parameter configures also the currently active

Parameter	Values	Role
default assertion enabling	true/false	Specify if assertion updates are enabled by default.
default ont. reasoning interval	number in seconds	Default time span between calls to ontology reasoner.
default TTL	number in seconds	Default time to live for any *ContextAssertion* in the runtime storage
default integrity constraint resolution	String in enumeration	Identifier of the service handling integrity constraint resolutions
default uniqueness constraint resolution	String in enumeration	Identifier of the service handling uniqueness constraint resolutions
default run window	number in seconds	Length of time window over which context usage statistics are computed
inference scheduling service	String in enumeration	Identifier of service providing priority scheduling for *ContextDerivationRules*

Table 5.2: List of general parameters available in a coordination policy.

ContextDerivationRules by specifying enabled *derived ContextAssertions*. The type of inference scheduling service (e.g. first-come first-served, priority based) is a parameter which has only a general value and controls the order in which enqueued *ContextDerivationRules* are to be executed.
With regard to constraint checking, remember from Section 4.4.1 that the CONSERT Engine can work with an external service which is called upon detection of an integrity constraint. These services are part of the agent environment of the given CMU and the *default* and *assertion-specific* constraint resolution parameters specify the *identifiers* of these services. Using the `Application Control Adaptor`, the application can change the desired resolution service by changing the initial identifier value of the corresponding control parameter.
Lastly, remember that in Section 4.3.1 we talked about how inference in the CONSERT Engine has both a dynamic (*ContextDerivationRules*) and a static (ontology reasoning for *ContextEntities* and *EntityDescriptions*) component. More specifically, we mentioned that there can be context models which define *ContextEntities* using OWL class construction axioms that involve *EntityDescriptions* and *ContextAssertions*. We hinted then, that in order to perform the required *instance realization* ontology reasoning procedure, we would have to invoke the ontology reasoner on every *ContextAssertion* update, which could become costly depending on the assertion update rate. In order to provide control over the invocations of the ontology reasoner, the *ont reasoning interval* parameter specifies a general time span between successive calls to the reasoner. However, this parameter can be customized for each *ContextAssertion* type, in which case a further optimization will be performed, namely the ontology reasoner will be loaded only with the *ContextEntity* instances for which their class axiom definition references the given *ContextAssertion* type, thereby reducing the size of the ABox. In Section 7.1.1 we will explain how our storage of *ContextEntities* and *ContextAssertions* enabled us to achieve this more easily.

Parameter	Values	Role
specific assertion enabling	true/false	Specify if updates are enabled for the specified *ContextAssertion*.
specific ont. reasoning interval	number in seconds	Time span between calls to ontology reasoner for specified *ContextAssertion*.
specific TTL	number in seconds	Time to live for specific *ContextAssertions* in the runtime storage
specific integrity constraint resolution	String in enumeration	Service handling integrity constraint resolutions for specific *ContextAssertion*
specific uniqueness constraint resolution	String in enumeration	Service handling uniqueness constraint resolutions for specific *ContextAssertion*
specific observation window	number in seconds	Length of time window over which context usage statistics are computed for specific *ContextAssertion*

Table 5.3: List of assertion-specific parameters available in a coordination policy.

To exemplify a setup of these parameters, we again refer to the AmI-Lab part of our scenario and provide the following pseudo-configuration excerpt.

General parameter values

`default assertion enabling`	:=	*false*
`default ont reasoning interval`	:=	10 s
`default observation window`	:=	20 s
`default TTL`	:=	100 s
`default uniqueness constraint resolution`	:=	*PreferNewest*
`default integrity constraint resolution`	:=	*PreferAccurate*

Assertion-specific parameter values

`assertion enabling`	:=	*device presence ContextAssertion* : *true*
`assertion enabling`	:=	*person location ContextAssertion* : *true*

The example shows us that updates for *ContextAssertions* are not enabled by default, except for device presence. The person location *ContextAssertion* is actually of a *derived* acquisition type. The *true* value for the specific *assertion enabling* parameter means that all the *ContextDerivationRules* that infer values of this *ContextAssertion* will be enabled. In dealing with uniqueness constraints, the service preferring the newer *ContextAssertion* instance is used by default. For general integrity constraints, we prefer the most accurate one (i.e. discrimination based on the certainty annotation).
Apart from transmission and inference, the *observation_window* parameter configures the length of the time window over which statistics of context information and inference usage are computed by the CONSERT Engine. This parameter can again have both general and *ContextAssertion*-specific configurations. In our scenario we only specified a default value for the parameter. The statistics gathered by the CONSERT Engine (presented in Section 5.3.1) together with snapshots of its current knowledge base constitute the triggering conditions of the *provisioning control rules*, which we discuss next.

Control rules are the concrete means to specify *adaptation actions* of the context provisioning process. Some of these actions can change the value of control parameters introduced earlier (e.g. *assertion enabling*, *inference scheduling service*), while others will have as consequence the alteration of sensing policy parameters discussed in the previous section (i.e. changes to the update mode or update rate for specific *ContextAssertions*). Parameters which can currently

not be affected by means of control rule output (e.g. type of constraint resolution service for violations triggered by a given *ContextAssertion* type) can still be altered at runtime by the application level by means of a more direct mechanism, namely the `Application Control Adaptor` (cf. Figure 5.1).
The currently available rule outcomes in the CONSERT Middleware are listed in Table 5.4.

Rule	Type	Effect
StartAssertionCommand	assertion specific	Enable updates for the specified *ContextAssertion*.
StartRuleCommand	assertion specific	Enable all *ContextDerivationRules* which derive the specified *ContextAssertion*.
StopAssertionCommand	assertion specific	Disable updates for the specified *ContextAssertion*.
StopRuleCommand	assertion specific	Disable all *ContextDerivationRules* which derive the specified *ContextAssertion*.
UpdateModeCommand	assertion specific	Alter the update mode or rate for the specified sensed *ContextAssertion*
InferenceScheduling Command	general	Set the active *ContextDerivationRule* scheduling service

Table 5.4: List of control rule output commands available for use in a coordination policy.

We mentioned in the previous chapter that the *ContextDerivationRules* used to infer new *ContextAssertion* instances are implemented as SPARQL CONSTRUCT queries. To maintain uniformity of the approach, we use the same principle to give shape to provisioning control rules. Details about the implementation and examples based on the reference scenario are provided in Section 7.1.3.

5.3 Context Provisioning Policy Execution

In this section we move to the operational aspect of context provisioning adaptation. We have seen previously that policies help set up the initial provisioning settings (sensing and coordination parameters) and supply the rules that specify when and how to change them.
We wish to investigate now how the provisioning adaptation processes are put into motion. We start first by discussing what kind of context usage statistics the CONSERT Engine is capable of providing during its runtime. Afterwards, we show how provisioning control rules are executed and how their results are further used by the `CtxCoord` agent.

5.3.1 Gathering Provisioning Statistics

The triggering conditions of a provisioning control rule are expressed based on a snapshot of the CONSERT Engine knowledge base (i.e. the information set currently relevant for the context provisioning process managed by the CMU agents used in an application) and a set of statistics regarding CONSERT Engine functionality aspects.
To see the use of these context usage statistics, consider the episodes from the reference scenario where Bob turns off the projector during the ad-hoc discussion in the AmI-Lab. After five minutes, updates for luminosity are disabled since they are deemed to be no longer required. Similarly, when all people have left the room, all sensors except the presence detection ones can

stop sending updates. At the CONSERT Engine level, detection of this non-usage situations can be translated into an inspection of the time interval since last there was any query for the *ContextAssertions* relevant to these situations.
The CONSERT Engine collects the set of statistics shown in Table 5.5.

Statistic	Value	Effect
nr. queries	integer number	Number of queries received for specified *ContextAssertion* during last OBSERVATION_WINDOW time interval.
nr. successful queries	integer number	Number of successfully answered queries for specified *ContextAssertion* during last OBSERVATION_WINDOW time interval.
nr. subscriptions	integer number	Number of existing subscriptions for specified *ContextAssertion*.
time since last query	number in ms	Elapsed time since last query received for specified *ContextAssertion*.
nr. derivations	integer number	Number of *ContextDerivationRule* inferring the specified *ContextAssertion* executed during last OBSERVATION_WINDOW time interval.
nr. successful derivations	integer number	Number of successful inferences of the specified derived *ContextAssertion* performed during last OBSERVATION_WINDOW time interval.
is derived assertion	true/false	*ContextAssertion* instance for which statistic is computed is obtained through derivation.
is enabled assertion	true/false	*ContextAssertion* instance for which statistic is computed is being currently active/inactive.

Table 5.5: List of context usage statistics gathered by the CONSERT Engine at runtime.

Notice that in almost all of the statistics depicted above the computation of their value occurs over the time interval specified by the *observation_window*, introduced in the previous section. The `CtxCoord` agent uses the CONSERT Engine command service (cf. Section 4.4.1) to collect such statistics every *observation_window* time spans. For this it uses the general value of the this parameter. However, the actual values of the statistics returned by the call to the CONSERT Engine service are computed based on *ContextAssertion*-specific values of the *observation_window* parameter, if they exist. As was the case of the control parameters and rules, the vocabulary used to express the context usage statistics is implemented using the same ontology, and its form will be discussed in Section 7.1.3.

5.3.2 Control Process

The execution of context provisioning control rules is carried out by the `CtxCoord` agent. Upon initialization, the agent will read all provisioning control parameters and create an index of all provisioning control rules.

CONSERT Engine setup The `CtxCoord` configures the functionality of the CONSERT Engine using the parameters relating to default enabled *ContextDerivationRules* and specified identifiers for constraint resolution and inference scheduling services. It also sets the values of the general and *ContextAssertion*-specific *observation_window* parameters which the CONSERT Engine will use to compute the context usage statistics.

Executing Provisioning Control Rules The `CtxCoord` agent itself uses the value of the general *observation_ window* to schedule the execution of the control rules it has indexed. Now, an important aspect to note is that control rules may have contradictory outcomes (e.g. one rule implies enabling a derivation rule while another one disables it). To help developers keep a consistent result, the control rules can be partitioned into *ordered* execution groups. For example, rules that specify deactivations of *ContextAssertions* can be grouped in an execution group which will be run *before* the group containing the rules that mandate activations. It is up to the designer to ensure that the rules within a group do not themselves have contradictory outcomes.
The rules are then executed in the order defined by the sequence of execution groups. However, the rules from later execution groups *override* contradictory results from rules in earlier groups. This consistency ensuring mechanism is a form of preference-based execution, where the preference of one rule over the other is explicitly given by its execution order.

Using Control Rules Outcomes After running the rules, the actions of the `CtxCoord` agent depend on the type of their output. For commands such as `StartAssertionCommand`, `StopAssertionCommand` or `UpdateModeCommand` the `CtxCoord` determines all *CtxSensors* that provide the *ContextAssertion* concerned by those rule outputs and sends them a *TaskingCommand* (cf. Section 5.4.1) request wrapping over the content of the control rule output. The `CtxSensor` agents receive the command and conform to its request. In this way, the `CtxCoord` agent effectively coordinates the provisioning behavior of the sensing agents.
On the other hand, if the control rule output imply changes in inference handling, the `CtxCoord` uses the CONSERT Engine command interface to perform the required adaptation. In case of `StartDerivation Command` or `StopDerivationCommand` the agent determines all *ContextDerivationRules* which can infer the *ContextAssertion* specified by these rule outputs and marks them as active. In case of an `InferenceSchedulingCommand`, the agent simply informs the CONSERT Engine of the new inference scheduling service identifier.

An additional aspect that influences the context provisioning process in terms of enabled/disabled *ContextAssertion* and *ContextDerivationRules* is represented by the activation requests coming from the `CtxQueryHandler` agent (cf. Section 5.4.2). Thus, in a typical application implemented using the CONSERT Middleware, context information update activations will be controlled in their majority by `CtxQueryHandler` requests which correspond to a query or subscription from a `CtxUser` agent. On the other hand, deactivations of updates will be performed based on developer specified conditions that form the body of a `StopAssertion` or `StopDerivation` context provisioning control rule.

5.4 Context Provisioning Interactions

In previous sections we presented the agent responsibilities and the means by which their context provisioning adaptation behavior is specified and executed. We now focus on the interactions that take place between the agents, given their responsibilities and the policies they manage. Essentially, the messages exchanged between them are the means by which both the flow of context information and the control / adaptation of this flow are ensured.

The context provisioning process is composed of two main interaction chains. The ***sensing chain*** concerns the updates that `CtxSensor` agents send to the `CtxCoord` of a CMU. The ***request chain*** regards queries and subscriptions that `CtxUsers` make to the `CtxQueryHandler` of a CMU. These two chains connect with each other through the working of the CONSERT Engine and will be detailed in the following subsections.

5.4.1 Provisioning Sensing Chain

The *sensing chain* comprises the communication protocol between the `CtxSensor` and `CtxCoord` agents. The principal messages exchanged between these agents are reported in Tables 5.6 and 5.7, where we detail the objective of the interaction, the type of protocol used (according to FIPA standards[1]) the role played in the protocol (initiator or receiver) as well as the condition that triggers the interaction. The interactions within the protocol can also be followed in their sequence in Figure 5.2.

Interaction	IP	Role	With	When
Register Agent	FIPA Request	Init.	*assigned* `OrgMgr`	Announce sensing service to *assigned* `OrgMgr`
Find Coordinator	FIPA Request	Init.	*assigned* `OrgMgr`	Intend to connect to a `CtxCoord`
Publish Assertions	FIPA Propose	Init.	`CtxCoord`	Intend to send info. to `CtxCoord`
Assertion Update	FIPA Inform	Init.	`CtxCoord`	New sensory observation
Tasking Command	FIPA Request	Recv.	`CtxCoord`	`CtxCoord` alters update method/rate

Table 5.6: Interaction table for `CtxSensor` shows conversation type, used Interaction Protocol (IP), role in the conversation, interaction counterpart and trigger condition.

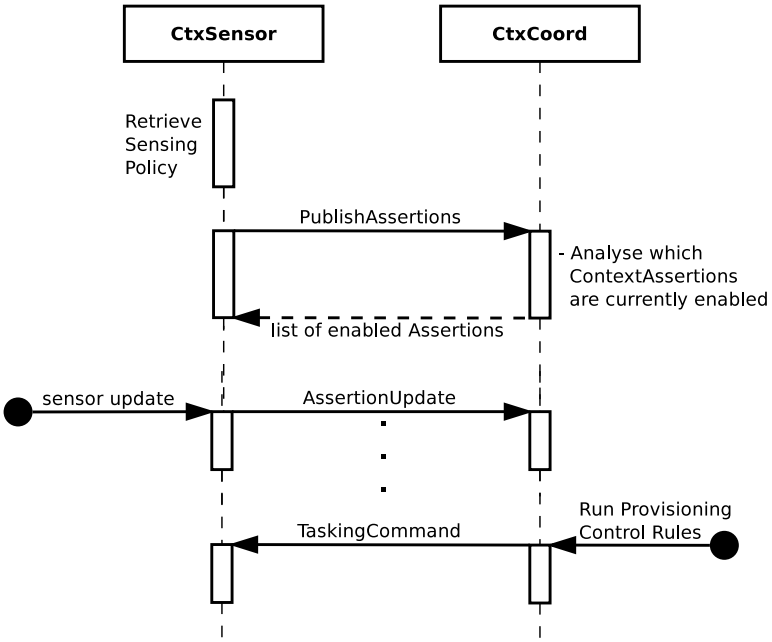

Figure 5.2: Provisioning interactions within the *sensing chain*.

After registering with the `CtxCoord`, a `CtxSensor` agent will look for a sensing policy that specifies the default (initial) indications for *how* to provide updates for the sensed *ContextAssertions* of which it is in charge. Thus we can see that the initial sensing behavior of a `CtxSensor` agent is governed by the sensing policy that it manages.
The sensor agent then starts a FIPA Propose protocol to publish its *ContextAssertion* update capabilities (as configured by the policies) to the `CtxCoord`. The `CtxCoord` acknowledges the publishing and returns the subset of *ContextAssertions* for which the sensor agent currently has to send updates. As explained in Section 5.2.2, the `CtxCoord` inspects the *default* and *assertion-specific assertion enabled* control parameters from its provisioning coordination pol-

[1] http://www.fipa.org/repository/ips.php3

icy, which tell it which *ContextAssertion* updates are declared by the application developer as being enabled by default.

To take a short example and see how this is useful, consider the case of the AmI Laboratory in our reference scenario. While Alice sits alone waiting for her colleagues to appear such that they may have their discussion, it is clear that she cannot be in an ad-hoc discussion by her self. Therefore, the sensors that provide information about noise level or body posture near each desk need not actively send updates. Thus, when the respective `CtxSensor` agent would first come online and publish their sensing capability, the `CtxCoord` managing the AmI-Lab smart room would instruct them to wait for its request to start sending updates. On the other hand, a basic condition for attempting to see if there is an ad-hoc meeting going on in the laboratory is the fact that there must be at least two people present in the room. Therefore, presence sensors must be active at all time and so the coordination policy guiding the behavior of the `CtxCoord` agent will specify that updates for person location related *ContextAssertions* are enabled by default.

Interaction	IP	Role	With	When
Register Agent	FIPA Request	Init.	*assigned* `OrgMgr`	Announce coordination service to *assigned* `OrgMgr`
Register Query Handler	FIPA Request	Recv.	`CtxQueryHandler`	`CtxQueryHandler` announces its services.
Publish Assertions	FIPA Propose	Recv.	`CtxSensor`, `CtxUser`	`CtxSensor` / `CtxUser` announce capabilities
Assertion Update	FIPA Inform	Recv.	`CtxSensor`, `CtxUser`	Context information update available
Activate Assertion	FIPA Request	Recv.	`CtxQueryHandler`	Query made for inactive *ContextAssertion*
Tasking Command	FIPA Request	Init.	`CtxSensor`, `CtxUser`	Activate/Stop/Alter context updates

Table 5.7: Interaction table for `CtxCoord`

For the *ContextAssertions* which are marked as active, the `CtxSensor` sends updates using FIPA Inform messages, at the rate initially specified by its *sensing policy*. However, as seen in the brief example above, the provisioning requirements related to the *ContextAssertions* update capabilities of a `CtxSensor` may change during runtime. When Bob and Cecille enter the laboratory and sit down at the desk with Alice for their discussion, the conditions for the *possibility* of an ad-hoc meeting are met. Therefore, at some point (see details of the request chain in the next subsection) the `CtxCoord` will request that `CtxSensor` agents managing the noise level and body posture *ContextAssertions* enable the updates for these assertions. Moreover, we explained that every *observation_window* period, the `CtxCoord` monitors the *usage* (e.g. how many queries were received for a given *ContextAssertion* during the specified time window) of the context information that he manages. The agent can then run provisioning control rules declared in its assigned coordination policy. Thus, when Bob and Cecille leave the laboratory, rules whose output is a `StopAssertionCommand` targeting noise level and body posture *ContextAssertions* will be triggered (since no more subscriptions for ad hoc meeting will exist).
The `CtxCoord` agent uses the `Tasking Commands` that result from the execution of the control rules to instructs the corresponding `CtxSensor` agents to stop sending updates for the targeted *ContextAssertions*. At the same time it will disable the *ContextDerivationRule* which infers the existence of the ad hoc meeting *ContextAssertion*.

5.4.2 Provisioning Request Chain

Within the *request chain*, a `CtxUser` that registered with a `CtxQueryHandler` can ask for information via FIPA Query or FIPA Subscribe protocols. There are two types of queries that a `CtxUser` can make: local and domain-based (cf. also Table 5.8). Here we will discuss about the first option, whereas the second one will be further explored in the next chapter in Section 6.4.1, since it involves the mechanisms related to the deployment and communication between

distributed CMUs. The typical sequence of interactions within the request chain protocol is shown in Figure 5.3 and the individual agent messages are summarized in Tables 5.8 and 5.9.

Interaction	IP	Role	With	When
Register Agent	FIPA Request	Init.	*assigned* `OrgMgr`	Announce query service to *assigned* `OrgMgr`
Find Coordinator	FIPA Request	Init.	*assigned* `OrgMgr`	Intend to connect to a `CtxCoord`
Register Query Handler	FIPA Request	Init.	`CtxCoord`	`CtxQueryHandler` connects to `CtxCoord`.
Register Query User	FIPA Request	Recv.	`CtxUser` / `CtxQueryHandler`	`CtxUser` / `CtxQueryHandler` register as context consumers
Local Query	FIPA Query, Subscribe	Recv.	`CtxUser`	CtxUser makes a *local* query
Domain-based Query	FIPA Query, Subscribe	Recv.	`CtxUser` / `CtxQueryHandler`	`CtxUser` or `CtxQueryHandler` make a *domain-based* query
Get Query Base	FIPA Query, Subscribe	Init.	`OrgMgr`	Determine routing path(s) for a *domain-based* query
Activate Assertion	FIPA Request	Init.	`CtxCoord`	Queried *ContextAssertion* not active

Table 5.8: Interaction table for `CtxQueryHandler`

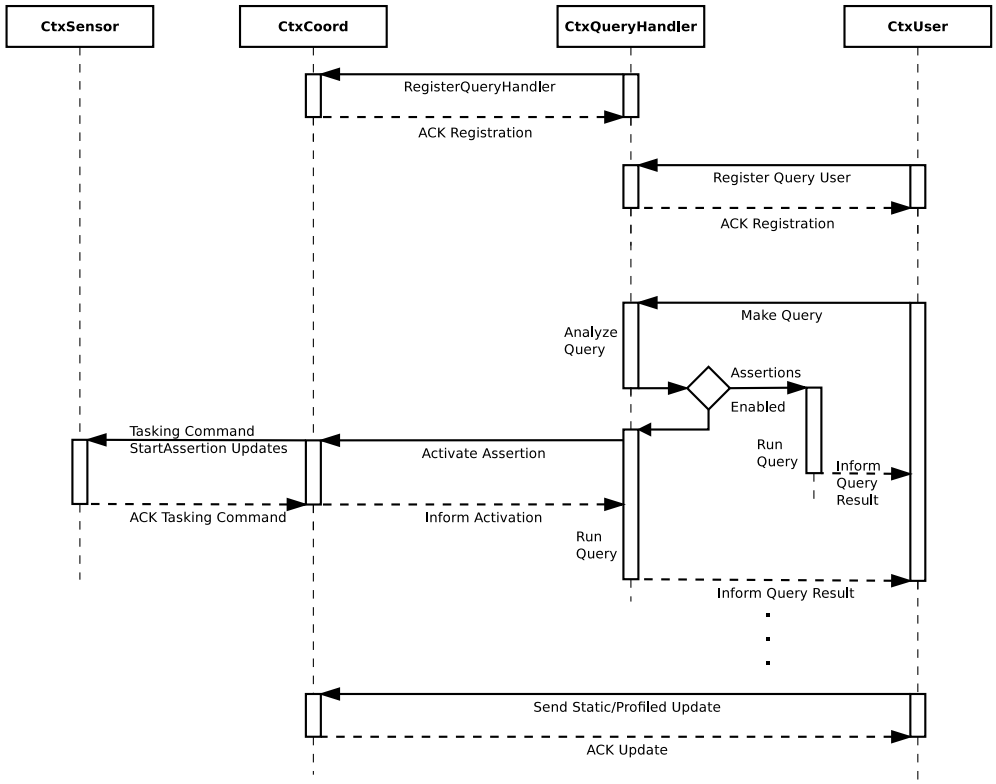

Figure 5.3: Provisioning interactions within the *request chain*.

In the case of the local query, the `CtxQueryHandler` will only attempt to provide answers using the knowledge base (i.e. the CONSERT Engine instance) it is currently connected to. However, remember from the short example in the previous section that a *ContextAssertion* can be inactive (i.e. there will be no instances of it in the runtime knowledge base of the CONSERT Engine). The `CtxQueryHandler` first analyzes the received request to see which type of *ContextAssertions* are referenced within. If all requested *ContextAssertion* types are active, the agent will immediately proceed with handling (i.e. posing the query or registering the subscription) the demand. Yet, if there exist *ContextAssertions* which are not enabled, the query/subscription protocol continues with the `CtxQueryHandler` notifying the `CtxCoord` of an activation need (*Activate Assertion* interaction in Table 5.8).

The coordinator handles the activation request based on the *acquisition type* of the *ContextAssertion* requested to be activated. It distinguishes between the *sensed ContextAssertions* (i.e. those updated by a `CtxSensor` agent) and the *derived* ones (i.e. those which are the output of a *ContextDerivationRule*). In the case of a sensed *ContextAssertion* the coordinator determines which *CtxSensors* can provide it and uses a *TaskingCommand* to tell them to start sending updates. In case of derived context, it uses the CONSERT Engine command interface to enable the corresponding *ContextDerivationRules*. Only when the `CtxQueryHandler` has the confirmation of *ContextAssertion* enabling will it pose the query to the CONSERT Engine.

Interaction	IP	Role	With	When
Register Agent	FIPA Request	Init.	*assigned* `OrgMgr`	Announce user service to *assigned* `OrgMgr`
Find Coordinator	FIPA Request	Init.	*assigned* `OrgMgr`	Intend to connect to a `CtxCoord`
Find Query Handler	FIPA Request	Init.	*assigned* `OrgMgr`	Intend to connect to a `CtxQueryHandler`
Find Context Domain	FIPA Request	Init.	*assigned* `OrgMgr`	Application wishes to know current *ContextDomain*
Register Query User	FIPA Request	Init.	`CtxQueryHandler`	`CtxUser` registers as context consumer
Publish Assertion	FIPA Propose	Init.	`CtxCoord`	`CtxUser` registers as producer
Static Context Update	FIPA Request	Init.	`CtxCoord`	EntityDescription update available
Profiled Context Update	FIPA Request	Init.	`CtxCoord`	Profiled ContextAssertion update available
Make Query	FIPA Query, Subs.	Init.	`CtxQueryHandler`	Initiate *local* / *domain-based* query / subscription

Table 5.9: Interaction table for `CtxUser`

This latter interaction protocol is an example of the coordination interactions that can modify the provisioning process within a CMU, as explainedat the end of Section 5.3.

One last set of provisioning interactions take place directly between a `CtxUser` and a `CtxCoord`. It is the case where the `CtxUser` agent acts also as a producer of context information, thus turning it into a context *prosumer* (i.e. producer and consumer). Since the updates coming from an `CtxUser` are the consequence of an explicit application action (via the `Application Client Adaptor`), rather than being acquired periodically from an environment, the information sent by the agent is either static context (i.e. *EntityDescriptions*) or profiled *ContextAssertions*.

An example taken again from our reference scenario is the case where the application on Alice's smartphone infers that she is in a ad hoc discussion and therefore determines that she is busy. The application then instructs the `CtxUser` agent to assert her profiled availability status to the `CtxCoord` of the AmI-Lab, such that queries for this type of information can be answered. An example of static information which can be asserted by a `CtxUser` is the fact that Alice is the owner of the smartphone with a given bluetooth MAC address. This information can be then used to derive the location of Alice within the laboratory (e.g. at which desk she is currently sitting). In Section 6.4.4 of the chapter on CMU deployment configurations we will again see that the `CtxUser` has several options when acting as a producer (i.e. local updates and domain-based updates).

5.5 Discussion

In this chapter we described our approach with respect to the architecture of context provisioning within the CONSERT Middleware. If in the previous chapter we talked about our contributions regarding context representation and reasoning based on the CONSERT Engine, we have now shown how those contributions are included within the larger scope of a set of context provisioning units that help bring about the provisioning of context information.
Specifically, we have seen that our architecture is based on design principles from the multi-agent system domain. Our context provisioning units are agents which encapsulate each functionality of the main context management life cycle as described first in Section 3.1.1. We have seen

that these agents are grouped into what constitute Context Management Units (CMUs), the flexible composition and deployment of which we mentioned we will describe in detail in chapter 6. Furthermore, we propose both Context Provisioning Policies and Context Provisioning Interaction Protocols that govern the communication and functionality of provisioning agents in a CMU. These capabilities increase application-development support by moving context provisioning adaptation concerns away from the application and towards the middleware level.
In what follows, let us make an analysis of how we address the operational and non-functional requirements of context management outlined in the state of the art chapter on context management solutions in Sections 3.1.1 and 3.1.2. The aspects for which we do not provide an analysis in this chapter will be detailed in the summary of the next chapter where we talk about the deployment options of the CONSERT Middleware.

Context Management Life Cycle

Context Acquisition is performed via the functionality of the `CtxSensor` and `CtxCoord` agents. A `CtxSensor` accesses physical or virtual sensors using `Context Assertion Adaptors`, thereby performing a middleware-based access. However, from the context-aware application point of view, access to context information is performed in a context-server based approach, since the application will use the `CtxUser` agent to query or subscribe for context that is stored in a CONSERT Engine knowledge based and managed by the `CtxCoord` agent. Therefore, the application level need not manage direct communication with several context providers.
Further, the `CtxSensor` actively push the data to the coordinator agent, but the parameters (update mode and update rate) of this process can be both initially specified and later adapted by the `CtxCoord`. Meanwhile, the communication with physical or virtual sensors can occur in both push and pull mode and is accommodated by the developer given implementation of the `Context Assertion Adaptors`.

Context Modeling and Provisioning Coordination is handled by the `CtxCoord` agent, which creates and manages the CONSERT Engine and coordinates the provisioning process based on control parameters and control rules operating on context usage statistics computed by the CONSERT Engine.

Context Dissemination is handled by the `CtxQueryHandler` agent which supports answering to both direct queries as well as long-lasting subscriptions. In this chapter we explored only one request option available to applications, namely local queries (i.e. those where the `CtxQueryHandler` only answers based on the contents of the knowledge base of the CONSERT Engine instance situated in the same CMU as itself). In the next chapter however, we explore additional and more complex query and subscription options which involve awareness of the distribution of several CMUs.
One other important point to note is that the `CtxUser` agent is able to express queries that exploit the full representation expressiveness of the CONSERT Ontology. As we will explain also in the implementation chapter, the application can formulate queries or subscriptions in SPARQL form and it can set conditions spanning both the desired *content* of *ContextAssertions* as well as their *ContextAnnotations*. In this way, the application can ensure that it receives context information of acceptable quality of context (e.g. certainty, adequate temporal validity).

Transverse Context Management Functionality

In this chapter we discuss the aspect of Provisioning Adaptation/Control, while the other two mentioned concerns (Context Producer Discovery and Mobility Management) are analyzed at the end of the next chapter. Our provisioning adaptation addresses functional changes of the behavior of context provisioning agents (i.e. `CtxSensor` and `CtxCoord`) and the CONSERT Engine. It is based on the existence of sensing and coordination policies which guide the

behavior of these agents. One strong suit of our approach, which will become more apparent in the implementation chapter, is the fact that these policies are expressed declaratively as opposed to being directly coded within the agent execution cycle. This makes their initial specification, change or extension much easier, which eases the development effort.
Currently, the adaptation capabilities offered by the CONSERT Middleware lie in terms of active/inactive *ContextAssertion* updates and *ContextDerivationRule* executions, as well as the custom scheduling of derivations. However, this already has strong implications on the number of performed sensing events and messages exchanged over a network as we will show in more detail in the evaluations of Section 8.2.3.
Further, we note that currently control parameters and rules can affect provisioning adaptation at the resolution of *ContextAssertion* types. For instance, the output of a `StartAssertionCommand` stipulates that all `CtxSensor` agents currently connected to the `CtxCoord` running the rule and which are able to provide the specified *ContextAssertion* will be requested to start providing updates for that assertion, regardless of the potential quality of those updates. Therefore, an immediate aspect of future work is the idea to augment the expressiveness of our provisioning policy vocabulary with the ability to express conditions and actions affecting *individual* context providers (`CtxSensors`) and `CtxDerivationRules`. The conditions for adaptation will be able to take into account quality of context related aspects (as can be already observed in works such as [Khedr and Karmouch, 2004; Corradi et al., 2010]). What's more, our agent-based architecture can be further exploited to allow the sensing and provisioning policies to set application-specific operation *goals* for each context provisioning agent. Remember that one of the agent attributes described in Section 5.1.1 refers to pro-activeness and goal oriented behavior. Thus, for instance, in an application where energy consumption of sensing equipment is of great importance, `CtxSensor` agents may have a goal to keep power spending within certain limits, which therefore impacts their supported update modes and rates. This, in turn, will impact their response to `TaskingCommands` issued by the `CtxCoord` which target, for example, an increase of update rate (i.e. they may refuse to comply because of the imposed energy savings requirements). This refusal can again impact the decisions of the `CtxCoord` which may elect to activate the updates for the same *ContextAssertion* type from a `CtxSensor` agent that is less constrained.
All these interactions can be collectively designated as establishing a *Context Level Agreement* (the context management equivalent of reaching a service-level agreement in service based application development), an idea which has been approached in related work [Khedr and Karmouch, 2004] and which constitutes an objective for future work in our case.

Non-Functional Aspects

Lastly, we revisit some of the non-functional requirements commonly set for context management system operation. We address the concern of *heterogeneity* of context producers by means of the `ContextAssertionAdaptors` which provide actual communication with physical sensors and translate sensed context into the constructs of the CONSERT Ontology which further ensures access of a uniform representation of acquired context information to all `CtxUser` agents.
Though we do currently do not offer support for traceability and control as defined in Section 3.1.2, we have an advantage over related work, given the monitorization of context usage carried out by the CONSERT Engine (which amounts to a reflection upon its own functionality). The mechanism can be augmented to provide explanations for activation/deactivation of *ContextAssertions* and *ContextDerivationRules*. Furthermore, given our rule-based inference implementation, mechanisms such as those in [Lim and Dey, 2010] can be used to provide *intelligibility* of the derived context information.
In this chapter we addressed ease of configuration from the point of view of establishing declarative policies that guide the adaptation of the context provisioning process within a CMU. Further aspects related to configuration of the deployment of the CONSERT Middleware will

be analyzed in chapter 6. These benefits will also become more obvious as we uncover the details of their implementation in chapter 7 and as we report on our experience of developing the evaluation simulation in chapter 8.
Aspects of scalability and robustness will be again discussed in the chapters on the deployment and implementation of the CONSERT Middleware, whereas privacy and security currently remain as issues for future work.

Chapter 6

Flexible Deployment of Context Provisioning

In the previous chapter we have seen that multi-agent oriented software engineering is a suitable paradigm for implementing the set of context provisioning related functionalities expected from a context management middleware solution. We saw that context provisioning agents act as units that encapsulate individual provisioning steps (e.g. acquisition, coordination, dissemination). We explained equally how the usual interactions between these agents can be guided and influenced by the development of *context provisioning policies*.

In this chapter we take another step towards our complete perspective over the context management problem by looking at the issue of deployment of context provisioning agents and, by extension, the deployment of an entire CONSERT Middleware instance. As in the previous chapter, we focus on showing that the CONSERT Middleware provides extensive development support by allowing for a declarative design-time agent deployment configuration and a runtime management of the deployed agents. We introduce configuration options that allow a developer to organize the implementation of a context-aware application according to logical partitioning of the context model into different usage domains, which relies on the notions of *ContextDimension* and *ContextDomain* introduced in Section 4.1.3.

Specifically, in Section 6.1 we describe our context domain-based view of provisioning agent deployment and show how it is organized in terms of the composition of a Context Management Unit (CMU - cf. Section 5.1.2) and the allocation of CMUs to the management of context from a domain.
In Section 6.2, we then inspect the configuration vocabulary used to specify middleware deployment and explain how the configurations are enforced at runtime.
Afterwards, in Section 6.4, we discuss the additional provisioning interactions that stem from the existence of a set of distributed context domains within the application space, before concluding the chapter with its summary and analysis in Section 6.5.

6.1 Deployment: A Domain-Based View

In many ambient intelligence applications there is a certain multidimensionality of the context model which reflects into *when* and *how* different context information is used by an application. To see what we mean by the above more clearly, consider an extension of our reference scenario, which we introduced in [Sorici et al., 2015]. In that instance we record the events of Alice who

wants to attend the CS Lecture in the AmI laboratory, happening prior to her arrival at the university.

Alice is in a tram on her way to class, which starts in 5 minutes. However, the smart application on her smartphone is informed that the tram is 7 minutes out from its next stop. The application knows to factor in an additional 3 minutes needed for Alice to walk from the tram stop to the university and therefore computes that she will be 5 minutes late to class. Since Alice is enrolled in the CS Lecture, the application will automatically send a notification of being late to the service managing teaching activities at the university. The CS professor is automatically subscribed to such notifications and decides to wait 5 minutes to begin the class, since he wants all students to be present for the current lecture.

Considering this extension and the interactions that follow afterwards we can take note of important features of context-aware application that impact the way context provisioning should be handled. As we can see from the application managing Alice's own schedule and her university related activities, the context information used by the application may be partitioned and structured along several logical *domains* such as places (the tram, the AmI lab.), activities (CS lecture, ad-hoc meeting) or organization (being enrolled as a student at the university). The scenario stresses also the need for flexible context provisioning deployment mechanisms. The domains *do not all have to be provisioned at the same time* (e.g. ad-hoc meeting in AmI lab, subscribing for updates from the tram only while in it), though provisioning needs may sometimes overlap (e.g. estimated arrival information from the tram, delay notifications sent by Alice to the lecture management service). Furthermore, these context usage sessions can be *dependent* (e.g. tram information influences course start time) or *independent* of one another (e.g. tram information and AmI lab meeting). Lastly, we observe that context reasoning needs to take place according to *varying degrees of complexity* (e.g. simple delay calculation in the tram vs. ad-hoc meeting detection in the AmI lab) and be performed both on fixed (e.g. the AmI lab context management service) as well as mobile computation nodes (e.g. Alice's smartphone).

The issues presented above motivate the discussion we hold in this section. In order to help a prospective developer structure his context-aware application according to the different context usage interaction sequences arising from the context model, we show how the agents composing a CMU are assigned to the provisioning of a specific *ContextDomain* formed along a preferred *dimension* of the context model.
Furthermore, we show that when the *ContextDomains* formed from a context model expose natural inclusion-like properties, this fact can be exploited to allow the developer to build domain hierarchies which impact the way in which inter-domain context provisioning can be performed.

6.1.1 Using ContextDimensions and ContextDomains

In Section 4.1.3 we introduced the formal notions of *ContextDimension* and *ContextDomain* and showed how they were extracted from the domain of discourse of the application context model. Specifically, we explained that a *ContextDimension* can be understood as a *privileged direction* (e.g. spatial location, user activity, organizational relation) along which the application will structure its context provisioning process. Further on, a *ContextDomain* establishes a *logical partition* of the global application context model along the chosen *ContextDimension*.

Identifying Context Dimensions In Figure 6.1 we show a domain based view of the reference scenario extension introduced earlier. Notice how we identify three distinct *ContextDimensions* which correspond to distinguished contextual interactions: `locatedIn(Person, PublicTransport)`, `engagedIn(Person, CourseActivity)` and `locatedIn(Person, UniversitySpace)`. Observe how these belong to two distinct context categories (i.e. spatial

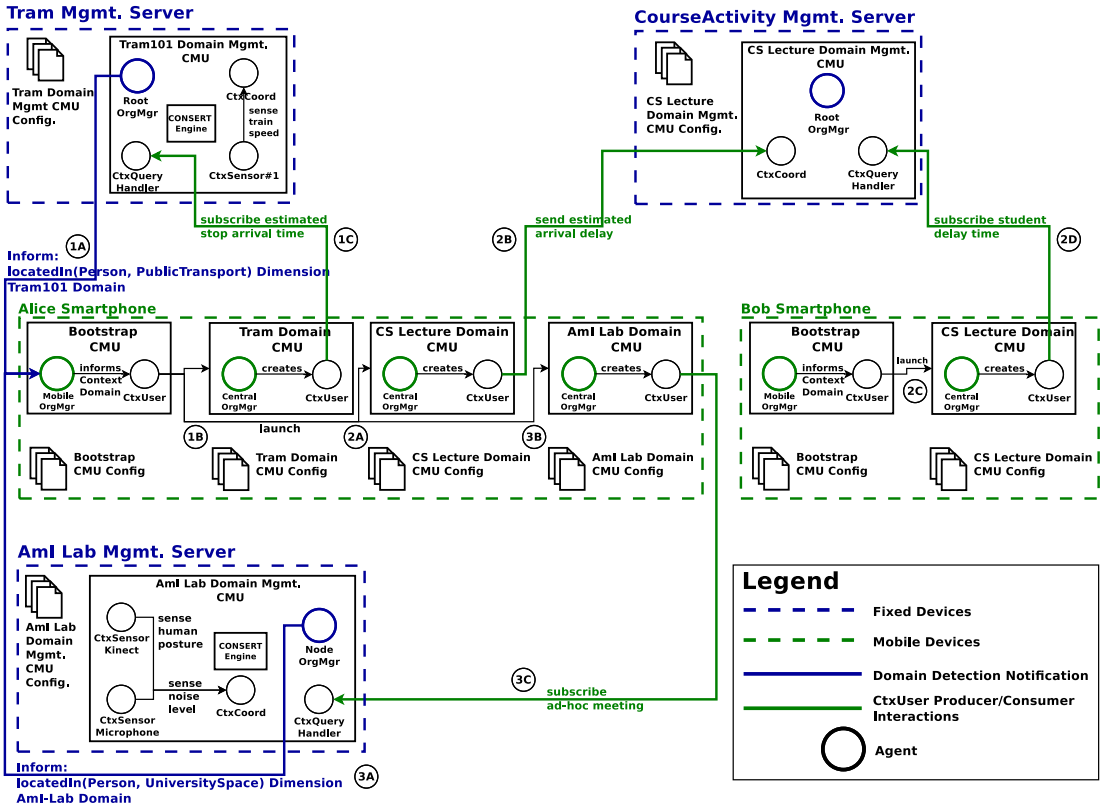

Figure 6.1: A domain-based view of the reference scenario extension discussed in this section. Notice how each CMU is assigned to a given *ContextDomain* and how its composition varies according to the attributions assigned to the machine on which it runs, e.g. usage of context (on mobile nodes) or management of context (domain management computational nodes). Numbered circles show the temporal order in which CMU management and context production/consumption interactions take place in the scenario.

and activity related) and how they correspond to the binary relation form explained in Section 4.1.3, having a subject that is a user-related *ContextEntity* and an object entity belonging to the spatial (`PublicTransport` and `UniversitySpace`) and activity (`CourseActivity`) *ContextEntity* .

Identifying Context Domains The distinguishable context related interactions can be further refined based on the specific *ContextDomains* formed along these dimensions. Thus, information about tram 101's speed and its estimated arrival time to the next stop is managed within the `Tram101` *ContextDomain* and consumed by Alice's smartphone only throughout the duration of her stay on board the tram. Information specific to the `CS_Lecture` course activity is exchanged with the lecture management service of the university and will be used, for instance, during the time period in which the user (i.e. Alice) has marked the CS lecture as active in the calendar. Lastly, the ad-hoc meeting related context interactions take place in the `AmI-Lab` *ContextDomain*, but only outside the normal course activity hours.

Relation between Context Domains and CMUs It is important to note that all throughout the above described situations, from Alice's perspective, it is the same application on her smartphone that handles all these contextual interactions. It is in this regard that support for structuring context-aware application development becomes important. Remember that in

the previous chapter we mentioned that a certain set of context provisioning agents forms a Context Management Unit (CMU). We also explained then, that the agent-based composition of that CMU depends on the *usage* of context information for which it is intended. In Figure 6.1 we can observe that several CMUs are allotted to the same *ContextDomain* (e.g. we have a `AmI-Lab` *management* CMU running on a server in the laboratory and a *consumption* (usage) CMU on Alice's smartphone), but have different compositions. On the AmI-Lab management machine, the CMU comprises the `CtxCoord` and `CtxQueryHandler` agents, a CONSERT Engine instance (created and managed by the `CtxCoord`) and the two `CtxSensor` agents responsible for detecting noise level and body posture specific *ContextAssertions*. Note that the latter two agents could have been themselves placed in a dedicated sensing machine, while still being assigned as a CMU used for provisioning of the same `AmI-Lab` *ContextDomain*. On the other hand, the CMU specific for the `AmI-Lab` *ContextDomain* running on Alice's smartphone consists only of a `CtxUser` instance, since her smartphone application is a *consumer* (i.e. user) of context information.

Considering the above example, we can now specify several deployment principals of the CONSERT Middleware:

(i) instantiation of the CONSERT Middleware in a context-aware application is thought in terms of *ContextDimensions* and *ContextDomains* arising from the application *context model*.

(ii) there is a *one-to-one mapping* between a CMU and the *ContextDomain* which it must service

(iii) the agent composition of the CMU running on a given computational node (e.g. server, laptop, smartphone) depends on the intended aspects of context provisioning (e.g. producer/acquisition, coordination, consumption/usage or combination thereof) that are set to run on that node

6.1.2 Using ContextDomain Hierarchies

We mentioned previously that context usage of an application can be structured along multiple *ContextDimensions*. The resulting *ContextDomains* can however be structured in two ways, depending on the relations that exist between them in the application context model.

A **flat** *ContextDomain* structure corresponds to a set of different values of the *object ContextEntity*, between which no other relation exists in the application context model. An example of this are the `AmI-Lab` and `room EF301` *ContextDomains* in our original reference scenario.

A *ContextDomain* **hierarchy** is a structuring of the *ContextDomains* of a *ContextDimension* based on the order induced by an *inclusion-like EntityDescription* characterizing the object *ContextEntities* of that dimension (cf. Definition 4.1.21 of Section 4.1.3).
Figure 6.2 shows an example of a hierarchical *ContextDomain* setup. We again consider a straightforward extension of our reference scenario to explain the above introduced ideas. In Figure 6.1 from the previous subsection we considered `UniversitySpace` as being a *ContextEntity* from which the values for the `locatedIn(Person, UniversitySpace)` *ContextDimension* (such as `AmI-Lab`) arise. Consider now the following subtypes of the `UniversitySpace` *ContextEntity*: `FacultyBuilding`, `LaboratoryRoom` and `RoomSection` (not shown in the figure). Instances of these entity types can be related with one another via the `includedIn` *EntityDescription*: Therefore, in the setup presented in Figure 6.2 we have

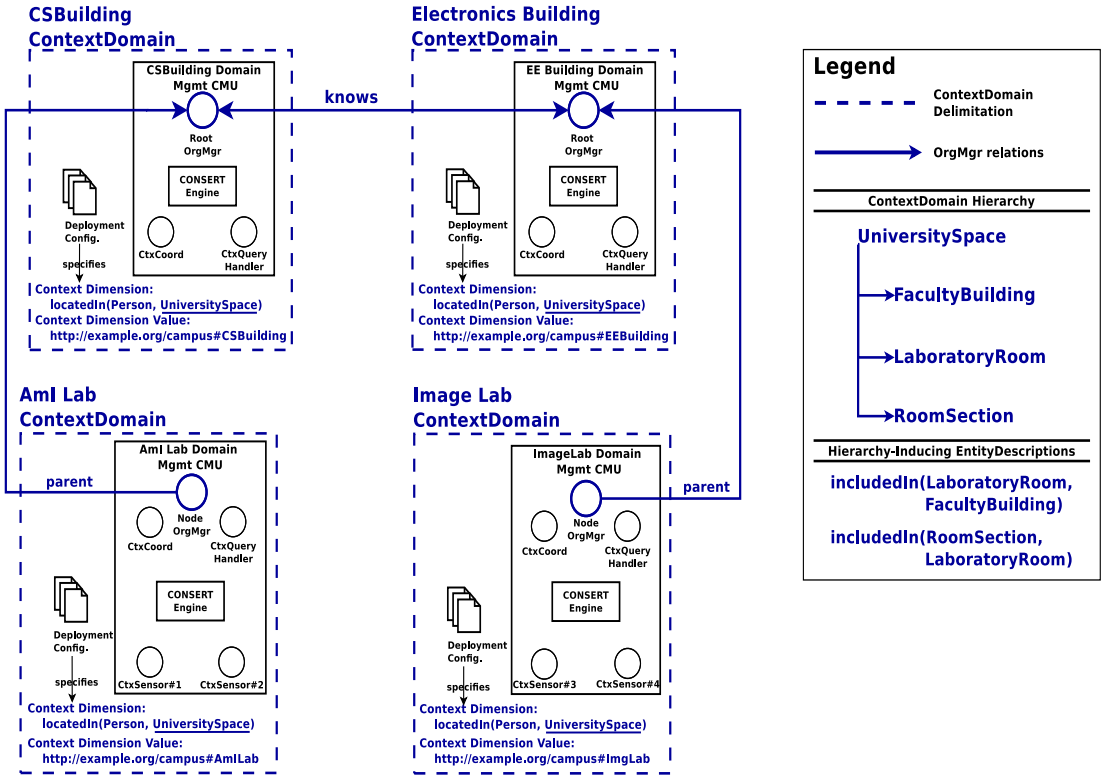

Figure 6.2: An example of a decentralized deployment constituted as a *ContextDomain* hierarchy of the spatial *ContextDimension* `locatedIn(Person, UniversitySpace)`.

the following:

```
includedIn(Desk_Alice, Ami_Lab)
includedIn(Ami_Lab, CS_Building)
includedIn(Desk_Dan, Img_Lab)
includedIn(Img_Lab, EE_Building)
```

Notice that the setup does not constitute a single tree-based hierarchy, but rather a forest, with the `CS_Building` and `EE_Building` domains lying on the same level (having a *knows* relation drawn between the `OrgMgr` agents managing the respective associated CMUs - more details in Section 6.3). However, each top-level *ContextDomain* is the root of a hierarchy based on the `includedIn` *EntityDescription*. In Section 6.4 we will show that the ability to consider such a hierarchical structure of application context management deployment can have substantial benefits in terms of context dissemination. The domain-based partitioning of context usage helps manage the issue of *locality*, that is, context information that is produced in a logical location should be consumed by application clients near that location. On the other hand, an application aware of the existence of several *ContextDomains* and the potential hierarchy they form can exploit query mechanisms which involve limited range request broadcasts (e.g. ask for information on Alice's availability status on every domain of type `LaboratoryRoom`).

6.1.3 CONSERT Middleware Deployment Schemes

In Section 3.3 we explained that one of the objectives of the implementation of our middleware is to provide it with the required flexibility such that it may be used for multiple scenario

scales and types. The deployment principles listed previously allow us to define two envisioned *deployment schemes*: *centralized* and *decentralized*.

A **centralized** scheme represents a setup where the application considers a single (default) *ContextDomain* and a single CMU that provides the context provisioning. The physical or virtual sensors (i.e. context producers) are managed by `CtxSensors` from this CMU and the application (i.e. context consumer) interacts with the `CtxUser` agent of the management unit. Such a scheme could be used in applications that target context-aware services running on a single device (e.g. smart document management as in [Pietschmann et al., 2008], computer-based guides with contextual navigation as in [Conan et al., 2007]) or even all-in-one smart home platforms as in [Sehic and Dustdar, 2010].

A **decentralized** deployment setup is configured in terms of one or more *ContextDimensions* and the *ContextDomains* they may form. The *ContextDomains* can be structured either in *flat* or a *hierarchical* layout, depending on the relations considered in the application context model.
Such a style targets applications of a larger scale, with distributed context models and multiple CMUs (organizable into hierarchies, if required) that comprise both fixed and mobile nodes. Sensors are managed by the `CtxSensor` agents of a single CMU (to preserve locality of sensed information), but consumers of this information can come from multiple CMUs (i.e. queries and subscriptions can be sent in between CMUs). Examples of this would range from the one in the scenario we presented to smart city applications [Da Rocha and Endler, 2012].

Having introduced the concepts and schemes that characterize deployment possibilities within the CONSERT Middleware, in what follows we present how this information is packaged as policies that specify CMU agent configurations and assignment of CMUs to *ContextDomains*.

6.2 Deployment Policies

As was the case in chapter 5, in order to support ease of application development, the CONSERT Middleware provides a vocabulary for declarative specification of **deployment policies**. These policies represent a collection of parameters which address various deployment settings such as platform setup, agent setup, CMU composition or context domain model.

As mentioned earlier, the deployment of the CONSERT Middleware is the process that binds together all aspects presented in previous chapters: modeling and reasoning about context information using the CONSERT Engine, creating and managing an adaptable context provisioning process. Deployment policies therefore specify the settings under which one or more instances of the agents and the service components in their environment have to execute.

In this section we will go over sets of parameters which concern the following aspects:

- *Platform Configuration*: setting up the runtime platform on which the agents of one or more CMUs run
- *ContextDomain Configuration*: specifying the deployment scheme for a *ContextDomain* and the context model for that particular domain.
- *Agent Configuration*: specifying the agent composition of a CMU and defining all CMU agent provisioning configurations (e.g. identifier of adaptor service implementations, assignment of provisioning policies)

We present here the intended purpose of each parameter and offer pseudo-configurations as examples, while in Section 7.4 of the middleware implementation chapter we will show how the effect of this parameters is handled at runtime.

6.2.1 Platform Configuration

The platform specification provides technical information for setting up a *container* for the physical machine that will host the agents running within the CMUs. For instance, in the reference scenario extension detailed in the beginning of this chapter, the application on Alice's smartphone is engaged with multiple *ContextDomains* (e.g. the tram, the AmI-Lab) at the same time. Therefore, as show in Figure 6.1, multiple CMUs must run on the same computational node (her smartphone) to help the application interact with these domains.

Parameter	Values	Role
platform name	string	Unique container identifier used to create application-wide unique agent names.
container host	string	String specifying the hostname part of the container URI
container port	integer number	Integer specifying the port part of the container URI
MTP host	string	Hostname for the URI of the messaging server that allows inter-container communication.
MTP port	integer number	Port number for the URI of the messaging server that allows inter-container communication.

Table 6.1: List of parameters available for CONSERT Middleware platform configurations.

Table 6.1 lists the parameters used to configure a CONSERT Middleware platform on a computational node. As we will see in Section 7.4.1 there is a single instance of these parameters for each physical machine on which one or more CMUs must be deployed. The developer configures a unique name for the runtime container in which CMU agents will run. As we will see in the agent-specific parameters below, CMU agents are given a container relative name. Therefore, the container name is used to establish an application-wide addressing means for each agent in a decentralized deployment. Further parameters refer to addressing configurations (hostname and port number) of the container itself and an HTTP-based communication service it exposes and will be further detailed in Section 7.4 of the chapter on CONSERT Middleware implementation.
Below we show a small pseudo-configuration excerpt that sets the platform configurations for the simulation of the AmI-Lab management server from our reference scenario (details in Section 8.2.4).

```
 platformName  :=  AmI-Lab
containerHost  :=  localhost
containerPort  :=  1099
      mtpHost  :=  localhost
      mtpPort  :=  7778
```

Since it is a configuration for a simulation, the container and message transport service hostnames are set to *localhost*, while the port numbers are typical for the agent development framework used in the CONSERT Middleware (i.e. JADE).

6.2.2 ContextDomain Configurations

The *ContextDomain* configurations specify the details that characterize a domain and are listed in Table 6.2.

Parameter	Values	Role
domain identifier	string	Identifies a *ContextDomain* from an application perspective.
deployment type	enumeration	Specify deployment scheme (centralized, decentralized) of which the *ContextDomain* is part.
domain dimension	URI string	(Optional) URI of the *ContextDimension* from which the domain arises.
domain range entity	URI string	(Optional) URI of the *ContextEntity* type playing the object role in the *ContextDimension*.
domain range value	URI string	(Optional) URI of the *ContextEntity* instance playing the object role in the *ContextDimension*, i.e. the domain value.
domain hierarchy property	URI string	(Optional) URI of the *EntityDescription* used to create the domain hierarchy.
domain hierarchy document	URI string	(Optional) URI of RDF document listing the *ContextDomains* that make up the hierarchy along a *ContextDimension*.
context model core / annotation / constraint / functions / rules document	URI string	URI of the RDF documents that contain the using the CONSERT Ontology based definition of the context model for this *ContextDomain*.

Table 6.2: List of parameters available for *ContextDomain* configurations.

A first thing that needs to be specified is an identifier for the *ContextDomain*. Note that, in theory, the domain can be uniquely identified within an application by the combination of *ContextDimension* and *ContextDomain* URIs that stem from the application context model. However, in more simple deployment scenarios (e.g. a centralized one on a single device), a developer can often require a single default domain. Defining the dimension and domain explicitly can become unnecessary in such cases. Therefore, to support identification from an implementation point of view, the designer can use the *domain identifier* parameter. Together with the agent local name and type (i.e. `CtxCoord`, `CtxQueryHandler`, `CtxSensor`, `CtxUser`), this parameter helps create the domain-relative agent name.
The *deployment type* parameter identifies the scheme of which the *ContextDomain* in question is part. It further serves to determine the actions taken by the `OrgMgr` agent, as we will see in Section 6.3.
As mentioned above, the URIs (as modeled using the CONSERT Ontology) that identify the *ContextDimension* and *ContextDomain* are marked as optional since they are not always necessary in a *centralized* deployment (e.g. consider the case of Alice's personal calendar management on her smartphone, decoupled from the interactions presented in the reference scenario extension at the beginning of this chapter).
When explicit *ContextDimension* and *ContextDomain* specifications are present and where *ContextDomains* can be constituted into a hierarchy along the given dimension, the relevant parameters (*domain hierarchy property* and *domain hierarchy document*) will give the means to identify the context model *EntityDescription* that allows for the construction of the hierarchy and the document that maintains the overview of this result.
The last set of parameters identify the URIs of the files that compose the application context model subset that is specific to the given *ContextDomain* (or the default one in case of a simple *centralized* deployment). To help model development, a designer can use different files to specify the required *ContextAssertions* (core file), the possible *ContextAnnotations* and

ContextConstraints (annotation and constraint files), the custom built functions that operate on the assertions and annotations (functions file) and the set of *ContextDerivationRules* which must be executed in the CONSERT Engine instance active within the *ContextDomain* (rules file).

To exemplify the discussed parameters, we list some pseudo-configuration from the reference scenario that shows the configurations of the `AmI-Lab` and `CS_Lecture` domains.

`domainIdentifier`	:=	*AmI-Lab-Smart-Classroom*
`deploymentType`	:=	*decentralized*
`domainDimension`	:=	`locatedIn(Person, UniversitySpace)`
`domainRangeEntity`	:=	`LaboratoryRoom`
`domainRangeValue`	:=	*AmI-Lab*
`domainHierarchyProperty`	:=	*includedIn(UniversitySpace, UniversitySpace)*
`contextModelCore`	:=	*http://pervasive.semanticweb.org/ont/2014/07/amilab/core*
`contextModelRules`	:=	*http://pervasive.semanticweb.org/ont/2014/07/amilab/rules*

Table 6.3: Configurations for the AmI-Lab *ContextDomain*.

`domainIdentifier`	:=	*CS_Lecture-Activity*
`deploymentType`	:=	*centralized*
`domainDimension`	:=	`engagedIn(Person, CourseActivity)`
`domainRangeEntity`	:=	`CourseActivity`
`domainRangeValue`	:=	*CS_Lecture*
`contextModelCore`	:=	*http://pervasive.semanticweb.org/ont/2014/07/courseactivity/core*

Table 6.4: Configurations for the *CS_Lecture ContextDomain*.

Notice that in the case of the *CS_Lecture*, the deployment type is set as *centralized*. In effect, the management of all course activity related context is performed on a dedicated server from the university, constituting a centralized deployment. However, even in this scheme, a *ContextDomain* (which is not part of a hierarchy though) can be specified for each type of lecture.

6.2.3 Agent Configurations

Once the configuration for the platform and *ContextDomain* are set, the developer is tasked with specifying what the context provisioning functionality assigned to each *ContextDomain* is. This is indicated in terms of the agent composition of the CMU allotted to that domain.

The above listed parameters configure the functionality of an agent. That is, for each instance of a middleware agent type that must be part of a CMU, these parameters have to be particularized. The name parameter helps identify the agent at CMU level and together with the rest of the identifiers discussed previously, the full and unique name of the agent is constructed.
In Section 6.1.1 we explained that certain CMUs, suchs as the AmI-Lab context usage CMU on Alice's smartphone, can run on dedicated machines. In that case, the agents on those computational nodes must know how to connect to their corresponding partners, depending on the interaction chain of which they are part (`CtxCoord` for the sensing chain, `CtxQueryHandler` for the request chain). This is done by specifying the address of the `OrgMgr` agent managing that specific CMU through the *assigned `OrgMgr` address* parameter.
The *parent* and *known* `OrgMgr` address configurations are specific to `OrgMgr` agents and determine the connections that need to be maintained by these agents in a given decentralized deployment. Examples of these relations can be observed in Figure 6.2 and the way in which they are created and exploited will be discussed further in Sections 6.3 and 6.4.

Parameter	Values	Role
agent name	string	The agent name local to its CMU.
assigned `OrgMgr` address	structured data	Agent name and container configurations that make up the address of the assigned `OrgMgr`.
***parent* `OrgMgr` address**	structured data	Address information for the parent of an `OrgMgr` agent in a domain hierarchy.
***known* `OrgMgr` address**	structured data	Address information for other manager agents that an `OrgMgr` agent *knows* about.
coordination policy document	URI string	URI of RDF document containing provisioning control parameters and rules as detailed in Section 5.2.2.
sensing policy document	integer number	URI of RDF document containing provisioning sensing parameters as detailed in Section 5.2.1.

Table 6.5: List of parameters available for CONSERT Middleware CMU agent configurations.

Lastly, as detailed in chapter 5, the CONSERT Middleware agents guide their context provisioning behavior according to declarative policies. The URIs for the documents containing the specific sensing or coordination provisioning parameters and rules are given using the *sensing* and *coordination policy document* parameters.

Continuing the pseudo-configurations from the extended reference scenario, we present example agent configuration in what follows, which show part of the composition of the CMU assigned to the management of the `AmI-Lab` *ContextDomain* on the server running in that laboratory.

`CtxCoord`	`agent name`	:=	*CtxCoord_AmI-Lab*
	`assigned OrgMgr`	:=	*AmI-Lab OrgMgr address*
	`coordination policy`	:=	`AmI-Lab coordination policy file`
`CtxSensor`	`agent name`	:=	*CtxSensor_NoiseLevel*
	`assigned OrgMgr`	:=	*AmI-Lab OrgMgr address*
	`sensing policy`	:=	*hasNoiseLevel ContextAssertion* sensing policy file

6.3 Managing Deployment: the OrgMgr agent

In the previous section we detailed the means by which the deployment setup of the CONSERT Middleware can be declaratively specified. We continue now with explaining how this specifications are managed at runtime. As mentioned in Section 5.1.2 where we introduced the agents that compose the functionality of the CONSERT Middleware, the `OrgMgr` agent provides the link between the application level and the CMU it is assigned to manage in terms of configuration and life cycle management of the provisioning agents that compose the CMU. In this section we present aspects of the `OrgMgr` agent functionality and sequences of steps that lead from the deployment specification to deployment runtime.

6.3.1 Launching Platform and CMUs

We explained earlier that a CMU lies in a one-to-one relation with a *ContextDomain*. When the application-level decides that interaction with a given *ContextDomain* is needed, it can request the launch of the CMU responsible for that domain. If this is the first CMU to be launched

on the platform, the configurations of the latter are inspected and the required container is deployed (further technical aspects are discussed in Section 7.4.2). The agents of the CMU are then created on this container.
Then, the `OrgMgr` is the first agent to be instantiated in the CMU and, as detailed in the chapter on context provisioning, it maintains the overview of *ContextDimension* and *ContextDomain* structure given to the context-aware application of which it is part. It gains this overview using the context domain structure and model configurations contained in the deployment policy (cf. Section 6.2.2).
From the agent configuration part of the deployment policy, the `OrgMgr` agent will read its own configuration. These settings and the value of the *deployment type* parameter contained in the *ContextDomain* configuration determine a specific role that the `OrgMgr` agent will play within the application. The specifics of each role are presented next.

6.3.2 OrgMgr Roles

In previous sections we discussed about the fact that the CONSERT Middleware allows for different deployment schemes (centralized and decentralized) and that more than one CMU can be assigned to the same *ContextDomain* depending on the *dedicated* provisioning aspect enacted by the agents in that CMU (context acquisition, coordination or usage). Depending on the CMU it has to manage and the *ContextDomain* to which it is assigned, an `OrgMgr` agent can play the following set of *roles*: *root*, *node*, *central* or *mobile*. These roles dictate a set of specific interactions that the `OrgMgr` will need to handle.

Root `OrgMgr` . When marked as *root* the `OrgMgr` supervises a CMU that manages and coordinates the provisioning of context within a *ContextDomain*. If the domains belonging to a *ContextDimension* do not form a hierarchy, than all `OrgMgr` agents supervising the CMUs deployed to provide coordination of those *ContextDomains* play a root role (e.g. refer to the case of the `CS_Lecture` and `Tram` domains in Figure 6.1).

Node `OrgMgr` If the *ContextDomains* can form a hierarchy, then `OrgMgr` agents overseeing CMUs for mid-level domain context coordination play a *node* role. This is the case for the `OrgMgr` agent from the `AmI-Lab` *ContextDomain*. As we explain further down, the role of the `OrgMgr` also affects the interactions between this type of agents.

Central `OrgMgr` A *central* role is assigned to an `OrgMgr` in a *centralized* deployment scheme, as is the case of the `Tram`, `CS_Lecture` or `AmI-Lab` CMUs running on Alice's smartphone. As explained previously, these CMUs are only launched when the context interactions they imply (i.e. consumption of production of context information from/for the *ContextDomains* to which they are assigned). Therefore, the role of the central `OrgMgr` that starts the corresponding `CtxUser` agents is just that of managing the life-cycle of these agents as dictated by application needs.

Mobile `OrgMgr` A *mobile* role is assigned to an `OrgMgr` that supervises the CMU agents running on a mobile computing node which is subject to *ContextDomain* changes. In our example in Figure 6.1 this is the case for the `Bootstrap` CMU deployed on Alice's smartphone. The purpose of the `OrgMgr` in this CMU is to be informed by other *node* or *root* `OrgMgr` agents (e.g. the one from the tram or the one in the AmI-Lab) of the fact that the mobile device on which it runs has entered/left the given *ContextDomains*. The CONSERT Middleware supports

two mechanisms by which domain detection can be performed and they will be discussed in Section 6.4.

We mentioned earlier that the role of an `OrgMgr` determines specific interactions that take place either during the initialization phase of these agents or later during runtime. The former interactions refer to the connections that are established between `OrgMgr` agents and are discussed shortly hereafter. The runtime interactions of the `OrgMgr` agents refer to the support they offer to context provisioning agents (notably `CtxQueryHandler` and `CtxCoord` agents) with the routing of inter-domain requests which have been briefly mentioned in chapter 5. The specific request types and routing protocols are discussed in Section 6.4.

6.3.3 Initialization and Provisioning Agent Setup

After reading its own deployment specifications and determining its role, the `OrgMgr` agent performs two sets of initialization interactions. In the first one it will connect to other `OrgMgr` agents in case of a decentralized deployment scheme. The other set refers to creation and launch of the provisioning agents which are configured in the deployment policy for the CMU managed by the `OrgMgr`.

OrgMgr Connection Interactions

The connections that need to be maintained between `OrgMgr` agents follow from the *parent* and *knows* parameters discussed in Section 6.2.3. Essentially, a *root* `OrgMgr` will connect to all other root `OrgMgr` agents defined for *ContextDomains* of the same *ContextDimension* (i.e. create a fully-connected network). *Node* `OrgMgrs` are aware they are part of a *hierarchical ContextDomain* deployment and will thus connect to a parent and possibly register several child `OrgMgr` agents. A *central* `OrgMgr` knows it is employed only for use on the local device, while in the *mobile* case the agent realizes that its "parent" will be determined dynamically at runtime.
For a decentralized deployment (i.e. having *root* and/or *node* `OrgMgrs`), in the messages exchanged within the registration request made by a child `OrgMgr` to its parent, both child and parent `OrgMgr` agents exchange the configured addresses of the `CtxCoord` and `CtxQueryHandler` agents that they manage in their respective CMUs. As we will see in Section 6.4, this information will help execute the routing algorithms for domain-based queries and broadcasts.

Provisioning Agent Initialization Interactions

After the `OrgMgr` initialization phase is over, the manager agent reads the provisioning agent configurations specified in the deployment policy corresponding to the CMU it supervises and creates them.
Remember from Section 5.4 that the communication between the provisioning agents is determined by their membership in a given interaction chain (sensing or request). The `OrgMgr` therefore starts the agents in a specified order to ensure that each agent can find its required conversation partner.
The `OrgMgr` first starts the `CtxCoord`, if coordination is required for the CMU in the charge of the manager agent (i.e. if the intended context provisioning aspects of the current CMU include that of coordination).
After initializing the coordinator agent, the `OrgMgr` starts the `CtxQueryHandler`. The coordinator and query handler agents will usually be deployed on the same physical machine. These first two agents constitute the provisioning coordination and control units. The remaining agents will each connect to one or both of these two agents to create the two mentioned interaction chains.

The OrgMgr next starts the CtxSensor agent(s) defined in his CMU and afterwards it instantiates the CtxUser agent, which the application uses to interact with the CONSERT Middleware.

In the case of a *central* OrgMgr, after all provisioning agents have been created and started, some additional interactions may take place. We mentioned that the composition of CMUs is flexible. For example, in the reference scenario, it may be the case that the CtxSensor agents that manage the noise level and posture detection sensors in the AmI-Lab are all deployed on a dedicated physical machine. Their initialization will be managed by an central OrgMgr agent local to the CMU of that machine.
Besides the central OrgMgr that creates them, these CtxSensor agents need a way to access the OrgMgr agents of the fixed coordination CMUs for their corresponding *ContextDomain*, in order to determine the interaction partners (CtxCoord or CtxQueryHandler) which are part of the desired interaction chain. In the reference scenario extension shown in Figure 6.1 we see that this is the case for the CtxUser agents from the Tram, CS_Lecture and AmI-Lab *ContextDomains* on Alice's smartphone.
Therefore, in order for the CtxSensors or CtxUsers to connect to the corresponding *remote* CtxCoords or CtxQueryHandlers, the former agents will register with the OrgMgr managing the remote CMU in order to obtain the addresses of their respective interaction counterparts. The way they know how to register with this OrgMgr is via the *assigned OrgMgr* parameter defined in Section 6.2.3 (cf. also Table 6.5).

This completes the CMU deployment procedure. In the next section we discuss how the different deployment schemes augment the capabilities of the context provisioning agents contained within a CMU.

6.4 Distributed Deployment Usage

Previously we have mentioned that the decentralized deployment scheme augments the context provisioning capabilities of an application that uses the CONSERT Middleware. Specifically, in chapter 5 we detailed the context provisioning messages exchanged as part of the interaction chains of a CMU and, by extension, a *ContextDomain*. However, in the lists of agent interactions described in Sections 5.4.1 and 5.4.2 it could be already noted that certain messages targeted *inter-domain* information exchanges.
Specifically, in this section we discuss three types of interactions that happen between agents deployed as part of different *ContextDomains*. We first talk about management *domain-queries* which bring increased dissemination capabilities. We then present the management of *domain-broadcasts* which complement the functionality of CtxUser agents with the ability of making broadcasts of context information along the domains of a given *ContextDimension*, in case of a *ContextDomain* hierarchy. Finally, we discuss aspects related to mobility management and how notifications about accessible *ContextDomains* is handled.

6.4.1 Domain Query Management

In our reference scenario we encounter a *domain-query* under the form of the request that Dan sends from his office in the EF301 room to inquire about Alice's availability status. We consider two types of domain queries.

An **exact domain** query or subscription specifies the exact *ContextDomain* from which an answer needs to be retrieved.
In the example of our reference scenario, if Dan knows the exact location of Alice he can use his application to issue a query for Alice's availability status directly to the CMU running on the management server in the AmI-Lab.

A **domain range** query or subscription is applicable given the existence of a *ContextDomain* hierarchy and specified an upper and a lower *ContextDomain type* limit from which answers to the query can be retrieved.
To continue the above example, if Dan does not know the exact position of Alice, but is certain that she must be somewhere in the building, he can issue a query for which he does not specify a *ContextDomain value*, but rather two *ContextDomain entity type limits*. Dan can, for instance, query the CMUs for all `FacultyBuilding` and `LaboratoryRoom` domain types (i.e. get answers regardless if Alice is in a room or on some hall of the building). In this case the upper limit is represented by the `FacultyBuilding` *ContextEntity* type and the lower one by the `LaboratoryRoom` *ContextEntity* type. However, the two limits can often coincide (e.g. Dan only wants to retrieve answers from laboratory rooms).

Consequently, we see that the CONSERT Middleware offers a flexible distributed query mechanism, with either direct match requirements or semantics-based range limitations. In what follows we describe the messages exchanged as part of the query routing mechanisms that implements the two request options.

Domain Query Protocol

A domain query is initiated by a `CtxUser` agent and sent to the `CtxQueryHandler` with which it is registered. If the `CtxQueryHandler` determines the CMU he belongs to does not manage the targeted *ContextDomain* of the query, it will start the routing protocol, collaborating with the `OrgMgr` agent. For both types of domain queries the protocol works the same, but the analysis done by the `OrgMgr` differs.

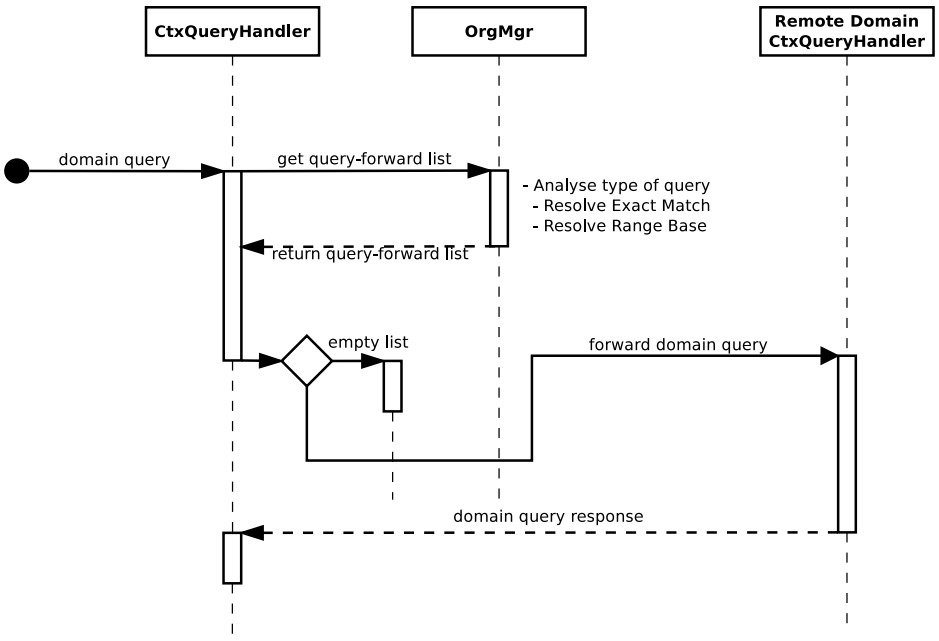

Figure 6.3: Sequence diagram for the domain-query routing interactions between `CtxQueryHandler` and `OrgMgr` agents.

The protocol is depicted in the sequence diagram from Figure 6.3 and works as follows:

- The `CtxQueryHandler` asks the `OrgMgr` of his CMU to provide a list of addresses for the `CtxQuery Handler` agent(s) to which he must forward the query (i.e. either from parent or child CMUs).

- The `OrgMgr`, knowing from which *ContextDomain* the query came and using the *domain hierarchy document* as well as his parent, child or known root manager connections, determines the list of `CtxQueryHandler` agents to which the query must be forwarded (Domain Query Forwarder Selection).
- If the query must not be forwarded anywhere (e.g. for a range query received from a parent domain where the child domains are out of range), then the list will be empty. If the list is not empty, the `CtxQueryHandler` will forward the query (i.e. as act a place holder `CtxUser`) to the agent addresses contained in the list. The answers to the query will thus take the exact reverse route back to the `CtxUser` that issued the request.

Domain Query Forwarder Selection

In what follows we detail the pseudo-code for the decision methods the `OrgMgr` applies to determine the list of forwarder `CtxQueryHandler` agents.

Algorithm 1 Resolve Query Forwarder Base - Exact Domain

```
1:  procedure RESOLVE-EXACT-BASE(queryMsg, domain)
2:      forward_list ← []
3:      if domainHierarchyExists() then
4:          if my_role == NodeMgr then
5:              forward_list.append(inspectHierarchy(queryMsg, domain))
6:          else[my role is RootMgr]
7:              forward_list.append(inspectHierarchy(queryMsg, domain))
8:              if forward_list == ∅ ∧ queryMsg.sender ∉ known_root_queryhandlers then
9:                  forward_list.append(known_root_queryhandlers)
10:     else[we are a Root OrgMgr with no hierarchy]
11:         if query_msg.sender ∉ known_root_queryhandlers then
12:             forward_list.append(known_root_queryhandlers)
        return forward_list
13: procedure INSPECTHIERARCHY(queryMsg, domain)
14:     list ← []
15:     if subsumes(my_domain, domain) then
16:         for all org_mgr ∈ child_org_mgrs do
17:             if subsumes(org_mgr, domain) then
18:                 list.append(org_mgr.query_handler)
19:                 break
20:     else
21:         if subsumed(my_domain, domain) ∨ my_type == NodeMgr then
22:             list.append(parent_mgr.query_handler)
        return list
```

Algorithm 1 presents the decision logic for an *exact domain* query. The procedure basically checks to see if there is any domain hierarchy model and if so, tries to determine if the searched domain is within the current hierarchy (i.e. anywhere in the tree up to the current root CMU). If either there is no domain hierarchy, or the domain is not in the current hierarchy (i.e. it is in the subtree of another root CMU) than the *root* `OrgMgr` will forward it to all other known peers, only if it did not already receive it from such a peer (this is to avoid redundant messages).

Algorithm 2 shows the decision logic of the `OrgMgr` in case of a *domain range query*.
Lines 4 and 5 check if the current domain (i.e. that of the `OrgMgr` making the decision) matches the type limits set by the query. If they do, the `CtxQueryHandler` of the local CMU is also included in the list. In this case, the local `CtxQueryHandler` will detect that its address is

Algorithm 2 Resolve Query Forwarder Base - Domain Range

```
1:  procedure RESOLVE-RANGE-BASE(queryMsg, upperLimitType, lowerLimitType)
2:      forward_list ← []
3:      if domainHierarchyExists() then
4:          if withinLimit(my_domain.type, upperLimitType, lowerLimitType) then
5:              forward_list.append(my_query_handler)
6:          if my_role == NodeMgr then
7:              forward_list.append(parent_mgr.query_handler)
8:          else[my type is RootMgr]
9:              if queryMsg.sender ∉ known_root_queryhandlers then
10:                 forward_list.append(known_root_queryhandlers)
11:         for all mgr ∈ child_org_mgrs do
12:             if withinLimit(mgr.domain.type, upperLimitType, lowerLimitType) then
13:                 forward_list.append(mgr.query_handler)
14:     else[we are a Root OrgMgr with no hierarchy]
15:         if query_msg.sender ∉ known_root_queryhandlers then
16:             forward_list.append(known_root_queryhandlers)
        return forward_list
```

also in the forwarder list and register the query locally.
Lines 6 through 10 ensure that all requests are forwarded higher up in the hierarchy and in between root managers such that they may reach the domain limits in all subtrees of root domains.
Lines 11 through 13 make sure that queries are forwarded back down the hierarchy, but no further than the lower *ContextEntity* type limit set by the initiator.

In the next section we perform a theoretical analysis of the complexity of the routing scheme and the expected number of exchanged messages, given some characteristics of the decentralized deployment setup.

6.4.2 Domain Query Complexity Analysis

We now perform an analysis of the complexity of the routing decision and the number of expected number of messages exchanged as part of the routing process.

In the case of *exact domain queries*, the main complexity degree comes from the *inspectHierarchy()* function which has to check whether the requested target *ContextDomain* lies in the current tree hierarchy or not. Since the `OrgMgr` agents holds a *domain hierarchy document* describing the existing *ContextDomains* and their inclusion relations, the function actually performs just two operations to determine whether the requested domain is in on the current *hierarchy branch*. The requested domain is either subsumed or it subsumes the current domain. Checking whether a node lies on a tree branch is an $O(h)$ operation, where h is the height of the tree. In this case h equals the number of different *ContextDomain* types defined for the current *ContextDimension*. In the example given at the beginning of this report h would equal 3 (`RoomSection`, `LaboratoryRoom` and `FacultyBuilding`).
If the requested domain is not on the current branch, the *inspectHierarchy()* function of Algorithm 1 will just forward it to the `CtxQueryHandler` of the parent CMU (see line 21).

In the case of the *domain range queries* the main complexity is given by the type limit range checks. Given that the `OrgMgr` agents have the *domain hierarchy document* and that this time they have to check for *ContextDomain* types instead of values, the worst case complexity is of the order of $O(dt)$, where dt is the number of *ContextDomain* types.

Considering now the messages exchanged between `CtxQueryHandler` and `OrgMgr` agents during a routing process, their expected number varies again depending on the type of domain based query. To determine a quantitative worst case value, we introduce the following assumptions and notations.
Let h be the maximum height of a *ContextDomain* tree hierarchy and r be the number of *root ContextDomains*. Further, let β be the maximum branching factor of the tree hierarchy.
To compute the number of exchanged messages for both request types we have to consider two terms: the messages exchanged in each CMU (i.e. between `CtxQueryHandler` and `OrgMgr`) and the ones exchanged in between CMUs (i.e. forwarding between `CtxQueryHandlers`).

In case of the *exact domain* request this amounts to:

$$\underbrace{2(h-1)+2r+2(h-1)}+\underbrace{2(h-1)+r-1} = 6h+3r-7$$

In the first term, we count the total number of nodes involved in the routing process. Since we compute for the worst case scenario (e.g. two *ContextDomains* lying at opposite domain hierarchy branches), there will $h-1$ nodes going "up" to the root, $h-1$ going back "down" in the other branch, plus the r *root* nodes which constitute a fully connected network. We multiply the obtained number by two to account for the request - reply messaging between the `CtxQueryHandler` and `OrgMgr`.
The second term counts the messages exchanged between `CtxQueryHandler` agents and, based on a similar logic, adds the *edges* going "up" on branch, and "down" the other, plus the $r-1$ messages between the fully connected root nodes.

For the *domain range* queries we apply a similar computation which takes the following form:

$$2S+(S-1)+rest, \text{where}$$
$$S = r \times (\sum_{i=1}^{h_{lower\_lim}} \beta^i) + r = r \times (\beta \frac{\beta^{lower\_lim}-1}{\beta-1} + 1)$$

In the above formula *rest* represents the messages that are caused by routing in the case where the original request start "below" the lower type limit of the query (i.e. from a *ContextDomain* lower in the hierarchy).
The formula basically shows that we are dealing with a bounded broadcast of messages, since all the nodes and edges of the *ContextDomain* tree hierarchies above the lower limit are involved in the routing.

The above analysis presented the maximum number of messages exchanged in the routing of queries. As mentioned earlier, the answers to these queries are returned on the routes created during the find phase. Therefore, in the case of *exact domain* queries, this will incur an additional $2(h-1)+1$ messages sent between `CtxQueryHandler` agents.
For the *domain range* query type, the number of answer messages depends on how many nodes from within the type limits set by the query can actually send a reply (i.e. the query matches information stored in the CONSERT Engine of their CMUs). In the worst case, the return path will resemble that of the *exact domain* query type.

6.4.3 Domain Broadcast Management

In the previous section we talked about *exact-domain* and *domain-range* query capabilities which complete the information retrieval functionality of the `CtxUser` agent.We now focus on the augmentation of the producer capabilities of such agents given the same *ContextDomain*-based structure awareness. Specifically, in Section 5.4.2, we mentioned that the `CtxUser` agent can provide *static EntityDescription* information or *profiled ContextAssertions* on instruction from its `Client Application Adaptor` service.

As was the initial case for the request chain, the insertion of *EntityDescriptions* or *profiled ContextAssertions* is by default limited to the current CMU (or more precisely, to the `CtxCoord` agent whose address was obtained from the *assigned* `OrgMgr` agent).

A **domain range broadcast** is applicable in case of a decentralized deployment where a *ContextDomain* hierarchy exists. As in the query case, it specifies upper and lower *ContextDomain type* limits between which context information given by the `CtxUser` can be forwarded.

However, the spread of information does not occur in the same way as for a query request. Given the *ContextDomain* type from which the broadcast starts, the information will be recursively forwarded to all child *ContextDomains* which respect the lower type limit. Instead, the forwarding to *ContextDomains* higher up in the hierarchy strictly follows the *parent* relation and stops either when it reaches either the upper type limit, or a *root ContextDomain*.
This decision attempts to maintain the locality principle, which, in our terms, translates to the fact that a piece of context information should only be directly accessible in those *ContextDomains* which have an explicit relation with one another (parent or child, recursively). Access to such information from other *ContextDomains* can be done via the domain-based query mechanisms described in the previous section.
To clarify this with an example, if Alice wishes to make her availability status known at faculty building level (whilst being in the ad-hoc meeting in the AmI laboratory) she will request a broadcast of this context with an upper type limit of `FacultyBuilding` and a lower type limit of `LaboratoryRoom`. Thus, her busy or free status will be sent to the `CtxCoord` agent running in the CMUs responsible for the coordination of context provisioning in the `AmI-Lab` and `CS_Building` *ContextDomains*. However, it is forwarded neither to the `EF301` office room, nor does it cross the *known root* relationship (as shown in Figure 6.2) to be sent to the `EE_Building`.

Algorithm 3 shows the decision logic for determining the forwarding list, once a **domain range broadcast request** is received. Notice that instead of the `CtxQueryHandler`, the `CtxCoord` agent is involved in the broadcast routing and it is charged with asking its assigned `OrgMgr` to determine the list of `CtxCoord` agents to which the request must be sent further.

Algorithm 3 Resolve Broadcast Forwarder Base - Domain Range

```
1:  procedure RESOLVE-RANGE-BASE(broadcastMsg, upperLimitType, lowerLimitType)
2:      forward_list ← []
3:      if hierarchyModelExists() then
4:          if my_type == NodeMgr then
5:              if broadcastMsg.sender ≠ parent_mgr.coordinator then
6:                  if parent_mgr.domain.type ⩽ upperLimitType then
7:                      forward_list.append(parent_mgr)
8:          if broadcastMsg.sender == parent_mgr.coordinator || my_coordinator then
9:              for all org_mgr ∈ child_org_mgrs do
10:                 if org_mgr.domain.typ ⩾ lowerLimitType then
11:                     forward_list.append(org_mgr.coordinator)
        return forward_list
```

Notice that no forwarding takes place unless the `OrgMgr` is aware of the existence of a hierarchy model (line 3 in the algorithm). *Node* `OrgMgr` will forward the request to their parent only if they did not receive it from them and if the broadcast upper limit permits it (lines 4 - 7). Lines 8 - 11 handle the other case, where the request is forwarded to child domains only if it did not come from a child `CtxCoord` and the broadcast lower limit permits it.

6.4.4 Context Prosuming Exemplification

The term *prosumer* was first coined by Toffler [Toffler et al., 1981] and used in an economic context to mean that the role of producers and consumers would begin to blur and merge. In the context of our work, we appropriate this attribute to the `CtxUser` agent which can act both as a consumer (i.e. member of the request interaction chain) as well as a producer of context information (i.e. member of the sensing interaction chain).

We have seen previously how extended query and broadcast abilities stem from a *ContextDomain*-based structuring of an application. We now want to discuss how the application-level can combine these behaviors across `CtxUser` agents from multiple CMUs and how this allows this type of agents to potentially act as a *relay* of context information.

To draw attention to aspects mentioned above, let us take a closer look at some of Alice's interactions while being engaged in the ad-hoc meeting taking place in the AmI laboratory. On the one hand, Alice's smartphone application is subscribed to the AmI-Lab management server for detecting the number of people at Alice's current `RoomSection`-level (i.e. which desk) location. If there are more than two people, the application subscribes for possible ad-hoc meeting notifications. If the AmI-Lab management server informs Alice's smartphone that she is in such a meeting, the application running on her mobile device knows to deduce that she is busy. The application agrees furthermore to disseminate this availability status at the level of the entire laboratory room.
At the level of structuring these interactions according to the provisioning connections and CMU deployment options discussed in this chapter and the previous one, there are two options, which are show in Figure 6.4 and are analyzed next.
We start by referring to a previous figure (Figure 6.1), where we explained that the role of the bootstrap CMU is to detect when Alice's smartphone application context falls within the incidence of a certain *ContextDomain*. The application-level is notified of when a *ContextDomain* becomes accessible or not. It can therefore decide to launch a CMU assigned for contextual interactions within this *ContextDomain*. While talking about the `OrgMgr` agent functionality we also mentioned that mobile `OrgMgrs` (which is the case of the `OrgMgr` of the bootstrap CMU) can notify the application level of the *remote* `OrgMgr` agent which needs to be *assigned* to the CMU agents that will be deployed to interact with the *ContextDomain* which has just been accessed.

This is the setting where the events and actions depicted in Figure 6.4 start. We consider that Alice and her colleagues are about to start their ad-hoc meeting. The `OrgMgr` in the bootstrap CMU has detected that Alice is in the AmI laboratory and has informed the application level of this fact and has provided the address of the *node* `OrgMgr` agent running in the CMU on the AmI-Lab management server. This latter agent becomes the *assigned* *`OrgMgr`* (shown as a blue arrow in Figure 6.4) for the provisioning agents in the CMU deployed on Alice's smartphone to interact with the `AmI-Lab` *ContextDomain* (shown as *AmI-Lab Domain CMU* in the lower part of the figure).
What we show in Figure 6.4 is that there are two means by which the application can interact with the `AmI-Lab` *ContextDomain*. It presents the different types of interplay that can take place between the application level and the CONSERT Middleware CMUs that are meant to facilitate context management within the application.
In the approach shown on the left side of the figure we see the case where the application decides to deploy a single CMU that will manage both context information coming from the `AmI-Lab` *ContextDomain* as well as information that is particular to Alice's own rules of specifying her availability status. In this setup, the `CtxUser` agent in the CMU will act as a direct relay of context from one CMU to the other. We notice that the `CtxUser` agent can only communicate with the local `CtxCoord` and `CtxQueryHandler` agents. These latter agents, however, have received the address of the remote *node* `OrgMgr` running in the CMU of the AmI-Lab management server. In this way they can participate in the routing of a domain-

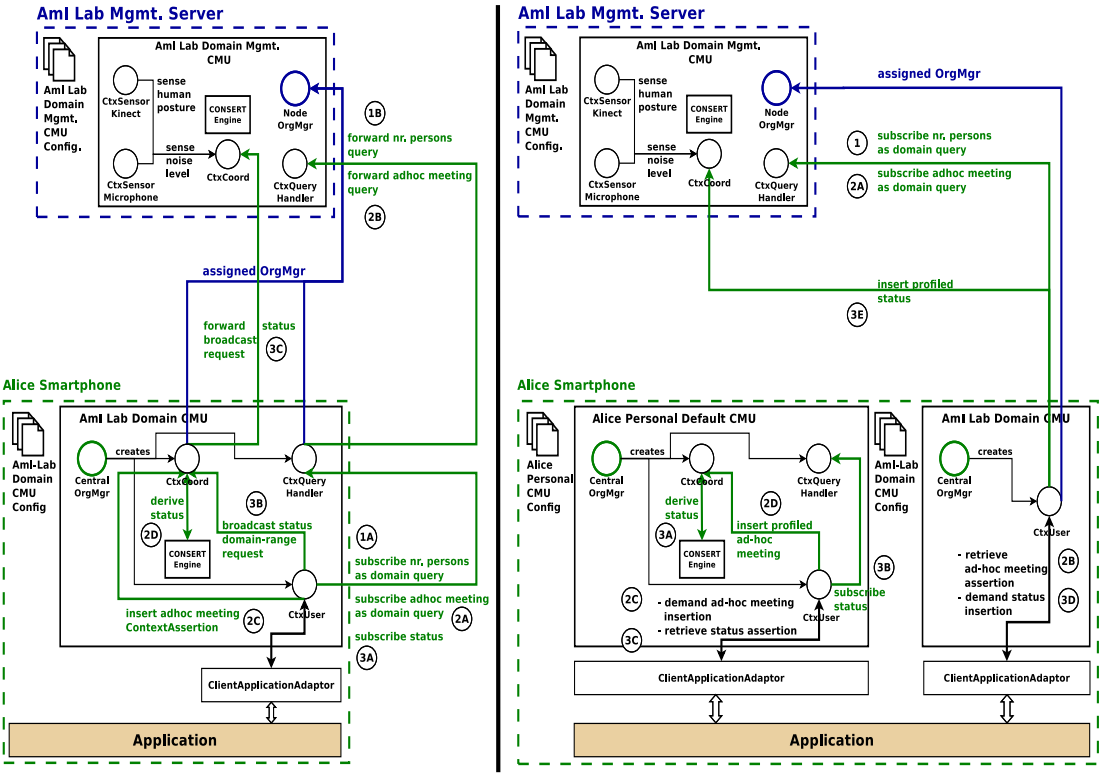

Figure 6.4: Two methods to implement context information relay using the `CtxUser` insertion and broadcast capabilities. Left side shows mechanism using a single CMU and domain-based range broadcast. Right side shows two CMUs linked by the application level and profiled insertion capability. Numbered circles shows temporal sequence of events and actions.

based query or broadcast as shown in the previous sections. The approach exploits precisely these options since the application instructs the `CtxUser` through the `Client Application Adaptor` to perform the following actions:

- an *exact domain* subscription for the number of people at the same location as Alice (step 1A)
- notifications of being in an ad-hoc meeting at that location (step 2A)
- make a local profiled insertion of the positive notifications of detected ad-hoc meeting situations (step 2C)
- make a local subscription for the availability status (step 3A)
- broadcast the availability status with an upper limit of the type `LaboratoryRoom`, meaning that it will stop at the `AmI-Lab` *ContextDomain* (step 3B)

Analyzing this approach we can note that it is suitable when the availability status (busy/free) is only relevant whilst Alice is in the AmI-Lab (since the CMU deployed on her smartphone that is assigned to this *ContextDomain* is only active during this time).

If the availability status is actually part of contextual information that is particular to Alice's everyday activities, then the approach shown on the right side of Figure 6.4 can be used. In this case the CMU deployed to interact with the `AmI-Lab` *ContextDomain* is decoupled from the CMU that was already deployed on Alice's smartphone and which is meant to handle her *personal* context, which includes the conditions she sets for being busy. This is the case show-

ing how the application-level can combine and relay context data from/between CMUs that run on the same device, but each handling different context information aspects. In the approach, the `CtxUser` running in the AmI-Lab Domain CMU communicates exclusively with the `CtxQueryHandler` and `CtxCoord` agents from the CMU running on the AmI-Lab management server. Its role is limited to subscribing for the number of people and ad-hoc meeting context information (steps 1 and 2A) and sending the profiled *ContextAssertion* informing of Alice's status which she wants to make available at `AmI-Lab` *ContextDomain* level.
The received query answers become available to the application level via the `Client Application Adaptor` used by the `CtxUser` agent in the `AmI-Lab` Domain CMU. The same adaptor type, employed by the `CtxUser` agent in the `Alice Personal` Domain CMU, is used to instruct the corresponding user agent to insert these query answers in the local CONSERT Engine by sending them as profiled *ContextAssertions* to the `CtxCoord` agent from the CMU. In this way, the application-level has performed a relay of context information between two different CMUs and the *ContextDerivationRules* running in the local CONSERT Engine instance can derive Alice's availability status based on this additional input. The assertion of the resulting busy status at `AmI-Lab` *ContextDomain* level (steps 3A - 3E) occurs in a similar manner, but in a reversed sense.

The example shown above illustrates two aspects. On the one hand it exemplifies the prosumer behavior of the `CtxUser` agent. Furthermore, it shows one of the key elements of the CONSERT Middleware, namely flexible deployment options. We noticed that the same context provisioning functionality can be engineered in several ways depending on the specific application design requirements which constitutes an important advantage.

6.4.5 Mobility Management

In Figure 6.1 we explained that the role of the bootstrap CMU on Alice's smartphone, managed by a *mobile* `OrgMgr`, is to detect when different *ContextDomains* become accessible/inaccessible. While the CONSERT Middleware does not currently offer a standard and readily exploitable method to perform discovery of mobile nodes and mobility management, the application level has two main possibilities to achieve this. Detection of *ContextDomain* changes (*entering* or *leaving* a domain) can be done in an active or passive way, from the perspective of the mobile node.
In an active mode, the CMU deployed to handle *ContextDomain* detection has to actively search for the information that tells it when in enters or leaves a *ContextDomain*.
In the passive mode, it is actually the *root* or *node* `OrgMgr` of the CMUs deployed on the machines that handle coordination of a *ContextDomain* (e.g. tram management service, AmI-Lab management server). In what follows we briefly discuss each possible approach.

Passive Mode In the passive mode, a *mobile* `OrgMgr` acts as a receiver of notifications for entering/leaving *ContextDomains*. The *node* or *root* `OrgMgrs` that manage the CMUs handling the coordination of a given *ContextDomain* perform a monitorization of the *ContextAssertions* which signal that a mobile node has access to their managed *ContextDomain*.
Facilitation for observing the *ContextDimension* instances which imply an entering of a *ContextDomain* exists in the CONSERT Middleware under the form of *ContextDomainEntered* and *ContextDomainLeft* rules. These are special forms of *ContextDerivationRules* which output information about the detected *ContextDimension* instance and, most importantly, the *ContextEntity* instance playing the subject role in the dimension.
To see an example of the passive mode in action, we refer again to Figure 6.1. In that case, both the tram and the AmI-Lab `OrgMgr` agents inform the bootstrap `OrgMgr` of entering/leaving the corresponding *ContextDomain*. To see how this could be achieved, we first remind that the *ContextDimension* from which domains arise comprises a subject and an object part.

To identify who/what has *entered* a *ContextDomain*, a *node* or *root* `OrgMgr` has to monitor the insertions of those *ContextAssertions* which actually represent *ContextDimension* instances. For example, for the `AmI-Lab` *ContextDomain* the *node* `OrgMgr` has to monitor instances such as `locatedIn(Alice, AmI-Lab)`, whereby `Alice` is the *subject ContextEntity* of type *Person*. After observing such an instance, the `OrgMgr` should be able to extract a profile linked to the subject *ContextEntity* (i.e. `Alice` in our example) to retrieve the addressing information of the *mobile* `OrgMgr` agent in the bootstrap CMU running on Alice's smartphone. As explained, there is currently no standardized support for how this information becomes accessible within an application, so it can be assumed that the profile information is given and updated via the mechanism falling outside the working of the CONSERT Middleware.
To continue the example, consider that location sensors within the laboratory use bluetooth MAC addresses to detect the existence of a smartphone device in their vicinity. If the profile information discussed above contains data specifying that Alice is the owner of a smartphone with a given bluetooth MAC address, than a designer can write a *ContextDomainEntered* rule which deduces the *ContextDimension* instance corresponding to Alice being in the AmI-Lab. The *ContextDomainLeft* rule could then, for instance, say that if the validity of `locatedIn(Alice, AmI-Lab)` *ContextDimension* instance has expired a given amount of time ago (i.e. there were no updates for the specified time interval), then it can be considered that Alice no longer has access to the `AmI-Lab` *ContextDomain*.
Both *ContextDomainEntered* and *ContextDomainLeft* rules are registered automatically by the `CtxCoord` agent from the given *ContextDomain* coordination CMU and run in the CONSERT Engine. Likewise, the `OrgMgr` subscribes automatically for the specific output of these special rules.

Active Mode In the active mode, a *mobile* `OrgMgr` is in charge of managing a CMU dedicated to determining the set of *ContextDomains* to which the mobile machine on which the CMU runs currently has access. The typical bootstrap CMU will contain a `CtxSensor` instance, whose role it is to detect *ContextDomain* announcements.
The active mode currently has a similar facilitation as the passive one and involves again application-specific engineering. For instance, consider the case where the AmI laboratory has its own wifi network on which the management service running on top of the `AmI-Lab` domain management CMU can constantly broadcast the *ContextDimension* and *ContextDomain* information, as well as the address of the corresponding *node* `OrgMgr`. The `CtxSensor` in the bootstrap CMU would be configured to work with a `Context Assertion Adaptor` that would allow them to receive these broadcasts. The bootstrap CMU would also include a `CtxCoord` agent instance which would run *ContextDomainEntered* and *ContextDomainLeft* that, like in the previous case, would output information about the detected dimension and domain (or lack of updates thereof, for the *domain left* case). The application designer would in this case instruct `CtxUser` on the bootstrap CMU to subscribe for the notification of the output of these special rules. In this way, the application-level would be notified (via the `Client Application Adaptor` interface) of different *ContextDomain* changes and, as shown in Figure 6.1, it can launch the appropriate CMUs that would handle contextual interaction with the newly detected *ContextDomain*.

6.5 Discussion

In this chapter we have completed the conceptual overview of the CONSERT Middleware architecture and functionality which started with the presentation of the context representation and reasoning method (Chapter 4), continued with the aspects of the provisioning process and its adaptation (Chapter 5) and ended with the explanation of context provisioning deployment options and mechanisms.

Specifically, in this chapter we have seen how the concepts of *ContextDimension* and *ContextDomain*, which arise from the multi-dimensional aspect of the application context model itself, are exploited to provide the logical deployment structure of context provisioning units (i.e. the agents of a Context Management Unit). We have analyzed how this structure elements allow us to consider two deployment schemes: centralized and decentralized. In the decentralized case the possibility exists to create a hierarchy of *ContextDomains* which, in turn, has benefits in terms of extended context query and broadcast mechanisms.
We then further addressed the issue of supporting design and engineering of this deployment flexibility via a declarative configuration approach, which is able to define computational platform, *ContextDomain* and agent-level specifications. We showed how a deployment configuration allows the application developer to specify the agent composition of a CMU and, therefore, assign it to handle one or more aspects of the main context provisioning life cycle (acquisition, coordination, dissemination) of a *ContextDomain*. We then explained how the life cycle of a CMU is controlled by the `OrgMgr` agent, which oversees all deployment-related aspects (including providing help with inter-domain routing and mobility management). As we will explore in the next chapter related to CONSERT Middleware implementation, support for the latter CMU life cycle aspects is further enhanced by the software component based design of our system.

At the end of the previous chapter we had begun an overview of how our CMM addresses several of the context management operational requirements introduced in Section 3.1.1. In what follows we comment on further aspects.
From the point of view of the main life cycle operations (acquisition, coordination and dissemination) we see an augmented functionality in the production and dissemination capabilities of the middleware. Specifically, the decentralized deployment scheme based on distributed *ContextDomains* allows for a domain-based query and broadcast functionality. This functionality has the added benefit of being able to impose dissemination limits based on *ContextDomain* types, that is, it is tied to the semantics of the application context model, not just to predetermined physical network configuration.
Context Producer Discovery and *Mobility Management* are not yet addressed completely. In future work we aim to define standard mechanisms of performing these aspects within the CONSERT Middleware. However, as explained in Section 6.4.5, partial support for these issues already exists in our CMM. Specifically, the *ContextDomainEntered* and *ContextDomainLeft* rules and the functionality of both *root/node* as well as *mobile* `OrgMgr` agents ensure facilitation of the application-level engineering of dynamic discovery and change of *ContextDomains*.
Non-functional aspects such as *scalability* and *robustness* are addressed by the flexible deployment options of our middleware. The *ContextDomain*-based *context model* partitioning and CMU-based *context provisioning* implementation allow our CMM to be employed in scenarios ranging from single device context management to smart campus applications such as in our reference scenario. The fact that CMUs that handle provisioning of the same *ContextDomain* can be physically deployed on different dedicated computing nodes further increases the foundation for scalability. Since these aspects also often depend on the implementation characteristics of a system, we will revisit them in chapter 7 where we discuss how the chosen agent development framework provides support scalability and robustness related issues.
Lastly, an important contribution is the facilitation of application development via *ease of configuration*. As in the case of context provisioning sensing and coordination policies, all relevant aspects of context provisioning deployment can be declaratively specified, which reduces implementation effort and constitutes one of the goals of our approach.

Chapter 7

CONSERT Middleware Implementation

The previous three chapters have presented the conceptual modeling and architectural elements that underpin our definition and design of the CONSERT context management middleware. This chapter is dedicated to exploring the software development practices, tools and frameworks which help implement it. We present our approach for each of the aspects discussed in previous chapters (modeling/reasoning, provisioning, deployment), explaining the motivation and details of our implementation choices in each case.

In Section 7.1 we begin our presentation by discussing aspects of context representation. We show the concrete way in which instances of an application context model built with the CONSERT Ontology are stored and accessed at runtime, as well as how *ContextDerivationRules* are encoded and included in a context model. We also talk about how the vocabularies for the provisioning and deployment policies introduced in Chapters 5 and 6 respectively are implemented as ontologies.
Section 7.2 explores the implementation of the CONSERT Engine, specifically its execution cycle and its implementation as a software service component.
In Section 7.3 we talk about the chosen implementation framework for the context provisioning agents, as well as the means by which these agents find and use the different adaptor services which are present in their environment and are required for their functionality. We then discuss implementation-specific details concerning the provisioning adaptation functionality of the CMM agents.
Finally, we present the method for packaging all context provisioning deployment configurations and the means by which they are handled at runtime in Section 7.4, before concluding the chapter with a discussion in Section 7.5.

7.1 Context Representation Implementation

Here we begin our overview of how the context modeling structures introduced in Chapter 4 are implemented. Specifically, we talk about the runtime storage specifics of *ContextAssertions* and *EntityDescriptions*, how *ContextAnnotations* are "attached" to their corresponding *ContextAssertion* instance, as well as how *ContextDerivationRules* and *ContextConstraints* are encoded and bound to the CONSERT Ontology based definition of the *ContextAssertion* type which they derive and constrain respectively.
Furthermore, we introduce the two ontologies which capture the parameter vocabulary for context provisioning and deployment policies. In this way, a uniform modeling and implementation

approach is maintained across all aspects of the CMM.

7.1.1 Using Named Graphs as Identifiers

In Section 4.2 we explained that the CONSERT context meta-model takes the form of an ontology capturing context content, annotation and constraint information. We now explain how instances of the context modeling elements are stored and related to each other.

RDF Named Graphs [Carroll et al., 2005] lie at the heart of our approach. These allow a set of RDF statements (subject - predicate - object triples), called a graph, to be grouped and associated with a URI (i.e. the graph *name*). In this way, a logical partitioning of an RDF dataset can be achieved. Furthermore, the URI which acts as an identifier of the named graph can be itself used in other RDF statements as their subject or object part.
Since instances of context information expressed using the CONSERT Ontology are in fact RDF statements, named graphs can be used to our advantage. They essentially facilitate the *identification* of individual *ContextAssertion* instances and thus allow the statements representing *ContextAnnotations* of those assertions to be assigned to their corresponding instance. Specifically, each individual RDF statement ($n = 2$) or set of statements ($n = 1, n \geqslant 3$ - cf. Examples 4.2.1 and 4.2.2) expressing a *ContextAssertion* is wrapped within its own graph and has the *name* of the graph as an identifier. The *ContextAnnotations* are then expressed as RDF statements which have the named graph URI as their subject.

We mentioned that named graphs act as logical separators of RDF datasets. Besides the use in *ContextAssertion* instance identification, as explained above, the CONSERT Middleware uses the named graph approach to establish separate runtime storage partitions for:

- static information: *ContextEntity* definitions and *EntityDescriptions*
- annotation information: all *ContextAnnotations* for the instances of a given *ContextAssertion* type are stored in the same named graph

Table 7.1 summarizes the form and use of named graphs for the structuring of context information in CONSERT.

Modeling Element	Named Graph URI form	Role
ContextEntity EntityDescription	http://purl.org/net/consert/entityStore	The named graph where all instances of *ContextEntities*, *ContextEntity* definition axioms and *EntityDescriptions* are kept.
ContextAssertion	URI of *ContextAssertion* + UUID seed	The named graph wrapping a *ContextAssertion* instance and acting as its identifier.
ContextAnnotation	URI of *ContextAssertion* + *Store*	The named graph where annotations of all instances of a *ContextAssertion* type are kept.

Table 7.1: List usage cases for named graphs in the CONSERT Middleware.

The table shows that static context information is kept in the *entity store*, which is assigned a specific named graph URI. Individual *ContextAssertion* instances are stored in a graph whose URI is obtained by adding the textual form of a generated UUID[1] to the URI denoting the *ContextAssertion* type.
Lastly, for each *ContextAssertion* type we create a special storage partition, called the *ContextAssertion Store*. The graph for this partition obtains its name by taking the *ContextAssertion*

[1] http://en.wikipedia.org/wiki/Universally_unique_identifier

URI, replacing all '#' occurrences with '/' and appending the word *Store* at the end. The obtained named graph contains all the annotations for the instances of the given *ContextAssertion* type.

Let us take an example from our reference scenario to illustrate all the above.

```
GRAPH  <ex:hostsAdHocDiscussion-UUID> {
 _:0 rdf:type ex:hostsAdHocDiscussion.
 _:0 core:assertionRole ex:AmI-Lab.




}
```

(a) A named graph identifying an instance of the hostsAdHocDiscussion UnaryContextAssertion

```
GRAPH <ex:hostsAdHocDiscussionStore> {
 ex:hostsAdHocDiscussion-UUID core:assertionType
   ctx:Sensed.
 ex:hostsAdHocDiscussion-UUID ann:hasTimestamp
   ex:tsAnn.
 ex:tsAnn ann:hasValue
   "2015-04-06T12:00:05Z"^^xsd:datetime
 . . .
}
```

(b) The "store" named graph of the hostsAdHocDiscussion *ContextAssertion* holding annotations

```
GRAPH  <http://purl.org/net/consert/entityStore> {
 ex:AmI-Lab rdf:type ex:LaboratoryRoom.
}
```

(c) The *EntityStore* holding information about the AmI-Lab *ContextEntity*

Figure 7.1: Example contents of *EntityStore* and *ContextAssertion Store* and *identifier* named graphs.

Figure 7.1 shows examples of each of the previously mentioned use cases for named graphs on hand of an instance of the hostsAdHocDiscussion `UnaryContextAssertion` taken from the AmI-Lab part of our reference scenario. Subfigure a) shows how the statements that make up an instance of the hostsAdHocDiscussion are stored in their own named graph whose URI consists of the assertion URI (ex:hostsAdHocDiscussion) to which an UUID is appended (shown in the figure just as UUID in order to obtain compact formatting of the picture). Subfigure b) shows the corresponding *ContextAssertion Store*, where annotation information about the assertion instance in subfigure a) is maintained. Lastly, the *EntityStore* shown in subfigure c) contains the definition of the AmI-Lab as an instance of a LaboratoryRoom *ContextEntity*.

As we explore in Section 7.2.1, these storage options are used at runtime by the CONSERT Engine to perform its insertion, inference and query execution cycles.

7.1.2 Rule Encoding using SPIN

In Section 4.3 we showed how context derivation and consistency maintenance are achieved by *ContextDerivationRules* and *ContextConstraint* expressions modeled as SPARQL CONSTRUCT queries. We now look at how these queries are serialized and assigned to the corresponding *ContextAssertion* types such that the CONSERT Engine can create its auxiliary data structures as discussed in Section 4.4.1.
Rule and constraint query encoding is made using the SPARQL Inferencing Notation (SPIN) [Knublauch et al., 2011] proposal. SPIN currently has the status of a W3C Member Submission, but it has become the de facto industry standard to represent SPARQL rules and constraints on Semantic Web models[1]. Apart from providing an RDF serialization of SPARQL queries, SPIN offers a vocabulary that enables definition of inference and constraint rule templates and "attaching" instances of such templates to elements (classes, properties) in an ontology model.

[1] http://spinrdf.org/

The CONSERT Ontology defines base templates for both derivation and constraint checking. These templates are subclasses of the spin:ConstructTemplates class. In the case of *DerivationRules* the base template for any rules shaped as SPARQL CONSTRUCT queries is called `DerivationRuleBase`. The SPIN specification allows for the definition of parameters of a template which may be used at runtime. The CONSERT Middleware uses two default arguments for the `DerivationRuleBase`: the `contextAssertionType` and `contextAssertionUUID` parameters. Both of these arguments identify the *ContextAssertion* instance which has triggered the execution of the current *DerivationRule*. They specify the type (URI of the *ContextAssertion* as given in the application context model built using the CONSERT Ontology) and the identifier URI (as explained in Section 7.1.1) of the triggering assertion. The values of these arguments are supplied at runtime by the CONSERT Engine who is executing the rule.
For *ContextConstraints* the base template is called `ContextConstraintTemplates` and it defines three subclasses, corresponding to the three types of constraints considered in the CONSERT Ontology: value (`ValueConstraint Templates`), uniqueness (`UniquenessConstraint Templates`) or general integrity (`IntegrityConstraint Templates`). The same two parameters (called `triggerAssertionType` and `triggerAssertionUUID` in this case) are defined by default for a `ContextConstraintTemplates` class.

Once created, a SPIN encoded *DerivationRule* or *ContextConstraint* needs to be included in the application context model and "assigned" to the *ContextAssertion* type it derives or constrains respectively. In this way the CONSERT Engine can build its auxiliary data structures which help it know which *ContextAssertions* are the object of a derivation or a constraint check. The CONSERT Ontology defines two OWL properties for this purpose.
The `spin:deriveassertion` is a sub-property of `spin:rule` and serves to bind an instance of a template extending the `DerivationRuleBase` to the *ContextAssertion* type it derives.

```
ex:AdHocDiscussionRule    rdfs:subClassOf        rules:DerivationRuleBase .
_:adhocrule               rdf:type               ex:AdHocDiscussionRule .
ex:hostsAdHocDiscussion   spin:deriveassertion   _:adhocrule .
```

The lines above continue the previous examples and show how an instance of the `AdHocDiscussion Rule` subclass of the `DerivationRuleBase` is attached to the `hostsAdHocDiscussion` *ContextAssertion*. In this case, our template instance is identified by an anonymous RDF node (`_:adhocrule`) since it does not need to be referenced anywhere else, other than in this assignment.
In the constraint case, the `spin:contextconstraint` property is used to bind an instance of a constraint template to the *ContextAssertion* type it constrains. In our reference scenario, a uniqueness constraint is defined on Alice's availability status, specifying that she cannot be free and busy at the same time. Thus, similar to the above example we have the following constraint assignment statements:

```
ex:AvailabilityConstraint   rdfs:subClassOf         constr:UniquenessConstraintTemplates .
_:statusconstr              rdf:type               ex:AvailabilityConstraint .
ex:hasAvailabilityStatus    spin:contextconstraint  _:statusconstr .
```

Statements such as the ones above are included together with the rest of statements that give shape to the context model of an application. In Section 7.2 we will see that in order to help developers separate the concerns of content, annotation, constraint and rule modeling, the set of RDF statements corresponding to each aspect can be included in their own file.

To complete the overview of representation-related implementation aspects, in the following two subsections we present the ontology-based realization of the vocabulary used in the development of provisioning and deployment policies.

7.1.3 Provisioning Ontology

In Section 5.2 we presented the structure of context provisioning policies made up of parameters and rules that control sensing and coordination behavior of the corresponding CONSERT Middleware agents. Here we explain how the concepts introduced in that chapter are implemented as the CONSERT Provisioning Ontology[1].

The choice for implementing the vocabulary as an ontology is based on the idea of uniformity of implementation. This means that the same CONSERT Middleware internal logic that reads and uses an application context model built using the CONSERT Ontology can do so as part of the initialization phase of the CONSERT Middleware agents.

Policy Parameter Definition

The ontology defines OWL classes for the concept of a policy instance itself, i.e. `SensingPolicy` for sensing and `ControlPolicy` for coordination. Further OWL classes define concepts such as provisioning control rule output commands (e.g. `StartAssertionCommand`, `StartRuleCommand`, `UpdateModeCommand`) or place-holders for *ContextAssertion*-specific provisioning control parameters (e.g. `AssertionSpecificEnableSpec`, `AssertionSpecificConstraintResolutionType`).
The actual parameters are implemented as OWL object properties which bind an instance of the `SensingPolicy` or `ControlPolicy` classes to the corresponding parameter value. To showcase the above, we revisit the examples given in Section 5.2 and present them in their corresponding ontology form.

```
:presenceSensingPolicy
    a  sensorconf:SensingPolicy ;
    coordconf:forContextAssertion  ex:sensesBluetoothAddress ;
    sensorconf:hasUpdateMode  coordconf:time-based ;
    sensorconf:hasUpdateRate  2 .

:luminositySensingPolicy
    a  sensorconf:SensingPolicy ;
    coordconf:forContextAssertion  ex:sensesLuminosity ;
    sensorconf:hasUpdateMode  coordconf:change-based ;
    sensorconf:hasUpdateRate  0 .
```

Figure 7.2: Example of sensing policy specifications for presence and luminosity *ContextAssertion* updates in CONSERT Provisioning Ontology form (Turtle syntax).

Figure 7.2 shows the initial sensing update mode configuration for the presence and luminosity *ContextAssertions* that were given as an example from our reference scenario in Section 5.2.1. Notice specifically the use of an OWL property called `forContextAssertion`. This property enables specifying *ContextAssertion*-specific parameters, such as is the case for the parameters of a sensing policy.

Figure 7.3 revisits the example on coordination policy control parameter specifications given initially in conceptual form in Section 5.2.2.

We see how `ProvisionPolicy_AmILab` is defined as an instance of a `ControlPolicy` and how the control parameters are assigned as properties of this instance. In the case of *ContextAssertion*-specific parameters (e.g. enabling updates for device presence via bluetooth sensing, enabling derivation for person location based on device presence) an instance of an `AssertionSpecificEnableSpec` class is created which has as OWL properties the type of

[1] http://purl.org/net/consert-provisioning-ont/control, http://purl.org/net/consert-provisioning-ont/sensing

```
:ProvisionPolicy_AmILab
    a  coordconf:ControlPolicy ;
    coordconf:enablesAssertionByDefault  false ;
    coordconf:hasDefaultOntReasoningInterval 10 ;
    coordconf:hasDefaultRunWindow  20 ;
    coordconf:hasDefaultTTLSpec    100 ;
    coordconf:hasDefaultUniquenessConstraintResolution coordconf:PreferNewest ;
    coordconf:hasDefaultIntegrityConstraintResolution coordconf:PreferAccurate ;

    coordconf:hasSpecificAssertionEnabling  [
      a  coordconf:AssertionSpecificEnableSpec ;
      coordconf:forContextAssertion  amilab:sensesBluetoothAddress ;
      coordconf:hasParameterValue true
    ]
    coordconf:hasSpecificAssertionEnabling  [
      a  coordconf:AssertionSpecificEnableSpec ;
      coordconf:forContextAssertion  person:locatedIn ;
      coordconf:hasParameterValue  true
    ] .
```

Figure 7.3: Example of coordination policy specifications in CONSERT Provisioning Ontology form (Turtle syntax).

ContextAssertion for which it applies (`forContextAssertion`) and the value of the parameter (`hasParameterValue`).

Policy Control Rule Definition

When we talked about provisioning control rules in Section 5.2.2 we explained that they as well are structured as SPARQL CONSTRUCT queries and we detailed the type of outcomes these rules can have. We now observe that the rule outcomes correspond to OWL classes from the CONSERT Provisioning Ontology such as `StartAssertionCommand`, `StopDerivationCommand`, etc.
As was the case for *DerivationRules* and *ContextConstraints*, the SPARQL expression of a provisioning control rule is encoded using the SPIN specification and is part of a `CommandRuleTemplate` defined by the CONSERT Provisioning Ontology. The ontology also contains a set of predefined subclasses of the `CommandRuleTemplate` which deal with commonly expected cases of provisioning adaptation/control. For example, there is a template which prescribes stopping the updates of a given *ContextAssertion*, if no queries and subscriptions (as collected from the CONSERT Engine statistics) for this assertion type have been received since a given time threshold (which is a parameter of the template). Another one prescribes the activation of *ContextAssertion* updates for a given assertion type if the current time is between a given interval (specified as parameter of the template). Naturally, in an application, the developer is able to create his own control rule templates besides the default ones.

Furthermore, just as in the cases for derivation rules and constraints, dedicated the CONSERT Provisioning Ontology defines a dedicated OWL property (`hasCommandRule`) which is used to "attach" a `CommandRuleTemplate` (or subclass thereof) instance to a provisioning coordination policy definition.
Remember also from Section 5.3.2 that provisioning control rules can be partitioned into ordered execution groups. At implementation level, the groups are created by defining sub-properties of `hasCommandRule`. The control rule instances (i.e. instances of a `CommandRuleTemplate` or its subclasses) that belong to a group are assigned to the provisioning coordination policy definition using the sub-property of `hasCommandRule` which determines that group. We refer the reader to Section 7.3.3 for more details on how ordering and execution of the rules is performed.

In Figure 7.4 we provide an example of the `CommandRuleTemplate` subclass mentioned above

(canceling updates if no query for given time threshold) and how an instance of this template is assigned to the coordination policy definition. In this case we are viewing a template instance for canceling updates for luminosity information, if no projector has made queries for this *ContextAssertion* in more than 20 seconds.

```
CONSTRUCT {
  _:b0 a :StopAssertionCommand.
  _:b0 :forContextAssertion ?assertion.
}
WHERE {
  ?stat a :AssertionSpecificStatistic.
  ?stat :forContextAssertion ?assertion.
  ?stat :isDerivedAssertion true.
  ?stat :nrSubscriptions 0.
  ?stat :timeSinceLastQuery ?time.
  FILTER (?time > ?elapsedThreshold).
}
```

```
coord:ControlPolicy
  coord:hasStopAssertionCommand [
    a  :QueryAbsenceAssertionCancellation;
    arg:contextAssertion ami:sensesLuminosity;
    arg:elapsedTimeThreshold 20;
  ];
```

Figure 7.4: SPARQL expression of derivation cancellation rule template (left) and control rule assignment (right)

In the WHERE clause of the CONSTRUCT query we can see some of the parameters collected as statistics by the CONSERT Engine (cf. Section 5.3.1 for details). The statistics parameters are also defined in the CONSERT Provisioning Ontology under the form of OWL properties. We refer again to Section **??** for explanations of their runtime retrieval. The CONSTRUCT clause of the query creates an instance of the `StopAssertionCommand`. The variables which start with a '?' (*assertion* and *elapsedThreshold*) are the parameters of the control rule template. They obtain their value when doing the template instance assignment, as shown on the right side of the figure.
For the control rule assignment part (right side of the figure) notice the `hasStopAssertionCommand` property which is used to attach the template instance to the `ControlPolicy` class. This property is a sub-property of `hasCommandRule` and it creates the execution group of all *ContextAssertions* which need to be stopped at the time of running the control rules. In Section 8.2.3 of the evaluation chapter we will show that for our AmI-Lab provisioning control as part of reference scenario we consider two execution groups: *ContextAssertions* and *DerivationRules* which need to be enabled and those that need to be disabled.

Provisioning Policy Verification

Some last noteworthy details about the implementation of context provisioning policies using an ontology definition are the aspects related to correctness verification and providing of defaults for a policy specification. The SPIN proposal, mentioned in the previous section, provides its own templates for inferring default values of an OWL property (`spl:InferDefaultValue`) and ensuring that a given amount of instances of a given property are present in an RDF document (`spl:ObjectCountPropertyConstraint`).
These elements can be exploited to perform a structural verification of a provisioning policy file. For example, an definition of a an `AssertionSpecific Command` instance is constrained to have a single value for the `forContextAssertion` OWL property. Similarly, for the `ControlPolicy` class, a set of defaults are defined for each of the general provisioning control parameters. For example, the default strategies for general integrity and uniqueness constraint resolution are always set to *prefer newest*, while all *ContextAssertion* updates are enabled by default.
These verification options ensure a stable functionality for all provisioning processes that depend on valid and complete contents of a context provisioning policy.

7.1.4 Deployment Ontology

In order to continue the uniformity of the development process for the CONSERT Middleware and the applications based on it, the vocabulary for deployment policy parameters too has been modeled as the CONSERT Deployment Ontology[1].
In Section 6.2 we explained that deployment specifications are related to three configuration aspects: the platform on which one or more CMUs run, the *ContextDomain* specifications and the configuration of the provisioning agents of a CMU. The CONSERT Deployment Ontology follows this partitioning and defines OWL classes for each such deployment aspect. We distinguish a `PlatformSpec`, an `ApplicationSpec` (for *ContextDomain* configurations) and several subclasses of the `AgentSpec` class corresponding to individual configurations for each CMM agent type (e.g. `OrgMgrSpec`, `CtxCoordSpec`).
The deployment policy parameters discussed throughout Section 6.2 are then implemented as OWL object or datatype properties of the instances of configuration classes such as the above. Additional OWL classes are defined by the CONSERT Deployment Ontology for parameters which are of a more complex nature. For example, an `AgentContainer` class acts as a place holder for all the coordinates related to the platform container specification. It will therefore have properties such as `containerHost`, `containerPort` and `platformName` which correspond to a part of the conceptual parameter definitions given in Section 6.2.1.

The full set of classes and properties can be explored by accessing the URL given as a footnote of this page. As was the case in the previous section, we will revisit some of the deployment configuration examples discussed in Chapter 6 and present them now in their corresponding ontology form (in Turtle syntax[2]).

```
:AmILabPlatformSpec
    a  orgconf:PlatformSpec ;
    orgconf:hasAgentContainer  :Container_AmILab .

:Container_AmILab
    a orgconf:AgentContainer ;
    orgconf:containerHost  "localhost"^^xsd:string ;
    orgconf:containerPort  1099 ;
    orgconf:hasMTPHost  "localhost"^^xsd:string ;
    orgconf:hasMTPPort  7778 ;
    orgconf:platformName  "EF210"^^xsd:string .
```

Figure 7.5: Example of platform configuration from the deployment policy for the AmI-Lab part of the reference scenario.

Figure 7.5 shows the configuration definition for the platform on which the coordination CMU of the AmI-Lab management server runs. Notice the use of the `AgentContainer` class discusses earlier as a holder of the parameters concerning the platform specification.

In Figure 7.6 we show the definition of the `AmI-Lab` *ContextDomain* specifications. The `appDeploymentType` specifies the type of application deployment on the AmI-Lab management server, namely a decentralized one with a domain hierarchy configuration, since the AmI-Lab is part of the CS Building. The `AmILab_Domain` instance of the `ContextDomain` class exposes properties which define the characteristics of the domain. For example, the *ContextDimension* is that of person localization, the object *ContextEntity* of this dimension is of type `LaboratoryRoom` and the domain hierarchy property is the one related to spatial subsumption (inclusion). The `hasDomainHierarchyDocument` is a property specifying the RDF document path where the overview of the *ContextDomain* hierarchy is maintained. We will discuss its contents and its use by the `OrgMgr` agent in Section 7.4.1.
As mentioned briefly in Section 7.1.2, a developer can partition the definition of context model

[1] http://purl.org/net/consert-deployment-ont
[2] http://www.w3.org/TeamSubmission/turtle/

```
:AmILabAppSpec
    a  orgconf:ApplicationSpec ;
    orgconf:appDeploymentType  orgconf:DecentralizedHierarchical ;
    orgconf:appIdentificationName  "AmI-Lab-Smart-Classroom"^^xsd:string ;
    orgconf:hasContextDomain  :AmILab_Domain .

:AmILab_Domain
    a  orgconf:ContextDomain ;
    orgconf:hasDomainDimension  person:locatedIn ;
    orgconf:hasDomainRangeEntity  amilab:LaboratoryRoom ;
    orgconf:hasDomainRangeValue  amilab:AmI-Lab
    orgconf:hasDomainHierarchyProperty space:spatiallySubsumedBy ;
    orgconf:hasDomainHierarchyDocument
      [ a  orgconf:ContentDocument ;
        orgconf:documentPath  "etc/cmm/domain-hierarchy-config.ttl"^^xsd:string
      ] ;
    orgconf:hasContextModel  :AmILab_ContextModel .

:AmILab_ContextModel
    a  orgconf:ContextModelDefinition ;
    orgconf:hasModelCoreDocument  [
      a orgconf:ContentDocument ;
      orgconf:documentURI  "http://purl.org/net/amilab/core"^^xsd:anyURI
    ] ;
    orgconf:hasModelRulesDocument  [
      a  orgconf:ContentDocument ;
      orgconf:documentURI  "http://purl.org/net/amilab/rules"^^xsd:anyURI
    ] .
```

Figure 7.6: Example of *ContextDomain* configuration from the deployment policy specifications for the AmI-Lab part of the reference scenario.

pertaining to a *ContextDomain* into several modules. In our example, we see this option under the form of *core* and *rule* documents whose contents compose the context model belonging to the AmI-Lab *ContextDomain* and managed by the agents of the CMU running on the AmI-Lab management server.

Lastly, Figure 7.7 shows an example of defining the agent specification for the `CtxCoord` and `CtxSensor` agents running in the CMU on the AmI-Lab management server. Notice specifically the use of the `hasControlPolicy` and `hasSensingPolicy` properties. These indicate paths to RDF documents which contain statements regarding sensing and coordination policy specifications as explained in Section 7.1.3. We will talk more about the inclusion of provisioning and deployment policy contents in different configuration files in Section 7.4.1.

7.2 CONSERT Engine Implementation

As explained in Chapter 4, the CONSERT Engine (cf. Figure 7.8) is the middleware component that leverages a context model created using the CONSERT Ontology. In Section 4.4 we discussed the conceptual architecture and execution cycle of the CONSERT Engine. In what follows, we explore the frameworks and design specifications which let us obtain the functionality of the engine.

7.2.1 Data Structures and Execution Cycle

The purpose of the CONSERT Engine is to provide storage, inference and consistency and query answering capabilities for the context information that it handles. The implementation

```
:CtxCoord_AmILab
    a orgconf:CtxCoordSpec ;
    orgconf:hasAgentAddress  :CtxCoord_AmILab_Address ;
    orgconf:hasControlPolicy  [
      a  orgconf:AgentPolicy ;
      orgconf:hasPolicyDocument [
        a  orgconf:ContentDocument ;
        orgconf:documentPath  "etc/cmm/coordconfig.ttl"^^xsd:string
      ]
    ] .

:CtxSensor_AmILab_NoiseLevel
    a orgconf:CtxSensorSpec ;
    orgconf:hasAgentAddress  :CtxSensor_AmILab_Address ;
    orgconf:hasSensingPolicy  [
      a  orgconf:CtxSensorPolicy ;
      orgconf:forContextAssertion amilab:hasNoiseLevel ;
      orgconf:hasPolicyDocument [
        a  orgconf:ContentDocument ;
        orgconf:documentPath  "etc/cmm/sensorconfig.ttl"^^xsd:string
      ]
    ] .
```

Figure 7.7: Example of `CtxCoord` and `CtxSensor` agent configurations from the deployment policy for the AmI-Lab part of the reference scenario.

of these capabilities relies heavily on two semantic web frameworks: the Apache Jena Framework[1] and the SPIN API[2]. In the following, we present how these frameworks are used in the implementation of the data structures and facilitation of the execution cycle of the CONSERT Engine.

Data Structures

The context knowledge base (called *ContextStore* in our middleware) is based on Apache Jena's TDB quadstore engine[3]. The *ContextStore* holds the *ContextAssertions* and their accompanying *ContextAnnotations*, as well as the *ContextEntity* and *EntityDescription* instances under the named graph form presented in Section 7.1.1. One advantage of TDB is that it offers an in-memory store variant with support for *transactions*, a feature we currently use to keep consistent views of the existing context information when handling the update, inference and query requests.
Using the features of Jena API and SPIN API the four auxiliary indexes (ContextAssertionIndex, ContextAnnotationIndex, ContextConstraintIndex and DerivationRuleDictionary) which facilitate the CONSERT Engine runtime functionality are created. As explained more in the next section, the engine's initial configuration contains the list of modules (i.e. RDF files) which compose the specification of the context model that needs to be handled by the CONSERT Engine instance. Thus, the creation of the ContextAssertionIndex entails loading the *core* part of the model (the one defining the *ContextAssertions*, *ContextEntities* and *EntityDescriptions*). The set of possible annotations for the *ContextAssertions* in the model is contained in the *annotation* module. Similarly, constraints and derivation rules are loaded from their respective module files.

Execution Cycle

The execution cycle of the CONSERT Engine starts with a *ContextAssertion*, *ContextEntity* or *EntityDescription* create/update event. The actual update or create request is implemented as

[1] https://jena.apache.org/
[2] http://topbraid.org/spin/api/
[3] https://jena.apache.org/documentation/tdb/index.html

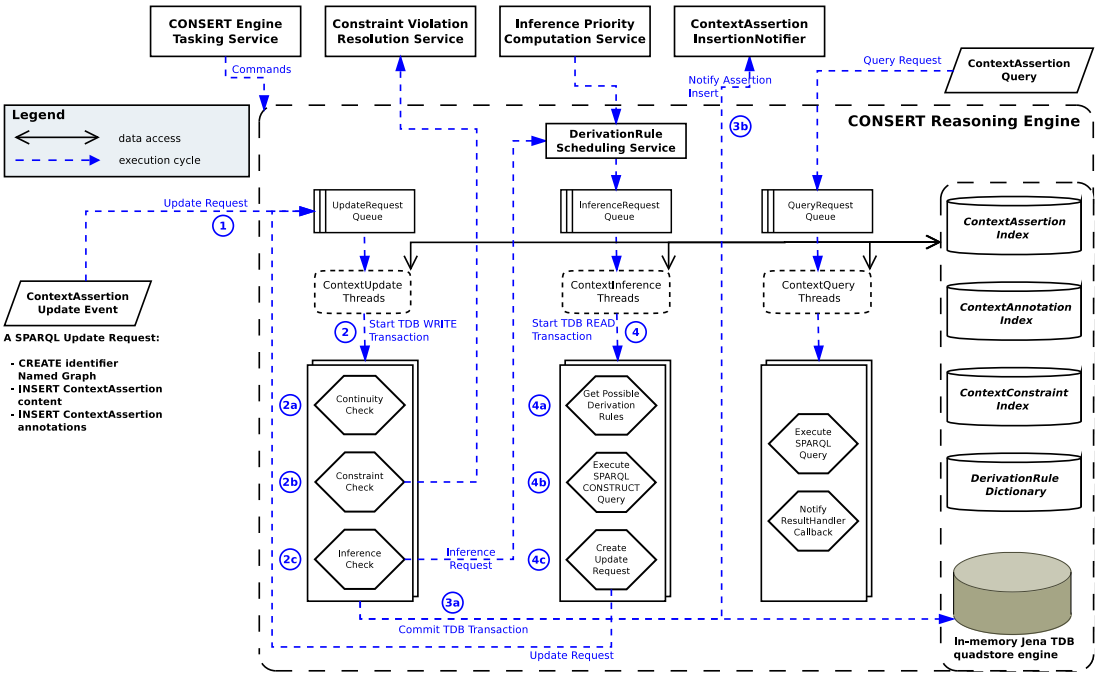

Figure 7.8: The CONSERT Engine architecture and main activity cycle

a SPARQL UPDATE query. In accordance with the named graph structure outlined in Section 7.1.1, the contents of the query are as follows:

(i) a `CREATE GRAPH` request with a unique URI naming the graph that will wrap the new assertion and act as its identifier. The unique URI is obtained by appending the output of a UUID (universally unique identifier) generator to the ontology resource URI corresponding to the *ContextAssertion* instance class.

(ii) (optional) a `SPARQL INSERT/DELETE` request that adds/removes *ContextEntity* or *EntityDescription* instance into the named graph corresponding to the *EntityStore* of the context knowledge base.

(iii) a `SPARQL INSERT` request to put the necessary RDF statements composing the *ContextAssertion* instance in the newly created named graph.

(iv) a `SPARQL INSERT` requests putting *ContextAnnotations* in the named graph corresponding to *ContextAssertion Store* of the newly inserted assertion type. The graph URI created at step (i) serves as the subject of the RDF triples expressing the annotations (cf. Figure 7.1).

The CONSERT engine has three separate thread pools together with corresponding task queues: one for handling *ContextAssertion* update requests, one for processing inferences and one for answering queries. The *ContextAssertion* insertion request which starts the execution cycle is placed in the update reuqest queue. From there it is picked up by the insertion thread which begins the series of checks described in Section 4.4.2. However, before starting the verifications, the thread enters a *TDB transaction*. The transaction behavior is similar to that of relational database management systems and has the purpose of keeping a consistent view of the *ContextStore* in face of concurrent read/write access to it by the three thread pools mentioned above. If any of the verification steps (continuity check, constrain check, inheritance check or inference check) from the insertion procedure return an error, the current transaction is rolled-back, such that the CONSERT Engine continues working as though the update never happened. Upon successful completion of the verifications the transaction is committed and

thus the *ContextAssertion* update becomes visible to the other thread pools (inference, query). An insertion thread will always start a `write` transaction (since it will modify the *ContextStore* in the end), while the inference and query ones will enter `read` transactions since they only access the context knowledge base in order to perform their execution steps.

7.2.2 CONSERT Engine: A Software Service Component

In Section 5.1.3 we mentioned that the CONSERT Engine (along with other adaptors) acts as a software service that makes up part of the context provisioning agent execution environment. The CONSERT Engine is indeed implemented as a software service component based on the OSGi service specification[1]. The engine code is wrapped in what is called a *bundle*, a Java Archive (JAR) file with special properties included in its MANIFEST specification. The advantage is that the CONSERT Engine can thus be used as an independent service component, usable even outside the usual CMU agent environment of which it is part within the CONSERT Middleware.

Since it is implemented as a service component, the CONSERT Engine exposes a set of interfaces (in the software development sense) which allow any application (including our context provisioning agents) to interact with it.
The CONSERT Engine exposes the following four interfaces: `InsertionHandler`, `QueryHandler`, `CommandHandler`, `StatsHandler`. The `InsertionHandler` interface sets the implementation-level interaction possibilities regarding insertion of static context information (*ContextEntities* and *EntityDescriptions*) and sensed or profiled *ContextAssertions*. Within the CONSERT Middleware it is used by the `CtxCoord` agent who controls the updates to the *ContextStore*.
The `QueryHandler` interface is used by the `CtxQueryHandler` agent and allows making *ask* (i.e. retrieve just a true/false answer) or select queries against the *ContextStore*. To handle subscriptions, the interface also allows registration of *ContextAssertion* update listeners which are notified by the *ContextAssertion Insertion Notifier* service mentioned in Section 4.4.1. In this way, the `CtxQueryHandler` agent can attempt the query associated with the subscription request every time a *ContextAssertion* referenced by the subscription is updated.
The `CommandHandler` interface is the one used by the `CtxCoord` agent to perform context provisioning adaptation/control actions. It allows setting the initial provisioning control parameters regarding the CONSERT Engine functionality (e.g. default enabled *ContextDerivationRules*, current *Inference Priority Computation* service name, current *observation_window* length). Furthermore, it allows sending control commands such as running an ontology reasoning session for the static contents of the *EntityStore*, or performing clean-up of the context knowledge base from *ContextAssertions* who have surpassed their time-to-live (TTL). Additionally, the `CtxCoord` can also use this interface to set/inspect information about currently enabled *ContextAssertion* updates and *ContextDerivationRules* as well as getting *snapshots* of the current *ContextStore* (i.e. entering a `read` transaction over the context knowledge base) to use during provisioning control rule execution.
Lastly, the `StatsHandler` interface is used by both `CtxQueryHandler` and `CtxCoord` agents to set, respectively retrieve, the statistics of runtime context information usage discussed in Section 5.3.1. Together with the snapshot of the *ContextStore* mentioned above, they constitute the set of possible trigger conditions for the body of a context provisioning control rule.

Apart from the exposed interfaces, the CONSERT Engine interacts with the *Inference Priority Computation Service* and the *Constraint Violation Resolution Service*. These services are provided by the developer and augment the functionality of the engine.
One of the advantages of implementing services respecting the OSGi specification is that running over OSGi-compatible platforms (e.g. Apache Felix[2]) allows for a complete control of the

[1] http://www.osgi.org/Technology/WhatIsOSGi
[2] http://felix.apache.org/

service lifecycle (i.e. install, start, stop, uninstall). Furthermore, within such a platform, mechanisms exist by which services can be tracked, even providing filters for observing only services with certain descriptive properties. The provisioning coordination policies and, therefore, the CONSERT Engine depend on these tracking and filtering mechanisms to obtain access to the specified inference priority computation or constraint resolution services.
Remember from Section 5.2.2 that the value of *inference scheduling service* or various *constraint resolution service* control parameters was given as an identifier (i.e. a string value). These identifiers are in fact properties which are attached to the OSGi-based description of the emphInference Priority Computation Service and the *Constraint Violation Resolution Service* instances by the application developer. The CONSERT Engine (as an OSGi-based service itself) can then use them to track and bind to the specific service implementation, as instructed by the `CtxCoord` agent through the `CommandHandler` interface. Even more, if the application level decides to upgrade the implementation of a currently used inference scheduling or constraint resolution service, the CONSERT Engine will automatically rebind with the new version of the services.

Lastly, use of the same service description and filtering mechanisms can be applied to the CONSERT Engine itself. As we observed in examples from Chapter 6, one physical machine (e.g. Alice's smartphone) may be host to a platform containing several CMUs, each requiring the use of a CONSERT Engine instance. To distinguish between the different service instances, the engine itself has a OSGi-description by which the agents from the CMU to which the engine belongs can identify it. In Section 6.2.2 we mentioned that the *domain identifier* parameter of the *ContextDomain* specification of a deployment policy is used for this exact purpose from the application perspective.

7.3 Context Provisioning Implementation

Having described the implementation of the CONSERT Middleware functionality elements which handle representation and reasoning aspects, we now present the implementation of the context provisioning units: the CMM agents. We discuss both the framework chosen to implement the agents as well as the specifics of their execution environment (the different agent adaptors). The overview of the provisioning architecture can be reviewed in Figure 7.9.

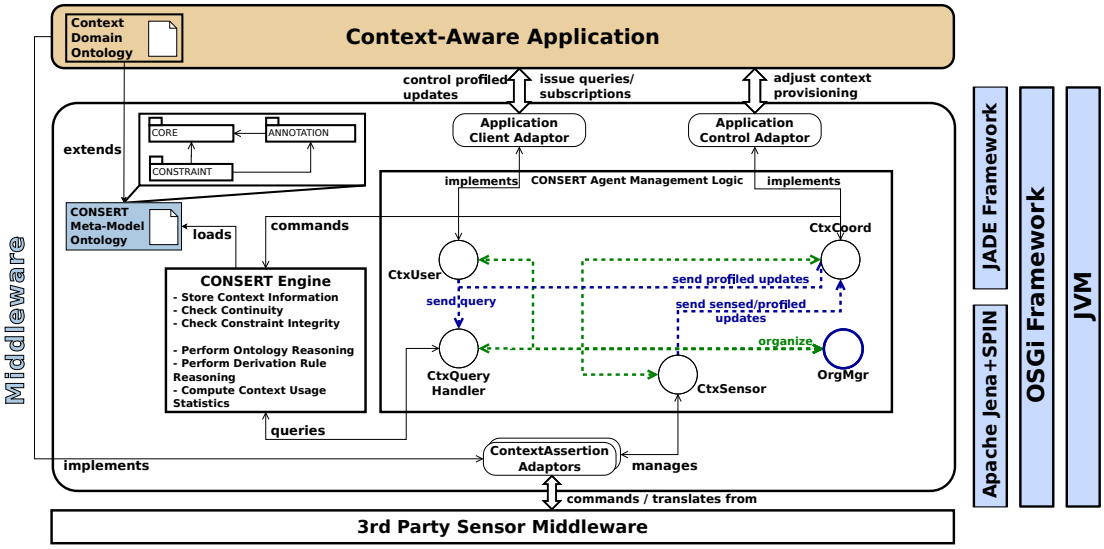

Figure 7.9: CONSERT Middleware: multi-agent architecture and interactions

7.3.1 Provisioning Agent Implementation with JADE

We have chosen JADE[1] as the agent development framework for the multi-agent based architecture of our CMM. JADE is a platform with a fairly long history over the course of which many improvements and add-ons have been brought to the framework. While other agent programming frameworks exist (e.g. JaCaMo[2], 2APL[3], GOAL[4] or CLAIM [El Fallah Seghrouchni and Suna, 2005]), they are either very high-level, BDI (belief, desire, intention) oriented languages (e.g. JaCaMo, 2APL, GOAL) or more simple reactive-based ones (CLAIM). Additionally, these referenced agent programming languages adopt a declarative programming style.

However, we have chosen JADE as our agent development language because of modeling and deployment factors which make it very suitable to the needs of our CMM agents.
To begin with, JADE's behaviour-based conceptualization of agent functionality overlaps well with the provisioning responsibility (e.g. sensing, coordination, dissemination, usage) centric model we made for each agent type of the proposed middleware.
Furthermore, from a deployment point of view, JADE has the advantage of already offering distribution (containers) and communication mechanisms (message transfer protocols - MTPs) as well as organizational utilities (which often lack in other of the referenced agent programming languages).

In the following, we provide an overview of how the JADE framework features discussed above enable the implementation of the CONSERT CMM agent-based provisioning architecture.
We start with the development of the agents themselves. While we do not perform an exhaustive mapping of each internal operation or interactions of an agent to the corresponding JADE Behavior class, we present the general guidelines used for agent implementation:

- use appropriate subclasses of the `AchieveREInitiator` or `AchieveREReceiver` Behaviours for each interaction from the FIPA-Request like family (e.g. FIPA-request, FIPA-query) as well as subclasses of the ProposeInitiator/Responder and Subscription-Initiator/Responder for interactions such as: publishing of *ContextAssertion* update capabilities (by `CtxSensors`), registration of agents with one another (e.g. a `CtxSensor` with a `CtxCoord` or a `CtxUser` with a `CtxQueryHandler`), or subscription requests (from `CtxUser` agents).
- construct subclasses of the `FSMBehaviour` for more involved protocol interaction which are not covered by FIPA standards (e.g. the sequence of interactions following a query/-subscription request of a `CtxUser` for *ContextAssertions* for which the updates or *DerivationRules* that provide them are not active - cf. Section 5.4.2 for more details).
- construct subclasses of `SimpleBehaviour` or `OneShotBehaviour` to handle notifications from the services present in the agent environment (e.g. a `CtxSensor` handles a new *ContextAssertion* update produced by a `ContextAssertion Adaptor`)
- use combinations of `CyclicBehaviour` and `SequentialBehaviour` to implement the non-interactional tasks within the responsibilities table of each agent (e.g. execution of the provisioning control rules by a `CtxCoord` agent at intervals configured by the *observation_window* provisioning control parameter).

Further, looking at provided deployment support, we see that the platform configuration parameters described in Section 6.2.1 (e.g. *container host*, `mtpHost`, `mtpPort`, `platform name`) apply specifically to the usual configuration requirements of a JADE Main Container.
Additionally, every `OrgMgr` agent is a subclass of the DF (Directory Facilitator) agent which

[1] http://jade.tilab.com/
[2] jacamo.sourceforge.net/
[3] http://apapl.sourceforge.net/
[4] http://ii.tudelft.nl/trac/goal

resides within the Main Container of a JADE instance. Thus, as mentioned in the agent responsibility descriptions in Section 5.1.2, the `OrgMgr` can function as a yellow pages service for the agents in its CMU (e.g. a `CtxSensor` agent will ask it about the `CtxCoord` agent address to which it must connect).
Lastly, a JADE platform instance launched within the CONSERT Middleware is configured to use an auxiliary *JadeOSGIBridgeService* that allows the agents to connect to their service based environment. As we explore in the next subsection, the adaptors used by the CMM agents are as well implemented following the OSGi service specification.

7.3.2 Provisioning Agent Adaptor Services

In Section 5.1.3 we explained that the agent environment of a CMU is constituted as a set of services that enable the agents within the CMU to communicate with sensing and application levels, as well as to perform actions for storing and reasoning about the context information they manage. Referring back to Figure 5.1, the reader can observe that these services are implemented based upon OSGi specifications.

In the previous subsection we have already seen how the CONSERT Engine itself is developed as such a service component. Apart from it, we mentioned that `CtxSensor` agents employ `ContextAssertion Adaptors` as interfaces towards the sensing infrastructure from which they must collect the information about the *ContextAssertion* of which they are in charge. We explained previously that these adaptors are implemented by the application developer and an initial issue for a `CtxSensor` agent is knowing which `ContextAssertion Adaptor` implementation corresponds to which *ContextAssertion* acquisition functionality. The solution is based on the OSGi-specific service tracking and filtering mechanisms described earlier in the case of the CONSERT Engine. Each *ContextAssertion Adaptor* is given a service description comprising the *domain identifier* parameter and the URI of the CONSERT Ontology based resource representing the *ContextAssertion* handled by the adaptor. Upon initialization, the `CtxSensor` agent will use the *JadeOSGiBridgeService* to access the tracking mechanisms that allow it to bind to the required adaptors.

The CONSERT Engine is created by the `CtxCoord` agent upon its initialization phase, whilst the `ContextAssertion Adaptors` pre-exist in the agent environment and are searched for by the `CtxSensor` agents. However, in Section 5.1.3 we mentioned that the `CtxCoord` and `CtxUser` agents expose services themselves in the environment, by which the application-level is able to directly interact with these agents (i.e. the `ApplicationUser Adaptor` and `ApplicationControl Adaptor`). In this case, the `CtxCoord` and `CtxUser` agents use the *JadeOSGiBridgeService* to expose themselves as the implementation of their corresponding adaptors. The description of the adaptor services again includes the value of the *domain identifier* parameter which acts to identify a CMU managing provisioning aspects of a *ContextDomain* from the application perspective.
Thus, the concrete way by which a context-aware application interacts with the context management functionality offered by the CONSERT Middleware is by tracking and binding to the `ApplicationUser` and `ApplicationControl` service adaptors.

7.3.3 Context Provisioning Adaptation

Previously we introduced the CONSERT Provisioning Ontology as the means for defining the vocabulary for provisioning control parameters and control rules. In this section we want to extend that discussion by explaining the implementation of the runtime execution of provisioning control actions by the `CtxCoord` agent.

We showed in Section 7.1.3 that the provisioning coordination policy is an RDF document

containing specifications in ontology form. Upon initialization, the `CtxCoord` agent reads this RDF document, retrieves the specified control parameters and creates an index of the control rules. As mentioned in Section 5.2.2, some of the provisioning control parameters concern the updates coming from `CtxSensor` agents (which *ContextAssertions* are enabled, what is their desired update mode). The rest of the parameters target the functionality of the CONSERT Engine. Therefore, in order to set these parameters, the `CtxCoord` agent will access the `CommandHandler` interface of the CONSERT Engine.
When creating the provisioning control rule index, the `CtxCoord` will also establish their partitioning into ordered execution groups. We mentioned in Section 7.1.3 that an instance of subclasses of a `CommandRuleTemplate` are attached to the `ControlPolicy` OWL class in the coordination policy RDF model using sub-properties of the `hasCommandRule` OWL property. All control rule instances attached using the same `hasCommandRule` sub-property form an execution group.
To establish the order of the execution groups, we rely on a facility of the SPIN specification which defines an OWL property called `spin:nextRuleProperty`. All (or some) of the sub-properties of `hasCommandRule` can be related to one another via this property. The order of the execution groups is then found by performing a topological sort of the graph formed by the sub-properties and their *next rule* relations.

After setup, the coordinator agent starts a cyclic behavior which runs once every *observation_window* time periods. Within this behavior the following actions are performed. The `CtxCoord` agent uses the `StatsHandler` and `CommandHandler` interfaces of the CONSERT Engine to retrieve the context usage statistics and the current snapshot of the *ContextStore*. It then combines the two into a single RDF dataset which will be used as knowledge base by the SPARQL CONSTRUCT queries of the provisioning control rules.
The agent then inspects the control rule index and retrieves the set of execution groups, in order. The `CtxCoord` then starts executing the rules within the current execution group and stores their outcome in a buffer. Every time the outcome of a control rule from a later execution group, overrides the prescription of a rule from a previous group, the previous outcome is replaced by the new one in the buffer. After executing all groups, the resulting control rule prescriptions from the buffer will thus constitute a consistent set of provisioning adaptation instructions.
The last step consists in applying these instructions either by means of `TaskingCommands` addressed to `CtxSensor` agents, either by using the `CommandHandler` interface to amend the value of provisioning control parameters specific to the CONSERT Engine (cf. Section 5.3.2).

7.4 Context Provisioning Deployment Implementation

The last implementation aspect we discuss is related to deployment functionality. Specifically, we detail the packaging of configuration information regarding the different platform, *ContextDomain* and CMU agent specifications which have been detailed in Sections 6.2 and 7.1.4. Furthermore, we explain the mechanism by which the application-level uses the CONSERT Middleware to control the lifecycle of deployed CMUs.

7.4.1 Deployment Specification Files

Throughout this chapter we have seen that both information describing the context model of an application, as well as the contents of provisioning and deployment policies are specified using ontologies and thus take the form of RDF datasets. In the following we want to explain how the CONSERT Middleware partitions and packages these RDF models into specific files which have a clear purpose and thus alleviate the effort concerning middleware deployment.

We start by presenting the file setup for the contents of the deployment policies themselves,

which the reader can observe in Figure 7.10. One can observe the existence of two files:

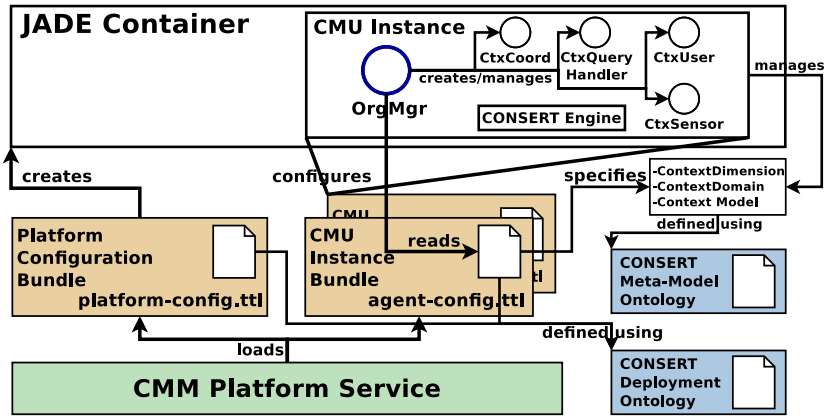

Figure 7.10: CONSERT Middleware Deployment Engineering

platform-config.ttl and **agent-config.ttl**. The first one contains all specifications (in ontology form) regarding the configuration of the platform (i.e. JADE Container) on which one or more CMUs will run on the given physical machine. The second one contains the *ContextDomain* configurations and agent specifications of a given CMU. That is, the **agent-config.ttl** file keeps the information related to the agent composition of the CMU, which links directly to the desired set of context provisioning aspects (e.g. acquisition, coordination, dissemination, usage) that the CMU needs to manage. Furthermore, it holds the specifications of the *ContextDomain* to which the configured CMU is assigned.
The names of these files are pre-defined in the CONSERT Middleware, such that the CMU agents that will inspect them will always know how to find the required configuration information.

A noteworthy observation is that these files are themselves packaged into OSGi Bundles. Thus, we gradually come to the conclusion that the software-level deployment of a CONSERT Middleware based application is equivalent to setting up an OSGi-compatible runtime (e.g. Apache Felix) and loading the required bundles into this platform.
The **platform-config.ttl** file is packaged into a bundle of its own, called the `cmm-instance-default` bundle. Though not depicted as such in Figure 7.10 (for purpose of clarity) this bundle can actually also contain (but is not required to) an instance of the **agent-config.ttl** file, thereby making it the bundle holding the configuration information for a default CMU, deployed every time a CMM platform is launched on a physical machine. An example of this is Alice's bootstrap CMU in the extended reference scenario from Section 6.1.1 (cf. Figure 6.1). Then, for each CMU that will need to be deployed at some point in the application's lifetime, a bundle containing the corresponding **agent-config.ttl** file is created. Such bundles maintain three important properties in their header section: the *domain identifier* value, the *ContextDimension* URI and the *Domain Range Value* URI. These values provide the coordinates which uniquely identify the *ContextDomain* to which the CMU is assigned. Furthermore, as we will explore in the next subsection, these values allow tracking of the different CMU configuration bundles that are included to run on a given CMM platform.
Apart from the **agent-config.ttl** file, the CMU bundle may contain other configuration files, depending on the composition of the CMU. As one could observe in the examples on the ontology form of deployment policy configurations from Section 7.1.4, the indication of sensing or coordination policy specifications, as well as context model make up are given as paths or URIs linking to RDF documents (e.g. `etc/cmm/coordconfig.ttl`, `etc/cmm/coordconfig.ttl`, `http://purl.org/net/amilab/core`). For file links that are given in URI form, a special file called the **domain-ontology-policy.rdf** file contains Apache Jena specific content which link the base URI of an ontology file (e.g. `http://purl.org`

`/net/amilab/core`) to a concrete RDF file (e.g. `etc/model/amilab-core.rdf`).
These files, together with the corresponding directory structure are included within a CMU bundle. The advantage is that each bundle provides its own class loader, such that the agents composing a CMU will only see the configuration files included in that CMU, thereby avoiding any naming confusion.

In the subsection that follows, we detail the mechanism by which the lifecycle of the configuration file packaging shown here is maintained at runtime.

7.4.2 Runtime Deployment Management

The packaging of all CMU configuration information into bundles helps us achieve the deployment flexibility objectives mentioned in Section 3.3. OSGI-based mechanisms explained previously allow us to create functionality that keeps track of all available/potential CMUs as well as their deployment state.
In the CONSERT Middleware, this happens via the `CMMPlatformService` (cf. Figure 7.10). As mentioned earlier, application development using our CMM resolves to loading the required bundles in an OSGi-based framework. The implementation (the code) of the CMM agents is itself packaged as a bundle, which will therefore be always present in a deployment on a given physical machine (e.g. PC, smartphone).
The `CMMPlatformService` is contained within this agent code bundle and will start automatically when the bundle is loaded in the given context-aware application. Its first task is to locate the bundle containing the **`platform-config.ttl`** file so as to create the CONSERT Middleware platform for that particular machine, following the configurations in the file.
After creating the platform, the service will open a tracker that searches for all bundles that contain a CMU configuration. An index is created which identifies each bundle by the three header properties mentioned in the previous subsection (*domain identifier*, *ContextDimension* URI, *Domain Range Value* URI).
The `CMMPlatformService` then provides an interface by which the application level can control the state of a CMU. CMUs can be installed (i.e. the CMU agents are created), started (i.e. the CMU agents start their execution cycle), stopped (i.e. CMU agents stop their execution cycle) or uninstalled (CMU is destroyed and resources are freed).
Furthermore, for mobile computational nodes, remember that in Section 6.4.5 we presented two possible approaches for management of their mobility on hand of the current facilitation given by our CMM. Both approaches involved in the end the sending of notifications for entering and leaving a *ContextDomain*. These notifications contain the information required to identify a *ContextDomain*. This information corresponds exactly to the header properties of the CMU bundle which would be required on the given physical machine to execute the intended provisioning aspect (e.g. sensing, usage) within that *ContextDomain*. Thus, the notifications are forwarded to the application-level which can use the `CMMPlatformService` to launch, (re)start, stop or destroy the required CMU, as explained above.

Once the application has requested the installation of a CMU, the `CMMPlatformService` identifies the necessary bundle holding its configuration, creates the `OrgMgr` agent responsible for that CMU and informs it of the means of accessing the configuration files within the bundle. From here on, the control is transferred to the `OrgMgr` agent which performs the initialization steps detailed in Section 6.3.3, thus deploying the CMU. As additional explanation, we mention the fact that upon creation of each agent whose specification lies in the **`agent-config.ttl`** file, the `OrgMgr` will supply the newly created agent with the reference to the bundle containing the CMU configuration files. Thus, for example, the `CtxCoord` or `CtxSensor` agents will be able to access the required coordination and sensing policy specification respectively.
All other CMU lifecycle operations (start, stop, uninstall) are all performed having the `OrgMgr` as a facilitator of the operation. After an uninstall request, the `OrgMgr` itself is destroyed.

From the above the reader can observe that the CONSERT Middleware does not only allow expressing the deployment and provisioning configuration in a declarative way, but it furthermore provides application developers with the ability to control it at runtime, thereby increasing the flexibility of our approach. We discuss this and other software development related advantages in the next and final section of this chapter.

7.5 Discussion

In this chapter we have reported on the frameworks and software design choices which help us implement the functionality of the CONSERT Middleware as it was presented in Chapters 4, 5 and 6.
The different modules of the CONSERT ontology (core, annotation, constraint), as well as the ontologies that implement the provisioning and deployment specification vocabularies are freely available for use and can be accessed at the links provided in the footnotes from Sections 4.2.1, 4.2.2, 4.2.3, 7.1.3 and 7.1.4 respectively.
The code of the CONSERT Middleware itself comes packaged into four OSGi bundles (JAR archives). The **`jena-bundle.jar`** bundle plays an auxiliary role as a wrapper over the JENA and SPIN APIs that makes them accessible in OSGi-based frameworks.
The **`consert-model.jar`** bundle contains the packages that enable manipulation of context model constructs (e.g. *ContextAssertion*, *ContextAnnotation*, *ContextConstraintViolation*) at application runtime. The component-based implementation of the CONSERT Engine is contained in the **`consert-engine.jar`** bundle, while the **`consert-middleware`** bundle includes the implementation of the multi-agent based provisioning logic and deployment specific APIs.
Since development and documentation of the CONSERT Middleware are still at an early stage, the specified bundles and more detailed descriptions of their utilization within an application are currently available only upon request.

Apart from implementing the functionality we explained that an important goal set out for this thesis was the idea of alleviation and uniformization of context-aware application development effort based on our CMM. In this section we want to perform an analysis of this support for *engineering* context-aware applications offered by the CONSERT Middleware. In doing so we will look at how we address some of to relevant attributes of software development.

Component-Based Design One of the strong feature of CONSERT Middleware is that it promotes a component-based design, both internally and at application level. Internally, both CONSERT Engine and context management agents are software units that encapsulate control-flow, providing a clearly defined context provisioning process. However, relevant aspects of this process (e.g. context model, sensing update modes, coordination policies) are entirely specifiable by the application. In this sense, CONSERT promotes **reuse by design**, as the agents and CONSERT Engine can be reused within many context-aware applications with different context management needs.
A CONSERT Middleware instance provides however key **variation-points** by means of **service implementations** (most notably at reasoning level) that allow an application to adapt and extend the context management process. Specifically, the application can implement custom constraint resolution and inference scheduling services and configure their usage in the CONSERT Engine either through the `ApplicationControlAdaptor` or through **declarative configurations** which we discuss next.

Configuration-Based Development CONSERT defines configuration options for almost all context management aspects it supports. It uses semantic web technology to create vocab-

ulary for **policies** that determine the deployment of context management units and control of the context provisioning process within a deployed context management group (sensing and coordination policies). Parameters controlling update modes, enabled inference rules, current inference scheduling services etc., are configurable at initialization and changeable at runtime by the application layer, leading to a very customizable context management process. As we will see in the evaluation chapter, our experience in developing the AmI-Lab simulation of the reference scenario shows that declarative configuration options significantly lower programming effort. For example, each CMU in the simulation (AmI-Lab management, Alice and Dan's smartphones, etc) requires the definition of a single *agent-config.ttl* file to specify both required agent types, connectivity information and context model elements. The ontology-based vocabulary means that these files can be edited using readily available semantic web IDEs (e.g. Protege, TopBraid) alleviating development effort even further.

Flexible Provisioning A last strong feature we want to point out is the availability of clear application structuring elements (*ContextDimensions* and *ContextDomains*) which depend on the application context model itself. The fact that a CMU lies in a 1-to-1 mapping with a *ContextDomain* and that its configuration is packaged as an OSGi bundle means that its lifecycle can be directly tied to dynamic use of the context information within that *ContextDomain*. This mapping promotes **separation of concerns** and makes the CMU a unit of **control encapsulation**. This can furthermore be exploited as an extensibility mechanism for context management. It means that an application can define several CMU configurations that manage, provide or consume information from the same *ContextDomain*. Each configuration will however specify different provisioning instructions for the agents of the CMU using declarative policies and custom services as explained in the previous two paragraphs. The application can then use the CMU lifecycle management support to dynamically choose among defined CMUs that perform the same management but in different ways. For example, to reduce energy consumption when events are less dynamic in the AmI-Lab, a managing CMU that configures `CtxSensors` with lower update times and a `CtxCoord` with less complex inference rules can be dynamically swapped instead of a more demanding configuration.

Chapter 8

Practice and Experimentation

The previous chapter gave an overview of the implementation of the architecture and functionality of the CONSERT context management middleware.
As seen throughout the chapters of this thesis, we have used the scenario introduced in Section 1.3 as a reference for all the given explanations and examples. The evaluation of the concrete CONSERT middleware implementation is based on this same scenario and in the following we report on the obtained results and the experience of *programming* and *working* with the CONSERT Middleware (as relevant indications of its support for context-aware application engineering).

In Section 8.1 we describe the evaluation process that we have conducted. We discuss the conception of the application handling the simulation of the reference scenario and what the objective is for the tests that measure various performance aspects of the CONSERT Middleware.
Section 8.2 then elaborates on how our CMM is used to model and provision the context information required by the reference scenario.
Setup, execution and analysis of the tests measuring the performance of the CONSERT Middleware is done in Section 8.3.
The chapter concludes with a discussion on overall evaluation results and the experience of developing with the CONSERT Middleware in Section 8.4.

8.1 Evaluation Considerations

The purpose of our evaluation is to show that the CONSERT Middleware can be successfully and effectively used to develop and control the context management aspects of an AmI application. We therefore follow two types of analysis: a *qualitative* one and a *quantitative* one.
In order to perform them, we have set up the evaluation system shown from a global perspective in Figure 8.1. We will refer to different aspects of this setup throughout this chapter.

In what follows we first detail the individual objectives of each analysis. Then we reiterate through our reference scenario and present the functionality of the application that implements it with the purpose of performing our desired evaluation. Lastly, we present the framework which allows us to simulate the sensing environment related events of the described scenario.

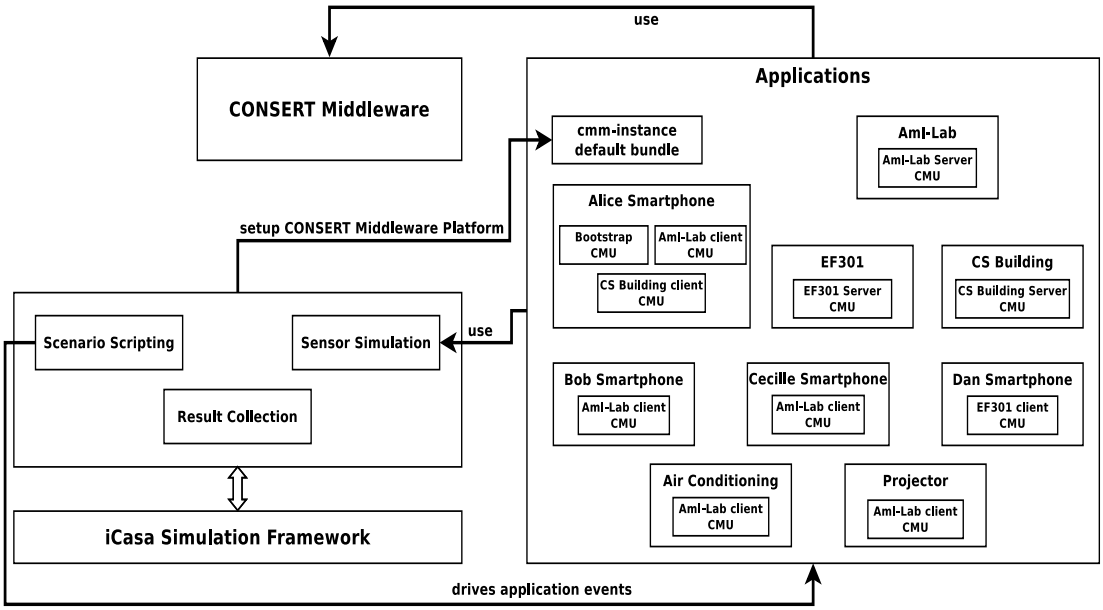

Figure 8.1: Global overview of the evaluation setup. iCasa Framework is used to simulate sensor functionality and provide scenario scripting. Applications use the CONSERT Middleware (specific CMUs) to respond to scenario events.

8.1.1 Evaluation Objectives

Qualitative Analysis From this point of view we are interested in seeing that the contributions introduced in Chapters 4, 5 and 6 offer the necessary support and advantage for development of context-aware applications.
With regard to modeling, we examine how the CONSERT Ontology provides suitable flexibility for representing context within the reference scenario. We look, for example, at how the ability to define and resolve constraint dependencies of context information is of a particular advantage in this case. From a reasoning perspective, we wish to observe the correct behavior of *ContextDerivationRules*, as well as the overall ease or difficulty of writing the rules themselves.
With respect to context provisioning, we analyze how the provided parametrization and control of the provisioning process create an advantage for the application developer in terms of computational resource usage.
Finally, from the point of view of middleware deployment we investigate how implementation of the application handling the reference scenario is facilitated by the proposed structuring and configuration options offered by the CONSERT Middleware.

Quantitative Analysis Given our focus on application engineering aspects that were laid out as objectives of this thesis, we want to make sure that the proposed implementation described in Chapter 7 offers an adequate computing performance.
In our evaluation we focuse on two performance aspects: the processing throughput of the CONSERT Engine and the query answering and routing functionality of the CMU agents (`CtxQueryHandlers` and `OrgMgrs` more particularly).
The CONSERT Engine performance analysis relates to the behavior of the engine under different loads of sensor updates and inference requests and will be discussed in detail in Section 8.3.2.
A theoretical overview of domain-based query routing functionality (in terms of exchanged messages and complexity analysis) has already been given in Section 6.4.2. In the performance tests detailed in Section 8.3.4 we simply attempt to put this analysis into numbers and examine the

different query response latencies, given different CMU deployment configurations and query targets.
In essence, the purpose of these types of test is to establish an initial benchmark of the performance capabilities of the CONSERT Middleware, against which all improvements proposed in future work will be measured.

8.1.2 Scenario Implementation

In order to perform the qualitative tests described earlier in a proper manner, we have split the reference scenario into two interaction episodes. We implement corresponding application code (as shown in the global view from Figure 8.1) for each distinct episode.

The first episode concerns the events happening while Alice is waiting for her friends to come join her in the AmI laboratory after having attended her CS Lecture. As mentioned in the scenario description, the lecture finishes early, so the smart application on Alice's smartphone has to determine that she is in fact no longer busy and can receive the call of her friend Bob, who wants to ask her where she is. As we will see in Section 8.2, we model this as a uniqueness constraint set on Alice's availability status which can be broken given the contradictory information of her personal schedule and the presence information within the AmI Laboratory.
Thus, this episode tests two important functionality aspects of the CONSERT Middleware. The first one is related to the ability of performing a more involved context prosuming activity, as was outlined in the exemplification from Section 6.4.4. This is required since, in order to trigger and resolve the availability status constraint, Alice's smartphone CMU needs to locally process information obtained from the CMU running on the AmI-Lab management server. The other important functionality aspect is that of the constraint management itself. The episode allows us to investigate if the constraint is correctly and timely recognized and solved in order to determine that Alice is in fact free while she is waiting for her friends, even though her personal calendar says otherwise.

The second episode regards the events happening during and after the ad hoc discussion in the AmI-Lab. In this case, we are interested in seeing that the context derivation capabilities of the CONSERT Engine are able to infer the ad hoc situation. On the other hand, the CONSERT Engine instances on the user smartphones, using the behavior of `CtxUser` agents in their CMU, have to determine that their users are busy (being in an ad hoc meeting). This status has to be broadcasted again at laboratory level, for the duration of their stay in the room.
In this way, the domain-based query functionality can also be tested, since the scenario specifies that Dan (who is in room EF301) sends out a query for Alice's availability status in order to find out if she can come join him for lunch. Thus the aspects of correct and timely domain-based routing are also qualitatively evaluated in this scenario episode.

8.1.3 Scenario Simulation Framework

A first thing to note is that the *simulation* only refers to the *sensing environment* (cf. Figure 8.1). That is, the physical spatial layout of the CS Building and the laboratory rooms within it, the placement of the sensors within the rooms and their sensing activity, as well as the overall *scripting* of the scenario events as they were described previously is performed using a simulation framework. The actual processing of context information is performed using live instances of CMUs developed using the CONSERT Middleware, such that its *qualitative* usage can be directly evaluated.

Simulation Framework To simulate the scenario environment and the actions of persons within this environment we use a simulation framework called iCasa[1]. ICasa is an execution platform on top of OSGi for digital home applications. It has been developed in the context of the Medical project by the ADELE Research Group and the Orange Labs. The very purpose of iCasa is to provide developers of pervasive applications with a simulated environment enabling complete control of the environment and time.
Given the CONSERT Middleware implementation details outlined in Chapter 7, the fact that iCasa is developed on top of OSGi makes it a suitable candidate for easy integration with our CMM. The scenario scripting features are another reason for choosing this framework.

Environment Layout The framework offers a browser-based GUI to define the layout of the simulated environment. In particular, our reference scenario includes three main spatial structures: there is the CS Building which includes the AmI Laboratory (where most of the scenario events take place) and the EF301 office from which Dan sends the queries for Alice's availability status.
Figure 8.2 shows a snapshot of the simulation GUI where the layout of the environment and the sensor placement is visible[2]. Notice that the AmI Laboratory has been further divided into individual room sections which correspond to areas where, for example, people in an ad hoc meeting can be recognized as sitting at the same desk.
Within each individual area of the AmI-Lab there is an instance of a presence sensor which works by detecting the bluetooth MAC address of user smartphones. The desks near the laboratory "windows" (PresenterArea, Section1_Right – Section3_Right) are equipped with luminosity sensors. The four corners of the room hold temperature sensors. Apart from this, all laboratory desks are equipped with an instance of microphone and body posture detection sensors (i.e. Kinect cameras). In total, this makes for a number of 29 simulated sensor instances.
The iCasa platform already provides implementations for the functionality of luminosity and

[1] http://adeleresearchgroup.github.io/iCasa/
[2] Sensor configuration is specified automatically in the scenario script, causing the unordered visual lineup noticeable in the figure

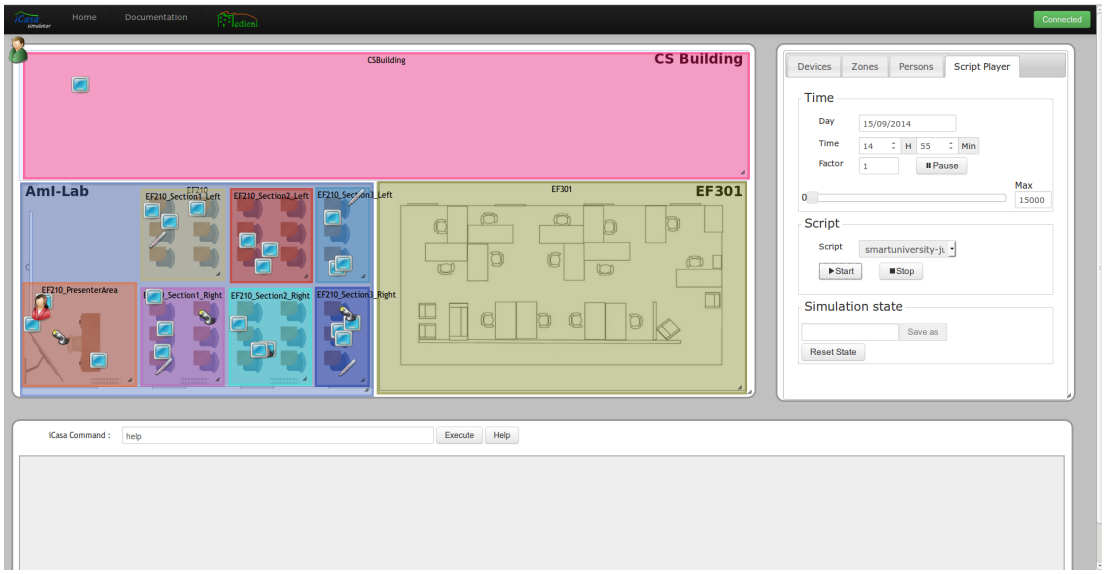

Figure 8.2: A snapshot of the iCasa Simulation GUI showing the modeling of the layout of physical spatial structures in the reference scenario, as well as sensor placement within the AmI-Lab. The AmI-Lab is divided further into individual room areas (i.e. desks).

temperature sensors. However, the functionality of the presence, microphone (noise level) and body posture sensors needed to be customly implemented.

Scenario Scripting As mentioned, another useful facility of the iCasa framework is the ability to *script* the events taking place in the scenario. iCasa defines by default a set of commands that offer the ability to create spatial areas, devices and users, move users and devices around, as well as setting the value of different environment variables (e.g. temperature, light level, noise level) which the sensors use to provide their readings. Furthermore, the framework allows easy development of custom commands, which in our case were used to perform initializations (e.g. set up a CONSERT Middleware platform instance, start the relevant CMUs) and measurement collection (for performance analysis). Figure 8.3 shows sample excerpts from the XML-based script language used to specify the scenario events. Commands such as `delay`, `move-person-zone` or `add-zone-variable` are provided by default by the iCasa framework. On the other hand commands such as `start-amilab` or `start-user` are custom implementations that serve our scenario explicitly.
A script such as the one in Figure 8.3 is defined for each of the two interaction episodes mentioned in Section 8.1.2.

```
1  <!-- ======== Person Section ======== -->
2  <create-person id="Alice" type="Woman" />
3  <create-person id="Bob" type="Man" />
4
5  <!-- ======== Init CMM Platform Section ======== -->
6  <start-cmm-platform />
7  <delay value="5" unit="s" />
8
9  <!-- ======== Init CMUs Section ======== -->
10 <start-amilab />
11 <delay value="5" unit="s" />
12
13 <start-alice-personal />
14 <delay value="5" unit="s" />
15
16 <!-- ======== Play Scenario Section ======== -->
17 <start-user userName="Alice" />
18 <move-person-zone personId="Alice" zoneId="EF210_Section1_Right" />
19 <move-person-zone personId="Bob" zoneId="EF210_Section1_Right" />
20
21 <modify-zone-variable zoneId="EF210_Section1_Right" variable="
      NoiseLevel" value="75" />
22 <add-zone-variable zoneId="EF210_Section1_Right" variable="skel-
      position-Alice" />
23 <modify-zone-variable zoneId="EF210_Section1_Right" variable="skel-
      position-Alice" value="SkeletonSitting" />
24 ...
```

Figure 8.3: Excerpt from the scripting of the reference scenario events.

8.2 Scenario Evaluation

In the following we describe the details of the two qualitative analysis episodes mentioned in the previous section. We evaluate the way in which the different context modeling, reasoning, provisioning and deployment contributions introduced throughout this work help address the context management requirements of the reference scenario.

8.2.1 Context Modeling Evaluation

We start with showing how the context information from reference scenario is modeled using the CONSERT Ontology. Our analysis reflects the episode-based partitioning of the scenario and will present extracts of the modeling relevant for the pre-meeting episode (corresponding to the context model used in the bootstrap CMU on Alice's smartphone) and the ad hoc meeting episode (the context model used in the `AmI-Lab` *ContextDomain* on the management server running the corresponding coordination CMU). Our goal is to inspect the suitability and expressiveness advantages of the CONSERT Ontology for the purpose of modeling the reference scenario simulation.

AmI-Lab Context Model We begin with the ad hoc meeting episode and the context model for the `AmI-Lab` *ContextDomain*, of which an excerpt can be observed in Figure 8.4.

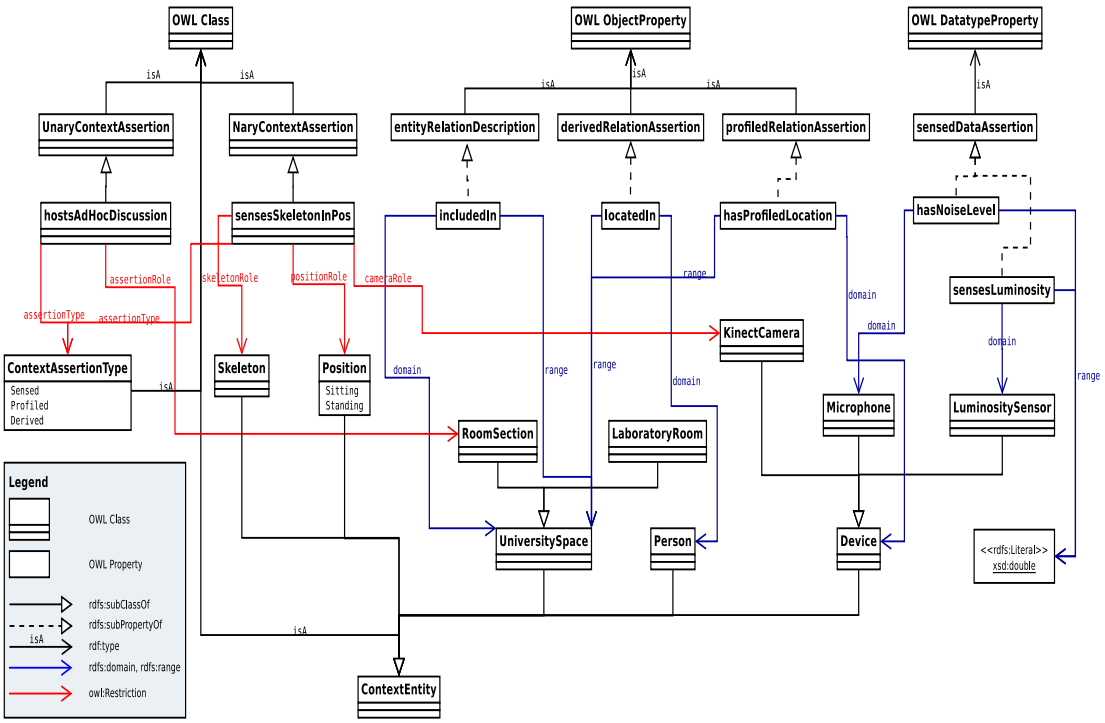

Figure 8.4: Excerpt of the context model for the AmI-Lab *ContextDomain*, created using the CONSERT Ontology.

The figure shows an example for each of the modeling constructs available in the core module of the CONSERT Ontology. The `hasProfiledLocation` binary *ContextAssertion* captures the positioning of fixed devices in the scenario (e.g. microphones, Kinect cameras, bluetooth detectors). Since these positions are known in advance, this represents a case for a *profiled*

ContextAssertion inserted at simulation initialization.
The example chosen for the *EntityDescription* construct (`includedIn`) is in fact the property that generates the *ContextDomain* hierarchy for this simulation (see also Sections 6.1.2 and 8.2.4).
On important thing to note is the modeling of the information that a person is sitting or standing as a *NaryContextAssertion* (`sensesSkeletonInPos`). The existence of this modeling capability brings several advantages. For one, the information about skeletons detected by the Kinect camera sensors is used for the sole purpose of determining their posture, as a condition for detecting ad hoc meetings. Therefore, the fact that the information about skeleton detection and skeleton position can be compacted into a single statement helps obtain a more focused model design.
Furthermore, in Section 8.2.2 we examine the *ContextDerivationRule* which infers the existence of an ad hoc discussion in a room section (desk) of the AmI laboratory. One can observe there that the detection is conditioned on the precision of the posture detection. This shows another advantage of having arbitrary arity *ContextAssertions*, namely the fact that such statements of arbitrary length can be characterized by required *ContextAnnotations* (in our case, validity and certainty are highly relevant; see Figure 8.2).
Though not shown in the diagram from Figure 8.4, for each *sensed ContextAssertion* instance (e.g. instances of `hasNoiseLevel` or `sensesLuminosity`) the simulation generates values of the common *ContextAnnotations* discussed in Section 4.2.2.

Alice Personal Context Model In the case of the pre-meeting episode, the relevant modeling concerns the information that leads the simulated application on Alice's smartphone to deduce her availability status. The noteworthy extract of the context model is shown in Figure 8.5.

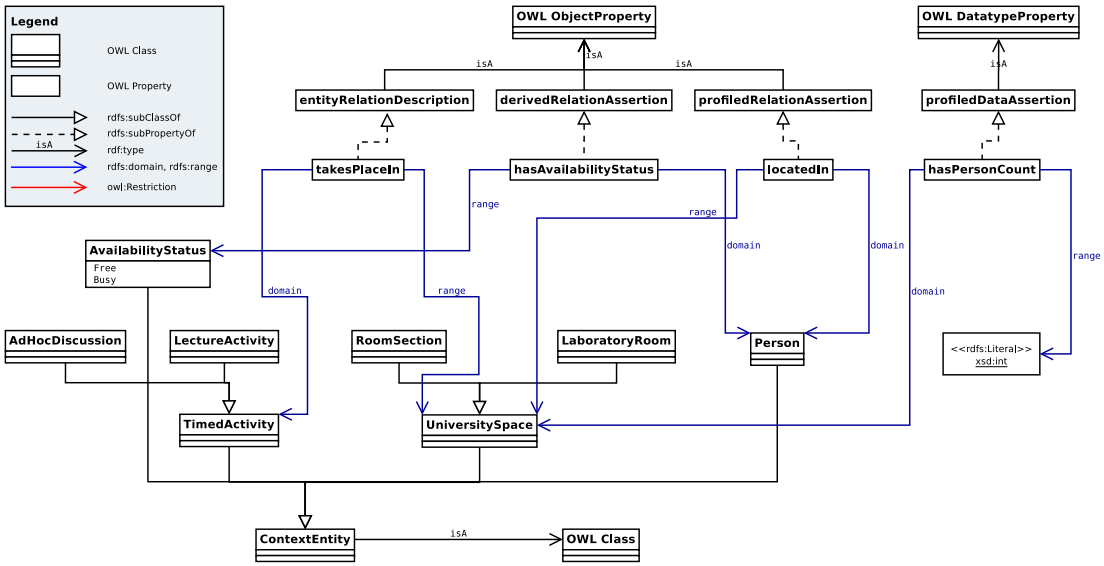

Figure 8.5: Excerpt of the context model for Alice's bootstrap CMU, created using the CONSERT Ontology.

Notice that in this case the `locatedIn` and `hasPersonCount` *ContextAssertions* are modeled as having a *profiled* acquisition type (as opposed, for example, to `locatedIn` being a *derived* assertion within the AmI-Lab context model). This stems from the fact that the instances of these *ContextAssertions* are obtained based on responses to subscriptions sent by a `CtxUser` from Alice's smartphone to the `CtxQueryHandler` agent running in the AmI-Lab management

server (more details in Section 8.2.4).

The figures above show that the CONSERT Ontology is flexible enough to focus in on all the content representation elements required by the reference scenario simulation.
In the pre-meeting episode, however, we also encounter the case of contradictory information (actual availability of Alice while she is waiting for her friends to arrive) that needs to be sorted out. Specifically, the application running on Alice's smartphone needs to detect that she is actually free to receive a call from Bob (since the lecture finished earlier), even though her calendar shows that the CS Lecture is normally still ongoing.
To handle this problem, we model the contradiction as a *context uniqueness constraint* on the `hasAvailabilityStatus` *ContextAssertion* stating that a person cannot be deemed *free* and *busy* at the same time.
Figure 8.6 shows how in the context model for Alice's personal context management an instance of the `AvailabilityStatusConstraint` context constraint template is attached to the `hasAvailabilityStatus` *ContextAssertion* such as to detect uniqueness violations. Further, remember from Section 5.2.2 that one type of provisioning control parameters was the one concerning specifications resolution services for *ContextAssertion*-specific constraints. In this case we create the `ResolveAvailabilityConflict` resolution service (discussed more in Section 8.2.2) and specify that this service is used to resolve uniqueness constraints for the `hasAvailabilityStatus` *ContextAssertion* (lower part of Figure 8.6).

```
sim:hasAvailabilityStatus
  spin:contextconstraint  [
    rdf:type  :AvailabilityStatusConstraint ;
    arg:anchorResource  person:Person
  ] .
:AvailabilityStatusConstraint
    rdf:type constraint:ContextConstraintTemplate ;
    rdfs:subClassOf constraint:UniquenessConstraintTemplates .


:ProvisionPolicy_AlicePersonal
  coordconf:hasSpecificUniquenessConstraintResolution [
  rdf:type coordconf:AssertionSpecificConstraintResolutionType ;
  coordconf:forContextAssertion sim:hasAvailabilityStatus ;
  coordconf:hasParameterValue :ResolveAvailabilityConflict;
  ] ;
```

Figure 8.6: Snippet showing how an instance of the `AvailabilityStatusConstraint` template is attached to the `hasAvailabilityStatus` *ContextAssertion* (up) and how a custom resolution service is setup to handle detected uniqueness constraint violations, as part of the provisioning control policy (down).

This example is again a validation of the ability of the CONSERT meta-model to capture context information dependencies using and ensure flexible resolution capabilities for detected context information inconsistencies. This distinguishes the CONSERT meta-model from most of the approaches reviewed in Chapter 2. The context meta-model based systems presented in Section 2.1.4 are capable of expressing the content and annotation properties showcased in the reference scenario (e.g. [Strang et al., 2003], [Fuchs et al., 2005]). However, only few of them (e.g. CML [Henricksen et al., 2005b]) are able to represent information dependencies, while still not providing methods to *resolve* detected conflicts.
The reviewed work that comes closest to our constraint handling functionality is [Bikakis and Antoniou, 2010], which proposes using defeasible logic rules to drive context reasoning. However, the work fails to consider context meta-properties such as timestamps or quality-of-context related information.

In the next Section we continue the exploration of the expressiveness and information depen-

dency modeling capabilities of the CONSERT meta-model by looking at reasoning performed during both scenario episodes.

8.2.2 Reasoning Evaluation

The need for reasoning by running *ContextDerivationRules* exists in both interaction episodes that form the reference scenario. Both before and during the ad hoc discussion phase, *DerivationRules* are required in order to establish the location of people in the AmI laboratory based on the presence of their smartphones (which have a bluetooth MAC address as shown in the previous section). Furthermore, in the pre-meeting episode *DerivationRules* running on Alice's own smartphone are used to infer her availability status. During the discussion phase, an additional rule is used to derive the actual fact that an ad hoc discussion is taking place in the AmI laboratory (more precisely, at the desk where the three friends are sitting).
In what follows, we explore the SPARQL form of two rules (one from the pre-meeting episode and the one deriving the ad hoc meeting situation) and observe their expected effect in console outputs from the simulation logs.

Pre-Meeting Episode

Before the meeting starts, Alice is waiting for her friends to arrive in the AmI laboratory. On her smartphone, the CONSERT Engine instance from her bootstrap CMU (confer Figure 6.1) there are two *ContextDerivationRules* that infer her current availability status. A first rule derives the fact that she is *busy* based on her calendar which specifies that she is engaged in a `TeachingActivity` from 10:00 to 12:00. On the other hand, since the class has finished early, a second *DerivationRule* triggers based on information obtained from the CMU of the AmI-Lab management server (count of people in the same room as Alice) and deduces that Alice is *free* since she is alone in the AmI laboratory.

```
1  CONSTRUCT {
2      ...
3      ?person smartclassroom:hasAvailabilityStatus smartclassroom:Busy .
4      _:b1 a contextannotation:DatetimeTimestamp .
5      _:b1 contextannotation:hasStructuredValue ?time .
6      _:b2 a contextannotation:TemporalValidity .
7      _:b2 contextannotation:hasStructuredValue ?validity .
8      _:b3 a contextannotation:NumericValueCertainty .
9      _:b3 contextannotation:hasStructuredValue ?certainty .
10     _:b4 a contextannotation:SourceAnnotation .
11     _:b4 contextannotation:hasUnstructuredValue ?src .
12 }
13 WHERE {
14     GRAPH ?personEngagedIn {
15         ?person smartclassroom:engagedIn ?activity .
16     } .
17     GRAPH ?personLoc {
18         ?person person:locatedIn ?RL .
19     } .
20     GRAPH <http://pervasive.semanticweb.org/ont/2004/06/person/locatedInStore> {
21         ?personLoc contextannotation:hasCertainty ?certAnnPersonLoc .
22         ?certAnnPersonLoc contextannotation:hasStructuredValue ?certaintyPersonLoc .
23     } .
24     GRAPH <http://pervasive.semanticweb.org/ont/2014/05/consert/entityStore> {
25         ?activity smartclassroom:takesPlaceIn ?RL .
26         ?activity tme:from ?start .
27         ?activity tme:to ?end .
28         ?start tme:at ?startTime .
29         ?end tme:at ?endTime .
30     } .
```

```
31      BIND (functions:now() AS ?now) .
32      BIND (functions:datetimeDelay(?now, -2) AS ?earlier) .
33      FILTER (((?certaintyPersonLoc >= 0.8) && (?now > ?startTime)) && (?now < ?endTime
            )) .
34      BIND (?certaintyPersonLoc AS ?certainty) .
35      BIND (?now AS ?time) .
36      BIND (functions:makeValidityInterval(?startTime, ?endTime) AS ?validity) .
37      BIND (functions:getCurrentAgent() AS ?src) .
38  }
```

Listing 8.1: SPARQL query of the `BusyLectureRule` *ContextDerivationRule* template

The SPARQL query of the `BusyLectureRule` template is shown in Listing 8.1. In the `CONSTRUCT` clause (not shown entirely - marked with ... - to reduce space) one can observe the assertion of the statement that the given person is *busy*. Underneath the *ContextAssertion* statement, all the derived annotations related to the statement are asserted. In the `WHERE` clause, the named graph based identification of *ContextAssertion* and *ContextAnnotation* instances explained in Section 7.1.1 can be observed. In this case, the rule examines the location of a person (Alice) and her personal activity calendar (`smartclassroom:engagedIn`) to determine if the person is currently located in the place that hosts the activity in which she is engaged. The details of the activity are modeled as *EntityDescriptions*, as shown also in the modeling from Section 8.2.1. This rule fires at the beginning of the simulation. However, soon after, the `UserAloneRule` *ContextDerivationRule* will be triggered by the fact that within the room where the lecture is supposed to take place, there is currently only one person (Alice). The `CONSTRUCT` clause of this rule will create statements similar to the ones from Figure 8.1, except that Alice's availability is set to *free*.

```
alex@delta: ~/work/PhD/resources/icasa-teaching-distribution-1.2.7 169x42
Task: load context model modules. Duration: 60 ms
Task: register custom RDF datatypes. Duration: 0 ms
Task: register custom SPARQL functions. Duration: 40 ms
Task: create the ContextAssertionIndex. Duration: 81 ms
Task: create the ContextAnnotationIndex. Duration: 0 ms
Task: create the ContextConstraintIndex. Duration: 40 ms
#### Constraint Index :
{{assertionType: Dynamic, assertionArity: 2, assertionOntologyResource: http://pervasive.semanticweb.org/ont/2014/07/smartclassroom/core#hasAvailabilityStatus}=[http://p
ervasive.semanticweb.org/ont/2015/01/alicepersonal/constraint#AvailabilityStatusConstraint]}

Task: compute derivation rule dictionary. Duration: 60 ms
[AlicePersonalActivator] INFO: Performing AlicePersonal bootstrapping.
[CtxCoord__AlicePersonal@EF210]: Done agent setup. Signaling outcome.
Starting agent: CtxQueryHandler__AlicePersonal
Starting agent: CtxUser__AlicePersonal
CMM INIT SUCCESSFUL!
[CtxUser] INFO: Setting queryHandler - CtxQueryHandler__AlicePersonal@EF210
[AliceUser] Simulation Manager dependency resolved.
[org.aimas.ami.cmm.simulation.users.AliceUser] Application Adaptor dependency resolved.
[org.aimas.ami.cmm.simulation.users.AliceUser] CSBuilding Application Adaptor dependency resolved.
[org.aimas.ami.cmm.simulation.users.AliceUser] Personal Application Adaptor dependency resolved.
[org.aimas.ami.cmm.simulation.users.PersonUser Alice] Describing myself
Moving Alice to Zone EF210_PresenterArea
[AliceUser] INFO: Subscribing for location information
[LocationInfoNotifier] INFO: Received Notification for Alice Location Info: http://pervasive.semanticweb.org/ont/2014/07/smartclassroom/bootstrap#EF210
[INFO AirConditioningUser] SUBSCRIBING FOR TEMPERATURE QUERY WITH ID: CtxUser__AirConditioning@EF210-query-1429718894501-6
[CheckConstraintHook] INFO: Constraint violations (1) detected for assertion: {assertionType: Dynamic, assertionArity: 2, assertionOntologyResource: http://pervasive.sem
anticweb.org/ont/2014/07/smartclassroom/core#hasAvailabilityStatus}
[ResolveAvailabilityConflict] INFO: Invocation of the Availability Status Constraint Resolver
[ResolveAvailabilityConflict] INFO: There really is just a single person in the room !!!
[ResolveAvailabilityConflict] INFO: Resolution Service keeps assertion with content:
[http://pervasive.semanticweb.org/ont/2014/07/smartclassroom/bootstrap#Alice, http://pervasive.semanticweb.org/ont/2014/07/smartclassroom/core#hasAvailabilityStatus, htt
p://pervasive.semanticweb.org/ont/2014/07/smartclassroom/core#Free]
[SimulateUserCallCommand] Simulating a CALL from user: Bob to user: Alice
[SimulateUserCallCommand] The simulation implies just a short lived subscription to the current availability status of the callee.
[AvailabilityInfoNotifier] INFO: We received news that Alice is Free at 2014-09-15T11:55:54.969Z^^http://www.w3.org/2001/XMLSchema#dateTime
[AvailabilityInfoNotifier] INFO: We received news that Alice is Free at 2014-09-15T11:55:59.96Z^^http://www.w3.org/2001/XMLSchema#dateTime
[AvailabilityInfoNotifier] INFO: We received news that Alice is Free at 2014-09-15T11:56:04.955Z^^http://www.w3.org/2001/XMLSchema#dateTime
##################### SIMULATION END #####################
#### Collecting Engine performance results
#### Collecting Sensing statistics
```

Figure 8.7: Console Log of the pre-meeting episode simulation using the iCasa platform.

As explained in Section 8.2.1, in the context model for Alice's personal information (from the *bootstrap* CMU on her smartphone) contains a uniqueness constraint specified for the `hasAvailabilityStatus` *ContextAssertion*. The two *DerivationRules* above infer instances of *ContextAssertions* that trigger the detection of this constraint. This is observable in the console log of the scenario simulation, show in Figure 8.7. Notice the line starting with `[CheckConstraintHook]` which shows the detection of a constraint violation for our mentioned *ContextAssertion*.

As explained in Sections 4.4.1 and 7.2.2, the CONSERT Engine is able to interact with custom constraint resolution services, which are implemented by the developer and assigned to handle detected constraints for a specific *ContextAssertion* using the provisioning coordination policy (cf. Section 5.2.2).
In this case, the service resolving the availability contradiction works by explicitly verifying if the recent (i.e. inserted less than 2 seconds ago) `hasPersonCount` *ContextAssertion* does indeed specify just the presence of a single person in the same physical location as that where the `TeachingActivity` instance of the `engagedIn` *ContextAssertion* takes place.
In the console output from Figure 8.7 one can observe the informative line starting with `[ResolveAvailabilityConflict]` which says that indeed, at the moment of evaluation of the resolution service, there really is only one person in the AmI laboratory (Alice). The constraint resolution service then decides which of the two contradicting *ContextAssertion* instances to keep as valid in the *ContextStore*. In this case, the console log shows that it is the one specifying that Alice is *free*.
Consequently, in the log output we can see that when a call to Alice's smartphone is simulated, the answer received (lines beginning with `[AvailabilityInfoNotifier]`) is that Alice is indeed *free*, even though it is before 12:00 PM.

Ad Hoc Meeting Episode

In the ad hoc meeting episode, the main reasoning task (apart from localization of users at the same desk) is that of deriving the ad hoc meeting itself.

```
CONSTRUCT {
        ...
    _:n1 a smartclassroom:HostsAdHocDiscussion .
    _:n1 core:assertionRole ?RL .

    _:n2 a annotation:DatetimeTimestamp .
    _:n2 annotation:hasStructuredValue ?t .
    _:n3 a annotation:TemporalValidity .
    _:n3 annotation:hasStructuredValue ?valid .
    _:n4 a annotation:NumericValueCertainty .
    _:n4 annotation:hasStructuredValue ?acc .
    _:n5 a annotation:SourceAnnotation .
    _:n5 annotation:hasUnstructuredValue ?src .
}
WHERE {
    {
        SELECT ?RL ((COUNT(DISTINCT ?S)) AS ?Ct) ((AVG(?accS)) AS ?AvgAccS)
        WHERE {
            GRAPH ?profiledLoc {
                ?K device:hasProfiledLocation ?RL .
            } .
            GRAPH ?gCamera {
                _:0 a smartclassroom:sensesSkeletonInPosition .
                _:0 smartclassroom:hasCameraRole ?K .
                _:0 smartclassroom:hasSkeletonRole ?S .
                _:0 smartclassroom:hasSkelPositionRole smartclassroom:SkeletonSitting
                    .
            } .
            GRAPH <http://pervasive.semanticweb.org/ont/2014/07/smartclassroom/core/
                sensesSkeletonInPositionStore> {
                ?gCamera annotation:hasValidity ?valAnn .
                ?valAnn annotation:hasStructuredValue ?validS .
                ?gCamera annotation:hasCertainty ?certAnn .
                ?certAnn annotation:hasStructuredValue ?accS .
            } .
            BIND (functions:now() AS ?now) .
            BIND (functions:datetimeDelay(?now, -2) AS ?close) .
```

```
36              BIND (functions:datetimeDelay(?now, -20) AS ?earlier) .
37              BIND (functions:makeValidityInterval(?earlier, ?close) AS ?interv) .
38              FILTER functions:validityIntervalsInclude(?validS, ?interv) .
39          }
40          GROUP BY ?RL
41      } .
42      GRAPH ?profiledLoc {
43          ?Mic device:hasProfiledLocation ?RL .
44      } .
45      GRAPH ?gNoise {
46          ?Mic smartclassroom:hasNoiseLevel ?lvl .
47      } .
48      GRAPH <http://pervasive.semanticweb.org/ont/2014/07/smartclassroom/core/
            hasNoiseLevelStore> {
49          ?gNoise annotation:hasCertainty ?certAnnMic .
50          ?certAnnMic annotation:hasStructuredValue ?accMic .
51          ?gNoise annotation:hasValidity ?validAnnMic .
52          ?validAnnMic annotation:hasStructuredValue ?validMic .
53      } .
54      BIND (functions:now() AS ?now) .
55      BIND (functions:datetimeDelay(?now, -2) AS ?close) .
56      BIND (functions:datetimeDelay(?now, -20) AS ?earlier) .
57      BIND (functions:makeValidityInterval(?earlier, ?close) AS ?interv) .
58      FILTER ((((?gNoise = ?contextAssertionUUID) || (?gCamera = ?contextAssertionUUID)
            ) && (((?Ct >= 2) && (?AvgAccS >= 0.75)) && (?lvl >= 60))) && functions:
            validityIntervalsInclude(?validMic, ?interv)) .
59      BIND (functions:certaintyMeetOp(?AvgAccS, ?accMic) AS ?acc) .
60      BIND (functions:getCurrentAgent() AS ?src) .
61      BIND (functions:now() AS ?t) .
62      BIND (functions:makeValidityInterval(functions:datetimeDelay(?t, -20), ?t) AS ?
            valid) .
63  }
```

Listing 8.2: SPARQL query of the `AdHocDiscussionRule` *ContextDerivationRule* template

The SPARQL query that infers that a given `RoomSection` (i.e. one of the desks from the AmI laboratory) hosts an ad hoc discussion is shown in Listing 8.2. In this case, notice that the inferred *ContextAssertion* is an instance of a *UnaryContextAssertion*, as explained in Section 8.2.1. In this case, its RDF representation requires two statements, one defining the assertion instance and one specifying the *ContextEntity* that plays a role in the assertion (using the `assertionRole` property).
In the `WHERE` clause of the query we can also see an example of the expressiveness advantage of SPARQL as a rule language, namely the use of aggregation expressions (e.g. count of observed body posture skeletons per room section) to specify the conditions for accurately detecting body postures.
The `BIND` statements at the end of the `WHERE` clause give the values of *ContextAnnotations* for the derived *ContextAssertion* instance. As specified in Section 4.3.1, they represent the concrete implementation of the *annotation assignment functions* from the formal CONSERT meta-model.

In Figure 8.8 one can see a part of the console log from simulation of the ad hoc discussion episode. Notice how after Bob and Cecille move to the desk code named *EF210_Section1_-Right*, Alice's smartphone submits a subscription (using the `CtxUser` agent from the CMU for the `AmI-Lab` *ContextDomain*) for detecting if she is in ad hoc meetings. This happens because her application specifies that as soon as more than two people are detected near the same desk, this can be an indication that they are part of a collective activity (in this case the ad hoc discussion).
A few lines further down in the console output, one can observe that the ad hoc discussion has been detected. Using a prosuming behavior like the one explained in Section 6.4.4, the information about being located at a desk that hosts an ad hoc discussion is forwarded to the CONSERT Engine of the bootstrap CMU on Alice's mobile device. The `BusyMeetingRule`

```
alex@delta: ~/work/PhD/resources/icasa-teaching-distribution-1.2.7 169x42
[org.aimas.ami.cmm.simulation.users.PersonUser Cecille] Describing myself
Moving Alice to Zone EF210_Section1_Right
[AliceUser] INFO: Subscribing for location information
Moving Bob to Zone EF210_Section1_Right
Moving Cecille to Zone EF210_Section1_Right
[LocationInfoNotifier] INFO: Received Notification for Alice Location Info: http://pervasive.semanticweb.org/ont/2014/07/smartclassroom/bootstrap#EF210
[INFO AliceUser] SUBSCRIBING FOR AD-HOC MEETING QUERY WITH ID: CtxUser__AliceUsage@EF210-query-1429777105257-12
[INFO AirConditioningUser] SUBSCRIBING FOR TEMPERATURE QUERY WITH ID: CtxUser__AirConditioning@EF210-query-1429777105295-13
Moving Dan to Zone EF301
[INFO DanUser] REMOTE SUBSCRIBE FOR ALICE'S STATUS WITH ID: CtxUser__DanUsage@EF210-query-1429777106410-14
[INFO org.aimas.ami.cmm.simulation.users.AliceUser] USER Alice IN AD-HOC DISCUSSION in EF210_Section1_Right at 2014-09-15T12:01:49.273+00:00^^http://www.w3.org/2001/XMLS
chema#dateTime !!!!!
[DanUser] INFO: We received news that Alice is Busy at 2014-09-15T12:01:49.273Z^^http://www.w3.org/2001/XMLSchema#dateTime
[INFO org.aimas.ami.cmm.simulation.users.AliceUser] USER Alice IN AD-HOC DISCUSSION in EF210_Section1_Right at 2014-09-15T12:01:51.405Z^^http://www.w3.org/2001/XMLSchema
#dateTime !!!!!
[DanUser] INFO: We received news that Alice is Busy at 2014-09-15T12:01:51.405Z^^http://www.w3.org/2001/XMLSchema#dateTime
[INFO] User Projector is leaving.
[INFO org.aimas.ami.cmm.simulation.users.ProjectorUser] CANCELING SUBSCRIPTION FOR LUMINOSITY: CtxUser__Projector@EF210-query-1429777111426-20
[INFO org.aimas.ami.cmm.simulation.users.AliceUser] USER Alice IN AD-HOC DISCUSSION in EF210_Section1_Right at 2014-09-15T12:01:53.376Z^^http://www.w3.org/2001/XMLSchema
#dateTime !!!!!
[DanUser] INFO: We received news that Alice is Busy at 2014-09-15T12:01:53.376Z^^http://www.w3.org/2001/XMLSchema#dateTime
[INFO org.aimas.ami.cmm.simulation.users.AliceUser] USER Alice IN AD-HOC DISCUSSION in EF210_Section1_Right at 2014-09-15T12:01:55.379Z^^http://www.w3.org/2001/XMLSchema
#dateTime !!!!!
[DanUser] INFO: We received news that Alice is Busy at 2014-09-15T12:01:55.379Z^^http://www.w3.org/2001/XMLSchema#dateTime
[INFO org.aimas.ami.cmm.simulation.users.AliceUser] USER Alice IN AD-HOC DISCUSSION in EF210_Section1_Right at 2014-09-15T12:01:57.358Z^^http://www.w3.org/2001/XMLSchema
#dateTime !!!!!
[DanUser] INFO: We received news that Alice is Busy at 2014-09-15T12:01:57.358Z^^http://www.w3.org/2001/XMLSchema#dateTime
[INFO] User Bob is leaving.
[INFO] User Cecille is leaving.
[INFO AliceUser] CANCELING SUBSCRIPTION FOR AD-HOC MEETING: CtxUser__AliceUsage@EF210-query-1429777105257-12
[DanUser] INFO: We received news that Alice is Free at 2014-09-15T12:02:10.74Z^^http://www.w3.org/2001/XMLSchema#dateTime
Moving Alice to Zone CSBuilding
[DanUser] INFO: We received news that Alice is Free at 2014-09-15T12:02:13.007Z^^http://www.w3.org/2001/XMLSchema#dateTime
[INFO org.aimas.ami.cmm.simulation.users.AirConditioningUser] CANCELING SUBSCRIPTION FOR TEMPERATURE: CtxUser__AirConditioning@EF210-query-1429777105295-13
[INFO] User Alice is leaving.
[INFO] User Dan is leaving.
[INFO DanUser] CANCELING REMOTE SUBSCRIPTION FOR ALICE'S STATUS: CtxUser__DanUsage@EF210-query-1429777106410-14
[INFO] User AirConditioning is leaving.
#################### SIMULATION END ####################
#### Collecting Engine performance results
#### Collecting Sensing statistics
```

Figure 8.8: Console Log of the ad hoc discussion episode simulation using the iCasa platform.

DerivationRule will then again infer that Alice is *busy*, as can be seen in the console output showing the response to Dan's query for the availability status of Alice.
Dan continues receiving the *busy* status until the meeting is over (seen in the log when Bob and Cecille are leaving). In that case, the ad hoc meeting is no longer derived, such that Alice is deemed to be *free*.

From the textual representation in Figures 8.1 and 8.2, one can observe that rule logic is easily understandable. Furthermore, an advantage over existing rule-based reasoning approaches studied in Section 2.2.4 comes from using SPARQL and relates to the ability of expressing conditions over aggregation expressions such as in lines 16 - 41 from Figure 8.2.
Nonetheless, one can observe that the current form of accessing annotations of a *ContextAssertion* instance within a rule leads to a very verbose representation, an issue that we attempt to mitigate in future work.

8.2.3 Provisioning Control Evaluation

In the description of the reference scenario (more particularly for the ad hoc meeting episode) we showed that the various sensors (temperature, luminosity, noise level, body posture) are engaged (i.e. their updates are enabled or disabled) based on the dynamic usage of the associated context information (e.g. using the projector, subscribing for ad hoc discussion which requires the enabling of the *DerivationRule*, which requires enabling of the noise level and body posture sensor updates).
In Sections 5.2 and 5.3 we explained how dynamic context usage is addressed by the context provisioning policies which guide the behavior of the CMU agents (especially `CtxSensor` and `CtxCoord`). In order to observe the effect of using such policies we devised two experiments based on the ad hoc meeting episode from the reference scenario.

Provisioning policy setup Remember from Sections 6.2.3 and 7.4.1 that the sensing and coordination provisioning policies are indicated to the CMU agents as files within the bundle that configures the CMU deployment.

Two configurations are used to evaluate the impact of using context provisioning policies. The first policy enables all *ContextAssertion* updates by default, without using *ContextAssertion* monitorization. The second one uses provisioning control parameters to enable only presence sensing and location derivation by default. All other context information updates are activated and deactivated on demand. Notifications from the `CtxQueryHandler` control update activation, while four provisioning control rules from the policy set the conditions for disabling specific updates and derivations (cf. Section 7.1.3 for one such example).

Provisioning policy influence Figure 8.9 shows the influence of provisioning policies on sensing behavior and update messages. In the simulation, the sensing policy configures updates every 2 seconds. As expected, the case where all sensor updates are enabled by default (left hand graph) shows an almost constant sensing activity (2563 sensor readings and 1168 message updates during the 3 minute simulation). The right hand graph (corresponding to adaptive

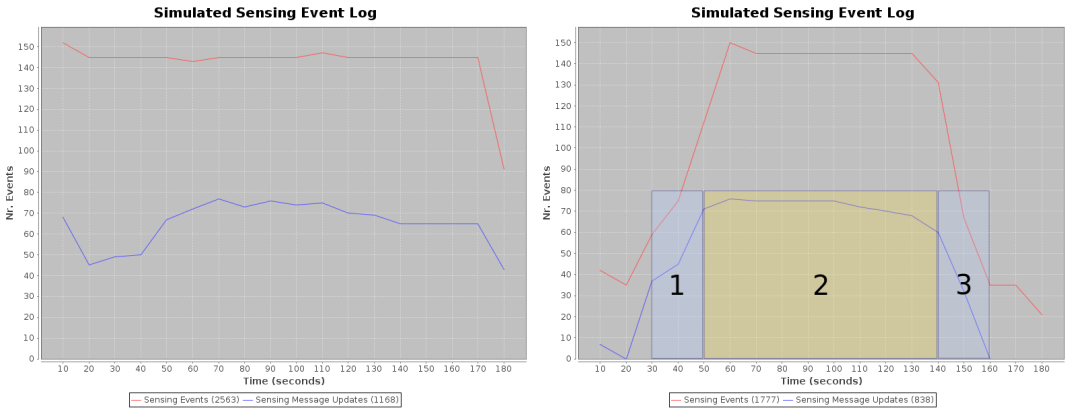

Figure 8.9: Sensing events and update messages graph. No provisioning control (left), with provisioning control (right)

provisioning) shows a dynamic evolution of the sensing behavior. At first, only presence sensors are active. When Bob and Cecille join the AmI laboratory, temperature sensors become active (interval 1 on chart). Further, when Bob turns on the projector, updates for luminosity are enabled (interval 1 on chart). When all 3 persons are at a desk talking, they subscribe for possible activity notifications and the noise level and body posture sensors needed for ad hoc meeting detections become active (interval 2). When the meeting is over, the provisioning control rules ensure that unnecessary sensing activity be ceased (interval 3), leaving only presence sensing enabled. Overall, *this reduces the sensing activity for the same simulation by 30%*. In a real environment, this could have significant benefits in terms of network load and power consumption.

Figure 8.10 shows measurements of CONSERT Engine insertion delay (time until processing starts) and processing duration for *ContextAssertion* updates. Charts similar to this one relating to CONSERT Engine performance will be discussed in more detail in Section 8.3.2.
While the average delay and processing times for the two simulation configurations are very similar, on the right hand graph we observe slightly higher insertion processing times (blue) and a greater density of insertion delay points (red) that are higher than the average. This is a consequence of having an additional periodic context knowledge base transaction for the evaluation of provisioning control rules (cf. Section 7.3.3). However, these results stem from the particular setup of our simulation (relatively reduced load and information diversity) and a technical limitation of the current implementation which relies on making transaction snapshots of the *ContextStore* when executing insertions, inferences or provisioning control rules. Furthermore, if the dynamic between active and non-active update and derivation periods is

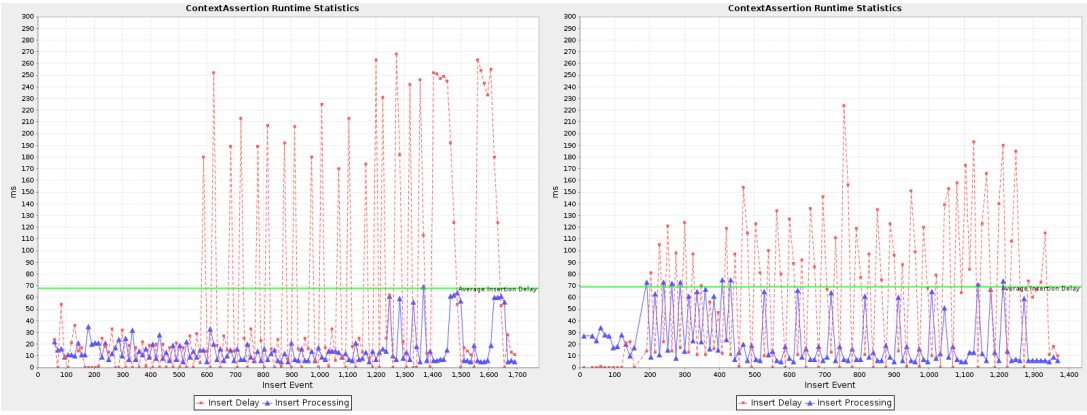

Figure 8.10: CONSERT Engine update performance showing insertion delay (red) and insertion processing (blue) times. No control rules (left), with provisioning control (right)

greater than the number of control rules, we expect that insertion delay and processing times will actually improve. We discuss this and other potential improvements in Section 8.4.

As mentioned in Section 3.1.3, from the reviewed provisioning architectures that rely on fixed communication infrastructures only COPAL [Sehic and Dustdar, 2010] and SOLAR [Chen et al., 2008] provide support for declarative specification of the provisioning functionality. The CONSERT Middleware distinguishes itself from these works by some of its characteristic parameters that influence the provisioning process, such as assigning constraint resolution services or defining time periods for ontology-based reasoning.

8.2.4 Deployment Evaluation

In the qualitative analysis of middleware deployment engineering we relate to the aspects discussed throughout chapter 6. First, we present the setup of the CONSERT Middleware in the simulation of the reference scenario in terms of the CMUs that need to be configured for all the actors that participate in the simulation. We then look at how the CMU setup is exploited in terms of prosuming behavior and domain-based query management.

Middleware Setup In terms of deployment configuration specification, we have already seen the setup for the CMU of the AmI-Lab management server, which has been exemplified throughout Section 7.1.4. Similarly structured configurations were developed for the other CMUs that were required as part of the applications driving the simulation of the two reference scenario episodes (cf. Figure 8.1 for the list of applications and the CMUs they require).
Speaking in terms of the CMU configuration bundles described in Section 7.4, Table 8.1 shows a list of the CMU configuration bundles that were deployed in the simulation.
As mentioned in the chapter on middleware implementation, there is a need for a bundle having a specific bundle name header (cmm-default-instance) to specify the configuration of the CONSERT Middleware platform on which all the other CMUs from the simulation run.
Further, notice that for each *ContextDomain* a *management* (i.e. coordination) CMU is created to run on the simulated server managing the given domain (e.g. `AmI-Lab`, `EF301`, `CS_-Building`). Apart from the necessary *ContextDomain* specifications, the **`agent-config.ttl`** files for these CMUs configure the deployment of a `CtxCoord` and `CtxQueryHandler` agent. The CMU for the `AmI-Lab` management server additionally configures the set of `CtxSensor` agents responsible for collecting the sensed temperature, luminosity, noise level and body posture data.

Client (i.e. usage) CMUs assigned to the same *ContextDomains* as the management ones are configured on the simulated smartphones of each person involved in the scenario episodes. These CMUs configure the deployment of a `CtxUser` agent whose `assignedOrgMgr` property points to the `OrgMgr` agent of the corresponding management server CMU.
Lastly, one can notice the bootstrap CMU configured in a *centralized* deployment scheme to manage Alice's personal context specifications (e.g. the *DerivationRules* determining her availability status, the service resolving the availability uniqueness constraint). The CMU configures an instance of the `CtxCoord`, `CtxQueryHandler` and `CtxUser` agents.

Prosuming Behavior In both episodes of the simulation scenario the `CtxUser` agents running on Alice's smartphone (the one from the bootstrap CMU and the one from the AmI-Lab client CMU) are required to perform both query and profiled insertion tasks (i.e. prosuming behavior).

In the pre-meeting episode, Alice's simulated smartphone application uses the `CtxUser` from the AmI-Lab client CMU (through the `ApplicationClientAdaptor` service) to query for Alice's location and determine the number of people located in the same room as her. When receiving answers to this queries, the application uses the `CtxUser` of the bootstrap CMU to insert the answers as *profiled ContextAssertion* instances inside the *ContextStore* of the

CMU	ContextDomain	Purpose
cmm-instance-default	–	Bundle containing CONSERT Middleware platform specification for the simulation test bed.
AmI-Lab server	`AmI-Lab`	Configuration bundle for CMU of AmI-Lab management server.
EF301 server	`EF301`	Configuration bundle for CMU of EF301 office management server.
CS Building server	`CS_Building`	Configuration bundle for CMU of CS Building management server.
Alice Bootstrap	–	Configuration bundle for CMU of Alice's personal context management application.
Alice AmI-Lab client	`AmI-Lab`	Configuration bundle for CMU of AmI-Lab context usage on Alice's smartphone.
Alice CS Bulding client	`CS_Building`	Configuration bundle for CMU of CS_Building context usage on Alice's smartphone.
Bob AmI-Lab client	`AmI-Lab`	Configuration bundle for CMU of AmI-Lab context usage on Bob's smartphone.
Cecille AmI-Lab client	`AmI-Lab`	Configuration bundle for CMU of AmI-Lab context usage on Cecille's smartphone.
Dan EF301 client	`EF301`	Configuration bundle for CMU of EF301 context usage on Dan's smartphone.
AirConditioning AmI-Lab client	`AmI-Lab`	Configuration bundle for CMU of AmI-Lab context usage by the air conditioning unit.
Projector AmI-Lab client	`AmI-Lab`	Configuration bundle for CMU of AmI-Lab context usage by the projector unit.

Table 8.1: List of CMU bundles, the *ContextDomain* to which they are assigned and their purpose within the simulation of the reference scenario.

CONSERT Engine running in the bootstrap CMU. The working of this sequence of actions is confirmed by the constraint detection and resolution messages that appear in the console log from Figure 8.7, since the triggering of the *DerivationRules* that violate the availability status uniqueness constraint is predicated on obtaining Alice's location information and the number of people in the laboratory.

In the ad hoc meeting episode, the `CtxUser` from the AmI-Lab client turns to a provider of context information himself. First, the AmI-Lab client `CtxUser` is used to subscribe for notifications of ad hoc meetings hosted in room sections from the AmI-Lab. The obtained information is relayed via the bootstrap CMU `CtxUser` to Alice's personal `CtxCoord` and inserted in the local CONSERT Engine instance. The `BusyMeetingRule` triggers and determines that Alice is busy as a consequence of being in an ad hoc discussion. This time, the application uses the bootstrap CMU to subscribe for such notifications (of being free or busy) and then uses the `CtxUser` of the AmI-Lab client CMU to send a profiled *ContextAssertion* towards the AmI-Lab management server CMU such that her status may become visible to queries coming from Dan. The effective working of these actions is confirmed by the messages from the simulation log in Figure 8.8 which show that Dan receives an answer to his domain-based queries for Alice's availability status.

The descriptions given above are a good example of the way in which a context-aware application is supposed to exploit the CONSERT Middleware from a context usage point of view. The deployment specifications of our CMM allow an application designer to perform a domain-based context model partitioning and use a CMU as a unit of control encapsulation for the context interactions specific to a *ContextDomain*.
However, it is still a matter of *application logic* to decide how context captured as part of one *ContextDomain* can be reused within another. Therefore, the control of the prosuming behavior of `CtxUser` agents from various CMUs deployed on the same physical machine is maintained by the context-aware application designer.

Domain-Based Query Management The domain-based query functionality is encountered in the ad hoc meeting episode when Dan, who is in the `EF301` office, uses the `CtxUser` from his `EF301` client CMU to make a *domain-range* query for Alice's availability status. Since he does not know Alice's current location, he opts to send a domain-range query having the `SchoolBuilding` and `LaboratoryRoom` *ContextDomain types* as its upper and lower limits respectively. In Listing 8.3 one can observe the content of the **`domain-hierarchy-config.ttl`** file which specifies the *ContextDomain* hierarchy overview (cf. Figure 7.6) required by the `OrgMgr` agents from the `CS_Building`, `AmI-Lab` and `EF301` management server CMUs to perform the distributed query routing according to the algorithms described in Section 6.4.1.

```
1  ...
2
3  sim:CS_Building
4    rdf:type sim:SchoolBuilding ;
5    rdfs:label "CS_Building"^^xsd:string ;
6  .
7  sim:AmI-Lab
8    rdf:type sim:LaboratoryRoom ;
9    space:includedIn sim:CS_Building ;
10   rdfs:label "EF210"^^xsd:string ;
11 .
12 sim:EF301
13   rdf:type sim:OfficeRoom ;
14   space:includedIn sim:CS_Building ;
15   rdfs:label "EF301"^^xsd:string ;
16 .
```

Listing 8.3: Content of the domain-hierarchy-config.ttl file for the simulation of the reference scenario.

As mentioned in the chapter on middleware deployment, the domain hierarchy overview file specifies an RDF model that indicates the *ContextDomain* instances and their *type* definitions (e.g. *sim:CS_Building rdf:type sim:SchoolBuilding*), as well as the "inclusion" relation that exists between the *ContextDomains* using the *EntityDescription* constituing the *domain hierarchy property* (e.g. *sim:AmI-Lab space:includedIn sim:CS_Building*).
The successful execution of the domain-based query routing protocol can again be observed based on the simulation log from Figure 8.8 which shows that Dan retrieves answers about Alice's availability status. Furthermore, notice that the log shows Dan receiving answers to his query both during the time in which Alice is in the `AmI-Lab`, as well as after she moves to the `CS_Building` hallway to head to the cafeteria. In the second case, Alice uses the `CtxUser` agent from her `CS_Building` client CMU to send a profiled *ContextAssertion* update specifying that she is free. This information is stored in the CONSERT Engine running in the CMU of the `CS_Building` management server. This way, when Alice moves to the hallway, Dan's domain-range query actually obtains answers from the `CtxQueryHandler` agent running in the CMU of `CS_Building` management server.

Most of the works reviewed in Chapter 3 have the ability to address the context management needs highlighted by the reference scenario. However, the distinguishing feature of the CONSERT Middleware, lacking in all of the reviewed works, is the explicit and context model related method to *structure* deployment of context provisioning units within an application. The proposed deployment conceptualization (in terms of *ContextDimensions* and *ContextDomains*) adds an explicit technical mean for developers to manage the life cycle of context provisioning units according to the anticipated *usage* of context information within the application.

8.3 Performance Testing

Apart from the qualitative assessments presented previously, we performed two types of quantitative evaluations. The first test evaluates the performance of the CONSERT Engine as a context information processing component. The second experiment looks at the distributed query routing capabilities of the context provisioning agents.

8.3.1 CONSERT Engine Test Setup

In performing the test, our interest is to verify that the proposed representation and reasoning engine can be successfully employed as part of a real-time context provisioning mechanism. In other words, we want to perform a load test on our proposed system.
We start by writing a simple scenario generating program that is able to create definitions of *ContextEntities*, *ContextAssertions* and *ContextDerivationRules*. The automated generation is controlled by the following parameters:

- number of *ContextEntity* classes and number of instances of each class
- total number of *ContextAssertion* classes of each arity (unary, binary or n-ary)
- number of derived *ContextAssertion* classes out of the total number of class types pertaining to each arity
- number of instances that would be generated during the test run for each type of *ContextAssertion*

The automatically generated context model is then used by a test script to generate a sequence of *ContextAssertion* update requests. The parameters which control the runtime dynamics are: the temporal validity duration of a *ContextAssertion* instance (in *ms*) and the instance pushrate (number of *ContextAssertion* instances generated per validity interval) for each arity class. The context model generation parameters influence the size and variety of the simulated domain, and they were used to study the memory footprint of the system, specifically in order to determine if an increasing number of named graphs (as would be expected by a larger number of *ContextAssertion* classes and instances) poses a concern. The parameters controlling the test script influence the responsiveness of the system and we wanted to determine under what kind of load it would still be able to perform at a level acceptable for real-time usage (e.g. a short enough time span passes from the moment a *ContextAssertion* that triggers an inference is created to the point where the inferred assertion becomes visible).
Note that for this automated test, we focused on checking the performance of the *event processing* capabilities of the CONSERT engine and did not trigger any ontology reasoning in between *ContextAssertion* updates. We discuss the influence that even simple RDFS reasoning can have on update processing in a second experiment. For the purpose of the automated test the *ContextDerivationRules* are kept at a low complexity in order to allow automatic generation and to ensure that a sufficient amount of rules fire during runtime.
The output of the scenario generation program are OWL files that contains the ontology definitions of the simulated context domain and the accompanying context derivation rules. After initialization, the test script is used to perform actual generation of *ContextAssertion* instances and collect runtime measures. We detail the type of collected information and its analysis in the following section.

8.3.2 CONSERT Engine Test Results

During the run of a test we collected information regarding the trace left by each created *ContextAssertion* within the system. We looked at insertion delay time (Step 1 in Figure 4.7: time spent from entering the update request queue until start of processing), insertion processing time (Steps 2a – 2d: how long it takes to apply all the verifications detailed in Section 4.4.2), inference delay (Step 4) and inference processing time (in the case the newly created *ContextAssertion* triggers an inference - Steps 4a – 4c) and overall deduction time (the amount it takes from the moment an inference triggering *ContextAssertion* enters the system until the deduced one is also observed). For each of these parameters we also computed minimum, average and maximum values.
To compute memory consumption we used the jProfiler[1] Java profiling framework to observe live memory usage. At the end of a test run, after performing garbage collection, we specifically recorded the number of instances and total size of several key data structures that would be directly influenced by the number of created context assertions. We also looked at the total memory size of the Java heap space, after garbage collection.

Runtime Processing Times

In Figure 8.11 we observe the runtime history of a test where the pushrate has been set to 20 *ContextAssertion* requests per second. The upper chart shows the insertion delay and insertion processing metrics, the middle one draws overlaid bars showing the inference delay and inference processing values, whereas the lower chart combines the two and shows a bar chart of the deduction duration information for those *ContextAssertion* instances that triggered the inference of a new one.

[1] http://www.ej-technologies.com/products/jprofiler/overview.html

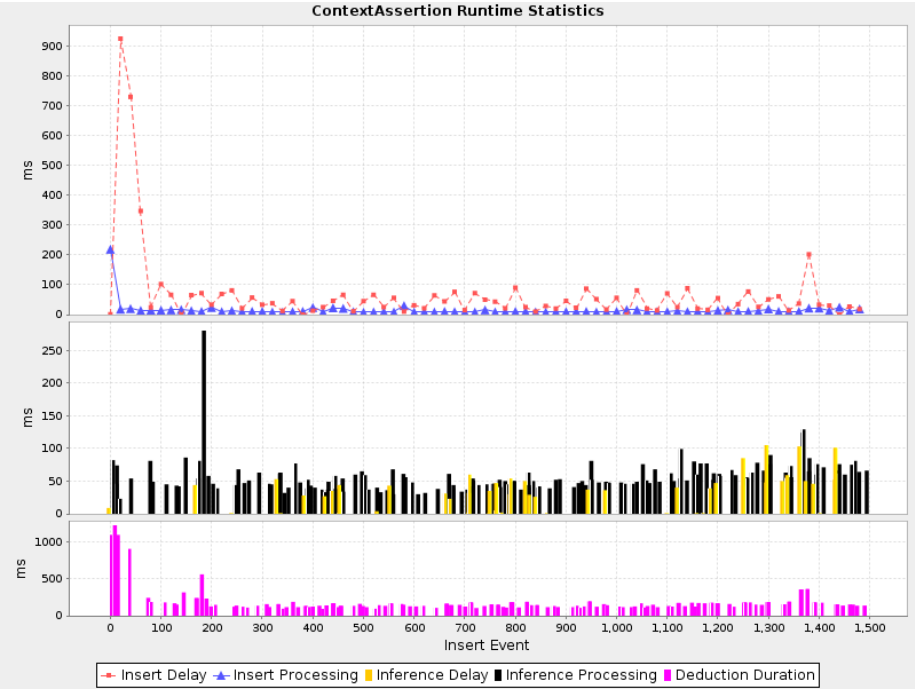

Figure 8.11: Runtime results for test run with 10 *ContextAssertion* classes of each arity type and 50 instances for each class. The validity duration of a *ContextAssertion* instance is set at 1000 ms and the pushrate has a value of 20 instances generated per validity interval (i.e., 20 events per second).

Table 8.2: Minimum, average and maximum values for the collected runtime parameters of the 20 events per second test run (in ms)

Insertion Delay			Insertion Processing			Inference Delay			Inference Processing			Deduction Duration		
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
0	71	952	7	15	272	0	14	141	23	59	255	85	206	1258

The spike at the beginning of the insertion delay plot is attributed to the “warm-up" of the thread pool handling the update, requests as well as, more decisively, that of the in-memory TDB quadstore engine where the inserted *ContextAssertions* are stored. The average value of the insertion delay metric is of only 71 ms, as can be seen in Table 8.2. The variation in the delay time is in sync with the peeks of the inference plot and it is due to the addition of the deduced *ContextAssertions* to the list of ones that have to be inserted as part of the next “batch" of events. Given the low complexity of the employed *Context Derivation Rules*, Table 8.2 shows that the inference duration time is fairly low and since the number of derived *ContextAssertions* is much lower than that of the created ones, there is no build up in the request queue of the inference thread pool and so the inference processing time dominates the inference delay time (the time that an inference request spends in the InferenceRequest Queue before actual execution).
For this test configuration the average deduction duration is set around 206 ms. Considering that one second can be seen as a decent response time for a realtime recognition of a situation, it means that the current load of 20 *ContextAssertion* insertion events per second can actually leave room to spare (in case the derivation rules are more complex and require more time for evaluation).

Deduction Time Analysis

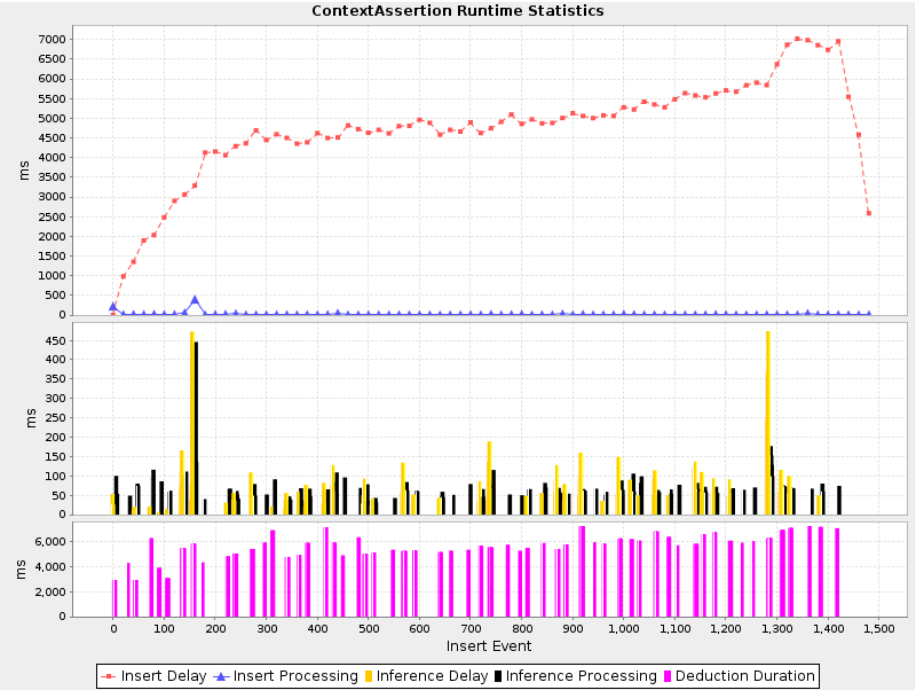

Figure 8.12: Same configuration as for the test case presented in Figure 8.11, but with a pushrate set at 60 instances generated per validity interval (i.e., 60 requests per second).

In general, the deduction duration plot helps us to determine two important aspects concerning the runtime dynamics of a system. First, depending on the value that is considered acceptable for the average deduction duration, we can set the corresponding maximum number of *ContextAssertion* update requests that can be handled per second. This implicitly translates into an upper threshold on the frequency with which different physical, logical or virtual sensors would provide updates to the data they perceive. Keeping a log of the deduction duration data could help a future version of the system determine how to automatically negotiate such sensor update rates. Second, the value of the deduction duration is also an indicator of the minimum temporal validity that a detected situation must have in order to be usefully utilized. That is, if the actual situation is shorter than the time it takes for the system to have it recognized and available for query or decision making, the effort to infer it will not have brought any added value.

We can see the effect of increasing the pushrate load in Figure 8.12 and Table 8.3. For both the insertion and inference of *ContextAssertions* we observe a dramatic increase and dominance of the delay component (the time spent in the request queues). Still, the insertion and inference processing metrics remain almost the same.

Table 8.3: Minimum, average and maximum values for the collected runtime parameters of the 60 events per second test run (in ms)

Insertion Delay			Insertion Processing			Inference Delay			Inference Processing			Deduction Duration		
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
1	4478	7035	8	17	347	0	46	306	31	65	268	2897	5466	7842

This line of experimentation has led us to observe that the true limitation factor in the current

instantiation of the CONSERT engine is given by Jena TDBs ability to handle multiple concurrent READ and WRITE transactions. Since every new *ContextAssertion* update request requires the creation of a WRITE transaction, a high amount of update events per second creates a contention with regard to synchronization after all the verification steps of the insertion processing cycle have been carried out. Under current test configurations we determined that the push rate value ensuring a less than one second deduction duration resides somewhere along the 30 update requests per seconds mark. This still represents a reasonable value for potential real life scenarios.

Runtime Memory Consumption

In terms of memory consumption, our main consideration was the creation of a large amount of named graphs (since every *ContextAssertion* has its own identifier graph). From an insertion and inference lookup point of view, the above analysis seems to indicate that this is not an issue, since Jena TDBs quad indexing scheme is able to efficiently handle the work. To investigate the evolution of the memory consumed by the in-memory TDB quadstore used by the CONSERT engine, we used a Java profiling framework and looked at classes that are in direct relation with the number of generated and updated *ContextAssertions*: com.hp.hpl.jena.graph.Node_URI and com.hp.hpl.jena.graph.Node_Literal. The first class is directly related to the number of named graphs created as each named graph is identified by an URI which becomes an instance of the class. The number of instances of the second class is largely influenced by the annotations of a *ContextAssertion* since it will refer to *ContextAnnotations* such as validity interval, certainty or timestamp which have datatype representations. Apart from the classes mentioned above, we also measured the total heap size at the end of a test run, after having performed garbage collection.

Table 8.4: Memory consumption and instance count for selected data structures during different test runs. Showing values for configurations with 30, 90 and 150 *ContextAssertion* class types and 50 instances per class

	30 x 50		90 x 50		150 x 50	
	instance count	mem. size	instance count	mem. size	instance count	mem. size
Node_URI	3825	60 KB	8227	131 KB	12478	199 KB
Node_Literal	4385	69 KB	8035	128 KB	11857	189 KB
Total Heap size	25.798 KB		40.676 KB		53.325 KB	

What we were most interested in seeing was how the memory consumption would scale with increasing number of *ContextAssertion* class types and instances. The results of 3 tests can be followed in Table 8.4. What is readily observable, and an important point, is that memory usage increase between the 3 test runs is sublinear. We consider that the main motive for this result is the existence of the *continuity check*. Though scenario *ContextAssertion* instances are generated randomly, during test runs we observed a fair amount of successful continuity checks (meaning that the content of a new *ContextAssertion* is the same as that of a previously existing one). In such cases, no additional identifying named graph has to be created and only the *ContextAssertion*'s annotation data has to change, leading to a very small memory increase.
Indeed, a theoretical analysis of the potential *ContextAssertions* of a context domain leads us to see that, given the existence of the continuity check, the number of named graphs that identify *ContextAssertions* is bounded by the number of distinct values that the *ContextEntities* involved in the assertion can have. If this amount is either naturally low, or can be made so by considering aggregations or discretization of physical or virtual sensor data, then the scalability of the system in terms of memory consumption can be decently addressed.

Influence of ontology reasoning

We mentioned that the automated test was meant to assess the performance of the event processing capability of the CONSERT Engine. To analyze the influence of ontology reasoning over the background knowledge in the entityStore, we implemented an additional test script using a subset of the *ContextAssertions* that make up the context model of the reference scenario. Specifically, we simulated part of the AmI-Lab room with several sensors that supplied the following *ContextAssertions*: sensesBluetoothAddress (1 sensor), hasNoiseLevel (7 sensors), sensesTemperature (4 sensors), sensesLuminosity (4 sensors). We set each sensor to be in sync, sending updates every 10 seconds, and furthermore ensured that at each update three *DerivationRules* would fire so as to infer the presence of the user's smartphone, the presence of the user in the room and, by inclusion, their presence within the conference building. We simulated the existence of 3 people in the laboratory (Alice, Bob and Cecille) and, in total, at every event cycle there are 18 updates coming in simultaneously from the sensors, plus the ones generated by the derivation rules. This creates conditions similar to the ones in the automated model generation test.
To see the influence of additional reasoning, we performed two test runs. In the first one, the sensors insert a description of themselves into the entityStore (as *ContextEntities* and *EntityDescriptions*) only in the beginning. In the second one, they change these descriptions whenever they make an update. The entityStore is meanwhile bound to a Jena RDFS reasoner, triggered whenever the entityStore is updated. Table 8.5 shows the results of the two test runs in terms

Table 8.5: RDFS reasoning influence in AmI-Lab simulation test: min., avg. and max. values for the collected runtime parameters (in ms)

Insertion Delay			Insertion Processing			Inference Delay			Inference Processing			Deduction Duration		
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
0	44	677	6	**21**	164	0	33	5254	2	17	82	35	127	935
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
0	1056	3756	6	**101**	667	0	34	5056	2	21	101	58	1221	4164

of the same measurement parameters used for the automated test. The upper rows show the values for the case where the entityStore is touched only in the beginning and the lower row for the one where it changes at every *ContextAssertion* update. While we can see that the values corresponding to the execution of *Derivation Rules* (inference delay and processing) stay the same, the important difference is highlighted for the insertion processing measure. Even though only RDFS reasoning is performed, the average processing time is almost five times greater when performing reasoning at every update. Cumulating this difference for 18 updates obviously leads to increased insertion delay times, especially for assertions which get inserted after having been inferred. The reported max values can be considered outliers, since they are always recorded only at the beginning of the test runs, where the combination of TDB initialization and RDFS reasoning drives the waiting and processing times up. This test shows why the *CONSERT Engine Tasking Service* described in section 4.4.1 is crucial in regulating both event processing and ontology related inferences.

8.3.3 Query Handling Test Setup

The query handling test aims to verify the routing performance of the context provisioning agents. In essence, we created a test bed able to measure the *response times* for queries with different types of routing behavior.
For the experiment we only considered *exact-domain* queries. In the case of *domain-range* queries the forward part of the routing process is similar to the single domain case. The

difference lies in the number of expected return messages (which can be much greater in the case of *domain-range* queries). However, the objective of our test is to observe the nature of the delay in query response times given our tree-based *ContextDomain* hierarchy, rather than testing the throughput capabilities of *CtxQueryHandler* agents. Consequently, the *exact-domain* case allows us to achieve our evaluation objective in a simple way.

In order to automate the testing we created a *ContextDomain* hierarchy generator, which is able to produce setups involving multiple hierarchy trees, of different heights and breadths. The generator is controlled by three parameters:

- `numTrees`: determines the number of *ContextDomain* based hierarchy trees. The `OrgMgr` agents from the CMU at the top of a tree play a *root* role and will therefore be connected with all root `OrgMgr` at the top of other hierarchy trees.
- `levels`: specifies the depth (in number of tree levels) of a *ContextDomain* hierarchy.
- `branchingFactor`: specifies the number of child *ContextDomains* for each domain node of the hierarchy.

For the purpose of automated generation we considered a simple application context model, wherein the *ContextDomain* hierarchy is generated based on a spatial *ContextDimension* (`locatedIn`). In each generated spatial environment, temperature and luminosity readings are collected periodically (every 2 seconds) by corresponding sensors. We implement `ContextAssertion Adaptors` that simulate sensed data collection, by generating values according to a simple formula. The query template then focuses on always demanding the temperature values registered during the past 2 seconds.
The generator produces the required instances of all CMU configuration files detailed in Section 7.4 (platform configuration file, CMU configuration file, domain hierarchy file, sensing and coordination provisioning policy files). It then packages these files in corresponding OSGi bundles. Each generated CMU specifies the creation of the following context provisioning agents: one `CtxCoord` agent, one `CtxQueryHandler` agent, two `CtxSensor` agents (one for handling temperature sensing and one for luminosity updates) and one `CtxUser` agent. Furthermore, the `OrgMgr` managing the CMU at the top of a generated *ContextDomain* hierarchy plays a *root* role, while all the others play a *node* role.

In order to perform the test in conditions closer to a real-life deployment we used two physical machines in the experiment. Since the CONSERT Middleware uses a tree-based *ContextDomain* hierarchy, it means that agents of a CMU for a given *ContextDomain* will perform inter-CMU communications only with agents from CMUs in parent or child *ContextDomains* (except for CMUs of root domains which are fully interconnected as explained in Section 6.3.3).
This effectively means that we can simulate a close-to-real network CMU deployment by having the CMUs from each level of a *ContextDomain* hierarchy tree run on a different physical machine. Figure 8.13 shows a graphical representation of this deployment principle. One can see that the CMU of each root domain is deployed on a different machine. Furthermore, the deployment of CMUs on the different levels of a hierarchy tree alternates between the two machines.

The figure also depicts the generated setup that was actually used to obtain the results that we discuss in the next section. Suspension dots are used in the diagram in some places to reduce clutter. They mark expansions of the *ContextDomain* hierarchy that is similar to the one already visible.
To obtain this setup we ran our generator with the following values for its parameters: `numTrees = 2`, `levels = 4`, `branchingFactor = 2`. We choose to report on this one setting because for the purpose of our query routing performance analysis, we observed that other setups produce comparable results.
In order to clearly distinguish between the target *ContextDomains* of *exact-domain* queries later on in the results, we give each generated *ContextDomain* a unique naming scheme.

In Figure 8.13 we show just the local names (i.e. without the full namespace) of the *ContextDomain* URIs, which use the following scheme: $Domain_{tree-number\_level\_index-in-level}$.
For example this means that $Domain_{1\_2\_1}$ is part of the first tree, on the second level and it is the second one (index starts from 0) to be generated on this level. The number of domains on a level depends of course on the branching factor.

One last detail visible in Figure 8.13 is the sequence of queries that are sent throughout the test (shown in green, curved arrows). The generator always picks a leaf node from one of the created *ContextDomain* hierarchy trees to be the one that emits all the queries.
The first query is always local, so as to set the baseline against which to compare the response times for all other queries that require routing. Then, as shown in the diagram, we explore requests made to target *ContextDomains* of increasing distance (hops) from the source one. First, we direct queries to each *ContextDomain* from the direct path to the root *ContextDomain* (requests 2 - 4) and then to sibling nodes of those which are on the direct path to the root from the source (requests 5 - 7).
Finally, the last request wishes to observe the response time for the longest route possible, one that crosses between hierarchy trees and targets another leaf node (request 8).
For each *ContextDomain* target, we repeated the request 5 times and then averaged the response times, yielding the results we present in the following section.

8.3.4 Query Handling Test Results

The discussion on the results of our query handling test focuses on query response time analysis and overview of the agent interaction trail. Figure 8.14 presents the observed response times (in ms) for the 8 *ContextDomain* targets detailed in the previous section.
The baseline (local domain query) shows a response time of just 26 ms for the handling of the query, with actual processing in the CONSERT Engine taking even less. This of course is intentional, as we explained previously, since the query complexity is reduced in order to evaluate just the overhead added by the routing protocol.
As observable in the graph, the effect of the latter on query response times shows a linear dependence, as was to be expected. The delay is strongly correlated with the number of hops

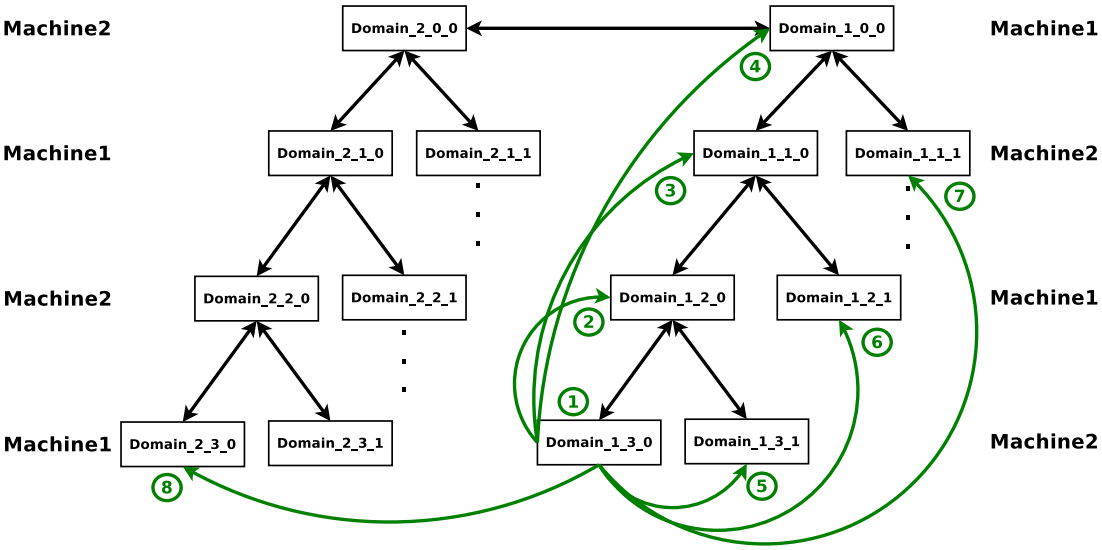

Figure 8.13: Example of CMU distribution on two physical machines such that each inter-CMU communication occurs in between the machines. The sequence of query request tasks is marked in green.

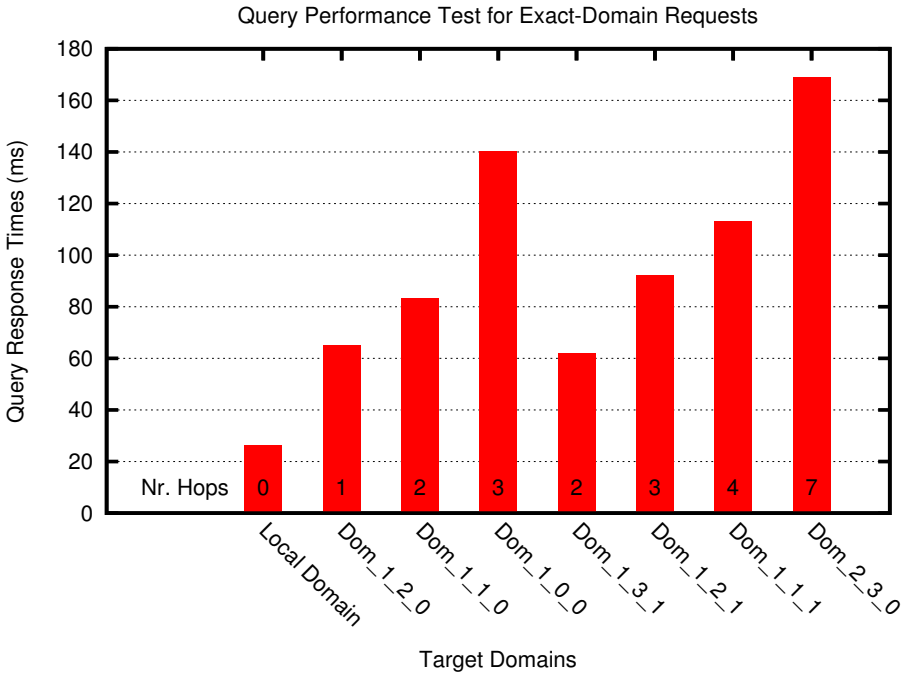

Figure 8.14: Request response times for the *exact-domain* queries carried out in the test. Labels on the bars show the number of hops (i.e. `CtxQueryHandler` agents) traversed by the query and its answer.

that the query has to traverse along its route. The 170 ms response time for the longest path (7 hops) shows that the routing overhead is completely manageable (e.g. a query for a *ContextAssertion* instance whose validity is of only one second will still be delivered in time for it to be still considered fresh information).

The graph shows that in the analyzed test run, the queries for the *ContextDomains* on a direct path to the hierarchy root (bars 2, 3 and 4 from left to right) take longer to reply than those to "sibling" domains of those on the direct path (bars 5, 6 and 7), even though in the case of $Dom_{1\_1\_1}$ the number of hops is greater (4) than that for $Dom_{1\_0\_0}$ (3).
This behavior is explainable by the way the CONSERT Engine functions. It is *not* caused by latencies in routing or actual processing, but rather by the time the query waits to be processed in the queue of the CONSERT Engine query execution thread (cf. Figure 4.7).
It is the same effect measured by the *insertion delay* and *inference delay* parameters for the CONSERT Engine reasoning performance tests discussed in Section 8.3.2. As explained previously, it is caused by the fact that the CONSERT Engine operates using full *ContextStore* transactions. If the queries were enqueued for execution when an insertion of temperature or luminosity updates was taking place, the request processing has to wait until those `WRITE` transactions are finished, before interrogating the *ContextStore*. This is why in Section 8.4 we talk about possible near-term future work improvements that address the current limitations of the CONSERT Engine execution cycle.

Figure 8.15 shows a snapshot of the agent interactions carried out as part of the query routing process for the longest possible path given our test setup ($Dom_{2\_3\_0}$ as target *ContextDomain*). The interaction log was captured using the Sniffer Agent[1] facility of the JADE platform. The Sniffer Agent can however only inspect the agents running in his own container (in this case, those running on machine 2). Since our experiments ran on two machines, Figure 8.15 shows

[1] http://jade.tilab.com/doc/tools/sniffer/html/intro.htm

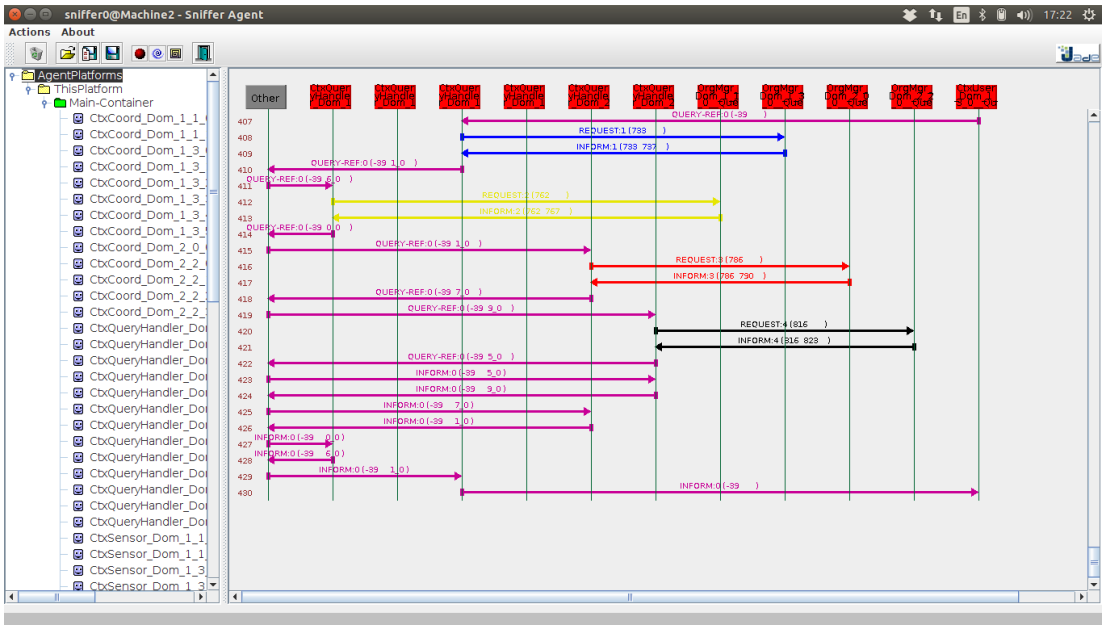

Figure 8.15: JADE Sniffer Agent snapshot of the sequence of agent interactions during the query routing protocol for the $Dom_{2\_3\_0}$ *ContextDomain* target.

all interactions with agents from CMUs on machine 1 as exchanges with *Other* (far left in the figure).
However, the depicted messages allow us to follow the steps of the *exact-domain* routing algorithm described in Section 6.4.1. The protocol starts with the request made by the `CtxUser` agent (shown on the far right) from the CMU of domain $Dom_{1\_3\_0}$. One can observe the query forwarding requests (using FIPA QUERY-REF calls) as magenta coloured arrows on the graph (i.e. all forwarded messages have the same conversation id). Notice the sequence of 7 QUERY-REF arrows (corresponding to the 7 required hops) that alternate between `CtxQueryHandler` agents from machine 1 and machine 2, until they reach the `CtxQueryHandler` from the CMU of domain $Dom_{2\_3\_0}$.
In between the magenta lines, one can observe the calls that are made by `CtxQueryHandler` agents to the `OrgMgr` agents of their respective CMUs, in order to determine the list of agents to which to forward the *exact-domain* query (cf. Algorithm 1).
The routing protocol ends with the FIPA INFORM messages (8 magenta lines at the end of the graph - 7 hops plus the response to the `CtxUser` agent) sent in reply along the same path followed by the forward query request.

The test on which we reported in this section aimed to establish the overhead in response time for the current implementation of domain-based query routing. The results show that the obtained delay is manageable and depends only on the number of hops through the *ContextDomain* tree hierarchy, in a linear fashion. However, response latency can sometimes increase due to the transaction based execution cycle of the CONSERT Engine, which determines us to investigate corresponding improvements in near-term future work.

8.4 Discussion

To conclude on the evaluation of the CONSERT Middleware we focus on two elements: a summarizing of the evaluation results and their interpretation, as well as a account of personal

experience of developing the reference scenario simulation using the CONSERT Middleware. The latter aspect in particular is one that motivates many of the future work directions that will be presented in the conclusions of this thesis.

8.4.1 CONSERT Middleware Evaluation Analysis

The reference scenario, even though reduced in scale and implemented as a simulated environment, featured many of the challenges that current and future context-aware applications are expected to encounter, as we mentioned in the introduction of this thesis.
In this chapter we have shown that the CONSERT Middleware is a suitable and often advantageous choice for the implementation of context management within an AmI application. For the most part, we provided a qualitative overview of usage of our CMM in the reference scenario simulation and one could observe that all context management challenges were well handled by CONSERT. Specifically, the scenario presented our CMM with issues requiring:

- the ability to model information in a flexible way (e.g. the n-ary *ContextAssertion* specifying the detected body posture of a person)
- the ability to use meta-properties to perform complex reasoning over context statements (e.g. the *ContextDerivationRule* for detecting ad hoc discussions)
- the ability to model, detect and resolve constraints stemming from contradictory context information (e.g. the availability uniqueness constraint)
- the ability to dynamically control the active context information update and inference processes (e.g. the provisioning control rules used in the scenario)
- the ability and need to perform consumption and production of context information at the same time (e.g. the relay of Alice's location information and person count to her personal context management CMU)
- the ability to structure the deployment of various context management control units throughout the simulation and exploit this structure for the purpose of context dissemination (e.g. *ContextDomain*-based deployment and domain-based queries used in the scenario)

Apart from the above, we performed a series of performance tests with the purpose of validating the effectiveness of the current CONSERT Middleware implementation. While the results from Sections 8.3.2 and 8.3.4 may be considered satisfactory, it is clear that improvements can and must be made in order to provide a middleware able to be used in real world application development.
In what follows we present and analyze some of the problems and solutions to those problems we can already identify.

Limitations of DerivationRule Writing A first concern relates to the definition of the *ContextDerivationRules* using the SPARQL syntax. In the current form, the SPARQL query that implements a rule is quite verbose, as can be observed in the examples from Listings 8.1 and 8.2. While our logical named graph separation of *ContextAssertion* instances and the *stores* that record their annotations are an advantage from an implementation point of view, the necessity to explicitly identify the graph names in the rule clauses could affect developer productivity significantly.
A custom CONSERT-specific interpretation engine for the SPARQL queries (as already explored by works such as [Anicic et al., 2011]) could help introduce syntactic sugar that drives down the complexity of *DerivationRule* writing.

Limitations of CONSERT Engine Execution A technical limitation observed in both Sections 8.3.2 and 8.3.4 was our usage of TDB transactions to allow consistent views of the context knowledge base when performing updates or inferences. This in turn translated into a limit on the number of *ContextAssertions* that can be updated per second, whilst still attaining decent real-time performance.
Besides considering to use alternative quadstore platforms (e.g. Sesame[1]) apart from Jena TDB to see if they provide different transactional behavior, one possible solution is to reconsider the update mechanism. The information in the CONSERT knowledge base has a good logical separation, given that every *ContextAssertion* has its own named graph identifier and the annotations of a given *ContextAssertion* class all reside within the same named graph. Thus, when performing a new insertion of a *ContextAssertion* it is possible to determine what logical partitions of the quadstore are going to be affected by the verification steps applied by the CONSERT engine during the insertion process. We can take advantage of this fact in attempting to implement custom locks or transaction behavior that creates views only for the named graphs concerned by an update. In this way update requests for different *ContextAssertions* could take place concurrently, since their locks or transaction views would not overlap.

8.4.2 Developing with the CONSERT Middleware

From a qualitative point of view, we report on the experience of developing the reference scenario simulation using CONSERT. This analysis is important, since one of the stated goals of this thesis is the ease of development and support for application engineering.

Modeling Experience Starting with the aspect of modeling the context information in the scenario, one noted advantage is the use of ontologies for the implementation of the context model. Model creation was aided by existing ontology development IDEs such as TopBraid Composer[2] or Protege[3] which allowed an easier management of the created model. In addition, the fact that the CONSERT Middleware allows a file based separation of the content, annotation, constraint, rule and function definition (cf. Section 6.2.2), provides additional structuring capability for the person developing the model.
However, despite the obvious advantage of IDE based development, during the creation of the context model for the reference scenario, it was still felt like the relations existing between the different instances of constructs from the CONSERT meta-model were not easily visible or manipulable. For example, there is currently no means to specify which annotations should be expected for instances of a given *ContextAssertion*. Likewise, while TopBraid Composer together with the SPIN API allow one to visually observe the context constraint templates attached to a given *ContextAssertion* type (i.e. an OWL ObjectProperty or Class), the impression is that the semantics of the CONSERT Meta-Model in itself escapes the capabilities of a general ontology development IDE (e.g. the various constraint templates are only identified as such because they are located within a designated constraint definition RDF file). Furthermore, the assignment of a corresponding constraint resolution service for a given constraint violation is deferred to the creation of another file (the provisioning coordination policy).

Provisioning Configuration Experience We find that implementation of a provisioning control and adaptation logic for an application is significantly aided by the existence of configuration files. Since the provisioning vocabulary is implemented as the CONSERT Provisioning Ontology, we found again support in existing ontology creation IDEs to edit the provisioning control parameters and rules for the context provisioning agents.

[1] http://www.openrdf.org/
[2] http://www.topquadrant.com/tools/modeling-topbraid-composer-standard-edition/
[3] http://protege.stanford.edu/

Specifically, the most useful fact is that the rules and parameters can be declaratively specified (as opposed to having to recompile built-in strategies after they are modified in-code) and it is thus easier to modify them and experiment with different parameter values, add or remove control rules.
Still, as in the case of modeling, it was felt that a visual editor giving the impression of a closer binding with the context model for which the provisioning policies are created would have reduced development time.

Middleware Deployment Experience As one could see from Table 8.1, a big number of deployment configurations was required even for a small scale scenario such as the one we presented. In this context, the ability to configure the deployment of CMUs for each simulated physical machine (e.g. AmI-Lab server, EF301 server, Alice's smartphone, projector unit) in a declarative way (i.e. as a configuration file) has positively affected the ease of development and the time taken to create each configuration.
Furthermore, we found that the existence of a deployment structure tied to the application context model and the packaging of the configuration files as OSGi bundles were of great support at simulation runtime, since it allowed an easy manipulation of the CMUs (tracking the existing configurations, managing their installation) with just a few lines of code.
As in the previous two cases, given that our CMU structuring is tied to application context model, it was felt that a custom editor and automatic builder of the configuration files would have reduced development effort even further.

Application Engineering Experience The previous paragraphs generally show that our experience in setting up the context *management* part of the simulation (which reduces to middleware configuration and deployment) is a positive and encouraging one.
However, the development of our scenario simulation involved writing code *around* the functionality of our middleware. Specifically, we refer to the implementation of `ContextAssertion Adaptors` and the code used to actually retrieve and relay context information from and to the middleware.

Writing of `ContextAssertionAdaptors` was definitely aided by the existence of service interfaces and the OSGi platform running underneath which allowed easy connection between an adaptor implementation and the `CtxSensor` agent that required it. The more time consuming task was the development of the adaptor logic used to retrieve data from the sensors simulated using the iCasa framework and converting that data into RDF statements built using the CONSERT Ontology.
Creating *ContextAssertion* content and annotations as RDF statements using the Apache Jena framework tends to be verbose. Consequently, wrapper code constructs that explicitly encapsulate the notions of *ContextAssertion*, *ContextAnnotation* or *ContextConstraint* and the relation existing between them according to the CONSERT Meta-Model represent an obvious development-time reducing solution.

The aspect of context information retrieval and manipulation was encumbered by the fact that queries had to be explicitly given in their SPARQL form. Many context middleware solutions from the ones reviewed in Chapter 3 fall short of providing a developer friendly way of accessing the context information managed by the respective middleware system. Currently, this is true for our CMM as well and it is an aspect that needs to be addressed quickly in future work.
Aside from this, we observed that the *ContextDomain*-based CMU structure of our CMM provides a good means to keep account of the different contextual interactions that are required by an application at runtime. However, due to the fact that response to a context query is obtained as an RDF result set, the relay of the relevant information instances from one CMU to another (i.e. context prosuming behavior) is made more difficult.
Both of the above observations suggest that improvements such as automatic query generation

for common *ContextAssertion* retrieval tasks, as well as templates or helper classes for inter CMU context information transferring would greatly benefit development efforts.

From the previous experience accounts it is clear that the CONSERT Middleware offers encouraging advantages in terms of AmI application engineering support. Yet in order to really become a productive solution for developers, it needs to be accompanied by modeling and configuration tools and automatic code generation functionality that closely follow the meta-model semantics and architectural design of the middleware, so as to alleviate boilerplate application development tasks. This insight motivates an important part of the envisioned future work, which will be explored in the next chapter.

Chapter 9

Conclusions

This chapter concludes the presented thesis. In closing, we will go over the initial objectives of this work and provide a holistic overview of the contributions brought to obtain these objectives (Section 9.1).
The work in this thesis has resulted in the creation of the CONSERT middleware for context management. Furthermore, as noted in the evaluation chapter, the possible improvements of the system bring about many perspectives for future work. In Section 9.2 we explore these perspectives, from conceptual, implementation and vision related points of view.

9.1 Contributions

This thesis has focused on designing and implementing a Context Management Middleware able to provide good support for the anticipated rapid innovation-oriented Ambient Intelligence projects.
Consequently, the central features that characterize our middleware are **flexibility in modeling and deployment**, as well as **explicit and declarative configuration**. Semantic Web technologies, Multi-Agent System design principles and techniques, as well as component-based software development are the key enabling factors that allow us to achieve these properties.

In the following we perform a recapitulation of the main take-away points from each chapter of this thesis (Section 9.1.1) and then present a list of point-wise specified contributions (Section 9.1.2).

9.1.1 Building a Flexible Context Management Middleware

In Chapter 1, we described **Ambient Intelligence** as being the **main research field** of which **context management** is an **integral and essential part**. We discussed the fact that current trends show that AmI is no longer a subject of pure technological research, but that it is expected to be the **object of many industry-backed innovation projects** (Section 1.1.1). Analyzing different kind of scenarios in Section 1.1.2, we identified a series of **challenges for the aspect of context management** within AmI application. We grouped these challenges according to **three different aspects** that characterize a context management system: **context modeling**, **context provisioning** and **context management deployment** (Section 1.2).
We determined that in order to be able to address the expected development trend of the Ambient Intelligence domain, the **key focus points** of the solution we develop should be **flexibility**

of modeling and deployment, **modularity of design** and **ease of configuration and development**.

In Chapter 2 we started to explore the requirements (in terms of information representation and reasoning) that make an approach successful for the specific task of context modeling. We then reviewed the solutions that have been proposed in the literature for this task. We covered the different types of representation approaches, from simple key-value pairs to **ontologies and meta-modeling solutions** and explained why we give more attention to the latter. We studied various reasoning methods and argued that **rule-based inferencing** provides the **best trade-off between expressiveness and understandability of a reasoning procedure and the effectiveness of the employed deduction mechanism**.
The analysis of reviewed works revealed a **need for an expressive, uniform representation model** able to capture **aspects of context content, meta-properties and dependencies**, as well as a reasoning approach combining **rule-based and ontology-based inferences**, while also sporting **temporal deduction capabilities**.

Chapter 3 then focused on how context models were incorporated into architectures for context management system. We investigated the steps of the main **context provisioning life cycle** (*acquisition, coordination, dissemination, usage*) and identified the set of **transverse (complementary) functionality blocks** (*producer discovery, mobility, adaptation management*) and **non-functional characteristics** (*support for heterogeneity, scalability, privacy and security, traceability/control, robustness, ease of deployment/configuration*) required of a context management system.
The subsequent review of context management solutions in terms of provisioning architecture and deployment capabilities discovered the fact that many approaches provided seemingly adequate responses to the identified context management requirements. However, the **main drawback of almost all reviewed works** was that they were **enclosed, monolithic systems** offering **little support for declarative, designer-friendly means of specifying how they should be employed within an application**.
These facts constituted the motivation for the **objectives** presented at the end of the chapter: **agentification of the provisioning architecture** and **component-based design** in support of **middleware flexibility and modularity**, as well as **declarative policy-based means** to **guide provisioning** behavior and **specify deployment configurations**.

Chapters 4 through 6 presented the conceptual overview of the contributions brought with the intent of achieving our objectives.
In terms of representation and reasoning (Chapter 4), we started with the presentation of the formal context meta-model we are proposing. The purpose of the formalization was to introduce the modeling concepts and the way they relate to one another. We then showed that semantic web technologies, namely ontology-based representation and SPARQL as a rule-based inference language, are an appropriate choice for the implementation of the CONSERT meta-model.
Noteworthy characteristics of the CONSERT context modeling approach are its **flexibility** (e.g. *ContextAssertions* of variable arity, **defining *ContextAnnotation* types** and the **semantics of how such annotations are combined during runtime inference**) and **uniformity** (**ontology-based implementation** of context information **content**, **annotations** and **constraint definitions**).
The CONSERT Engine is implemented in such a way as to exploit the characteristics of the CONSERT meta-model. It uses an **execution cycle** that emphasises **temporal continuity** of sensed or derived *ContextAssertions*, leading to the identification of **semantically distinguishable situations**. The engine offers support for **expressive rule-based reasoning**, since the rule-based derivation formalism is translated to **SPARQL inference queries** which allow **complex expressions such as aggregation and grouping**.
The CONSERT Engine is also implemented as an **extensible service component**. It presents **control options** that affect the reasoning process (e.g. integrate ontology-based reasoning into the rule-processing cycle, clean up runtime history) and uses **pluggable services** that imple-

ment **derivation scheduling** and **constraint resolution**.

The CONSERT Engine becomes a central piece of the **agent-based context provisioning architecture** of the **CONSERT Middleware** described in Chapter 5. We explained that, in the context of goals for *flexible deployment* and *configuration*, the **characteristics of multi-agent oriented programming** (*autonomy, social ability, reactivity, pro-activity*) are an **optimal fit**. The proposed multi-agent defines agent types (**`CtxCoord`, `CtxQueryHandler`, `CtxSensor`, `CtxUser`**) that **encapsulate the functionality** of each main context management life cycle aspect.
Control of the provisioning process is specified through **declarative policies** that govern the **functionality** of the **`CtxSensor`** and **`CtxCoord` agents** and the messages exchanged between agents (**provisioning interaction protocols**).

The multi-agent based architecture offers the ability of a flexible deployment, that is, different agents can run on different machines. Chapter 6 presented the means by which an **explicit deployment structure** can be configured. We introduced the concepts of ***ContextDimension*** and ***ContextDomain*** as the elements that **bind a deployment structure** to the **multi-dimensionality of the application context model**. The defined concepts permit **two deployment schemes**: **centralized** and **decentralized** (**with the ability to form a tree-based hierarchy of *ContextDomains***).
Context provisioning agents defined in Chapter 5 are **assembled into Context Management Units (CMU)** which are assigned to **handle one or several aspects of provisioning** (acquisition, coordination, dissemination, usage) pertaining to the **contextual interactions within a *ContextDomain***.
We then introduced the **`OrgMgr` agent** as the entity that **supervises the lifecycle** (install, start, stop, uninstall) of the agents within a CMU. Furthermore, we explained that **declarative deployment policies** allow the application designer to **specify the configuration** in terms of: **ContextDomain settings** (context model of the domain, domain hierarchy of which it may be part), **CMU agent composition** for the given *ContextDomain* and **individual agent specifications** (e.g. policy files that define agent behavior).
All these **deployment options** confer the CONSERT Middleware the **ability to be used** in **multiple scenario situations** of **different scale sizes**. On hand of the reference scenario, we explained how our CMM enabled **complex, but structured prosuming behavior** and **effective distributed query routing**.

In Chapter 7 the **service component based** implementation of the CONSERT Middleware became apparent as a means to ensure additional **modularity** and **runtime flexibility**.
We showed how **ontology-based vocabularies** are used to **give shape to context provisioning** and **deployment policies** and how these **CMU configurations** are packaged as **OSGi bundles**. **Adaptors and services** (e.g. `ContextAssertion Adaptor` for sensing, `ConstraintResolution Services`) used by context provisioning agents are implemented in an **OSGi-compatible** way.
The **component-based middleware design** and **bundle-based configuration packaging** offer **benefits** in terms of **runtime management and tracking of deployed CMUs**. Furthermore, we explained why the **JADE agent development language** is a good fit for the proposed agent functionality in terms of behavior and interaction implementation, as well as agent deployment infrastructure support.

Finally, Chapter 8, evaluated the contributions of this thesis on hand of the reference scenario. We **simulated the environment** of the scenario and offered a **complete application implementation** based on the CONSERT Middleware functionality.
The **qualitative evaluation** of modeling, provisioning and deployment aspects showed **substantial application engineering benefits** stemming from the flexibility of context model design and declarative specification of provisioning and deployment policies.
Quantitative evaluation of CONSERT Engine reasoning and distributed query handling

performance showed **adequate runtime performance** and **identified necessary improvement requirements**.
The directions of future work arising from evaluation discussions are presented later in Section 9.2.

9.1.2 Contribution Summary List

The main contributions of this work can be point-wise grouped into the categories shown below and summarized by the following list:

- **Context Modeling and Reasoning**:
  - Ontology-based Context Meta-Model providing *uniform representation support* for the main context modeling concerns: content, annotation, dependencies. Main difference with state-of-the-art is the *uniformity* of representation which leads to ease of conception for context model developers. (Section 4.2)
  - Extensibility in terms of *annotation definition* (this includes both the *kind* of annotation, as well as the functions that implement the annotation inference operators - Section 4.2.2)
  - Definition of annotation *inference usage semantics* (a specification of *how* and *when* annotation information is combined during inference). This is not explicitly handled in related works (Sections 4.2.2 and 4.4.2).
  - Reasoning cycle which includes automatic computation of temporal continuity of events, leading to *semantically distinguishable situations*. This facilitates complex situation definition, and is different from state-of-the-art, because the mechanism is *implicit*, rather than *on-demand* (Section 4.4.2).
  - Reasoning engine supporting customizable *constraint resolution* services and definition of derivation rule scheduling heuristics. We exploit our component-based design to provide both default services, as well as configurable customizations. The difference from state-of-the-art lies in the offered customization support (Sections 4.4.1 and 7.2.2).
- **Context Information Provisioning**:
  - Agentification of context provisioning units (sensing, coordination, dissemination, usage). Difference with state-of-the-art lies in better encapsulation and the *potential* for increased autonomy of individual provisioning units (Section 5.1.2).
  - Declarative policy-based control specification for the context provisioning process. Difference with state-of-the-art is in providing the policy vocabulary, which helps reduce development effort (Section 5.2).
- **Context Management Middleware Deployment**:
  - Definition of concepts that use the dimensionality of a context model to induce *decentralized structure of context management units (CMUs)*. Difference with state-of-the-art is that we provide a notion of *ContextDomain* which is *tied* to a context model. This guides application design and eases application development by making an explicit mapping between a CMU and a *ContextDomain* (Section 6.1).
  - Vocabulary for *declaring* a context provisioning deployment structure (Section 6.2).
  - Support for *runtime management* of *CMU life cycles*. This aspect is not explicitly addressed in state-of-the-art and is a consequence of our component-based design (Sections 6.3 and 7.4).

- **Service component based Middleware Implementation**:
  - OSGi-compatible implementation of the CONSERT Engine, the services it uses (e.g. inference scheduling, constraint resolution) and of the adaptors used by provisioning agents to interact with sensor and application layers. This brings an advantage in term of modularization and runtime management (e.g. set/alter service implementation) of deployed agents and services (Section 7.2).
  - OSGi Bundle based packaging of CMU configuration, giving the capability of tracking and managing possible CMU deployments at runtime (Section 7.4).

9.2 The Future of the CONSERT Middleware

In terms of future work, we consider two aspects. The first one refers to improvements that are easily observable and that constitute near-term objectives. The other aspect refers to more elaborate visions of development which will require more time to investigate, but which have an increased potential of rendering the CONSERT Middleware as a noteworthy tool for context-aware application development.

9.2.1 Improvements there for the taking

We discuss the near future improvement objectives grouped by the aspects of context management as they were presented in this thesis.

Model related Regarding the CONSERT context meta-model, one potential line of work revolves around the idea of a better structure for *ContextAnnotations* and a more elaborate set of *ContextAnnotation* types.
The need for structuring is based on observations made by Marie et al., who explain that, despite the more than a decade long research into context modeling, a consensus has not been reached as to what constitutes an exhaustive list of quality of context (QoC) criteria [Marie et al., 2013]. Their proposal to this problem, as in our case, is that of a meta-model for context annotations. However, their QoC modeling efforts include aspects of computability and ability to more easily and generically manipulate meta-properties, which are currently absent from our annotation model. They therefore constitute the object of immediate possible extensions.
Furthermore, aside from aspects of QoC, *ContextAnnotations* can serve additional information processing purposes, as for example in works such as [Henricksen and Indulska, 2004a; Henricksen et al., 2005c]. Henricksen et al. propose models that capture *context access control* indications and *user preference* specifications (e.g. the preference of a user for a given value of a *ContextAssertion* statement). Though the authors of these papers do not mention if these modeling efforts are part of a more general attempt to represent context meta-properties, the methods they propose could be adapted and included in our *ContextAnnotation* extension endeavours. This would grant the `CtxCoord` agent the ability to understand the nature of context annotation information provided by `CtxSensor` or `CtxUser` agents to a greater extent and, therefore, make more informed provisioning control decisions.

CONSERT Engine related Improvements to the CONSERT Engine relate both to a bettering of existing functionality, as well as to the addition of new processing capabilities. Some improvements to existing services (e.g. syntactic sugar to reduce complexity of *ContextDerivationRules*, alternatives to the current transaction-based processing of the CONSERT Engine) have already been suggested in Section 8.4.

Other possible optimizations concern the performance of *ContextDerivationRule* and subscription query execution. Currently, both rule and query executions are being triggered every time an update to a *ContextAssertion* type referenced in the body of the rule/query occurs. However, the analysis made in [Kang et al., 2008] shows that the possibility exists where updates to *ContextAssertion* instances (especially environment sensing such as temperature, noise, light level, etc) will not actually trigger any changes in the outcome of registered rules or queries. The authors therefore propose performing a more thorough analysis of query or rule trigger conditions and creating an index of these elements based on "change points", i.e. regions in the value domain of a *ContextAssertion* where the output of a rule/query changes.
Furthermore, in his thesis, Gero Mühl proposes different optimization techniques for content-based publish/subscribe systems [Mühl, 2002] (of which our CONSERT Engine is an example). For example, one option to reduce the number of registered subscriptions is to perform a *query coverage* analysis, that is, determine whether the conditions in a subscription made by a new client are already "covered" by an already existing (but not necessarily identical) one.
Both suggestions are subject of near-term investigation and will require an improvement to the current *ContextDerivationRule* and query analysis capabilities of the CONSERT Engine.

Lastly, an alternative, but more elaborate, approach to CONSERT Engine optimization is the idea to investigate the possibility of adapting/extending existing SPARQL-based RDF stream processors (e.g. C-SPARQL [Barbieri et al., 2010], EP-SPARQL [Anicic et al., 2011], ETALIS [Teymourian et al., 2012] or INSTANS [Rinne et al., 2012]) to/with the execution cycle of the CONSERT Engine (most notably, the handling of *ContextAnnotations* and the continuity check).
Furthermore, one existing feature of the CONSERT Engine which has not been thoroughly explored and evaluated in the current experiments is the aspect of mixed ontology and rule-based reasoning. Specifically, we are interested in offering application developers clearer guidelines and means to specify how ontology-based reasoning integrates with the event-processing behavior of the CONSERT Engine. In this sense, an analysis of the functionality of EP-SPARQL, for example, could help provide valuable insights into issues and performance trade-offs that are to be expected in this attempt.

Context Provisioning related The provisioning specific aspect that we wish to improve in the near term concerns the introduction of an additional provisioning agent: the `CtxAggregator`. In some of the works reviewed in chapter 3 (e.g. [Sehic and Dustdar, 2010], [Conan et al., 2007], [Chen et al., 2008]), the proposed context management solutions offer a pre-processing step (e.g. simple value filters, computation of average/min/max values from several sensors providing the same *ContextAssertion* type), before more complex reasoning is applied to information coming from sensors.
Though it could be argued that some of this functionality could be taken care of by corresponding implementation of `ContextAssertionAdaptors`, the objective of flexibility and modularity in middleware design leads us to the desire of creating a special provisioning agent who would be assigned to manage such tasks. A `CtxAggregator` agent would have policies specifying the *ContextAssertion* types and the nature of the pre-processing operations that it needs to manage. Furthermore, at deployment specification level, `CtxSensor` agents would no longer connect only to `CtxCoord` agents. Instead, it will be possible for several `CtxSensor` and `CtxAggregator` agents to constitute a processing pipeline, before the *ContextAssertions* managed by these agents reach the `CtxCoord` where reasoning on hand of the CONSERT Engine can be applied.

Context Middleware Deployment related One of the deployment specific topics which requires immediate attention is the appropriate support for mobility and resource discovery management. In Section 6.5 we explained that these functionality aspects are currently incompletely handled by the CONSERT Middleware. While the existence of *ContextDomainEn-*

tered and *ContextDomainLeft* rules provides some level of support for detecting *ContextDomain* changes, these features are not yet fully fledged, nor were they evaluated by the current simulation experiments.
Further work is needed to ensure adequate support for mobility management to application developers. On the one hand, given our *ContextDimension* and *ContextDomain* based deployment structure, we consider it relevant to study the practical translation of the operational concepts of *change-of-focus* and *shift-of-attention* defined in [Zimmermann et al., 2007] into actions that manage the lifecycle of deployed CMUs (e.g. start/stop an installed CMU, install another CMU in place of an existing one to handle the context information that is now in focus). Some support for this idea, but *within* the frame of a single CMU, is already available under the form of context provisioning rules, which can inspect the *ContextStore* and try to "anticipate" the change in the relevance (i.e. shift-of-attention) of certain *ContextAssertion* updates coming from `CtxSensor` agents that are part of the CMU.
On the other hand, besides these conceptual inquiries, the API provided by the `Application Client Adaptor` exposed by a `CtxUser` agent needs to be augmented with functionality that allows the application to take command of session handover during mobility (e.g. decide how the subscriptions and ongoing queries that were submitted to the `CtxQueryHandler` of an existing *ContextDomain* are transferred over to the query handler of the new *ContextDomain* during a detected domain change operation).

Another deployment related improvement stems from the observations discussed in Section 8.4.2, where we talk about the experience of developing applications with the CONSERT Middleware. One concern was tied to the prosumer behavior of the `CtxUser` agent and the ability to more easily (i.e. have clearer and more straightforward code support) perform the relay of context information from a CMU responsible for one *ContextDomain* to another.

Finally, in terms of domain-based query management, the tests discussed in Sections 8.3.3 and 8.3.4 measured that the overhead in query response times introduced by the CONSERT Middleware routing protocol is reasonably negligible. However, the experiment only attempted to evaluate the routing procedure itself. Further, more complex and real application based evaluations are required to determine the throughput and load handling capabilities of the current `CtxQueryHandler` implementation. This too constitutes the object of near term investigation.

9.2.2 Hidden Potentials

The ideas presented previously constitute extensions of the CONSERT Middleware which are mostly based on its current design and functionality and can therefore be regarded as near future investigations.
The "hidden potentials" aspects that we want to discuss here relate more closely to the goals expressed in the introduction to this thesis, namely to provide a context management middleware solution that is able to address the trend towards technological innovation in Ambient Intelligence, by means of strong support for application development. As in the previous section, we present the features that we have in mind grouped by context management aspects.

Exploiting the CONSERT Meta-Model The first feature we envision is the introduction of a visual context model development environment, i.e. an IDE for context modeling on hand of the CONSERT Ontology. This idea was already noted as a nice-to-have feature in the analysis from Section 8.4.2. As opposed to simple ontology editor software, this IDE would give the developer a much clearer overview of the context model he is developing, since it would be directly aligned with the CONSERT meta-model. Furthermore, the fact that an OSGi-based packaging of the resulting model files has been established means that the output of the IDE build process is already known and can be readily exploited within applications.

The beginning of development of this feature is already planned under the form of student internships that will start this summer within the AI-MAS laboratory of the University Politehnica of Bucharest.

Exploiting the agent-based provisioning architecture When we performed the motivation for using multi-agent based design principles and technologies, we mentioned that one advantage of using MAS techniques is the potential for increased autonomy of individual provisioning agents.
In the chapter on problem definition, we presented challenges related to management of body worn sensors (e.g. in the "Care for the elderly" scenario), where the need to employ the right sensors at the right moment became clearly apparent. This need was largely driven by the context usage patterns, i.e. the nature of queries and subscriptions sent to a context management system by the context consumption clients (e.g. the smartphone of the elder).
One possibility to implement management functionality as the one described above revolves around the idea of Context Level Agreements (CLA). CLAs are the context-management equivalent of service level agreements (SLA) in current network- or web-service implementations. They have already been explored in the context management literature in works such as [Khedr and Karmouch, 2004].
However, in addition to what is proposed by Khedr and Karmouch, our vision for CLAs in the CONSERT Middleware would include *policy-based goals* of `CtxCoord` and `CtxSensor` agents. For example, a `CtxCoord` could have a goal of executing the least amount of *ContextDerivationRules* as possible, while a `CtxSensor` agent could be configured to try and limit the amount of power spent sending *ContextAssertion* updates.
Thus, when a `CtxUser` agent would make a subscription request, the agreement would involve determining *which DerivationRules* and *which* `CtxSensor` agents would be required to function and in *what way*, so as to adequately respond to the user request (for example in terms of providing query answers of acceptable accuracy and within acceptable delay). The establishing of such CLAs, could then help create an execution strategy (who is active, when and how) for all the `CtxSensor` agents coordinated by the `CtxCoord` in the same CMU.
Still, important research questions arise. For example, an interesting consideration is how new requests are handled. Does determining a new CLA consider only currently available resources or can existing CLAs be modified/altered in favour of the new one? On what criteria would such concessions be made?
This is why the aspect of CLAs is left as an element of long-term future work.

Exploiting the declarative configuration options This aspect is closely tied to the IDE for context model development. The idea is to extend the previous IDE with complementary functionality such that the IDE build process would be able to generate desired CMM deployment scheme. This relies on the fact that within CONSERT Middleware, such schemes are tied to the existing context model. Once again, the OSGi based packaging of deployment and provisioning policies means that the output of the editor build process is already clearly determined.

Exploiting the OSGi-based implementation The ideas discussed next come back to the challenges discussed initially, in the chapter on problem definition. One specific set of inquiries was based on the "Maria" scenario where we saw that many, unrelated and briefly-used services were employed by the same application running on Maria's smart watch. Consequently, the main questions raised during the discussion on that scenario focused on the very aspect of how such an application would be engineered, given that it relied on a sense of global ubiquity (insofar as the AmI infrastructure in the foreign country would seamlessly interact with Maria's smart watch which would have been developed by entirely different engineers).
One possible vision regarding the context management aspects of the "Maria" scenario is based

on the deployment modularity and flexibility features of the CONSERT Middleware and its OSGi-based implementation in particular. We have seen that a CMU is conceptually designed as a control encapsulation unit for the context management requirements of a given *ContextDomain*. Furthermore, each *ContextDomain* comes with its own context model definition.
This brings to mind the perspective of a *CONSERT App Store* similar in functionality to Google Play[1] or the iOS App Store[2]. In the CONSERT App Store, CMU code (e.g. for the different adaptor implementations) and configuration bundles become downloadable for the contextual interactions specific to a place, activity, organization or service.
The end user device would only be required to host a *CONSERT Base Platform* where the CMUs could be installed and their lifetime managed. The user could manage the downloads him-/herself or, as in the extended reference scenario from Section 6.1.1, there could be a bootstrap CMU that could detect the various *ContextDomains* and demand the installation of corresponding CMU configurations from the CONSERT App Store.
Furthermore, services or organizations could be configured to automatically *push* the CMU bundles that would be required for contextual interactions within their domain. Engineering support for the latter idea has even already been developed in [Boujbel et al., 2014], where a domain specific language (MuScADeL) for multiscale and autonomic software deployment is presented.
Altogether, the vision of the CONSERT App Store would provide a practical and feasible solution to the questions raised in the "Maria" scenario.

[1] https://play.google.com/store
[2] store.apple.com/us

List of Publications

The work from and leading up to this thesis has been published in 9 research papers, of which 1 in an ISI indexed journal, 4 in ISI indexed conference proceedings and 4 in the proceedings of international, peer-reviewed conferences. All papers were published as first author.

Sorici, A., Boissier, O., Picard, G. and Santi, A. (2011, October). Exploiting the jacamo framework for realising an adaptive room governance application. In Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11 (pp. 239-242). ACM.

Sorici, A., Picard, G., Boissier, O., Santi, A. and Hübner, J. F. (2012, June). Multi-Agent Oriented Reorganisation within the JaCaMo infrastructure. In Proceedings of The Third International Workshop on Iinfraestructures and tools for multiagent systems: ITMAS (pp. 135-148).

Sorici, A., Boissier, O., Picard, G., and Zimmermann, A. (2013). Applying semantic web technologies to context modeling in ambient intelligence. In Evolving Ambient Intelligence (pp. 217-229). Springer International Publishing. (ISI Proceedings)

Sorici, A., Picard, G., and Boissier, O. (2014, September). Towards an Agent enabled Context Management Middleware. In Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing (pp. 1-2). ACM.

Sorici A., Picard G., Boissier O., Zimmermann A. and Florea A. (2015). CONSERT: Applying semantic web technologies to context modeling in ambient intelligence. In Computers and Electrical Engineering: In Press, Accepted Manuscript, DOI: 10.1016/j.compeleceng.2015.03.012. (ISI Indexed Journal)

Sorici, A., Boissier, O., Picard, G., and Florea, A. M. (2015). Policy-based Adaptation of Context Provisioning in AmI. In 6th International Symposium on Ambient Intelligence (ISAmI'15). Springer. (ISI Proceedings)

Sorici, A., Boissier, O., Picard, G., and Florea, A. M. (2015). Multi-Agent based Context Provisioning Deployment in AmI Applications. In 13th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'15). Springer. (ISI Proceedings)

Sorici, A., Picard, G., and Florea, A. M. (2015). Multi-agent based context management in AmI applications. In International Workshop on Agent Technology for Ambient Intelligence at the the 20th International Conference on Control Systems and Computer Science (CSCS). IEEE CPS. (ISI Proceedings)

Sorici, A., Picard, G., Boissier, O., and Florea, A. M. (2015). Gestionnaire multi-agent de contexte pour les applications d'intelligence ambiante. In 23es Journées Francophones sur les Systèmes Multi-Agents (JFSMA'15). Cépaduès.

Bibliography

Aguiar, R. L., Sarma, A., Bijwaard, D., Marchetti, L., and Pacyna, P. (2007). Pervasiveness in a competitive multi-operator environment: the daidalos project. *Communications Magazine, IEEE*, 45(10):22–26.

Alonso, E., D'inverno, M., Kudenko, D., Luck, M., and Noble, J. (2001). Learning in multi-agent systems. *The Knowledge Engineering Review*, 16(03):277–284.

Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N. (2011). Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM.

Augusto, J. C., Nakashima, H., and Aghajan, H. (2010). Ambient intelligence and smart environments: A state of the art. In *Handbook of ambient intelligence and smart environments*, pages 3–31. Springer.

Balazinska, M., Balakrishnan, H., and Karger, D. (2002). Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Pervasive Computing*, pages 195–210. Springer.

Baldauf, M., Dustdar, S., and Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263.

Barbieri, D. F., Braga, D., Ceri, S., VALLE, E. D., and Grossniklaus, M. (2010). C-sparql: a continuous query language for rdf data streams. *International Journal of Semantic Computing*, 4(01):3–25.

Bates, J. et al. (1994). The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125.

Bellavista, P. and Corradi, A. (2012). A survey of context data distribution for mobile ubiquitous systems. *ACM Computing Surveys (...*, 44(4):1–45.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., and Riboni, D. (2010). A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180.

Bettini, C., Maggiorini, D., and Riboni, D. (2007). Distributed context monitoring for the adaptation of continuous services. *World Wide Web*, 10(4):503–528.

Bevan, N. (2009). Usability. In *Encyclopedia of Database Systems*, pages 3247–3251. Springer.

Bikakis, A. and Antoniou, G. (2010). Defeasible contextual reasoning with arguments in ambient intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1492–1506.

Bikakis, A., Antoniou, G., and Hasapis, P. (2011). Strategies for contextual reasoning with conflicts in ambient intelligence. *Knowledge and Information Systems*, 27(1):45–84.

Bikakis, A. and Patkos, T. (2008). A survey of semantics-based approaches for context reasoning in ambient intelligence. *Constructing Ambient Intelligence*, pages 14–23.

Bolchini, C., Curino, C. a., Quintarelli, E., Schreiber, F. a., and Tanca, L. (2007a). A data-oriented survey of context models. *ACM SIGMOD Record*, 36(4):19.

Bolchini, C., Schreiber, F. A., and Tanca, L. (2007b). A methodology for a very small data base design. *Information Systems*, 32(1):61–82.

Bordini, R. H., Dastani, M., Dix, J., and Seghrouchni, A. E. F. (2005). *Multi-Agent Programming*. Springer.

Boujbel, R., Rottenberg, S., Leriche, S., Taconet, C., Arcangeli, J.-P., and Lecocq, C. (2014). MuScADeL: A Deployment DSL Based on a Multiscale Characterization Framework. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, pages 708–715. Ieee.

Brdiczka, O., Crowley, J. L., and Reignier, P. (2009). Learning situation models in a smart home. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 39(1):56–63.

Broda, K., Clark, K., Miller, R., and Russo, A. (2009). SAGE: A logical agent-based environment monitoring and control system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5859 LNCS:112–117.

Brodie, M. L. (1984). On the development of data models. In *On conceptual modelling*, pages 19–47. Springer.

Brown, P. J. (1995). The stick-e document: a framework for creating context-aware applications. *ELECTRONIC PUBLISHING-CHICHESTER-*, 8:259–272.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284.

Buchholz, S., Hamann, T., and Hubsch, G. (2004). Comprehensive structured context profiles (cscp): Design and experiences. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 43–47. IEEE.

Buchmann, A. and Koldehofe, B. (2009). Complex event processing. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 51(5):241–242.

Bucur, O., Beaune, P., and Boissier, O. (2006). Steps towards making contextualized decisions. *Lnai*.

Calvanese, D., Lenzerini, M., and Nardi, D. (1998). Description logics for conceptual data modeling. In *Logics for databases and information systems*, pages 229–263. Springer.

Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):247–267.

Chang, K.-H., Chen, M. Y., and Canny, J. (2007). *Tracking free-weight exercises*. Springer.

Chen, G., Li, M., and Kotz, D. (2008). Data-centric middleware for context-aware pervasive computing. *Pervasive and Mobile Computing*, 4:216–253.

Chen, H., Finin, T., and Joshi, A. (2003). An intelligent broker for context-aware systems. *Adjunct proceedings of Ubicomp*, pages 183–184.

Chen, H., Finin, T., and Joshi, A. (2005). The SOUPA Ontology for Pervasive Computing. *Computing Systems*, pages 233–258.

Chen, H., Finin, T., Joshi, A., Kagal, L., Perich, F., and Chakraborty, D. (2004a). Meet the Semantic Web in Smart Spaces. *IEEE Internet Computing*, 8(October):69–79.

Chen, H., Perich, F., Finin, T., and Joshi, A. (2004b). SOUPA: Standard ontology for ubiquitous and pervasive applications. *Proceedings of MOBIQUITOUS 2004 - 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pages 258–267.

Chen, L. and Nugent, C. (2009). Ontology-based activity recognition in intelligent pervasive environments. *International Journal of Web Information Systems*, 5(4):410–430.

Conan, D., Rouvoy, R., and Seinturier, L. (2007). Scalable processing of context information with cosmos. In *Distributed Applications and Interoperable Systems*, pages 210–224. Springer.

Cook, D. J., Augusto, J. C., and Jakkula, V. R. (2009a). Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277–298.

Cook, D. J., Augusto, J. C., and Jakkula, V. R. (2009b). Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277–298.

Corradi, A., Fanelli, M., and Foschini, L. (2010). Adaptive context data distribution with guaranteed quality for mobile environments. In *Wireless Pervasive Computing (ISWPC), 2010 5th IEEE International Symposium on*, pages 373–380. IEEE.

Courtrai, L., Guidec, F., Le Sommer, N., and Mahéo, Y. (2003). Resource management for parallel adaptive components. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE.

Da Rocha, R. C. A. and Endler, M. (2012). *Context management for distributed and dynamic context-aware computing*. Springer Science & Business Media.

Dargie, W. (2007). The role of probabilistic schemes in multisensor context-awareness. *Workshops, 2007. PerCom Workshops' 07. Fifth*, pages 1–6.

Dey, A. K. (2001). Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7.

Doukas, C., Maglogiannis, I., Tragas, P., Liapis, D., and Yovanof, G. (2007). Patient fall detection using support vector machines. In *Artificial Intelligence and Innovations 2007: from Theory to Applications*, pages 147–156. Springer.

Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., and Burgelman, J.-C. (2001). *Scenarios for ambient intelligence in 2010*. Office for official publications of the European Communities.

Ejigu, D., Liris-umr cnrs, L., Lyon, I. D., Scuturici, M., and Brunie, L. (2008). Hybrid Approach to Collaborative Context-Aware Service Platform for Pervasive Computing. *Proceedings of the IEEE*, 3(1):40–50.

El Fallah Seghrouchni, A. and Suna, A. (2005). Claim and sympa: A programming environment for intelligent and mobile agents. In Bordini, R., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 95–122. Springer US.

Fuchs, F., Hochstatter, I., Krause, M., and Berger, M. (2005). A metamodel approach to context information. *Third IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom 2005 Workshops*, 2005:8–14.

Giunchiglia, F. and Serafini, L. (1994). Multilanguage hierarchical logics, or: how we can do without modal logics. *Artificial intelligence*, 65(1):29–70.

Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. (2003). Description logic programs: Combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web*, pages 48–57. ACM.

Gross, T. and Specht, M. (2001). Awareness in context-aware information systems. In *Mensch & Computer 2001*, pages 173–182. Springer.

Gu, T., Pung, H. K., and Zhang, D. Q. (2005a). A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18.

Gu, T., Qian, H., Yao, J. K., and Pung, H. K. (2003). An architecture for flexible service discovery in octopus. In *Computer Communications and Networks, 2003. ICCCN 2003. Proceedings. The 12th International Conference on*, pages 291–296. IEEE.

Gu, T., Tan, E., Pung, H., and Zhang, D. (2005b). A peer-to-peer architecture for context lookup. *The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pages 333–341.

Gu, T., Wang, X. H., Pung, H. K., and Zhang, D. Q. (2004). An Ontology-based Context Model in Intelligent Environments. In *in Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 270–275.

Guo, B. and Zhang, D. (2010). The architecture design of a cross-domain context management system. *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 499–504.

Hearst, M. A., Dumais, S. T., Osman, E., Platt, J., and Scholkopf, B. (1998). Support vector machines. *Intelligent Systems and their Applications, IEEE*, 13(4):18–28.

Henricksen, K. (2003). *(Thesis) A framework for context-aware pervasive computing applications,*. PhD thesis, The University of Queensland.

Henricksen, K. and Henricksen, K. (2006). Indulska, J.: Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2(July 2005):37–64.

Henricksen, K. and Indulska, J. (2004a). A software engineering framework for context-aware pervasive computing. *Proceedings - Second IEEE Annual Conference on Pervasive Computing and Communications, PerCom*, pages 77–86.

Henricksen, K. and Indulska, J. (2004b). Modelling and using imperfect context information. *Proceedings - Second IEEE Annual Conference on Pervasive Computing and Communications, Workshops, PerCom*, pages 33–37.

Henricksen, K., Indulska, J., and Mcfadden, T. (2005a). Middleware for Distributed Context-Aware Systems. *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 846 – 863.

Henricksen, K., Indulska, J., and McFadden, T. (2005b). Modelling context information with ORM. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3762 LNCS:626–635.

Henricksen, K., Wishart, R., McFadden, T., and Indulska, J. (2005c). Extending context models for privacy in pervasive computing environments. In *Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops. Third IEEE International Conference on*, pages 20–24. IEEE.

Hirschheim, R., Klein, H. K., and Lyytinen, K. (1995). *Information systems development and data modeling: conceptual and philosophical foundations*, volume 9. Cambridge University Press.

Hong, J.-y., Suh, E.-h., and Kim, S.-J. (2009). Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509–8522.

Horrocks, I., Patel-Schneider, P. F., and Van Harmelen, F. (2003). From shiq and rdf to owl: The making of a web ontology language. *Web semantics: science, services and agents on the World Wide Web*, 1(1):7–26.

Huang, S.-H., Wu, T.-T., Chu, H.-C., and Hwang, G.-J. (2008). A decision tree approach to conducting dynamic assessment in a context-aware ubiquitous learning environment. In *Wireless, Mobile, and Ubiquitous Technology in Education, 2008. WMUTE 2008. Fifth IEEE International Conference on*, pages 89–94. IEEE.

Hull, R., Neaves, P., and Bedford-Roberts, J. (1997). Towards situated computing. In *Wearable Computers, 1997. Digest of Papers., First International Symposium on*, pages 146–153. IEEE.

Indulska, J., Robinson, R., Rakotonirainy, A., and Henricksen, K. (2003). Experiences in using CC/PP in Context-Aware Systems. In *Mobile Data Management.4th International Conference, MDM 2003 Melbourne, Australia, January 21–24, 2003 Proceedings*, pages 247–261.

Juszczyk, L., Psaier, H., Manzoor, A., and Dustdar, S. (2009). Adaptive Query Routing on Distributed Context - The COSINE Framework. *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, pages 588–593.

Kang, S., Lee, J., Jang, H., Lee, H., Lee, Y., Park, S., Park, T., and Song, J. (2008). Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 267–280. ACM.

Kang, S., Lee, Y., Min, C., Ju, Y., Park, T., Lee, J., Rhee, Y., and Song, J. (2010). Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 135–144. IEEE.

Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. (2002). An efficient k-means clustering algorithm: Analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881–892.

Khedr, M. and Karmouch, A. (2004). Negotiating context information in context-aware systems. *Intelligent Systems, IEEE*, 19(6):21–29.

Khedr, M. and Karmouch, A. (2005). ACAI: agent-based context-aware infrastructure for spontaneous applications. *Journal of Network and Computer Applications*, 28(1):19–44.

Knappmeyer, M., Kiani, S. L., Frà, C., Moltchanov, B., and Baker, N. (2010). ContextML: A light-weight context representation and context management schema. *ISWPC 2010 - IEEE 5th International Symposium on Wireless Pervasive Computing 2010*, pages 367–372.

Knublauch, H., Hendler, J. A., and Idehen, K. (2011). Spin-overview and motivation. *W3C Member Submission, W3C*.

Kohonen, T. (2001). *Self-organizing maps*, volume 30. Springer Science & Business Media.

Korel, B. T., Koo, S. G., et al. (2010). A survey on context-aware sensing for body sensor networks. *Wireless Sensor Network*, 2(08):571.

Lim, B. Y. and Dey, A. K. (2010). Toolkit to support intelligibility in context-aware applications. *Proceedings of the 12th ACM international conference on Ubiquitous computing - Ubicomp '10*, pages 13–22.

Luckham, D. (2008). *The power of events: An introduction to complex event processing in distributed enterprise systems*. Springer.

Lyu, C. H., Choi, M. S., Li, Z. Y., and Youn, H. Y. (2010). Reasoning with imprecise context using improved dempster-shafer theory. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 2, pages 475–478. IEEE.

Ma, J., Russo, A., Broda, K., and Clark, K. (2008). Dare: a system for distributed abductive reasoning. *Autonomous Agents and Multi-Agent Systems*, 16(3):271–297.

Marie, P., Desprats, T., Chabridon, S., and Sibilla, M. (2013). QoCIM : un méta-modèle de qualité de contexte. In *Ubimob 2013*.

Meditskos, G., Dasiopoulou, S., Efstathiou, V., and Kompatsiaris, I. (2013). SP-ACT : A Hybrid Framework for Complex Activity Recognition Combining OWL and SPARQL Rules. (March):25–30.

Mühl, G. (2002). *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis.

Müller, J. P. and Fischer, K. (2014). Application impact of multi-agent systems and technologies: A survey. In *Agent-Oriented Software Engineering*, pages 27–53. Springer.

Olaru, A. (2011). A context-aware multi-agent system for ami environments. *PhD Thesis*.

Olaru, A., Florea, A. M., and Seghrouchni, A. E. F. (2011). Graphs and patterns for context-awareness. In *Ambient Intelligence-Software and Applications*, pages 165–172. Springer.

Olaru, A., Florea, A. M., and Seghrouchni, A. E. F. (2013). A context-aware multi-agent system as a middleware for ambient intelligence. *Mobile Networks and Applications*, 18(3):429–443.

Peizhi, L. and Jian, Z. (2008). A context-aware application infrastructure with reasoning mechanism based on dempster-shafer evidence theory. In *Vehicular Technology Conference, 2008. VTC Spring 2008. IEEE*, pages 2834–2838. IEEE.

Pereira, D., Oliveira, E., and Moreira, N. (2008). Formal modelling of emotions in bdi agents. In *Computational Logic in Multi-Agent Systems*, pages 62–81. Springer.

Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2012). Ca4iot: Context awareness for internet of things. In *2012 IEEE International Conference on Green Computing and Communications (GreenCom)*, pages 775–782.

Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2014a). Context aware computing for the internet of things: A survey. *Communications Surveys & Tutorials, IEEE*, 16(1):414–454.

Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2014b). Sensing as a service model for smart cities supported by internet of things. *Transactions on Emerging Telecommunications Technologies*, 25(1):81–93.

Pietschmann, S., Mitschick, A., Winkler, R., and Meißner, K. (2008). Croco: Ontology-based, cross-application context management. In *Semantic Media Adaptation and Personalization, 2008. SMAP'08. Third International Workshop on*, pages 88–93. IEEE.

Preuveneers, D., Bergh, J. V. D., Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., and Coninx, K. (2004). Towards an extensible context ontology for Ambient Intelligence. *Electronics*.

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106.

Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Riboni, D. and Bettini, C. (2009). Context-aware activity recognition through a combination of ontological and statistical reasoning. *Ubiquitous Intelligence and Computing*.

Riboni, D. and Bettini, C. (2011). OWL 2 modeling and reasoning with complex human activities. *Pervasive and Mobile Computing*, 7(3):379–395.

Rinne, M., Törmä, S., and Nuutila, E. (2012). Sparql-based applications for rdf-encoded sensor data. *SSN*, 904:81–96.

Robinson, R., Henricksen, K., and Indulska, J. (2007). XCML: A runtime representation for the Context Modelling Language. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pages 20–26. IEEE.

Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. (2002). A middleware infrastructure for active spaces. *IEEE pervasive computing*, 1(4):74–83.

Schilit, B., Adams, N., and Want, R. (1994). Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90. IEEE.

Schmidt, A. (2006). Ontology-based user context management: The challenges of imperfection and time-dependence. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 995–1011. Springer.

Sehic, S. and Dustdar, S. (2010). COPAL: An adaptive approach to context provisioning. *2010 IEEE 6th International Conference on Wireless and Mobile Computing, Networking and Communications*, pages 286–293.

Sharon, G. and Etzion, O. (2008). Event-processing network model and implementation. *IBM Systems Journal*, 47(2):321–334.

Shoham, Y. (1993). Agent-oriented programming. *Artificial intelligence*, 60(1):51–92.

Sorici, A., Picard, G., Boissier, O., and Florea, A. M. (2015). Multi-agent based flexible deployment of context management in ambient intelligence applications. In *Practical Applications of Agents and Multi-Agent Systems, 2015 13th International Conference on*, volume in print. Springer.

Strang, T., Linnhoff-Popien, C., and Frank, K. (2003). CoOL: A context ontology language to enable contextual interoperability. *Distributed applications and interoperable systems*, pages 236–247.

Studer, R., Benjamins, V. R., and Fensel, D. (1998). Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25(1):161–197.

Tapia, E. M., Intille, S. S., and Larson, K. (2004). *Activity recognition in the home using simple and ubiquitous sensors*. Springer.

Teymourian, K., Rohde, M., and Paschke, A. (2012). Fusion of background knowledge and streams of events. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 302–313. ACM.

Toffler, A., Longul, W., and Forbes, H. (1981). *The third wave*. Bantam books New York.

Toninelli, A., Montanari, R., Kagal, L., and Lassila, O. (2006). A semantic context-aware access control framework for secure collaborations in pervasive computing environments. In *The Semantic Web-ISWC 2006*, pages 473–486. Springer.

Turhan, A.-Y., Springer, T., and Berger, M. (2006). Pushing doors for modeling contexts with owl dl-a case study. In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, pages 5–pp. IEEE.

Van Kasteren, T., Noulas, A., Englebienne, G., and Kröse, B. (2008). Accurate activity recognition in a home setting. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 1–9. ACM.

Van Laerhoven, K. (2001). Combining the self-organizing map and k-means clustering for online classification of sensor data. In *Artificial Neural Networks-ICANN 2001*, pages 464–469. Springer.

Weiser, M. (1991). The computer for the 21st century. *Scientific american*, 265(3):94–104.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(02):115–152.

Yager, R., Fedrizzi, M., and Kacprzyk, J. (1994). Advances in the dempster-shafer theory of evidence.

Yegnanarayana, B. (2009). *Artificial neural networks*. PHI Learning Pvt. Ltd.

Youssef, M. A., Agrawala, A., and Udaya Shankar, A. (2003). Wlan location determination via clustering and probability distributions. In *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 143–150. IEEE.

Zahid, N., Abouelala, O., Limouri, M., and Essaid, A. (2001). Fuzzy clustering based on k-nearest-neighbours rule. *Fuzzy Sets and Systems*, 120(2):239–247.

Zhang, D., Cao, J., Zhou, J., and Guo, M. (2009). Extended dempster-shafer theory in context reasoning for ubiquitous computing environments. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 205–212. IEEE.

Zimmermann, A., Lopes, N., Polleres, A., and Straccia, U. (2012). A general framework for representing, reasoning and querying with annotated semantic web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 11:72–95.

Zimmermann, A., Lorenz, A., and Oppermann, R. (2007). An operational definition of context. In *Modeling and using context*, pages 558–571. Springer.