JYVÄSKYLÄ STUDIES IN COMPUTING
187

Michal Nagy

Self-Management in Distributed Systems

Smart Adaptive Framework for Pervasive Computing Environments

JYVÄSKYLÄN YLIOPISTO

JYVÄSKYLÄ STUDIES IN COMPUTING 187

Michal Nagy

Self-Management in Distributed Systems

Smart Adaptive Framework for Pervasive Computing Environments

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston Agora-rakennuksen auditoriossa 3
joulukuun 19. päivänä 2013 kello 12.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in building Agora, Auditorium 3, on December 19, 2013 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2013

Self-Management in Distributed Systems

Smart Adaptive Framework for Pervasive Computing Environments

JYVÄSKYLÄ STUDIES IN COMPUTING 187

Michal Nagy

Self-Management in Distributed Systems

Smart Adaptive Framework for Pervasive Computing Environments

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2013

Editors
Timo Männikkö
Department of Mathematical Information Technology, University of Jyväskylä
Pekka Olsbo, Ville Korkiakangas
Publishing Unit, University Library of Jyväskylä

URN:ISBN:978-951-39-5554-0
ISBN 978-951-39-5554-0 (PDF)

ISBN 978-951-39-5553-3 (nid.)
ISSN 1456-5390

Jyväskylä University Printing House, Jyväskylä 2013

ABSTRACT

Nagy, Michal
Self-Management in Distributed Systems. Smart adaptive framework for pervasive computing environments.
Jyväskylä: University of Jyväskylä, 2013, 192 p.
(Jyväskylä Studies in Computing
ISSN 1456-5390; 187)
ISBN 978-951-39-5553-3 (nid.)
ISBN 978-951-39-5554-0 (PDF)
Finnish summary
Diss.

Nowadays, majority of the population has access to various powerful mobile devices such as smartphones or tablets. On the other hand, these devices are becoming increasingly complex and soon we might become overwhelmed by this situation.

There is a need for a new type of software that would be able to manage itself with very little human involvement, or none at all. We call it self-managed (or adaptive) software. In order to make this vision possible, several years ago a new middleware platform has been created as a part of the Ubiware project. The platform introduces a novel approach to Agent-oriented Software Engineering based on a declarative semantic agent programming language called S-APL (Semantic Agent Programming Language).

This work introduces three improvements to the S-APL language and the Ubiware platform – utility functions, belief safeguards and agent's ability to plan. Moreover, the dissertation proposes a framework for self-management called SAF (Smart adaptive framework). The framework follows a hybrid approach based on a three-layered model for adaptive software and a closed planning loop.

SAF is built as an extension of the Ubiware middleware platform. The framework introduces a new set of tools and techniques that allow the developer to describe the goals, policies, data structures or plans of the software being developed. Based on these descriptions, SAF takes care of software management in an autonomous or semi-autonomous way. With the help of newly introduced S-APL improvements, each module of the framework is described. Also, the software and framework configuration techniques are provided. Moreover, a formal ontology for the operating knowledge is described. Lastly, a sample scenario from the medical area is described, where SAF is used to manage the environment of a ward patient and the nursing staff. SAF aids the nurse by performing certain actions automatically or semi-automatically based on high-level goals and policies specified by various stakeholders.

Keywords: autonomic computing, self-management, Semantic Web, agent-based system, pervasive computing, ubiquitous computing

Author

Michal Nagy
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Supervisors

Professor Dr. Vagan Terziyan
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Professor Dr. Timo Tiihonen
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Reviewers

Associate Professor Dr. Vadim Ermolayev
Department of Information Technologies
Zaporozhye National University
Ukraine

Associate Professor Dr. Michal Zábovský
Deptartment of Informatics
University of Žilina
Slovakia

Opponent

Dr. Ilkka Seilonen
Department of Automation and Systems Technology
Aalto University
Finland

ACKNOWLEDGEMENTS

There are many people in my life that have influenced me in some way and thus directly or indirectly affected this work. The most important influence were my parents Robert and Iveta. I am very grateful for their support through all these years. My thanks also go to my sister Jana, my grandparents and the rest of the family for providing me with a peaceful family environment, both in my childhood and in the later years.

I would like to thank my supervisors, Prof. Vagan Terziyan and Prof. Timo Tiihonen, for guiding me through this process. Our common discussions always provided me with additional stimuli that perpetuated this work forward. Also, despite being very busy, they always responded to my questions with haste.

Moreover, many thanks go to my colleagues from the Industrial Ontologies Group, namely (in alphabetical order) Michael Cochez, Artem Katasonov, Olena Kaykova, Joonas Kesäniemi, Oleksiy Khriyenko, Sergiy Nikitin and Michał Szydłowski. Our numerous (and often long) discussions helped me to improve my theories and provided me with new thoughts.

Furthermore, I would like to thank the reviewers, Assoc.Prof. Vadim Ermolayev and Assoc.Prof. Michal Zábovský, for their quick review. Their comments were very constructive and really helped me improve the manuscript. Moreover, I want to thank Dr. Ilkka Seilonen for being my opponent.

My gratitude also goes to the Agora Center, the Department of Mathematical Information Technology and the COMAS graduate school who all financially supported this research.

Last, but not least, I would like to thank my girlfriend Fotini for her psychological support and for proofreading the manuscript draft.

Göteborg
December 2013

Michal Nagy

LIST OF ACRONYMS

ACL	Agent Communication Language
ACM	Association for Computing Machinery
AML	Agent Modeling Language
AMS	Agent Management System
AOP	Agent-oriented programming
AOSE	Agent-oriented Software Engineering
API	Application Programming Interface
APL	Agent Programming Language
AmI	Ambient Intelligence
BDI	Beliefs, Desires and Intentions
DF	Directory Facilitator
DNS	Domain Name System
FIPA	Foundation for Intelligent Physical Agents
GPS	Global Positioning System
GUI	Graphical User Interface
HCI	Human-Computer Interaction
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IOG	Industrial Ontologies Group
IPC	International Planning Competition
IPS	Indoor positioning systems
IoT	Internet of Things
JADE	Java Agent DEvelopment Framework
LAN	Local Area Network
MAS	Multi-Agent System
NIST	National Institute of Standards and Technology
OOP	Object Oriented Programming
OOSE	Object-oriented Software Engineering
ORB	Object Request Broker
OS	Operating system
OWL	Web Ontology Language
OWL-DL	Description Logic profile of OWL
PAN	Personal Area Network
PARC	Xerox Palo Alto Research Center
PDDL	Planning Domain Definition Language
PES	Physical environmental sensor
PerComp	Pervasive computing
QoS	Quality of Service
RAB	Reusable Atomic Behavior

RDF	Resource Description Framework
RDFS	RDF Schema
RFC	Request for Comments
RFID	Radio-Frequency IDentification
RMI	Remote Method Invocation
S-APL	Semantic Agent Programming Language
SAC	Situated and Autonomic Communications
SMS	Short Message Service
SOA	Service-Oriented Architecture
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SW	Software
SWRL	Semantic Web Rule Language
TCP	Transmission Control Protocol
UBIWARE	Smart Semantic Middleware for Ubiquitous Computing
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
UbiComp	Ubiquitous Computing
VES	Virtual environmental sensor
W3C	World Wide Web Consortium
WAN	Wide Area Network
WLAN	Wireless Local Area Network
WWW	World Wide Web
XML	eXtensible Markup Language
XSD	XML Schema

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

1.1 Problem statement

We arrived to the point in history, where computers are becoming an everyday part of our lives. According to Smith (2013) over 90% of US inhabitants own a phone and most of them own a smartphone. While the price and size of today's computers is decreasing rapidly, their computing power and memory size is increasing. With respect to their computational capabilities, today's smartphones are on par with personal desktop computers from 10 years ago. In the last years, new device types have been introduced, such as tablets and smart watches. With this kind of development, the vision of Ubiquitous computing (Weiser, 1991) might soon become a reality.

Moreover, several single-board microcontrollers such as Arduino are available for the public at very low prices. Various companies have started to invest in the area of smart home solutions. Not only hardware producers, but also the consumers have now access to various sensors and actuators that can be used for personal automation projects. The advancements in the area of low-energy wireless transmission make the interconnection of small battery-powered devices possible. Moreover, the capacity of batteries has risen due to strong emphasis on research in this area. It seems that also the visions of Ambient Intelligence and the Internet of Things might soon become possible.

Despite this positive development in the computing hardware area, these visions are still not the reality. The devices, sensors, actuators, microcontrollers and other components exist. Naturally, one must ask, why are we not in the era of Ubiquitous computing? The available hardware is only one prerequisite. There are social- and software-related issues as well (Lyytinen and Yoo, 2002). This dissertation is trying to address the later. The complexity of devices is increasing and so is the software that operates them. Moreover, the devices are capable of interaction with each other, their environment and humans. Pervasive computing environments are inherently open, dynamic and complex (Zhang and Hansen, 2008b). There is a need for a new kind of software that is able to deal with changing

conditions with very little human intervention or none at all (Kephart and Chess, 2003). We present the following 3 research questions:

Q1: What are the key elements of a middleware for adaptive software?
Q2: How can such a middleware be implemented?
Q3: How can such a middleware be used in the domain of pervasive computing?

The question Q1 is answered in Chapter 3, where a self-management framework called Smart Adaptive Framework (SAF) is introduced. The framework uses a hybrid approach based on a modified MAPE-K cycle (Oreizy et al., 1999; Kephart and Chess, 2003) and the three-layered architecture by Kramer and Magee (2007). Later in the chapter, we elaborate on the requirements for such a framework.

The answer to the second question is provided in Chapters 4 and 5. We chose Ubiware (Katasonov et al., 2008; Katasonov and Terziyan, 2008) as the implementation platform for the SAF framework. Despite the fact that Ubiware already has many of the required capabilities, it is still missing some features. In Chapter 4 we introduce 3 new concepts into the Ubiware platform – safeguards, utility functions and the planning ability. For each improvement, we discuss the motivation and implementation. In Chapter 5 we show how the SAF framework is implemented in terms of Ubiware and S-APL concepts, including the newly introduced ones.

Finally, Chapters 5 and 6 provide an answer to the question Q3. Chapter 5 describes the proper use of SAF and software built on top of it. We show how the developer can use S-APL to describe the information about policy definitions, sensor and actuator configuration, software configuration, action descriptions, etc. In Chapter 6, a scenario from the area of healthcare is described. This scenario shows a hospital with self-managed software based on the SAF framework. By using such software, the hospital staff can divert more of their attention to the interaction with their patients instead of the interaction with their devices. Table 1 summarizes the answers to the research questions.

1.2 Research approach

In this work we are investigating the possibilities of semantic and agent technologies in the area of self-managed software. This has been partially done as a part of the Ubiware project, which is an intermediate step towards the GUN vision (Terziyan, 2003; Kaykova et al., 2005; Terziyan, 2011). One of the results of the Ubiware project and its predecessors was the Semantic Agent Programming Language (S-APL) (Katasonov and Terziyan, 2008).

This work started with a **literature review** of the domains in question. Based on the review we concluded that even though there have been some attempts to use the semantic technology in the area of autonomic computing, S-APL and the Ubiware platform had a feature that did not seem to be used anywhere else – S-APL was used not only to describe the data, but also to describe the behavior

TABLE 1 Answers to research questions

Question	Answer	
	Where	How
Q1	Chapter 3	We introduce a concept of an adaptive middleware called Smart Adaptive Framework (SAF). We provide the motivation, requirements and general description of the framework.
Q2	Chapter 4	We provide Ubiware platform improvements needed for a Ubiware-based implementation of SAF.
	Chapter 5	We provide a Ubiware-based implementation of SAF.
Q3	Chapter 5	We describe how SAF can be used to develop self-managed applications
	Chapter 6	We provide a sample SAF-based scenario from the area of healthcare.

of the agent. Even though the literature review has become crucial to finding the solution to the problem, it was also often a source of frustration in the later phase. Due to a large number of scientific publications being published every year and the broad nature of the problem, it was impossible to read all the relevant publications. Sometimes, when we came up with a seemingly new approach or technique, we discovered that it has been introduced in some previously published article.

During the last several years, the Ubiware platform and S-APL were being improved and as a result new versions of the platform were released. By taking part in the platform development process, we were able to understand the benefits and limitations of the Ubiware-based approach. By implementing various **prototypes** for the industrial members of the Ubiware consortium we learned new S-APL programming techniques. These were later tested and improved in other projects such as the SCOPE project[1] and the Tivit SHOK Cloud Software program[2].

The research group members were meeting regularly and trying to solve various problems related to the objectives of the projects being solved at the time. Such meetings were a good place for **brainstorming**. Also, group discussions have proven to be an effective tool to evaluate one's hypotheses.

Finally, **interviews** with the members of the industry contributed to our understanding of the problem by showing us the practical point of view. This helped us improve the prototypes and in the end SAF as well.

The Ubiware platform has been used in the **teaching** process as a part of two courses – *Semantic Web and Ontology Engineering* and *Agent-oriented Software Engineering*. Traditionally, teaching has not been considered a research method, therefore mentioning it as part of the research approach might be disputable.

1 More information available at http://www.cs.jyu.fi/ai/OntoGroup/projects.htm

2 More information available at http://www.cloudsoftwareprogram.org/

However, it provided us with new challenges for the design and implementation of Ubiware-based systems and thus it contributed to this work as well.

1.3 Structure of the thesis

This thesis is divided into seven chapters. **Chapter 1** introduces the scope of the work and three research questions. For each question, a short answer is provided together with a reference to a chapter containing a more comprehensive answer.

Chapter 2 discusses the theoretical background. Firstly, four scope-relevant research visions are introduced – Ubiquitous computing, Pervasive computing, Ambient intelligence and the Internet of Things. A historical view is provided as well, since it helps the reader to understand the context in which these visions were conceived. Secondly, a concise overview of research in the area of agent-based systems is presented. We describe a typical multi-agent system in terms of its components and their interactions. We also discuss the role of an agent in software engineering process. Thirdly, Semantic Web technologies are introduced. We provide a general overview and elaborate on the related standards. Moreover, the research related to self-managed software is introduced. We describe the properties and architecture of such systems. Lastly, we provide the description of the Ubiware platform and the S-APL language.

Chapter 3 provides a conceptual description of a middleware for self-managed systems called *Smart Adaptive Framework* (SAF). Firstly, we elaborate on the nature of typical pervasive computing environments. Consequently several requirements on adaptation in such environments are presented. Moreover, we introduce a general approach to adaptation based on SAF. Finally, we discuss the use of the Ubiware platform as an implementation candidate for SAF.

Chapter 4 introduces three new Ubiware platform improvements. Firstly, utility functions and their evaluation are described. Secondly, belief safeguards are defined and situations when they are broken are described. Lastly, a generic approach to planning is introduced, including the S-APL constructs for plan descriptions and action descriptions.

Chapter 5 presents a Ubiware-based implementation of SAF. Firstly, formal OWL ontologies for each of the platform components are introduced. Secondly, platform and software configurations are described, including their effects on the adaptation. Lastly, each of the SAF elements is described in terms of their implementation.

Chapter 6 provides a sample scenario from the area of healthcare. This scenario shows how SAF can be used by the hospital personnel to improve their work efficiency.

Chapter 7 concludes the thesis. Firstly, it compares the SAF approach to other self-management approaches. Secondly, it discusses the future research topics in this area.

2 THEORETICAL BACKGROUND

2.1 Ubiquitous computing

2.1.1 The original vision

The vision of ubiquitous computing was firstly introduced in 1988 in Xerox Palo Alto Research Center (PARC) (Weiser, 1993). This term is associated with an influential paper titled "The computer for the 21st century" written by Mark Weiser in 1991 (Weiser, 1991). Weiser presents a vision where computers become an integral part of everyday lives to such extent that they become indistinguishable from it. An example of a ubiquitous technology could be electricity. When one turns on the light switch, one does not think of the way the electric energy is made, transported and distributed. One does not have to. Another way of identifying a ubiquitous technology is that people will not realize its presence, but will realize the lack of it. In other words they will take it for granted. According to the UbiComp vision, the same should happen to computers. They should become invisible and unnoticeable.

indistinguishable indistinguishable

Depending on their size, Weiser suggests three types of devices that could carry out this functionality – tabs, pads and boards (Table 2). Tabs represent inch-scale devices that can be attached to people and things. They can for example provide the location of the object and other basic functions. Pads are foot-scale devices that are compared to scrap paper. They are not personalized (thus do not belong to anybody) and they are not carried with the user's laptops. The user can grab a pad, use it and then just return it back. They can be physically used to display different types of activities same as windows on a modern computer desktop. Boards are yard-scale devices and they act as whiteboards and bulletin boards. A typical room may contain approximately a hundred tabs, ten pads and one board.

There are three technical milestones that must be reached. Firstly, computers should become cheap enough to be used in every aspect of human lives. These

TABLE 2 Weiser's three device types

Name	Size	Example	Amount per room
tab	$\sim$ 1 inch (= 2,5 cm)	active badge	$\sim$ 100
pad	$\sim$ 1 foot (= 30 cm)	scrap paper	$\sim$ 10
board	$\sim$ 1 yard (= 91 cm)	white board, bulletin board	$\sim$ 1

computers should have low energy consumption, be lightweight and still able to display all necessary information. Secondly, the software for ubiquitous applications must be easily reconfigurable and capable of adapting to the current situation. Lastly, computer networks must provide wired and especially wireless access among the devices. The speed, spatial range and low energy consumption of transceivers are the challenges.

Weiser argues that "personal" computers did not yet reach this point and the user of such a device still requires a complex jargon to interact with it (Weiser, 1991). He believes that it is not only a problem of the graphical user interface (GUI) or the display capabilities of computers. He argues that the problem is about a new relationship between people and computers and as such it is a very complex problem (Weiser, 1993).

In 1990s Weiser and his team identified several research areas that needed to be explored in order to make this vision possible. Among these areas there were:

- hardware components
- network protocols
- interaction substrates (e.g. software for screens and pens)
- applications
- privacy
- computational methods

Firstly, there was the issue of hardware components. There was a need for a high-speed wireless network capable of connecting several hundreds of devices, but still requiring only very little energy to run. Also, smaller and more energy-efficient microprocessors were needed.

Secondly, new wireless network protocols were required. At that time there was no common wireless network protocol such as IEEE 802.11 (also known as WiFi™). Since the UbiComp environment is very dynamic and devices can join and leave the network at any time, there was a need for a very dynamic media access wireless network protocol. Wired media access techniques were not applicable to UbiComp environment, due to the inherently different nature of wireless communication and media access control. Other network-related issues involved high-speed wired and wireless networks.

The third problem addressed by Weiser and his team was the problem of interaction substrates. Tabs, pads and boards each required a different way of

human-computer interaction (HCI). Tabs were small and therefore it was impossible to use a keyboard. Voice commands are not always possible due to the situation (e.g. during a concert) or the nature of the commands (e.g. some private information). Boards, for example, had the opposite problem. They were too large for traditional window-based GUIs, because the user would be required to walk around the whole room to perform an action.

Applications represent the next problem. Since devices and their users can freely move, one of the new aspects of UbiComp applications is the location-awareness. The application changes its behavior depending where the device and/or user is located. Another issue related to UbiComp is the issue of human-computer interaction. In such environments, users and devices are interacting with each other more often than in typical computer usage scenarios.

The next problem is the privacy, especially the privacy of location. For example in cellular wireless networks it is possible to track user's movement by determining his or her connection to a particular cell. These issues are not only related to technological solutions, but they have a social dimension as well.

Lastly, the issue of computational methods is addressed. The problem of resource caching is pointed out. If a relatively slow wireless network is used, then data has to be cached on a device in order to reduce the amount of remote data transferred.

Some of the problems mentioned in the original work by Weiser and his team are completely or partially solved. Wireless computer networks have become more reliable and can achieve better quality of service (QoS). They can easily be used for such QoS-sensistive applications as videotelephony. It appears that even the problem of interaction substrates has been partially solved. Nowadays even tab-sized devices feature color touch screens, powerful enough speaker and even haptic feedback capabilities. As an example of an inch-size device we could mention 6th generation of Apple iPod Nano that has 1.54 inch multi-touch screen (Apple, 2011). Tablets (pad-size devices) are even more sophisticated. Also, the location-awareness problem has been partially solved by incorporating GPS modules into devices. However, in some places (e.g. indoors) the GPS signal might not be available. There are techniques, where publicly broadcast wireless data from wireless access points is used to determine an approximate position of the device. This approach is used by for example Google (Google, 2011). The problem of privacy still persists, but as Weiser mentiones, it is not only a technical problem, but also a social one. It seems that many problems have been solved. Naturally a question arises: So why aren't we in the era of Ubiquitous computing?

Lyytinen and Yoo (2002) argue that apart from technical issues, there are also social and organizational challenges. This is due to the fact that the realization of ubiquitous computing leads to the creation of a sociotechnical system. According to Jessup and Robey (2002), ubiquitous computing will introduce new social actions or change the existing ones. This might be met with a resistance by public.

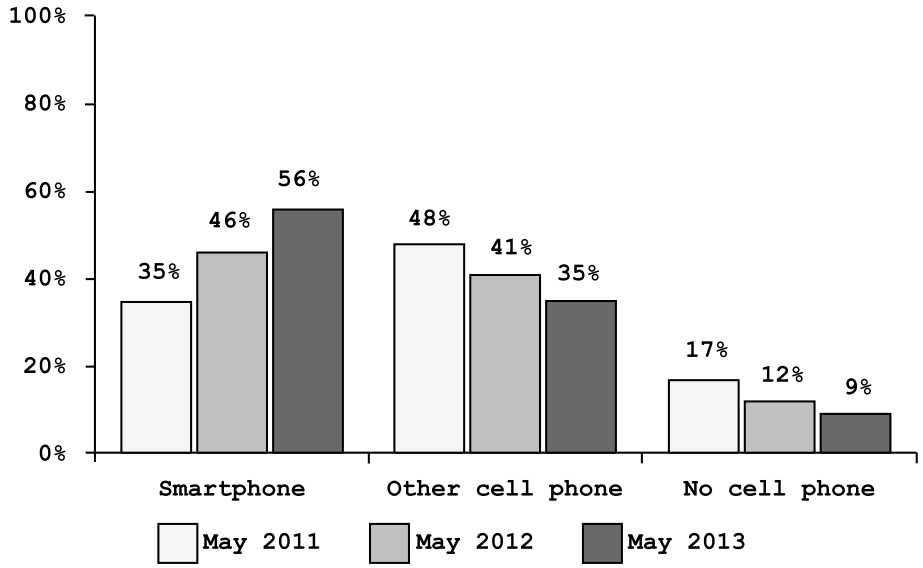


FIGURE 1 "Smartphone Ownership 2013" study

2.1.2 Ubiquitous computing today

Devices such as smartphones and tables became a part of our lives. According to Pew Research Center's Internet & Life American project surveys conducted between April 2010 and May 2013 (Smith, 2013), 91% of the adults in the United States own a phone (Figure 1). More than half of the US adult population owns a smartphone and over a third owns a feature phone [1]. The number of smartphone users has increased rapidly over the last 3 years – by 10% of the population per year. The usage of a smartphone is more prevalent (79%) in the population of age 18-34. The relationship between the age and the percentage of smartphone users is apparent. The older the demographic group, the lower relative amount of smartphone users that it contains. It is our belief that the number of smart phone users will increase also in the higher age groups. Young people already got used to own a smart phone and they will most likely keep on owning a smart phone as they grow older.

2.2 Pervasive computing

In 2009, Ronzani (2009) conducted a study related to the usage of words "ubiquitous computing", "pervasive computing" and "Internet of things" in newspapers between years 1990 and 2006. One of the conclusions is the fact that while the

[1] According to the authors of the study, a feature phone is any mobile phone that is not a smartphone or a PDA phone. Feature phones have proprietary operating system firmware, and generally do not support third-party software applications in the way that smartphones can. Compared to smartphones, feature phones typically are less powerful, with fewer features and capabilities.

expression ubiquitous computing (UbiComp) has been appearing in newspapers since 1990, the expression pervasive computing (PerComp) started to appear in 1994. Both of them reached their peak between 1999 and 2001. Also, while the amount of articles related to ubicomp stayed relatively constant, the amount of articles about PerComp fluctuated significantly more.

Lyytinen and Yoo (2002) say that "Though these terms [PerComp and UbiComp] are often used interchangeably, they are conceptually different and employ different ideas of organizing and managing computing services (see the accompanying figure)". They see pervasive computing as a similar, but separate, effort. The difference between UbiComp and PerComp is the fact that the former works with a higher level of mobility and the later works with a lower level of mobility. However, they both are striving for something that they call "a higher level of embeddedness".

There are some researchers, who use terms ubicomp and PerComp interchangeably. Saha and Mukherjee (2003) present PerComp as an answer to Mark Weiser's vision of 21st century computing. However, Mark Weiser himself referes to his vision as the vision of UbiComp in (Weiser, 1993). In their book titled "Pervasive computing: The Mobile World", Hansmann et al. (2003) write about the use of mobile phones and PDAs as machines, through which PerComp can be achieved. The authors are members of IBM, one of them being directly a member of IBM Pervasive Computing Division. They also emphasize the importance of wireless networks (e.g. Personal Area Networks) as an enabler of PerComp. This suggests that indeed PerComp is dealing with computer mobility.

There is a number of conferences and journals that accept work both in ubiquitous and pervasive computing. An example of such a conference is International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN). Another conference that seems to use these terms interchangeably is "International Conference on Pervasive Computing Technologies for Healthcare". According to their "Call for Papers" document, they state that they they concentrate on technologies related to the use of ubiquitous computing in healthcare (PervasiveHealth, 2013). However, later in the text they state that they accept contributions from the area of pervasive computing. Some of the publication channels claim directly that these two terms are synonymous. According to the scope of "Pervasive and Mobile Computing Journal":

> Pervasive computing, often synonymously called ubiquitous computing, is an emerging field of research that brings in revolutionary paradigms for computing models in the 21st century (Elsevier, 2013).

It is our belief that the boundaries between pervasive computing and ubiquitous computing have softened. The issue of mobility does not exist anymore. Nowadays, computer mobility has become a part of our everyday lives. Therefore we consider both terms interchangeable.

2.3 Ambient intelligence

2.3.1 History

This term was coined in 1998 in a series of workshops at Phillips Research (Aarts, 2003). At this time consumer electronics was represented by standalone devices, each providing a certain feature. Researchers at Philips believed that by connecting these devices one could obtain additional functionality. In its beginnings, Ambient Intelligence (AmI) represented a vision of 2020s where consumer electronics would be fully integrated, user-friendly and with the ability to support ubiquitous information, communication and entertainment (Aarts, 2003). Such environments were supposed to be aware of human presence (Aarts et al., 2002). The term *ambient intelligence* first publicly appeared in a Dutch journal in an article named "Ambient Intelligence: thuisomgevingen van de toekomst" (Aarts and Appelo, 1999).

In early 2000s Ambient Intelligence vision was adopted by public research institutions as well. In 2000, a working group under the European Commission called ISTAG (IST Advisory Group) was established to come up with usage scenarios of AmI. Their results directly influenced the content of the European Commission's Sixth Framework Programme (Ducatel et al., 2001). In 2002, Philips Research opened their HomeLab laboratory (Aarts and Eggen, 2002). The goal of the laboratory was to provide an environment for prototyping of future electronic systems. Around the same time, many leading universities around the world opened their own research laboratories devoted to the vision of Ambient intelligence (Wildstrom, 2000; GeorgiaTech, 2012). This type of research was supported by various companies such as Hewlett-Packard, Nokia, Intel, Motorola and others. In 2012, when ACM published the new version of The ACM Computing Classification System, AmI was added as a subtopic of "Ubiquitous and mobile computing" (ACM, 2012). To this day (May, 2013) the IEEE database contains over 1400 articles published between 1999 and 2012, and having "Ambient intelligence" as one of the keywords.

2.3.2 AmI Scenarios

According to ISTAG, AmI is based on three important technologies – Ubiquitous Computing, Ubiquitous Communication and Intelligent User Friendly Interfaces. It should lead to a state where:

> People are surrounded by intelligent intuitive interfaces that are embedded in all kinds of objects and an environment that is capable of recognising and responding to the presence of different individuals in a seamless, unobtrusive and often invisible way. (Ducatel et al., 2001)

As an example of real-world use, in 2001, ISTAG published four different scenarios taking place in 2010 where AmI would assist people in their daily activities. They were supposed to show the power of AmI. The first two scenarios are based

on existing technologies and represent near future. The other two technologies represent further future with higher AmI penetration into humans' lives.

The first scenario is about Maria, a businesswoman travelling to a foreign country. Using her phone she is able to stay in touch with her family and friends and at the same time find all travel-related information in a personalized and location-aware way. These kinds of services already exist and are offered on many smartphone operating systems.

The second scenario is about Dimitrios and his *Digital Me* (D-Me). D-Me is a digital avatar representing him in the digital world. It can be programmed, but it also has the ability to learn and get to know Dimitrios' preferences and routines. Its goal is to relieve him of unnecessary interaction with his environment by acting on his behalf.

The third scenario depicts the future of intelligent traffic and commerce. The main protagonist, Carmen, uses AmI to share a morning ride to work with somebody else. She also uses her intelligent fridge to do the shopping remotely and collects the goods on the way home. The system pays automatically for every service she uses on the way.

The last scenario is about a relatively futuristic environment for learning called *ambient*. A course about environmental studies is taking place in a conference room. People can come and ambient helps them achieve their own goals individually and in groups. Every learning tool and every participant are connected with each other and with ambient.

Based on these four scenarios, five key technology requirements have been identified:

1. Very unobtrusive hardware
2. A seamless mobile and fixed communications infrastructure
3. Dynamic and massively distributed device networks
4. Natural feeling human interfaces
5. Dependability and security

2.3.3 Criticism of AmI

It was hoped that by making human-computer interfaces intelligent, the interaction with them will become easier or even invisible. However, several years later, Alahuhta et al. (2005) present SWAMI (Safeguards in a world of AmI) and argue that the environment is not intelligent in a way that people define intelligence. They believe that it appears intelligent only from the outside. Also, they raise the question of privacy and security of such environments. They provide four model situations called "dark scenarios" where some vulnerability could be exploited. Despite its effort to find vulnerabilities, SWAMI still believes in the vision of AmI (Alahuhta et al., 2005). They just argue that in order to overcome these shortcomings, all stakeholders must cooperate and ensure the necessary technological, socio-economic, legal and regulatory safeguards.

2.3.4 Ambient Intelligence 2.0

In 2009, Aarts and Grotenhuis (2009) published an article describing a revised vision of AmI. They called this vision Ambient Intelligence 2.0. They realized that the original ideas and the way they wanted AmI to develop were not valid anymore. They argue that AmI did not become a user centered technology as promised in the beginning. Also, it seems that people's desires changed. Therefore the old AmI does not reflect the current wishes of the society. The society wishes to have balanced lives where the technology would play only a supporting role. Therefore a more human-centered approach is needed.

The new approach is based on three elements – People, Planet and Profit. These elements should be in balance called Synergic Prosperity. Four views are presented based on the three elements. The People element provides Body and Mind view. The Body is related to personal wellbeing and the Mind is about a balanced personal lifestyle. The Planet element generates Community and Environment view. The Community relates to common welfare and the Environment talks about sustainable development of the planet.

The Profit is understood as Prosperity. Aarts and Grotenhuis (2009) believe that true prosperity stems from the right combination of components related to People and Planed. They define Synergic Prosperity as "development and application of eco-affluent innovations that allow all people to flourish" (Aarts and Grotenhuis, 2009). The goal of AmI 2.0 should be to create components, that act as synergic satisfiers contributing to this prosperity. A synergic satisfier is something that satisfies several needs at the same time (Max-neef et al., 1991).

It is clear that Ambient Intelligence 2.0 takes into account economic and socio-political issues. This is the major shift from the original technology-oriented vision.

2.4 The Internet of Things

2.4.1 The original idea

According to Ashton (2009), the first use of the term *Internet of Things* (IoT) is dated to 1999. It was used to link two popular ideas at that time – RFID (Radio-frequency identification) technology and the Internet. The goal was to make them look appealing to the management of the company. Since 1999 the idea has developed further. However, the original idea can be summarized in one sentence as follows:

> If we had computers that knew everything there was to know about things – using data they gathered without any help from us – we would be able to track and count everything, and greatly reduce waste, loss and cost (Ashton, 2009).

The content of today's Internet is mostly generated by humans. If one watches a video on Youtube, reads an article from an online newspaper, sends or receives

an email or communicates through some social media, all this data is human-generated. In some usage scenarios of Internet this is not a problem. When people communicate with each other (exchange ideas and opinions), naturally the data must (and should) be of human origin. We may call this leisure use of the Internet. However, if precise and accurate data about some real-world object is needed, then humans are not always the best data source. The problem of humans as data creators is that they are not always accurate, they have limited attention span and they don't always have time. In these cases we may talk about an "industrial use" of the Internet. The goal of the Internet of Things is to allow computers to gather data about the world themselves with minimal human participation.

Gershenfeld et al. (2004) use the example of the Internet's architecture as a successful method for integration of heterogeneous local networks. Similarly, the IoT vision should be the answer to the problem of connecting heterogeneous devices. Sometimes this idea is also called *Internet-0* (I0), because the data speeds required for efficient communication between several simple devices are very low in comparison to the capabilities of today's Internet. Instead of speed, the emphasis is given on the interoperability.

Buckley (2006) define the Internet of Things as a worldwide network of intercommunicating devices. As potential benefits they list public good, economic growth and personal enrichment of life. The technology should enable pervasiveness of communication technologies in many sectors. Buckley (2006) also claim that the idea has grown from the concepts of ubiquitous computing, pervasive computing and ambient intelligence.

2.4.2 Auto ID project

One of the associations trying to achieve the IoT vision is Auto ID (Auto-IDLabs, 2012). Auto ID is a network of academic research laboratories in the field of networked RFID (MIT, 2012). It consists of seven universities from four different continents. It is also the name of a system used in intelligent manufacturing control and supply chain management (McFarlane et al., 2003).

Traditionaly, the equipment/resources used to manufacture and distribute products are already networked. However products themselves were not (McFarlane et al., 2003). Products only contain an identifier (usually a bar code) that can be scanned by some equipment. The location of the product is then estimated based on the location of the equipment used to scan it. Also, in order to obtain the location, a scan has to be performed and therefore real-time location awareness is not always possible. The goal of Auto ID is to provide an infrastructure where each product is given a so-called *Smart Tag* (McFarlane, 2002). Then a set of wireless sensors is capable of locating the product and obtaining its identity based on the Electronic Product Code (ePC) written on the Smart Tag. This infrastructure is supported by a service similar to DNS (Domain Name System). Also, the products are described using so-called Product Markup Language, similar to HTML (Hypertext Markup Language). This architecture enables the concept of networked products, not only networked resources.

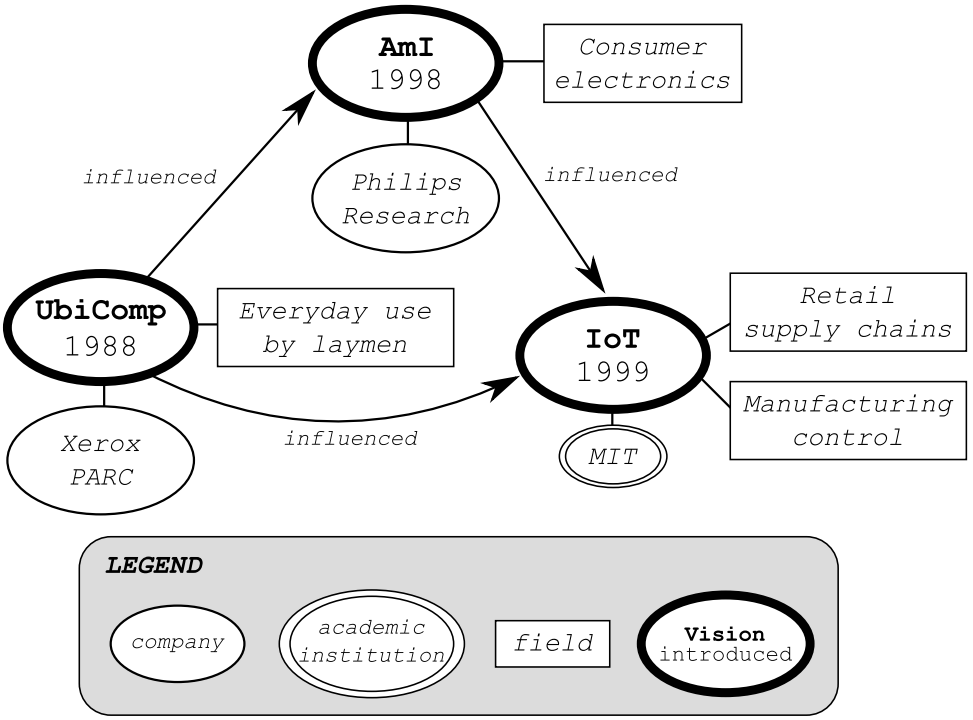


FIGURE 2 Various research visions and their relationship

2.5 The relationship between the visions

We have introduced three important visions related to the topic of this thesis – Internet of Things (IoT), Ubiquitous Computing (UbiComp) and Ambient Intelligence (AmI). These three terms are understood in many ways, some synonymously, depending on the literature that one is reading. However, if we track these terms historically, we can see that originally many of them were driven by different motives and were aimed at different domains. The oldest of these three is UbiComp and it is intended for the broadest audience – for everybody. The AmI vision was mostly concerned with consumer electronics, since it was developed by a consumer electronics producer. The IoT vision was developed by an academic institution. It is associated with the RFID technology and it targets the area of retail supply chain and manufacturing control. Probably the biggest contrast is between the IoT and UbiComp visions. While UbiComp puts emphasis on humans as technology users, IoT mostly concentrates on machine-to-machine interactions.

These visions intersect. All of these technologies are trying to lower the complexity of human-machine or machine-machine interactions. Some of the technologies and principles can be applied to all of them. For example, the RFID technology is useful both in the area of UbiComp and AmI. Also, ubiquitous computing environments can accelerate the adoption of the other two visions. The vision of fully integrated consumer electronics coming from the area of AmI is fully compatible with the vision of UbiComp. The relationship between these three technologies can be seen in Figure 2.

2.6 Agents and agent-based systems

2.6.1 What is an agent?

The word *agent* has many meanings inside and outside the world of Computer Science. We start our exploration of agents by first looking at the word itself. According to Merriam-Webster dictionary, the word *agent* has the following meanings:

- 1: one that acts or exerts power
- 2a: something that produces or is capable of producing an effect : an active or efficient cause
- 2b: a chemically, physically, or biologically active principle
- 3: a means or instrument by which a guiding intelligence achieves a result
- 4: one who is authorized to act for or in the place of another: as
  - a : a representative, emissary, or official of a government <crown agent> <federal agent>
  - b : one engaged in undercover activities (as espionage) : spy <secret agent>
  - c : a business representative (as of an athlete or entertainer) <a theatrical agent>
- 5: a computer application designed to automate certain tasks (as gathering information online)

The meaning 2b is related to chemistry and therefore it is not relevant for to this work. All the other meanings are very closely related to agents as Computer Science (CS) and Artifitial Intelligence (AI) understands them. However, even within these fields various meanings are provided. Wooldridge and Jennings (1995b) distinguish between the two views on agents – a weak notion of agency and a strong notion of agency. According to the weak notion of agency, an agent is a hardware or software-based system that has four properties – autonomy, social ability, reactivity and proactiveness. The definition of these properties is depicted in Table 3.

Autonomy is arguably the most important property that separates an agent from any other software (Wooldridge, 1997; Jennings and Wooldridge, 1995). As a result of this, an agent is usually implemented as a software component with its own thread of control.

We say that agents have social abilities, if they are able to communicate with each other. Agents communicate with each other using some kind of *agent communication language* (ACL). The communication is necessary in order to perform social actions such as negotiation, cooperation, etc. It is also important in order to communicate with the human, since in most cases the human is the one who is being represented by the agent.

TABLE 3 Agent's properties. Adapted from Wooldridge and Jennings (1995b)

Property	Definition
autonomy	agents operate without the direct intervention of other entities, and can control their actions and internal state
social ability	agents interact with other agents (and possibly humans)
responsiveness	agents perceive (sense) their environment and respond to changes that occur in it
proactiveness	agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative

Reactivity and proactivity (or proactiveness) are closely related. Reactivity is a lower form of behavior. A reactive agent just perceives the environment and based on the input and its current state it performs an action that potentially changes the environment. On the other hand, proactivity is a higher form of behavior. Proactive agents are goal-driven. It means that they don't perform an action just because some change was sensed. They do so, because they are trying to achieve some goal. They plan ahead.

Apart from the weak notion of agency, Wooldridge and Jennings (1995b) offer the so-called strong notion of agency. According to it, an agent has all the properties mentioned in Table 3, however it should also be designed and/or implemented using concepts related to humans and human thinking. Various researchers use various viewpoints to achieve it. Shoham (1993) uses so-called mentalistic notions such as knowledge, belief, intention and obligation. Rao and Georgeff (1995) use another set of mentalistic notions to model and implement agents – beliefs, desires and intetions. Bates (1994) uses emotions to design and implement agents. These are just a few of them.

According to Wooldridge (2002), most of the researchers agree that an agent should be autonomous. However, they do not always agree on the rest of the properties. An *intelligent agent* is and agent capable of flexible autonomous action in order to meet its design objectives. Flexibility is achieved by three properties – reactivity, pro-activeness and social ability.

A simple and wide definition of an agent is offered by Russell and Norvig (2003) in their book "Artificial intelligence: Modern approach":

> An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators (Russell and Norvig (2003), page 32).

There are three important words mentioned in this definition – sensors, actuators and environment (see Figure 3). Environment is a place where the agent resides. It can be virtual (e.g. computer program) or physical (e.g. real world robot). A sensor is something that is used to perceive the environment. In the case of physical environment, it can be some equipment to sense light, sound, movement and

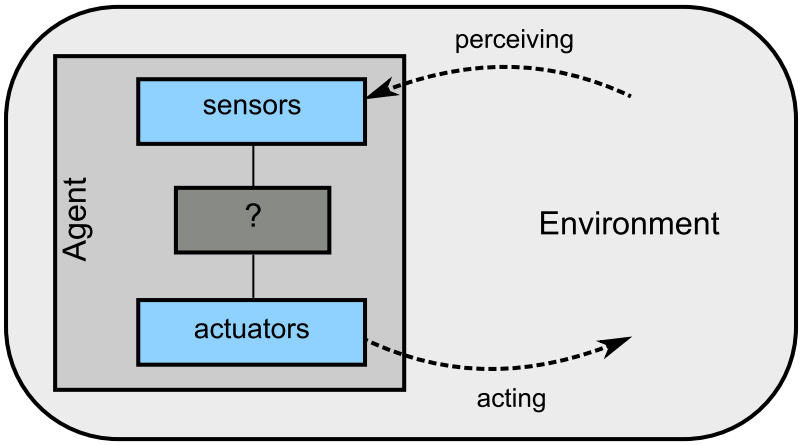


FIGURE 3 Scheme of a typical agent

others. Or, in the case of virtual environments, it can be a piece of code capable of reading files, web pages or computer's hardware status. An actuator is something used to influence the environment. In physical environments it could be a robotic arm, speaker, light, etc. In virtual environments it can be a file writer, database writer and so on. Simply put, sensors are agent's eyes and ears, whereas actuators are agent's arms and legs.

Russell and Norvig (2003) also provide a more narrow definition of an agent. A *rational agent* is an agent that always chooses the most benefitial action based on the sequence of perceived data and the built-in knowledge. The level of benefitiality of an action is determined by a *performance measure*.

For the purposes of this dissertation, we will use the weak notion of agency as defined by Wooldridge and Jennings (1995b). In other words, we will not require agents to be designed using concepts related to the human mind, but we will require the four properties depicted in Table 3.

According to Jennings (1999), a single agent might be insufficient when trying to model a real-world scenarios using an agent-oriented view. On the other hand, multi-agent systems (MAS) can easily be used to model decentralized control, cooperation, competition, etc. Often, agents are used as intermediaries that act on behalf of other entities (e.g. people or organizations). In general, multi-agent systems are dynamic and the interactions among agents change in time. Since agents are autonomous and social, agent interactions might lead to unpredictability at runtime (Jennings, 1999). This unpredictability stems from the fact that the behavior of the agent system cannot be understood only as a sum of behaviors of individual agents. A potential solution to this problem is the use of *organizational context* by explictly modelling organizational relationships and structures.

2.6.2 Agents and their influence on programming

The term agent-oriented programming (AOP) was coined in 1988 by Yoav Shoham (Shoham, 1997). Traditional programming paradigms like structural programming

or object-oriented programming (OOP) deal with terms like program, method, function, object, property, etc. AOP is based on congnitive and societal view of computation. As mentioned earlier, Shoham (1993) clearly defines agents as entities, whose state is defined using terms like beliefs, capabilities, choices and commmitments. AOP is then a programming paradigm, where the programs consist of agents and agents consist of elements mentioned previously. The agents communicate by exchanging speech acts.

Shoham (1997) argues that a complete AOP framework should contain the following three elements. First of all, a formal language for description of agent's mental state must exist. Secondly, an interpreted language for agent programming should be used. This language would depend on the mental state description language. Lastly, there should be a method of conversion of neutral devices into agents.

Shoham (1991) presents an AOP system based on the Agent-0 language. The agent's mental state is described using quantified multi-modal logic. This allows the agent to reference to time. The agent programming language is called Agent-0, where actions are triggered by commitment rules. The action can either be communicative (sending a message) or private (triggering an internal subroutine). One of the drawbacks of Agent-0 was the inability to create plans (Wooldridge and Jennings, 1995a).

Apart from AOP, there are also other agent-based programming paradigms, such as agent-based software engineering (Genesereth and Ketchpel, 1994) and agent-oriented software engineering (AOSE) (Wooldridge, 1997; Jennings, 1999). Strictly speaking, these are two different paradigms, but they share many similarities. Jennings (1999) presents a hypothesis that for certain types of problems, a multi-agent approach to system development brings benefits over contemporary methods. Agent interactions are facilitated using a high-level communication language, often called agent communication language (ACL). In contrast to traditional systems, in MAS, agents can exchange not only the data, but also logical information, commands and whole scripts (Genesereth and Ketchpel, 1994). The inter-agent messages are interpreted as speech acts. Moreover, agents are capable of reasoning and planning, which allows them to make context-dependent decisions.

2.6.3 Agent-related standards

A large number of multi-agent platforms has been developed in 1990s. As a response to this, in 1996, the Foundation for Intelligent Physical Agents (FIPA) has been formed (FIPA, 2012). The goal of this organization is to promote agent-based technology and the interoperability of its standards with other technologies. FIPA provides a collection of open standards from varios areas such as agent communication, agent architectures, agent management and others. According to FIPA00001 named "FIPA Abstract Architecture Specification", an agent system consists of an agent platform hosting agents and services (FIPA, 2002a). Each agent has a unique identifier and one or more transport addresses. FIPA00001

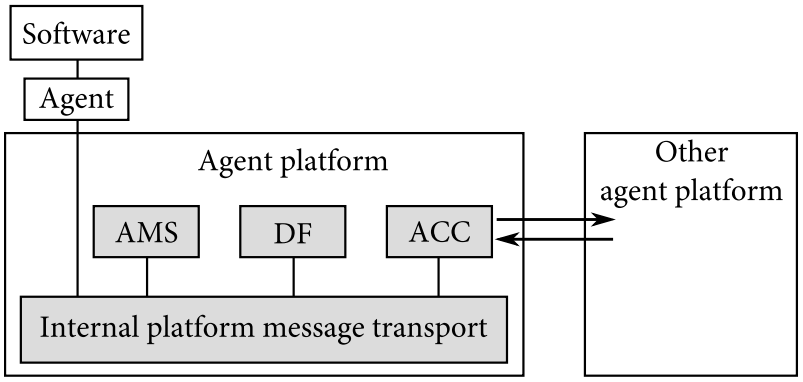


FIGURE 4 FIPA abstract agent platform (FIPA, 2002a)

also specifies agent's lifecycle by defining six states in which agent may exist. According to FIPA00023, each agent platform contains an agent management system (AMS) acting as an agent directory (FIPA, 2004). An agent platform may contain a directory facilitator (DF) that stores the information about agent descriptions and their capabilities. Several DFs are allowed. A typical FIPA platform architecture can be seen in Figure 4.

The agents communicate by exchanging messages encoded in an ACL (FIPA, 2002c). Each ACL message has a set of parameters associated with it. *Performative* is the only mandatory parameter. It represents the nature of the communication with respect to the speach act theory. Other parameters determine the sender, receiver, content, language, protocol, etc. FIPA00025 and FIPA00037 define typical patterns of message exchange called interaction protocols (FIPA, 2003, 2002b). An example of such a protocol is a request interaction protocol that allows one agent to request an action from another agent. The interaction protocol defines what type of messages (e.g. performative) is allowed at certain points of message exchange. It also clearly defines the acceptance or denial of an action. There are several other protocols defined by FIPA, such as Query, Request, When or Contract Net.

One of the most well-known agent platforms following FIPA is JADE (Java Agent Development Framework). As the name suggests, it is a software framework for the development of multi-agent applications written in Java programming language (Bellifemine et al., 2000). JADE is a distributed platform that may run on several computers. In order to achieve it, it uses a concept of an agent container, where each platform has exactly one main container and zero or more secondary containers (Bellifemine et al., 1999). The developer receives a toolkit consisting of the platform runtime, Java class library used to build JADE agents and debugging tools. Each agent is implemented as a Java object by extending a specific Java class provided by the toolkit. Each agent may have several behaviors, which are implemented as Java classes as well. Agents communicate by exchanging FIPA-compliant messages.

2.7 Semantic Web

2.7.1 From World Wide Web to Semantic Web

The World Wide Web (WWW) is nowadays one of the most important services on the Internet . It became so well-known that some people wrongly assume that WWW and the Internet are the same thing. The father of the World Wide Web is considered Tim Berners-Lee (Berners-Lee, 1999). In 1980, while he was working at CERN (European Organization for Nuclear Research), he realized that a lot of data and software was being lost – not due to technical failures, but due to organizational problems. People working at CERN were hierarchically organized. However, they were communicating with each other across all levels – upstream, downstream, cross branches, etc. Sometimes this led to loss of information or duplication of effort. Also, there was a very high turnover of people working at CERN. People leaving a project were often taking valuable knowledge with themselves. Another problem was that the environment was very dynamic. New experiments were introduced, old experiments modified, etc. On a daily basis researchers were facing questions such as: Which document does this refer to? Who wrote this code or document? What does this software module depend on? Which organizations were involved in this project?

Berners-Lee assumed that all these questions could be answered if there was a network of linked documents, people, projects, institutions, etc. He started to call this idea *the web*. He developed an application called Enquire that implemented this idea. Enquire was the predecessor of WWW (Berners-Lee, 2012). The idea of the web was to link various nodes with each other and thus increase the value of the data they represent. As mentioned earlier, these nodes were people, groups, software modules, projects, documents, concepts, etc. Nodes were connected with each other using directed arcs (links) such as “depends on”, “refers to” and so on. Berner-Lee’s web was therefore a directed graph (see Figure 5).

The basic idea of Enquire eventually evolved into the World Wide Web (WWW) (Berners-Lee, 1993). WWW documents (Web pages) refer to each other using hyperlinks. A typical Web page is written in the Hypertext Markup Language (HTML). HTML is a text enriched by the so-called tags (Figure 6). Tags are used to express text formatting, hyperlinks to other documents and so on. HTML was designed to create documents that are visually appealing and easy to read by humans. However, a machine (computer) is able to obtain very little information from HTML documents due to the fact that these documents are too complicated and unstructured (Berners-Lee et al., 2001).

If also computers were capable of reading documents on the Web, it would allow them to make decisions and take actions on behalf of humans. There would be no (or very little) need for humans to be involved in certain tasks. Nowadays, a human is able to read weather forecast, stock exchange information, book a flight ticket or a hotel. If somehow computers would be capable of doing all this, people would just specify the criteria and some application (agent) would do it for them.

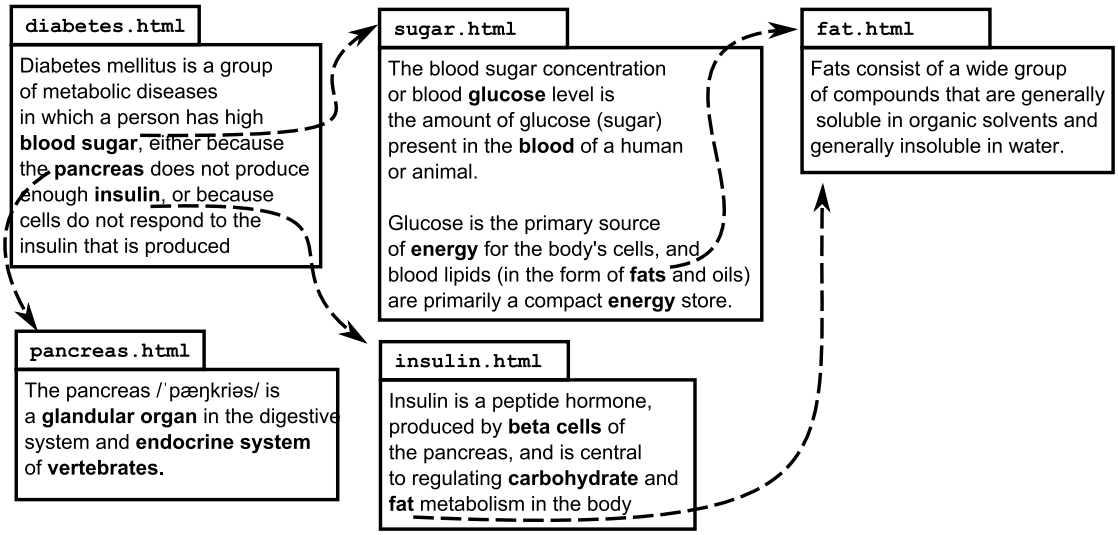


FIGURE 5 Example of the structure of Berner-Lee's original web

diabetes.html

```
<body>
  <div class="desc">
    Diabetes mellitus is a group
    of metabolic diseases in which a person has high
    <a href="sugar.html">blood sugar</a>, either because
    the <a href="pancreas.html">pancreas</a> does not produce
    enough <a href="insulin.html">insulin</a>, or because
    cells do not respond to the insulin that is produced
  </div>
</body>
```

FIGURE 6 Fragment of HTML text with tags printed in bold

Unfortunately, this is not yet possible on the traditional Web. We need a web for machines. Researchers came up with an answer to this problem, a vision called Semantic Web.

Semantic Web is not another type of Web. It is merely an extension of the current one . It builds on the same assumptions and principles as the Web for humans. The difference is that in the Semantic Web computers have access to highly structured information together with sets of inference rules that they can use to perform logical reasoning.

2.7.2 Common features of World Wide Web and Semantic Web

Same as in Berners-Lee's original Enquire, the main building blocks of the World Wide Web are nodes (documents) and links. Both Enquire and WWW are using the idea of hyperlinks and hypertext (Berners-Lee, 1993). According to the Merriam-Webster dictionary, hypertext is "a database format in which information related to that on a display can be accessed directly from the display" (Merriam-Webster, 2013). A hyperlink then provides access from one place in a hypertext to another. It is important to note that the hyperlinks are only unidirectional. The existence of a link from A to B does not automatically imply the existence of a link from B to A. In other words, there are no "backward links", unless they are explicitly specified. One of the implications is that the creator of a certain document on the Web cannot know how many other documents refer to it. This is true for WWW as well as for Semantic Web.

Another feature of the World Wide Web is so-called *AAA principle*. The *AAA principle* says that "Anyone can say Anything about Any topic" . It means that every user of WWW is free to publish any information about any topic and refer to already existing documents (Web pages or others) potentially written by somebody else. This is not a juridical rule – law. It is a technical rule. The WWW is technically designed with this principle in mind. Nowadays it is obvious, but in the early days of WWW it created a lot of confusion, which was the fuel for many myths. Some myths are presented on Berner-Lee's Web page (Berners-Lee, 1997). Web page owners thought that if some person wants to refer to their Web page, that person must obtain a permission from the creator. Some even thought that it gives them the right to ask money for every link pointing to their document. Another myth was that linking to their documents is a copyright infringement. None of these are true. Nowadays it might seem obvious, but in the past WWW was too novel for non-professionals to understand.

2.7.3 Ontologies

As mentioned earlier, Semantic Web is an evolutionary step from the World Wide Web. Whereas WWW is a web of documents for people, Semantic Web is a web of documents for machines (computers). The World Wide Web utilizes HTML as the main language for document creation. HTML documents are easy to read by humans, but almost impossible to understand by computers. In general there are

two ways to deal with this problem – make computers smart enough to understand the current WWW, or make the data simple enough for the current computers to understand it. Semantic Web is trying to achieve this goal using the second approach. The goal is for the Web to contain data that is written in a highly structured form easily understandable by a machine.

Once the computers are able to understand the meaning of data, they can interpret it and thus obtain information. Information can then be used to determine other logically-related facts. The process of computing logically-related consequences is called *reasoning*. In order to reason about the data, machines need to be given some understanding of the domain including or some reasoning rules. For example if a computer understands that person A is married to person B without any additional knowledge, it is missing a very important point – that also person B is married to person A. Humans naturally understand it, but a machine needs to know that being married is a symmetric property. Some argue that there should be one language that can express both data and rules (Berners-Lee et al., 2001). Others argue that it is sufficient to have two separate languages .

Apart from the ability to reason about the data, there should be some mechanism that can verify the consistency of the data. For example if the data says that "dog Rocky is married to dog Daisy", then humans naturally conclude that it does not make sense, because only humans can marry each other. In other words, the data is inconsistent with respect to the "being married" property. Again, humans know this naturally, but machines require an explicit definition of "being married" – a definition which specifies that only humans can be married. Therefore there is a need for some specification containing information about what kind of connections between things (e.g. people, dogs, food, feelings, etc.) are allowed to be made. We call this specification an *ontology*.

The word ontology comes from the philosophy. According to the Merriam-Webster dictionary, ontology is "a branch of metaphysics concerned with the nature and relations of being" (Merriam-Webster, 2013). The meaning of the word ontology in the domain of the Semantic Web is similiar. According to van Harmelen and McGuinness (2004), an ontology is used to "explicitly represent the meaning of terms in vocabularies and the relationships between those terms".

A narrower definition is provided by Noy et al. (2000). According to them, an ontology formally and explicitly describes a domain of discourse in terms of concepts (classes), properties (slots, roles) and property restrictions (facets, role restrictions). Classes describe groups of concrete or abstract entities (e.g. student, skill, bread). A property describes features and attributes of a class. For example a student can have a name, student ID or address. In the human resource (HR) domain, properties of the class skill can determine the area of skill, level of skill, way it was learned, etc. Lastly, a property restriction describes property value types, allowed values, property cardinality and other features of properties. An example in the HR domain would be a restriction on the "level of skill" property specifying that the cardinality is either zero or one and the value can be an integer between 1 and 5.

Amann and Fundulaki (1999) formally define an ontology as a triple $O =$

FIGURE 7 Dependency among W3C standards for the Semantic Web

(C, S, isa) where:

1. $C = \{c_1, c_2, \ldots, c_n\}$ is a set of concepts, where each concept c_i refers to a set of real-world objects (concept instances),
2. $R = \{r_1, r_2, \ldots, r_m\}$ is a set of binary typed roles between concepts,
3. *isa* is a set of inheritance relationships defined between concepts. Inheritance relationships carry subset semantics and define a partial order over concepts.

2.7.4 Semantic Web and related standards

The Semantic Web standards are developed under the umbrella of the World Wide Web Consortium (W3C). The corner-stone of the Semantic Web is the Resource Description Framework (RDF). It is a language for representing information about resources in the World Wide Web (Miller and Manola, 2004). RDF relies on another stadnard called URI for unique representation of terms. RDF as a description language can be expressed in textual form in various ways. These ways are called serializations and the process of expressing RDF in these forms is called serialization. Some of the serializations are easier to read by humans, some are easier to read by machines. Another important standard by W3C is Web Ontology Language (OWL). As the name suggests it is a language that allows the user to describe an ontology. In 2012, a new version of an ontology language has been released by W3C. It has been named Web Ontology Language 2 (OWL2). It is based on OWL and it introduces new concepts and new sublanguages. In the next subsections we are going to describe these standards and their relationship. Figure 7 depicts which standards depend on each other.

2.7.4.1 Uniform Resource Identifier and related standards

URIs and related standards have a long history with several informative and normative documents by IETF (Internet Engineering Task Force) such as RFC 1630, RFC 2396 and RFC 2732. Some of these documents have been obsolete by newer ones and some of the documents are still valid. This text is based on the latest normative document RFC 3986 (Berners-Lee et al., 2005) and related valid (non-obsolete) standards. According to this specification, URI is defined as a "compact sequence of characters that identifies an abstract or physical resource". Even though URIs are mostly used on the Internet (e.g. World Wide Web), URIs may identify both resources accessible via the Internet (e.g. Web pages or images) and resouces outside the Internet (e.g. real-world objects or abstract concepts). These resouces may be identified by name, by its location or both. URIs that identify resources by their location are called URLs (Uniform Resource Locators).

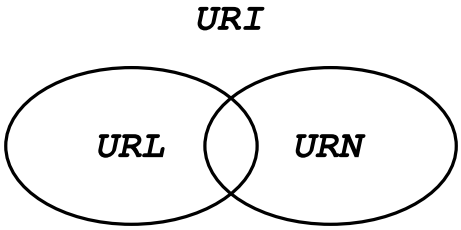


FIGURE 8 The relationship between URI, URL and URN

URIs that identify resources by their name are called URNs (Uniform Resource Names). In some cases a URI can be both URL and URN. The relationship between them is depicted in Figure 8.

The generic URI syntax as defined by RFC 3986 is as follows:

```
<scheme>:<hierarchical-part>["?"<query>]["#"<fragment>]
```

The `<scheme>` part is a string identifying the scheme. Based on the scheme, the `<hierarchical-part>`, `<query>` and `<fragment>` are interpreted. Using this approach URIs are extensible. Each new scheme should be registered at IANA (Internet Assigned Numbers Authority), which keeps track of existing URI schemes. This should happen according to the procedure described in RFC 4395 (Hansen et al., 2006). However, there is no technical restriction that would prohibit the use of private un-registred URI schemes. One of the most well-known shcemes is http that is used to locate network resources via the Hypertext Transfer Protocol (HTTP). Since this scheme is used to *locate* resources, we may consider http URIs also URLs. RFC 2616 defines HTTP/1.1 and according to it the http URL looks as follows (Fielding et al., 1999):

```
"http:" "//" <host> [":"<port>] [<abs_path> ["?"<query>]]
```

Another well-known schema used for remote access to terminals is telnet. According to RFC 4248 (Hoffman, 2005), a telnet URL looks as follows:

```
telnet://<user>:<password>@<host>:<port>/
```

Even though there might be some similarities in the scheme specific part of various schemes (e.g. in telnet and http), these parts are different and in general they are not interpreted the same way. The interpretation is always given by the scheme specification. One of the limitations of URIs is the fact that according to the specification, one can use only US-ASCII characters to identify a resource. As URIs became more popular and started to be used for identification of resources in other languages than English, this became a serious issue.

Internationalized Resource Identifier (IRI) is a complement to URI defined in RFC 3987 (Dürst and Suignard, 2005). IRIs use Unicode characters, while URIs use a subset of US-ASCII characters. The introduction of Unicode characters allows the use of non-English characters, including non-Latin characters (e.g. Greek, Chinese

FIGURE 9 An example of an RDF graph

or Cyrilic). For the reasons of compatibility, IRIs can be mapped to URIs and vice versa. The syntax of IRIs is the same as the syntax of URIs defined in RFC 3986 with one exception. The class of unreserved ("useable") characters is extended by Unicode characters starting from U+0080 (beyond the original US-ASCII block). Characters other than US-ASCII must not be used for syntactical purposes.

2.7.4.2 RDF

The main goal of RDF is to provide a language that can be used to represent information about resources in the World Wide Web (Miller and Manola, 2004). Apart from representing Web resources, RDF can be used to describe resources that can be identified on the Web, but are not retrievable from the Web (e.g. people or real-world objects). RDF relies on URI for identification of things (on and off the Web) and their properties. The information about resources is expressed in a form of a directed graph. Nodes of the graph represent resources (e.g. human John, Web page of W3C, my car) or property values (e.g. green, 35, true). The directed edges of the graph represent relationships (properties) between two resources, or between a resource and a property value.

An example of such a graph can be seen in Figure 9. One can notice three types of elements used – circles, rectangles and arrows. Circles represent resources and they are identified by URIs. Rectangles represent property values – in this particular case string values. In general, property values are expressed by literals such as numbers, boolean values true/false, etc. Arrows represent properties and are indentified by URIs. Therefore we may say that an RDF graph can consist of three elements – resources, properties and property values. Both resources and properties are expressed using URIs. Property values are expressed using literals.

In general, there are several ways of storing the graph representation of RDF data. These ways are called serializations. The RDF specification provides a serialization called RDF/XML, that expresses the data in a form of an XML

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:org="http://www.jyu.fi/organization#"
   xmlns:pim="http://www.w3.org/2000/10/swap/pim/contact#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://users.jyu.fi/jdoe">
    <pim:lastName>Doe</pim:lastName>
    <org:worksFor rdf:resource="http://www.jyu.fi/organization#OntoGroup"/>
    <pim:firstName>John</pim:firstName>
    <rdf:type rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#Person"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.jyu.fi/organization#OntoGroup">
    <org:belongsTo rdf:resource="http://www.jyu.fi/organization#jyu"/>
  </rdf:Description>
</rdf:RDF>
```

FIGURE 10 An example of an RDF/XML document

document (Miller and Manola, 2004). This serialization is useful for data exchange between computers due to the fact that many programming languages and computer systems contain XML parsers. However, RDF/XML serialization is not easily readable for humans. The example of RDF data from Figure 9 serialized in RDF/XML is shown in Figure 10.

The RDF model is based on so-called triples. A triple is a statement consisting of three elements – subject, predicate and object. Subject represents a resource in question. Predicate represents a property of that resource and object represents a value or another resource. One triple can be graphically expressed as two nodes (subject and object) connected with a directed link (predicate). The direction of the link determines which of the nodes is the subject and which is the object. The link always travels from the subject to the object. Based on this knowledge an RDF document can be described as a set of subject-predicate-object triples (SPO triples). Sometimes SPO triples are called SPO statements or just statements.

As mentioned earlier that RDF elements can either be expressed using URIs (resources and properties) or literals (property values). A subject of a statement can only be a URI. There has been some discussion among W3C members about the possibility of having literals as subjects, but at the moment it is not directly supported (W3C, 2013). A predicate of a statement also can only be a URI. Finally, the object of a statement can be either a URI or a literal.

RDF is an assertional langauge capable of expressing propositions using formal vocabularies (Hayes, 2004). Statements in RDF act as propositions. Subjects, predicates and objects of a statement may be defined in a vocabulary with a precise meaning. In the example in Figure 9 one can see the predicate `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`, which has been defined in a vocabulary `http://www.w3.org/1999/02/22-rdf-syntax-ns`. This predicate defines the relationship between the subject and object in the following way: subject of this stament belong to the class defined by the object. In our example, this means that the resource with URI `http://users.jyu.fi/jdoe` belongs to the class `http://www.w3.org/2000/10/swap/pim/contact#-Person`. In other words it means that John is a person. In general predicates used in an RDF document can be defined in the same vocabulary or in different

```
<http://users.jyu.fi/jdoe> <http://www.w3.org/2000/10/swap/pim/contact#firstName> "John" .
<http://users.jyu.fi/jdoe> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> %
<http://www.w3.org/2000/10/swap/pim/contact#Person> .
<http://users.jyu.fi/jdoe> <http://www.w3.org/2000/10/swap/pim/contact#lastName> "Doe" .
<http://users.jyu.fi/jdoe> <http://www.jyu.fi/organization#worksFor> %
<http://www.jyu.fi/organization#OntoGroup> .
<http://www.jyu.fi/organization#OntoGroup> <http://www.jyu.fi/organization#belongsTo> %
<http://www.jyu.fi/organization#jyu> .
```

FIGURE 11 An example of an RDF document in N-Triples notation

vocabularies. We will discuss the topic of vocabularies in Section 2.7.4.4.

2.7.4.3 RDF serializations

As mentioned earlier, RDF/XML is not suitable for data exchange between humans due to its complexity. Knowing that RDF data can be expressed as a set of SPO triples, this method has become popular among the W3C contributors and has led to a serialization called N-triples (Carothers, 2013). As of October 2013, N-triples specification has the status of "Last Call Working Draft" and it is intended to become a W3C standard. According to the specification "N-Triples is a line-based, plain text format for encoding an RDF graph" (Carothers, 2013). An N-triples document is a plain-text document consisting of SPO statements, one statement per line. Each statement consists of three strings (URIs or values) separated by either a tab or a space. Each statement must end with a dot (".") and a new line. The N-triples representation of RDF data from Figure 9 can be seen in Figure 11. Due to the fact that URIs used in this example are too long and they do not fit the page width the `%` symbol is used at the end of the line to indicate that the line continues. Note that in N-Triples one is not allowed to insert a new line between the S, P and O portions of a statement.

N-triples is a step forward towards more human readable RDF data. However, it has one serious limitation – all URIs must be expressed using their full names. Due to the "one statement per row" rule and the fact that full URI names must be used, the document becomes very wide (a single line is very long). Also, from the N-triples example one can clearly see that some portions of the URI are being repeated throughout the document, which is wasteful.

Another type of serialization is Turtle (Terse RDF Triple Language). At the time of writing (October 2013) Turtle is a W3C Candidate Recommendation (Carothers and Prud'hommeaux, 2013). According to the W3C Turtle document, Turtle is trying to express RDF data in "a compact and natural text form, with abbreviations for common usage patterns and datatypes". The full syntax of Turtle goes beyond the scope of this work. However, we would like to point out several important differences between Turtle and N-triples. Every N-triples document is also a Turtle document. The Turtle syntax adds several new features to the N-triples syntax. Table 4 contains a list of five major differences marked from D_1 to D_5.

The first difference implies that a Turtle document does not have to contain

TABLE 4 A comparison between various RDF serializations

Diff	N-Triples	Turtle
D1	Only tabs and spaces are condisered white spaces	Tabs, spaces, new lines and comments are considered white spaces
D2	Encoded in ASCII	Encoded in UTF-8
D3	–	Introduction of @prefix
D4	No QNames possible	QNames allowed
D5	–	New abbreviations such as ",",";","[]" and "()"

exactly one statement per line. One can put either several statements per line or a portion of a statement per line. The D_2 difference means that one can use Unicode characters in a Turtle document, which is very important if string values are in a language different than English. The third and fourth difference (D_3 and D_4) are closely related. As mentioned earlier, in a N-Triples document some portions of the URI are being repeated throughout the document. More precisely, one can see that some URIs share the same begining. D_3 and D_4 allow URIs to be abbreviated and thus improve readability of the document. D_3 introduces a special statement in the following form:

```
@prefix <prefixName>: <uriref>
```

The `<prefixName>` parameter is optional and it indicates the name of the prefix that can later be used in the document. The `<uriref>` parameter defines the starting portion of the URI. Once a prefix is defined, one can use QNames (D_4) with defined prefix names instead of URIs. The last difference (D_5) introduces new constructs that abbreviate the statements. The symbol ";" is used in cases when several statements are describing the same subject. One can then use it in the following form: $SP_1O_1; P_2O_2; P_3O_3$. The symbol "," is used in a similar way. When both the subject and the object are the same, one can use the following form: SPO_1, O_2, O_3. The two remaining symbols are used for blank nodes and lists. For more information about these constructs, refer to Carothers and Prud'hommeaux (2013). Figure 12 shows the data from Figure 11 written in Turtle. One can see that such a document is more readable and compact.

Another approach to RDF serialization is Notation3 (shortly N3). Notation3 is a superset of RDF. Apart from providing a syntaxt for RDF, it also adds formulae, variables, logical implications and functional predicates (Berners-Lee and Connolly, 2011). N3 exists of a form of a W3C Team Submission and it is not a W3C recommendation. The addition of formulae makes it possible to annotate not only resources, but also sets of statements. The set of statements that is being annotated is enclosed in brackets. Variables in N3 are expressed either as a string starting with a question mark "?" or with a semicolon ":". By making formulae and variables possible, one can also define logical implications. An implication is a statement

```
@prefix con: <http://www.w3.org/2000/10/swap/pim/contact#> .
@prefix org: <http://www.jyu.fi/organization#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://users.jyu.fi/jdoe> rdf:type con:Person ;
    org:worksFor org:OntoGroup ;
    con:firstName "John" ;
    con:lastName "Doe" .

org:OntoGroup org:belongsTo org:jyu .
```

FIGURE 12 An example of an RDF document in Turtle notation

```
@prefix org: <http://www.jyu.fi/organization#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

{?x org:worksFor org:OntoGroup} => {?x rdf:type org:OntoGroupMember} .
{?x org:worksFor ?org . ?y org:worksFor ?org}
  => {?x org:coworkerOf ?y . ?y org:coworkerOf ?x} .
```

FIGURE 13 An example of an RDF document in Notation3

with subject and object being a container. The predicate is a "=>" symbol, which is a shorthand for `<http://www.w3.org/2000/10/swap/log#implies>`. An example of two implications can be seen in Figure 13.

The relationship between N-Triples, Turtle and Notation3 is depicted in Figure 14. We can see that N-Triples is a subset of Turtle and that is a subset of Notation3. That means that every N-Triples document is also a legal Turtle document. Similarly, every Turtle document is also a legal Notation3 document. For the remainder of this document we will use Turtle, unless specified otherwise.

2.7.4.4 RDFS

The previous sections introduced RDF and its four serializations as a formal way to express propositions. However, there also has to be a way to formally define vocabularies. This role can be played by RDF Schema (RDFS) (Guha and Brickley,

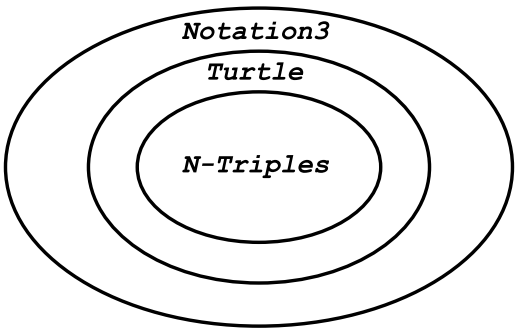


FIGURE 14 The relationship between various serialization methods

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix org: <http://www.jyu.fi/organization#> .

org:Student rdf:type rdfs:Class .
org:PhdStudent rdf:type rdfs:Class .
org:PhdStudent rdfs:subclassOf org:Student .
org:john rdf:type org:PhDStudent .
```

FIGURE 15 An RDFS example – class hierarchy

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix org: <http://www.jyu.fi/organization#> .

org:attends rdf:type rdf:Property ;
  ; rdfs:domain org:Student ;
  ; rdfs:range org:Course .
```

FIGURE 16 An RDFS example – property definition

2004). Using RDFS one can define classes, class hierarchy, properties, lists, sets and other concepts.

Classes are groups of resources sharing some common signs. RDFS allows one to specify that a certain URI is a class by expressing that this URI is of type `rdfs:Class`. One can also define the class hierarchy by using `rdfs:subclassOf` property. The example in Figure 15 specifies two classes (student and PhD student) and their relationship (every PhD student is also a student). Also, by using the `rdf:type` predicate, it indicates that the resource `org:john` is a phd student.

Any resource that is a member of `rdf:Property` class is considered a property. In order for the property definition to be complete, one should define its domain and range. Both the domain and the range are classes. If S P O is a statement and P is the property one is trying to define, then S must be a member of the property domain and O must be a member of the property range. In other words, if one defines a property `org:attends` with `org:Student` as the domain and `org:Course` as the range, then one is only allowed to make statements `X org:attends Y` where `X` is is a student and `Y` is a course. RDFS allows definitions of property hierarchies using `rdfs:subPropertyOf`. An example of a property definition is shown in Figure 16.

2.7.4.5 OWL

Another language used to represent the meaning of terms in vocabularies and the relationship between the terms is OWL (van Harmelen and McGuinness, 2004). OWL has higher expressive power than RDFS, which makes it more suitable for machine reasoning. OWL is a W3C Recommendation and it is closely related to XML, XSD, RDF and RDFS. Note that there is also OWL2, which is an improved

version. It will be discussed later in the text.

OWL consists of three langauges with increasing expressive power – OWL Lite, OWL DL and OWL Full. More precisely, OWL Lite is a subset of OWL DL and that is a subset of OWL Full. OWL Lite is a simple language used for classification and taxonomies. Due to its simplicity an OWL Lite reasoner is relatively fast in comparison to OWL DL. OWL DL provides the maximum of expressive power while keeping computation complete and decisive. DL stands for description logic – the formal foundation of OWL. Lastly, OWL Full provides the most expressive power with no guarantees for any computational characteristics. There are many reasonser capable of full reasoning within OWL Lite or OWL DL. However, it is very unlikely that there will ever be a reasonser capable of full reasoning within OWL Full .

The full description of OWL's capabilities is beyond the scope of this text. Only a few important features of OWL DL will be described. The first difference between RDFS and OWL DL is the ability to describe properties more precisely. One can define if a property is an object property or a datatype property. An object property allows only URIs as the objects of a statement – e.g. `org:attends` from an earlier example. A datatype property allows only literals as the objects of a statement – e.g. `con:firstName`, where a person's name as a string is expected. Object properties and datatype properties are disjoint in OWL DL (Schreiber and Dean, 2004).

Both object and datatype properties can be defined as functional (`owl:FunctionalProperty`). It means that such a property has no more than one value for a given individual. In addition to that, two object properties (`x:p1` and `x:p2`) can be specified as inverse. Property `x:p1` is inverse of `x:p2` if and only if:

```
{?a x:p1 ?b} => {?b x:p2 ?a}
```

An example of such properties is employeeOf and employerOf. Moreover, an object property may be specified as symmetric, transitive and inverse functional. A symmetric property `x:p1` has the following feature:

```
{?a x:p1 ?b} => {?b x:p1 ?a}
```

An example of a symmetric property would be marriedTo. For a transitive property `x:p1` the following is true:

```
{?a x:p1 ?b . ?b x:p1 ?c} => {?a x:p1 ?c}
```

A very well known transitive property is the mathematical property $\leq$. Property `x:p1` is inverse functional if and only if for every `O` in `S x:p1 O` there is exactly one `S`. In other words, the object of a statement with `x:p1` as the predicate fully determines the subject. For example `x:hasStudentID` is inverse functional, because no two students have the same ID. Finally, properties can have minimal cardinality and maximal cardinality associated with them. The cardinality defines how many statements with the property can be made about the same subject. For

example `x:hasBiologicalParent` is a property that has cardinality exactly two, because each human has exactly two biological parents. However, property `x:hasChild` has allowed cardinality range from 0 to n.

OWL DL introduces new concepts for class descriptions as well (Hayes et al., 2004). A class can be defined by enumerating all the individuals (instances) that belong to the class. This is not possible in OWL Lite or RDFS. Such a class is called enumerated class. Moreover, a class can be defined as a union, intersection or compliment of other classes. Also, one can describe relationships between two classes such as disjointness or equivalence. Sevaral examples of OWL ontologies can be found in as appendices. Note that while metaclasses are allowed in OWL Full, they are illegal in OWL DL (Schreiber and Dean, 2004). A metaclass is a class of classes. For example, class Vehicle would be a metaclass, if Car was a class and Car would also be an instance of Vehicle.

An DL-based ontology consists of three parts – TBox, RBox and ABox (Fokoue et al., 2006). TBox expresses relationships among concepts such as class-subclass relationship or class equivalence. An example of a TBox assertion would be `x:Human rdfs:subClassOf x:Mammal`. RBox contains relationships between roles (properties), e.g. property-subproperty relationships. An example would be `x:hasFather rdfs:subPropertyOf x:hasParent`. Finally, ABox consists of statements about individuals using terms from TBox and RBox. An example is a statement `x:John x:hasFather x:Bill`. In general, there can be ontologies with various TBox-RBox-ABox ratios. Some ontologies might have a very simple taxonomy (small TBox) and many descriptions of individuals (large ABox). Others might be the oposite – complex TBox with very little ABox.

2.7.4.6 OWL2

OWL2 is an ontology language based on OWL. Every legal OWL ontology is also an OWL2 ontology (W3C, 2009). OWL2 is a W3C Recommendation, which has been released in two editions. This work focuses on the second edition, which became a recommendation in December 2012. Similarly to OWL, OWL2 ontologies can be written in OWL2 DL or OWL2 Full. OWL2 DL is a restricted subset of OWL2 Full (Krötzsch et al., 2012). There are two ways to interpret OWL2 ontologies – using RDF-based Semantics (Carroll et al., 2012) or Direct Semantics (Horrocks et al., 2012). Direct Semantics interprets OWL2 in terms of Description Logics and under this interpretation the reasoning problem is decideable. This allows the construction of OWL2 reasoners in Decription Logics. The RDF-based Semantics interprets OWL2 ontologies in a way similar to RDFS. According to Krötzsch et al. (2012) RDF-based Semantics is an extension of RDFS semantics. Under the RDF-Based Semantics OWL2 Full is undecidable and therefore an OWL2 Full reasoner would not always be able to return an answer about the correctness of the ontology or data.

OWL2 introduces three profiles – OWL2 EL, OWL2 RL and OWL2 QL. Profiles are language subsets that guarantee good computational properties for a particular task. OWL2 EL is based on the EL family of Description Logics. This

property allows one to build ontologies with large amounts of classes (TBox) and properties (RBox), while still reason in polynomial time with respect to the size of the ontology (Wu et al., 2012). OWL2 RL represent ontologies, where reasoning can be implement using rule-based reasoning engines. OWL2 QL is implemented in a way that allows large amounts of individual assertions (ABox), while being able to respond to conjunctive queries in logaritmic time with respect to the size of the ABox. A query for OWL2 QL data can be rewritten into a standard relational query language (Wu et al., 2012).

There are several new features in OWL2 over the older OWL. This work mentions only a few of them. For more information, refer to W3C (2009) or Parsia et al. (2012). In OWL2 one can express property chains. This allows a property to be defined through a chain of two or more properties. For example hasGrandchild is a property that can be expressed through a chain of two hasChild properties. Therefore an individual has a grandchild x, where x is his/her child's child. Moreover, OWL2 can define new datatypes as a combination or restriction of the existing ones. For example one can define a datatype based on `xsd:integer` with the restriction that only values between 0 and 10 are allowed. Furthermore, in OWL2 one can define classed based on property cardinality restrictions. For example one can define a class `ex:LargeFamily` as a `ex:Family` class with at least 6 `ex:hasMember` properties.

2.7.4.7 SPARQL

SPARQL is a recursive abbreviation that stands for *SPARQL query language*. SPARQL is used to define queries across multiple RDF data sources that either store RDF data natively or expose a different form of data as RDF via some middleware (Prud'hommeaux and Seaborne, 2008). SPARQL fulfils a similar function as SQL language used in relational databases. The result of a SPARQL query is either a result set or an RDF graph. A result set is a set of variable-value mappings, similar to SQL result set. An RDF graph is a graph constructed based on the definition of a query. One can consider the RDF graph result a transformation of the original RDF data.

SPARQL exists in two versions – 1.0 and 1.1. Both of them are W3C Recommendations. The full description of SPARQL is beyond the scope of this work, therefore only basic syntax is described. For more information see "SPARQL 1.1 Query Language" documentation (Harris and Seaborne, 2013). A simple query returning a result set is shown in Figure 17. This query asks for first name and surname of each person that works for an institute/group belonging to the University of Jyväskylä. If the query was used on the data from Figure 12, it would return a set $\{\{?fn = John, ?ln = Doe\}\}$. Figure 18 shows a construct query, which returns an RDF graph. This query returns an RDF document consisting of statements describing people as employees of the University of Jyväskylä. One can see that in this particular case a similar result could be achieved by using a rule (e.g. Notation3 implication).

```
PREFIX con: <http://www.w3.org/2000/10/swap/pim/contact#>
PREFIX org: <http://www.jyu.fi/organization#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?fn ?ln
WHERE {
 ?p rdf:type con:Person ;
    org:worksFor ?inst ;
    con:firstName ?fn ;
    con:lastName ?ln .
 ?inst org:belongsTo org:jyu
}
```

FIGURE 17 A SPARQL example

```
PREFIX con: <http://www.w3.org/2000/10/swap/pim/contact#>
PREFIX org: <http://www.jyu.fi/organization#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT { ?p rdf:type org:JyuEmployee }
WHERE {
 ?p rdf:type con:Person ;
   org:worksFor ?inst .
 ?inst org:belongsTo org:jyu
}
```

FIGURE 18 A SPARQL construct example

2.8 Ontology visualization

In the upcoming chapters we will present various OWL-DL ontologies. Despite the fact that a textual representation of an ontology in OWL is formal and thus unambiguous, a visual representation depicts the ontology in a more human-readable way. In this section we discuss various ontology visualization techniques with respect to the ontologies being depicted later in the text.

Firstly, we provide the requirements for visualization of the ontologies. Secondly, available visualization methods are discussed. Moreover, an evaluation of visualization methods with respect to the requirements is presented. Lastly, we describe a visualization technique suitable for the category of ontologies that will be presented.

2.8.1 Requirements

We present three requirements for the ontology visualization techniques. The first requirement (R_1) states that the visualization method should be able to convientiently display the following type of ontologies. The ontologies have less than 30 frames (classes and instances) and thus according to Ernst and Storey (2003) they are considered to be very small. Moreover, TBox and RBox are considerabely larger than ABox. Lastly, the size and complexity of TBox in relation to RBox

varies.

The second requirement (R_2) deals with the medium being used to depict an ontology. The ontologies should be readeable both on-screen and in print. This implies that no interaction between the viewer and the medium is possible.

The third requirement (R_3) specifies that the visualization method should be able to express basic features of OWL-DL, namely:

- Classes – hierarchy, equivalence, disjointness, union, intersection, property restrictions
- Properties – type (object or datatype), domain, range, cardinality, symmetricity, asymmetricity, reflexiveness, irreflexiveness, functionality, inverse functionality
- Individuals – class membership, assertions

2.8.2 Analysis and evaluation of visualization techniques

Katifori et al. (2007) investigate various techniques for ontology visualization. Despite the fact that the words *technique* and *tool* are used interchangeably, we distiguish between these two. An ontology visualization technique is a set of principles according to which an ontology can be visualized. An ontology visualization tool is a software that visualizes an ontology according to one or more visualization techniques.

According to the survey by Katifori et al. (2007) there are six main visualization types. For easier reference, each of them will be given an identifier V_i. Firstly, an *indented list* (V_1) is a tree representation similar to the representation of file system structure. A typical representative of such a method is the class hierarchy representation in Protégé[2]. Secondly, *node-link and tree* (V_2) representation consists of a 2D or 3D graph representation of a taxonomy with a top-down or left-to-right layout. This approach is used by the OntoViz tool [3], IsaViz tool [4] or OntoTrack tool (Liebig, 2004). Third of all, *zoomable visualizations* (V_3) represent low-level entities inside their respective parent entities. The user is able to zoom in or out to see either low-level or high-level entities. An example of such a tool is Jambalaya (Storey et al., 2001) or Grokker (Rivadeneira, 2003). Moreover, *space-filling techniques* (V_4) represent the ontology by subdividing the screen space of a node among its children. Examples include TreeMaps (Shneiderman, 1992) and SequoiaView [5], which is based on Squarified treemaps (a variation of the original TreeMaps). The fifth group of techniques is called *focus + context or distortion* (V_5). In this technique a context is chosen by focusing on a certain entity within the ontology. The rest of the ontology is distorted (e.g. blurred) and only the node in focus with related nodes is displayed. A member of this category is for example TGVizTab (Alani, 2003). Lastly, *3D information landscapes* (V_6) visualize entities

[2] Available at http://protege.stanford.edu/

[3] Available at http://protegewiki.stanford.edu/wiki/OntoViz

[4] Available at http://www.w3.org/2001/11/IsaViz/

[5] Available at http://w3.win.tue.nl/en/research/research_computer_science/visualization/sequoiaview/

TABLE 5 Suitability of various ontology visualization techniques with respect to requirements R_1, R_2, R_3

Visualization	Name	R_1	R_2	R_3
V_1	indented list	+	+	-
V_2	node-link and tree	++	++	++
V_3	zoomable visualizations	++	-	+
V_4	space-filling techniques	++	-	+
V_5	focus + context or distortion	+	- -	+
V_6	3D information landscapes	++	- -	+

of an ontology in form of color-coded 3D objects on a plane. Properties between the entities are represented as links between the objects. File System Navigator (FSN) (Tesler and Strasnick, 1992) and Harmony Information Landscape (Wolf, 1996) are two representatives of such a technique. A similar approach was used by Khriyenko (2008) to represent RDF resource closeness based on a user-specified context. Note that the six presented visualization types are not disjoint.

The evaluation of the visualization techniques with respect to the requirements is summarized in Table 5. While an indented list (V_1) is suitable for representation of simple class hierarchies, the method is not suitable for the visualization of properties and multiple inheritance. V_2 is capable of very good visualization of class hierarchy (including multiple inheritance) and good visualization of role (property) relations. On the other hand it may become difficult to read for larger ontologies, especially when the amount graph edges increases. V_3 is suitable for on-screen visualization if zoomable visualizations are used (vector or high-resolution bitmap). However, it is not suitable for print due to the fact that it relies on the ability to zoom the visualization. V_4 suffers from the same issues as V_3. For larger or more complex ontologies it requires the zooming functionality, which is not available for print. Both V_5 and V_6 are irrelevant with regards to the requirements. V_5 is interaction-based, which is in conflict with the print requirement. V_6 requires 3D navigation, which is not possible in print. Based on the evaluation, the most suitable method is *node-link and tree* (V_2).

The next step is to analyze the available visualization methods within the V_2 group. IsaViz is a visual environment for browsing and authoring RDF models represented as graphs. The tool and its visualization method is suitable mostly for on-screen view, but it can also be used for print view. OntoViz provides a more compact representation, where a role (property) can be displayed both inside the class node and as a link between nodes. SpaceTree and Tree Plus are both very suitable for an on-screen interactive view. However, they are not suitable for print, which is non-interactive. OntoTrack uses a pseudotree visualization method, where each node is a class. Similarly to OntoViz, OntoTrack can express certain property restrictions in a compact way as a part of the class node. Despite the fact that it is primarily intended to be used on-screen, it is also suitable for

print. GoSurfer (Zhong et al., 2004) and GOBar (Lee et al., 2005) are both used to visualize the Gene Ontology (GO). GoSurfer uses a tree-like structure and GOBar is based on OntoViz. While both are most likely suitable for representation of such a large ontology as GO, they do not introduce any new visualization features that other previously mentioned tools do not have. Based on the analysis, the visualization method should be based on OntoViz and OntoTrack.

2.8.3 Proposed visualization technique

The proposed visualization technique is based on the graph methods used by OntoViz and IsaViz. Figure 19 provides an example of an ontology together with a legend. There are four types of vertices – a rounded rectangle representing a class from within the ontology, a non-rounded rectangle representing a class from a different (e.g. imported) ontology, a rhombus representing a datatype and a circle with the letter D in the middle (D-circle) represending disjoint classes.

Each rounded rectangle (class description) consists of three parts – class name (N-section), equivalent classes (E-section) and properties (P-section). The class name is represented by the class URI. If the default namespace is used, it means that the URI belongs to the namespace defined by the ontology. The representation of equivalent classes is based on the OWL Manchester syntax (Patel-Schneider and Horridge, 2012) that is also used in Protégé. Each line of the E-section represents one statement. The equivalence is given by the conjunction of the E-section lines. Finally, the list of properties (P-section) contains one property definition per line. Each property definition consists of five portions. The first part defines whether it is an object property (O) or a datatype property (D). The second part defines the property name as a URI. The third part defines the range of the property. The fourth part defines the minimal and maximal cardinality. The last, fifth, part specifies the characteristics of the property. A property can be functional (F), inverse function (IF), transitive (T), symmetric (S), asymmetric (aS), reflexive (R) and irreflexive (iR).

A non-rounded rectangle represents a class from a different (imported) ontology and it only contains the URI of the imported class. A D-circle represents disjointness. If a set of classes is connected through dashed lines to a D-circle, it means that the classes are disjoint.

The edges of the graph are represented by lines (links), which can be directed and undirected. There are three types of edges (lines). Firstly, a dashed directed line is used to represent disjointness as mentioned previously. The line may connect only a class and a D-circle. The second type is a full (non-dashed) directed line with a triangular arrow, which represents class inheritance. The arrow points to the parent class. In case several child classes share the same parent, it is allowed to aggreggate the inheritance links into a link with a single arrow, which improves clarity. Lastly, a dotted oriented line represents an object or datatype property that has been defined in the class definition node (rounded rectangle). The line leaves a node that represents the domain of the property and enters a node that represents the range of the property (rhombus for a datatype property, class node

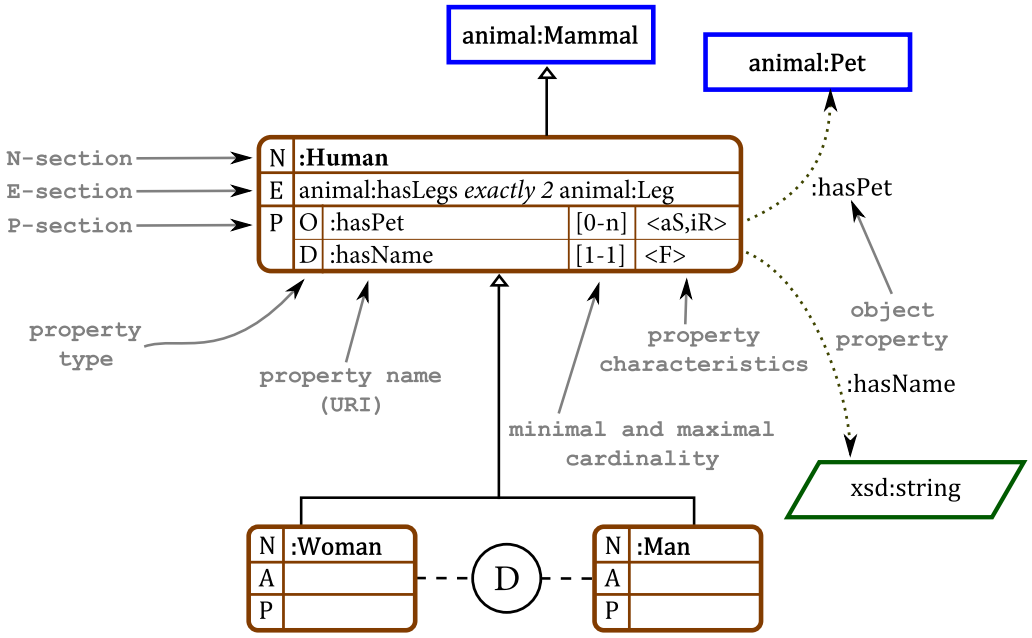


FIGURE 19 Visualization of a sample ontology together with a legend

for an object property).

Based on the visualization technique, the example from Figure 19 can be read as follows. Classes `animal:Mammal` and `animal:Pet` are imported from a different ontology. The ontology being depicted contains three classes – `:Human`, `:Man` and `:Woman`. The `:Human` class is defined as a child class of `animal:Mammal` and at the same time an equivalent class to a class with property restriction stating that it should have exactly two legs. Both `:Man` and `:Woman` are subclasses of `:Human` and they are disjoint (note the D-circle). The `:Human` class has two properties. The object property `:hasPet` has range `:Pet`, cardinality from 0 to n, and it is asymmetric and irreflexive. The datatype property `:hasName` accepts a string as its value and it is functional.

2.9 Self-managed software

2.9.1 The vision of Autonomic computing

In the last 70 years the complexity of computer systems has increased significantly. Nowadays, computer systems can be seen everywhere – from our homes to hospitals, power plants and space shuttles. They take form of small music players through mobile phones, laptops, desktops and up to supercomputers. This means that more people are using computers, but at the same time computer systems are becoming more complicated to manage and control. At some point computer systems will become so complicated, that humans will not be able to directly control all their features anymore. Instead some other way of indirect control is

needed. This fact was the motivation for the vision of autonomic computing, where computer systems can autonomously control their behavior based on operator's high level goals rather than direct control.

The vision of autonomic computing was presented by IBM in March 2001 (Ganek and Corbi, 2003). The motivation behind this vision is the fact that soon we will reach a point where computer systems' complexity will surpass the limits of human capability to control them (Kephart and Chess, 2003). This is especially important in the area of pervasive computing where potentially millions of devices can create a single distributed computer system. The complexity does not only stem from the complexity of a single computer system, but from the interactions of several interconnected computer systems. Kephart and Chess (2003) argue that "the only option remaining is autonomic computing – computing systems that can manage themselves given high-level objectives from administrators". An argument in favor of autonomic computing is the fact that very complex systems such as human body or stock exchange are largely autonomous in their nature.

If autonomic computing is the problem, then self-management is a solution. Achieving self-management in computer systems is a very complex task and therefore Kephart and Chess (2003) presented four aspects of it, namely self-configuration, self-healing, self-optimization and self-protection. Some call these features *self-* properties* (Salehie and Tahvildari, 2009; Al-Shishtawy et al., 2009) and some *self-X properties* (Mühl et al., 2007).

Some researchers, when refering to autonomic software, use the terms self-adaptation, self-adaptive software (Cheng et al., 2009; Ghosh et al., 2007) or just adaptive software (Salehie et al., 2009; Salehie and Tahvildari, 2009; Sharmin et al., 2005, 2006; Laddaga, 1999). Laddaga (1999) uses the following definition of self-adaptive software:

> Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.

Salehie and Tahvildari (2009) use a narrower definition based on the terms *context* and *self*. The *context* is everything in the system's environment that influces its properties or behavior. The *self* refers to the body of the software, which is often implemented in form of layers. The definition is as follows:

> Self-adaptive software is a closed-loop system with feedback from the self and the context.

For the purpose of this work we are going to use the definition by Salehie and Tahvildari (2009) with one exception. We believe that a better name for *self* and *context* would be *internal* and *external context* due to the fact that according to the definition of context by Dey (2001), the information about self classifies as context as well. We discuss it further in Section 2.9.3 (page 58).

The vision of autonomic computing shares some similarities with other visions as well. *Situated and Autonomic Communications* (SAC) is a vision that was presented by the European Commission as a part of the Framework Programme 6

(2002-2006). The authors believe that the main feature of future communication systems will be the ability to adapt to chaning conditions and that self-organizing networks are a solution that will make the vision possible (Sestinim, 2004). Such networks should be radically distributed, fully scalable and autonomous (Sestinim, 2006). These features should be achieved by making computer networks task- and knowledge-driven. Dobson et al. (2006) consider autonomic computing to be oriented towards application software and management of computing resources, while autonomic communication is oriented towards the management of network resources at both infrastructure and user levels. Despite different goals, these two visions are not disjoint (Quitadamo and Zambonelli, 2008).

2.9.2 Self-* properties

Self-configuration is the ability of a computer system to autonomously install, update, itegrate and compose/decompose software entities as a response to changes in its environment or itself (Salehie and Tahvildari, 2009). The human administrator does not have to specify which steps need to be taken. Instead, he/she only needs to specify the desired goal. The system is aware of its components and their capabilities and it will choose a configuration that will satisfy administrator's goals. Also, if a new component is added, the system will recognize it and make it available to the system automatically.

Many computer systems can be tuned by changing a wide variety of configuration parameters. A system with self-optimization capabilities is constantly seeking the best combination of these parameters in order to achieve its goal. Such system is proactive and it tries to find new, better ways of performing its task. A self-optimizing system can even experiment in order to find a better combination of parameters. Self-optimization is sometimes called self-tuning or self-adjusting (Sterritt et al., 2005).

Self-healing is the ability to detect, diagnose and fix problems at runtime. These problems might stem from software bugs or hardware problems. It is necessary for the system to be aware of its own configuration and dependencies between its components.

Lastly, self-protecting system is capable of defending itself from malicious attacks or mission-critical cascading failures. Such systems can be both reactive and proactive. The proactive nature can be achieved by following the sensory information and using it to predict attacks or critical problems.

Throughout the years, the term self-management has been used in many different contexts. Originally, Kephart and Chess (2003) used the term autonomic computing and self-management in connection to the computer system as a whole. However, one might want to talk about such capabilities in the context of only one aspect of computing (e.g. self-management in software). Therefore new terms were created. Salehie and Tahvildari (2009) use the term *self-adaptive software* to indicate the use of self-management in software only.

Also, new terms appeared such as self-manageable, self-managing, self-organizing, etc. Some authors use them almost synonymously and some differ-

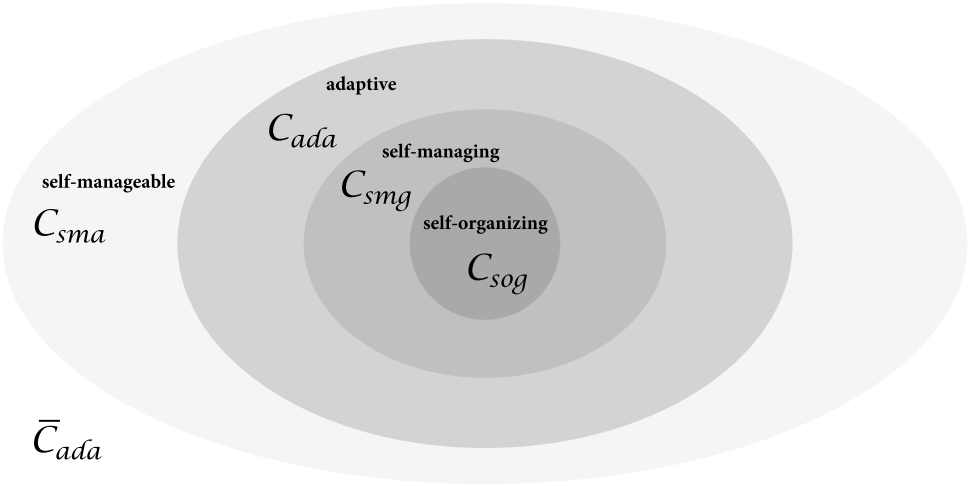


FIGURE 20 Hierarchy of self-* properties (adopted from Mühl et al. (2007))

entiate between them. Mühl et al. (2007) argue that most publications give only informal definitions of these terms. They proposed a formal system of several terms where adaptive systems are the most general ones and self-organizing systems are the most specific ones (see Figure 20).

2.9.3 Sensors and actuators

Based on the description mentioned earlier, a self-managing system must be well-informed in order to make an autonomous decision. Also, such a system must have a means to change its own behavior. Therefore a self-managing system requires sensors to receive the information and actuators (or effectors) to perform changes to its behavior and manage itself. The field of autonomic computing adopted these concepts from the field of agent-based systems.

Sensors are responsible for gathering information about system resources, component status, other systems interconnections, etc. (Hinchey and Sterritt, 2006). Sometimes the aquisition of sensory information and its propagation to the applications is called *context provisioning* (Hochstatter et al., 2008). Many authors distinguish two types of sensory information – information about the system itself and information about the external environment. However, these authors do not always agree on the nomenclature. For the purpose of this work, we call these two types *internal context* and *external context*. We understand the context as any information that is relevant to the operation of the system, including the information about the system itself. This is consistent with the definition of context given by Dey (2001). Table 6 shows synonymous terms used by other authors. Salehie and Tahvildari (2009) use the word context to refer to the information about the external environment and the word self to refer to the internal context. Hinchey and Sterritt (2006) call the ability to perceive the internal contextual information as self-awareness. The ability to perceive the information related to the external context is called environment-awareness.

TABLE 6 Synonyms for "context"

This publication	internal context	external context
Salehie and Tahvildari (2009)	self	context
Hinchey and Sterritt (2006)	self-awareness	environment-awareness

2.9.4 Classification of adaptive systems

Mckinley et al. (2004) recognize two approaches to adaptation – parameter adaptation and compositional adaptation. Parameter adaptation is a less intrusive approach, where component parameters are modified in order to change the behavior of the component and thus the system as a whole. Since no component addition or deletion is required, the cost of the change is low and there is no need to reconfigure component connections. The drawback of this approach is that only anticipated changes are possible, because parametrization must be implemented in the design and implementation phase of the application. Strategies implemented after the implementation of the application cannot be adopted. On the other hand, compositional adaptation may replace the whole structural parts of the application. Using this approach also unforseen changes may be handled. In general, these two approaches are not mutually exclusive and can coexist in one system as we will show later.

Weyns et al. (2008) distiguish between exogenous and endogenous self-management. Endogenous self-management is based on the idea that components should cooperatively adapt their structure and behavior based on the changing requirements. Exogenous self-management is based on a control loop, which monitors relevant parameters and makes decisions based on high-level objectives.

2.9.5 Control loop

Each software application contains the so-called application logic. Application logic is a set of programming elements that are trying to achieve the goal (or goals) of the application. A self-managed software application contains also adaptation logic, which is responsible for self-adaptive properties of the application. There are two main approaches how these two logics can be incorporated into one application – internal and external approach (Salehie and Tahvildari, 2009).

In the internal approach, both the application logic and the adaptation logic are interweaved (Figure 21a). The adaptation is achieved using parametrization, conditional expressions and exceptions. These constructs might be native to the programming language used to define the application logic or the language might be extended to handle them. The advantage of this approach is that all the code is in one place and very often no special language is needed to define the adaptation logic. The disadvantage is that such applications scale worse and are more difficult to maintain. The reason for this is that both features are written in the same piece

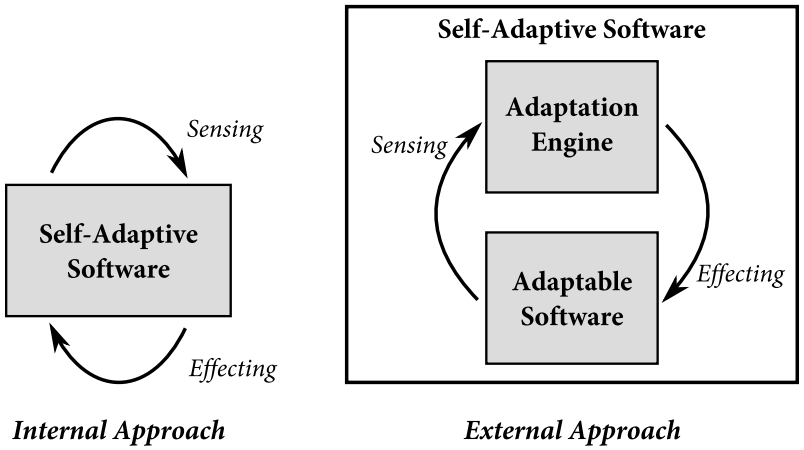


FIGURE 21 Internal vs. external approach to self-management (adopted from Salehie and Tahvildari (2009))

of code and thus are tightly coupled.

The external approach makes the adaptation a separate process (Figure 21b). The adaptation is controlled by a separate component that is called adaptation manager or adaptation engine. This component is often implemented in form of a middleware. The adaptation component controls the software. Naturally, there is a need to design and build the software with adaptation in mind. The software must be adaptable in order to be controlled/managed by the adaptation engine.

In general, an adaptive system can be based on an open loop or a closed loop control. A closed loop has a feedback loop, while an open loop does not.

2.9.6 Autonomic architectures and frameworks

An architectural approach to the problem of autonomic computing has been a favorite method for several researchers (White et al., 2004, 2006; Kramer and Magee, 2007). White et al. (2004) base their approach on a service-oriented architecture. Their system resembles a multi-agent system (MAS) in that aspect that it consists of autonomous or semi-autonomous goal-driven components. They argue that the key to an autonomic system are autonomic elements with respect to their definition by Kephart and Chess (2003) mentioned earlier. According to White et al. (2004), these elements must have the following three behaviors. Firstly, such an element must handle problems locally whenever possible. Secondly, elements must be interconnected for service provision including service description and contract negotiation. Lastly, the element must refuse any kind of activity that is in conflict with its obligations. In White et al. (2006), they provide a prototype called Unity, on which they show the effiency of thier approach.

Kramer and Magee (2007) are in favor of an architectural approach as well. They adopted a model by Gat et al. (1997), which originated from the field of robotics. Gat et al. (1997) noticed that in practical applications, a robot's actions can be categorized into three types – simple reactive control algorithms, more complicated techniques for governing routine sequences (plans) and the most

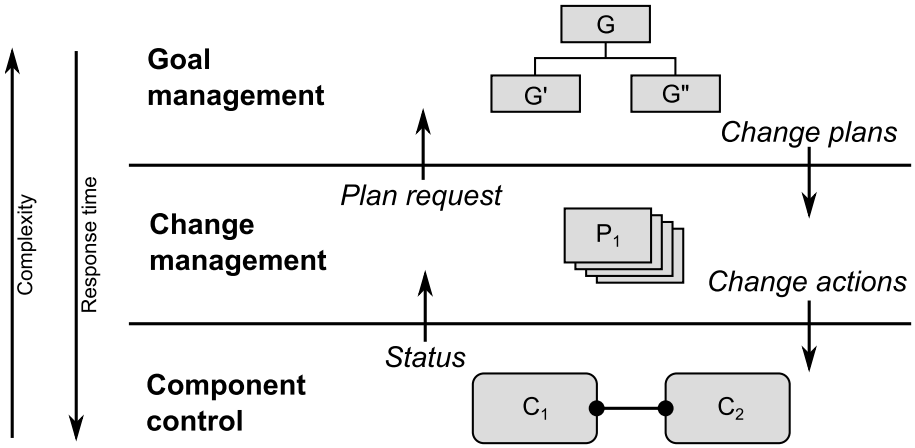


FIGURE 22 The three layered model for self-management

complicated search-based algorithms (e.g. planners). Kramer and Magee (2007) altered this model for the purposes of autonomic computing. Their model consist of three layers – Component control, Change management and Goal management (Figure 22). The Component control layer consists of self-tuned components. These components use control loops to read the data from the sensors, make decisions and manipulate the environment through the actuators. They also should support component replacement and reconfiguration. Everything that can be solved locally within the Component control layer is solved there. However, if an unsolvable error occurs, it is forwarded to the second layer. The Change management layer is responsible for component replacement and reconfiguration according to predefined plans. In other words, this layer's goal is to reactively respond to an existing problem. If a suitable plan is not found, the Goal management layer is contacted. The top layer is responsible for high-level planning with respect to predefined goals and policies.

Sharmin et al. (2006) present an architectural approach to the problem of self-management. Their project named MARKS (Middleware Adaptability for Resource Discovery, Knowledge Usability and Self-healing) aims to handle a new attribute in the area of pervasive computing, which they call "knowledge usability". Knowledge Usability is an autonomous technique to determine the user's preferences and related actions with respect to his/her past, present and future information (Ahmed et al., 2005). The communication with other devices is done through an Object Request Broker (ORB). In Sharmin et al. (2005), they utilize MARKS for resource discovery in pervasive computing environments. Ahmed et al. (2009) present their results of using MARKS+ to achieve self-healing.

2.10 Ubiware

2.10.1 Introduction

Ubiware is a smart semantic middleware platform for ubiquitous computing developed at the University of Jyväskylä by the Industrial Ontologies Group (IOG). The platform development started in 2004 as a part of the SmartResource project (Terziyan, 2008) and continued on in the Ubiware project (Terziyan, 2011). The goal was to create a new generation middleware platform enabling the creation of self-managed complex systems, especially industrial ones (Katasonov et al., 2008). This would enable the realization of the IoT and UbiComp visions. In today's computer systems, a great amount of resources can be connected to various IT systems. These devices are often very heterogeneous – human operators, software, sensor networks, RFIDs, etc. Some of these resources can be shared and some can be accessed only exclusively. Some are controlled by the system owner, some are coming from outside of the system. Integration of such resources is complicated from the design, implementation and maintenance point of view (Kephart and Chess, 2003).

Ubiware is implemented as a multi-agent system. Each agent is implemented using Semantic Agent Programming Language (S-APL). This approach was chosen due to the following benefits. Firstly, agent-based systems are suitable for design of large and complex software systems (Jennings, 2001). Secondly, MAS is suitable for spontaneous (re)configuration, adaptive reorganization and they can recover from partial failures (Mamei and Zambonelli, 2006). Moreover, distributed systems such as MAS are known to scale better with growing number of resources in play.Lastly, interoperability in UbiComp environments would be very difficult to achieve only by imposing standards and making agents comply to them. Using semantic technologies, agents are also capable of communicating their beliefs (Lassila, 2005).

Ubiware was designed as a toolbox for developers that helps them achieve smooth cooperation of various heterogeneous resources such as devices, sensors, RFIDs, web-services, software applications, humans, etc. Each of the resources is represented by an agent that acts on its behalf in the MAS. Developers use S-APL to design these agents. S-APL is a declarative rule-based language with special semantic constructs. S-APL code is interpreted by the S-APL engine, which is an integral part of the Ubiware platform. Also, a developer has the ability to call special code snippets from within the S-APL code. These snippets are called Reusable Atomic Behaviors (RABs) and they are written in Java programming language. There is a database of essential RABs and S-APL scripts delivered with the Ubiware platform. However, the developers are free to add new S-APL scripts or RABs if needed. For a deeper technical description of Ubiware read Section 2.10.3 (page 65).

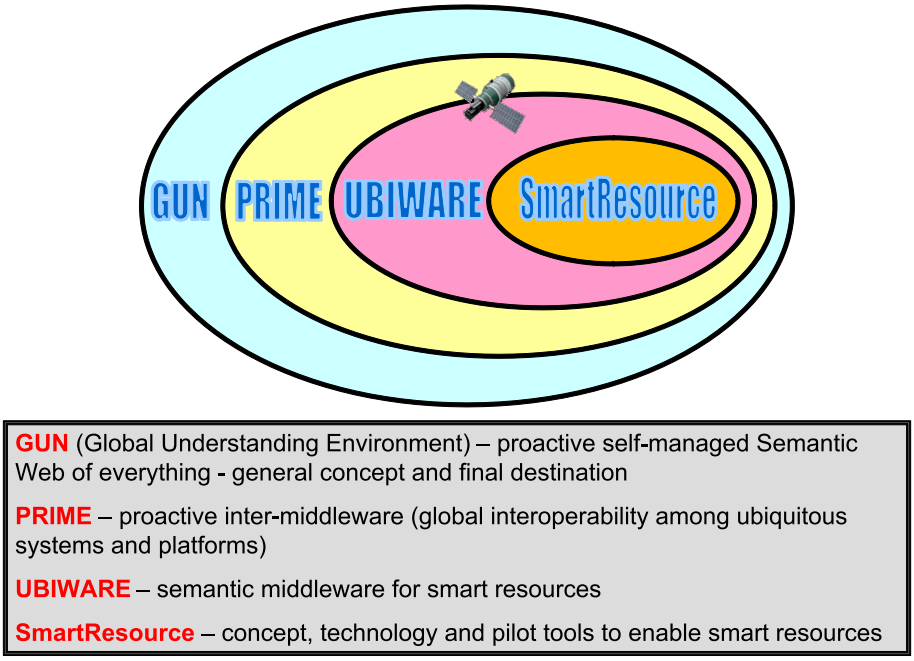


FIGURE 23 GUN research roadmap (adopted from Nagy et al. (2009))

2.10.2 Ubiware and GUN

Despite the fact that the platform development officially started in 2004, Ubiware was not an isolated project. Ubiware is a part of a bigger vision called Global Understanding Environment (GUN) (Terziyan, 2003, 2005; Kaykova et al., 2005). The relationship between GUN and other IOG projects is depicted in Figure 23. The first step towards GUN was the Smart Resource project (2004-2006), which was trying to research and develop a large-scale environment for integration of industrial smart devices, web services and human experts (Terziyan and al., 2007). In the end of the project, a set of semantic tools was developed. These tools then became the starting point of the Ubiware project (2008-2010). At the end of the project, a semantic agent platform was developed that acts as the technical base for the implementation of GERI (Global Enterprise Resource Integration) and GUN. GERI acts as a step between Ubiware and GUN, where the emphasis is given on the implementation in the industrial domain.

Originally, GUN was proposed by Terziyan (2003) as an answer to the problem of *field device management* (FDM). The idea of intelligent agent systems was not new to the area of FDM (Cederlof and Pyotsia, 1999). However, several problems still persisted. The GUN approach was based both on the agent theory and Semantic Web technologies. In GUN, each device was represented by a software agent following the "health" of the device. This way, during the lifecycle of the agent, the agent can take advantage of the collected information and utilize it in its decision making. This idea was presented as the adaptation of a physical object into a Semantic Web environment using a so-called GUN adapter. This early representation of GUN also introduced the idea of agent encapsulation and

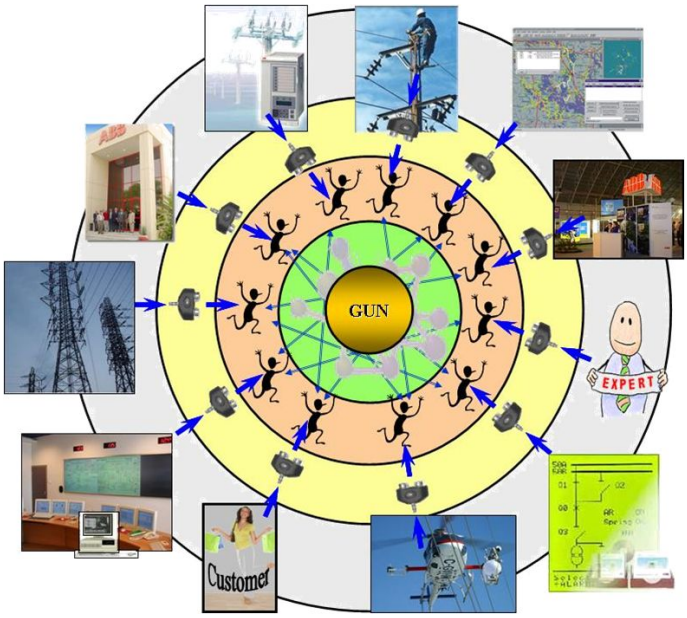


FIGURE 24 GUN overview (adopted from Terziyan and Katasonov (2009))

transparency. Agents were able to join into clusters called OntoShells, where these acted as a single entity towards the external environment (Figure 24).

In their later paper, Terziyan and Katasonov (2009) describe GUN in more detail and show its strengths in the domain of industrial automation. Traditionally, semantic technology has been used to describe resources from the functional and non-functional point of view. They however suggest the use of semantic technology as a tool for description of resources' behavior – proactivity, communication and coordination. The word global then has two meanings. Firstly, it denotes the fact that resources communicate and cooperate across the whole organizational domain and beyond. Secondly, it stands for "global understanding", where each resource can understand other resource's properties, state, behavior and business processes that it is involved in. They also believe that GUN might be the so-called "killer application" for the Semantic Web.

Despite all the benefits of agent-based systems, they still face one serious problem. Since agents are autonomous, the effects of their interactions are uncertain and difficult to predict at run time (Jennings, 2000). In order to keep these interactions under control, agents must incorporate effective mechanisms of coordination, cooperation and negotiation. One of the solutions is based on the implementation of static protocols that govern these. On the other hand, these rigid rules restrict the capabilities of MAS. Jennings (2000) believes that a long term solution must be found.

Terziyan and Katasonov (2009) recognize two main directions in the scientific literature – social level characterization of agent-based systems and ontological approaches to coordination. The first approach emphasizes the importance of social interactions among the agents and the way they influence the individual's behavior. Some of the proponents of this system believe that agent-based systems

should be designed with agent roles in mind (Vázquez-Salceda et al., 2005). The second approach based on ontological coordination emphasizes the need for a semantic description of agent's state, intentions, beliefs, etc. By describing agent's mental and physical state, other agents can coordinate their efforts accordingly. This gives the agents the ability to "understand" each other.

There are several methodologies for MAS design and implementation such as Gaia (Wooldridge et al., 2000), TROPOS (Bresciani et al., 2004) or OMNI (Vázquez-Salceda et al., 2005). They vary in their approach to design and implementation of MAS, but most of them are trying to blur the distinctions between these two. For example, OMNI uses the roles as atomic units of behavior, where roles are defined in a declarative manner. Roles are used not only in the development process, but also at runtime, where each agent can play several roles at the same time. In order to bridge the gap between the design and implementation, TROPOS methodology uses Belief-Desire-Intention (BDI) agent architecture (Rao and Georgeff, 1995).

In most cases, agent programming language (APL) is used to describe the agent and then it is compiled into an executable code or interpreted by an interpreter. APL is considered to be static code that does not leave the agent and it is not shared among other agents. GUN is going beyond this concepts and considers APL to be dynamic, modifiable and available at runtime, not only during compilation time.

In order to give APL the mentioned properties, APL must overcome the following issues. Firstly, agent roles described in APL should be easily accessible to all agents at runtime. Therefore there has to be some APL storage or database that can efficiently distribute APL code among agents. Secondly, since the same code (role) is distributed by the organization to several agents, the code must be understood by all agents in the same way. For example, many APLs are based on the first order logic, which is suitable for a single agent description. However, when one piece of code is used to describe several agents, all the language constructs (e.g. n-ary predicates) must be understood the same way by all the agents. Therefore, there has to be some common vocabulary shared by the agent.

Based on the requirements mentioned above, Terziyan and Katasonov (2009) suggest to treat agent programs as data, which can be stored in a database. They also suggest to utilize the Semantic Web technology to overcome the ambiguity of the code. Firstly, the RDF model can be used to unambiguously describe the data by using binary predicates. Secondly, URIs can be used to unambiguously identify not only the resources, but also the predicates. Lastly, ontologies can be used to explicitly describe the semantics of predicates and thus allow semantic inference of data.

2.10.3 Ubiware architecture

As mentioned previously, an agent perceives its environment through sensors and acts upon that environment through actuators. In Figure 3 (page 33) a general schema was depicted that shows an agent with sensors, actuators, the environment

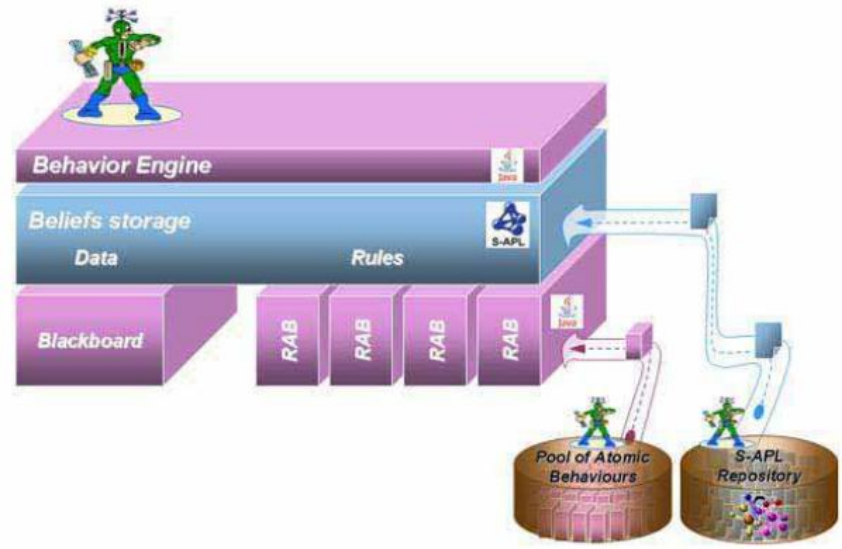


FIGURE 25 The three layered model of Ubiware

and a central decision-making component (the "brain" of the agent) marked with the symbol "?". In this subsection we will discuss the decision-making component and how it is implemented in Ubiware.

As mentioned earlier, Ubiware is a multi-agent platform based on semantic technologies. A Ubiware agent can play several roles, where each role is specified in S-APL. S-APL is a declarative language based on RDF. More specifically, S-APL is a superset of RDF written in N3 notation. Figure 25 shows a three-layered architecture of a Ubiware agent (Katasonov and Terziyan, 2008). In the next paragraphs we will describe each of these layers. Then, we will describe how these layers relate to the platform.

The top layer is a behavior engine. Each agent has its own instance of it. The engine is written in Java programming language. Originally, it was based on JADE (Java Agent Development Framework). However, the engine can be separated from JADE and a different architecture (including a custom one) can be used instead (Cochez, 2012). The behavior engine contains an S-APL interpreter. It is responsible for the evaluation of agent's beliefs and their interpretation with respect to the ontology. Some of the beliefs are interpreted as actions, where one or more actuators may be invoked. Other beliefs are interpreted as sensory input requests, where sensors are queried for information. Another category are beliefs, which represent behavioral rules. Lastly, there is a category of beliefs that may add, delete or modify other beliefs of the agent.

The middle layer is a belief storage. It is the place where agent's beliefs are stored at runtime and inspected by the behavior engine. The storage may be implemented in several ways – in-memory storage, RDF storage, etc. In the current implementation, the storage is implemented as an in-memory volatile storage. However, there is an option to store a portion of beliefs in a persistent file storage. When the agent starts its operation without any initial script, it contains

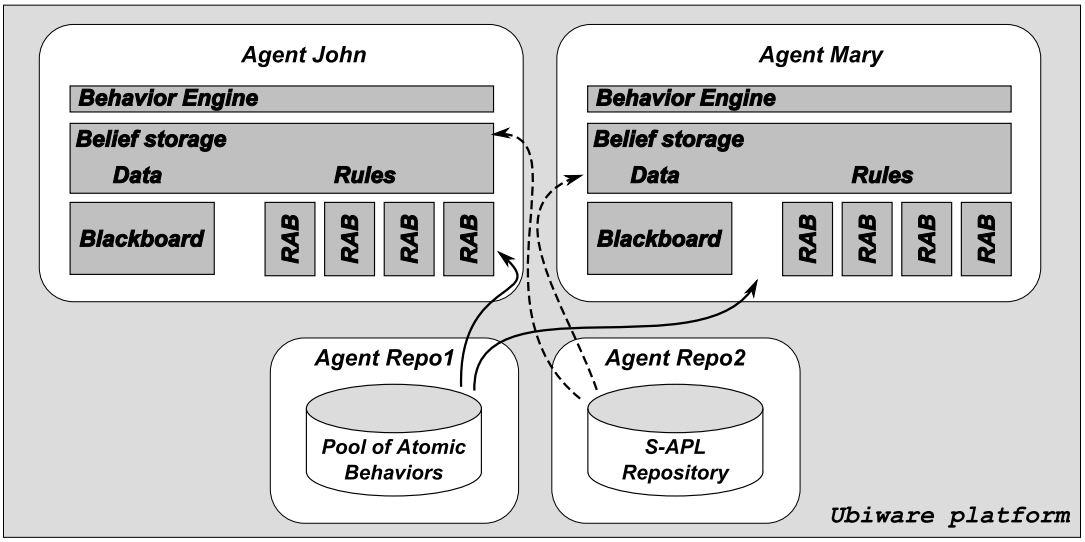


FIGURE 26 An example of a Ubiware platform deployment

very few default beliefs in its storage (e.g. its name, date, time). These are scenario-independent. The scenario-dependent code is loaded from an S-APL repository. The S-APL repository contains agent roles written by agent developers. Note that the belief storage is a runtime storage and thus it is dynamic, whereas S-APL repository is essentially a database of S-APL scripts and therefore it is static. Also, while it only makes sense to talk about the belief storage in terms of a single agent, the S-APL repository stores scripts for all agents on the platform.

The bottom layer consists of RABs (Reusable Atomic Behaviors). RABs are Java components that can act as actuators, sensors or other agent-related components (e.g. reasoners). There is a set of standard RABs that are provided with the platform. The developer can however create custom RABs. Both custom and standard RABs can be stored in a shared pool of atomic behaviors. Each agent has access to a blackboard, where RABs can exchange Java objects between each other. These objects are often related to communication (especially asynchronous). Without the blackboard, agents would have to serialize Java objects into RDF, store it into the belief storage and then read it again. This would be inefficient from the performance point of view and also cumbersome for Ubiware developers.

With respect to the three-layered architecture, the Ubiware platform offers three elements – the engine (including the blackboard), the standard set of RABs and the standard set of S-APL models. If Figure 25 shows the three layers of a single agent, then Figure 26 shows an example of a scenario hosted on the Ubiware platform. In this particular scenario there are two agents (Repo1 and Repo2) responsible for the infrastructure – namely distribution of RABs and distribution of S-APL models. Then there are two case agents (John and Mary) that internally consist of three layers as specified earlier. Technically speaking, also agents Repo1 and Repo2 contain the same three layers, but for the sake of clarity these layers are not displayed.

2.10.4 Semantic Agent Programing Language

S-APL, as it was introduced in 2008 in Katasonov et al. (2008), is a rule-based declarative language serialized in N3 notation [6]. However, in later work by Cochez (2012), S-APL is redefined in a formal way with several modifications including numerous serializations. For the purpose of this publication we consider the original version from Katasonov et al. (2008), because the platform itself uses that version of the language. We provide the following literal description of S-APL based on the "Semantic Agent Programming Language (S-APL) Developer's Guide" (Katasonov et al., 2012):

- Everything is a belief. All other mental attitudes such as desires, goals, commitments, behavioral rules are just complex beliefs.
- Every belief is either a semantic statement (subject-predicate-object triple) or a linked set of such statements.
- Every belief has the context container that restricts the scope of validity of that belief. Beliefs have any meaning only inside their respective contexts.
- Statements can be made about context, i.e. contexts may appear as subjects or/and objects of triples. Such statements give meaning to contexts. This also leads to a hierarchy of contexts (not necessarily a tree structure though).
- There is the General Context G, which is the root of the hierarchy. G is the context for the global beliefs of the agent (what it believes to be true here and now). Nevertheless, every local belief, through the hierarchical chain of its contexts, is linked to G.
- Making statements about other statements directly (without mediation of a context container) is not allowed. The only exception is when a statement appears as the object of one of the following predicates: `sapl:add`, `sapl:remove` and `sapl:erase`.

The most important part of the definition is the first sentence – everything is a belief. The behavior engine is responsible for identification of the belief types. The second most important thing is the fact that Ubiware uses the idea of context containers that form a hierarchy with the global context G being the parent. Figure 27 contains a sample S-APL code. Before the engine analyzes the meaning of the code, it parses it into the following structure. For each subject-predicate-object triple (SPO triple) the engine remembers which context container it is associated with. Then it remembers the relationships between the containers and creates a graph with the general context container as the vertex through which all other containers (vertices) are reacheable. In the case of the example from Figure 27 the inner structure will look like in Figure 28.

Note that since S-APL is based on Notation3, it follows these rules. Firstly, a subject can either be a URI, a container or a literal. S-APL allows annotations of literals, even though it is not recommended. Secondly, a predicate is a URI. Strictly speaking, the S-APL parser allows the use of literals as predicates. However, it is not recommended and it does not make sense with respect to the RDF semantics. Lastly, an object can either be a URI, a container or a literal.

[6] If you are unfamiliar with the N3 notation, refer to Section 2.7.4.3 on page 44

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@prefix java: <http://www.ubiware.jyu.fi/rab#> .
@prefix p: <http://www.ubiware.jyu.fi/rab_parameters#> .
@prefix com: <http://www.ubiware.jyu.fi/communication#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
x:Mary rdfs:type x:Female .
x:Mary x:age "25" .
x:Mary com:thinks {x:John x:age "23"} .
/*Receive messages with SAPL ontology*/
{sapl:I sapl:do java:ubiware.shared.MessageSenderBehavior}
  sapl:configuredAs {
    p:receiver sapl:is x:agentB .
    p:ontology sapl:is "SAPL" .
    p:content sapl:is {
      x:Mary x:knows x:John .
    } .
  } .
```

FIGURE 27 A fragment of S-APL code

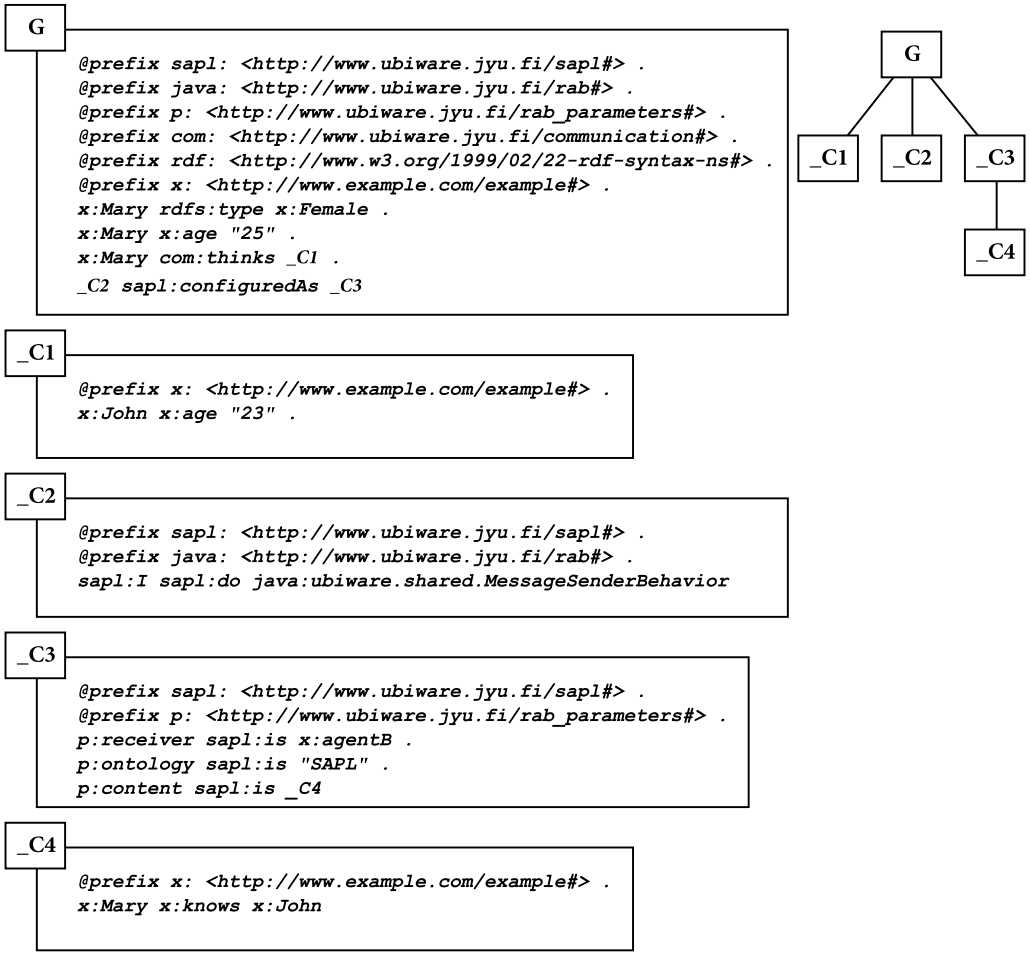


FIGURE 28 An example of an S-APL container hierarchy

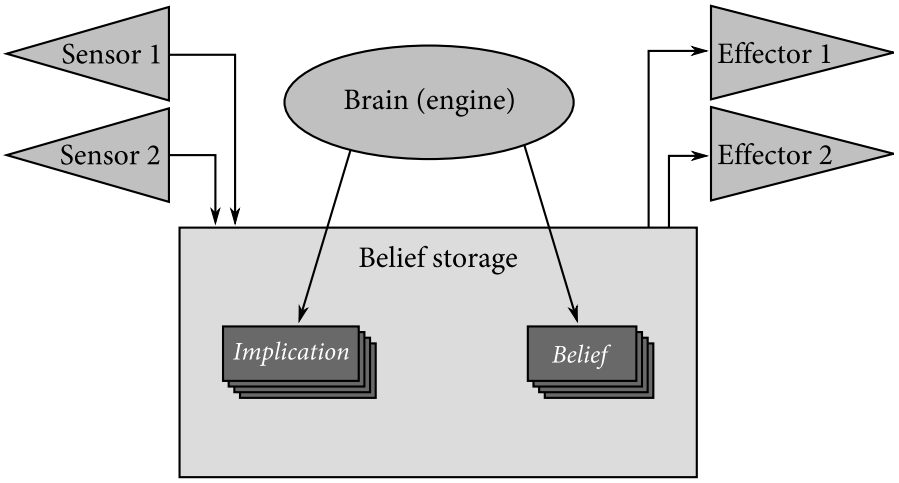


FIGURE 29 A different perspective to the three layered model of Ubiware

2.10.5 Agent's lifecycle

In order to understand how the Ubiware engine operates, we provide a different representation of the three layered model depicted in Figure 25. Figure 29 provides the same three layers. However, they are repositioned with the engine in the center. In our experience, this depiction is simpler to comprehend when trying to understand how the engine works, e.g. how the rules and commitments are being executed.

The behavior engine of the agent operates in cycles (also called lifecycles). During each lifecycle the engine performs actions related to belief manipulation, rule execution, RAB execution, etc. The execution of rules, commitments and RABs will be described later in the text. The agent lifecycle is relatively complicated and consists of several steps. However, for the purpose of this publication we introduce a simplified version of the cycle depicted in Figure 30.

First of all, the engine looks for triples with predicates `sapl:implies` or `sapl:impliesNow` in the MetaRule context. These triples are categorized as meta rules and they are executed. Secondly, the engine is looking for the same type of triples both in the Rule context and in the general context. Then they are executed. In the third step the engine performs the same actions as in the first step – it is looking for the metarules. The fourth step is to look for all triples where subject and object are containers and the predicate is `sapl:configuredAs`. These triples are identified as RAB calls. For each such belief the engine calls the specified RAB and deletes the RAB call belief, so that the RAB is not triggered again in the next cycle. After this step the engine checks if there have been any changes made to the beliefs. If yes, it proceeds to the next iteration. If no, the agent enters the sleep mode, when the agent's engine is inactive. In the sleep mode the agent only listens for the incoming inter-agent communication and other triggers. Once such a trigger is detected, the agent is woken up and the engine starts to perform again.

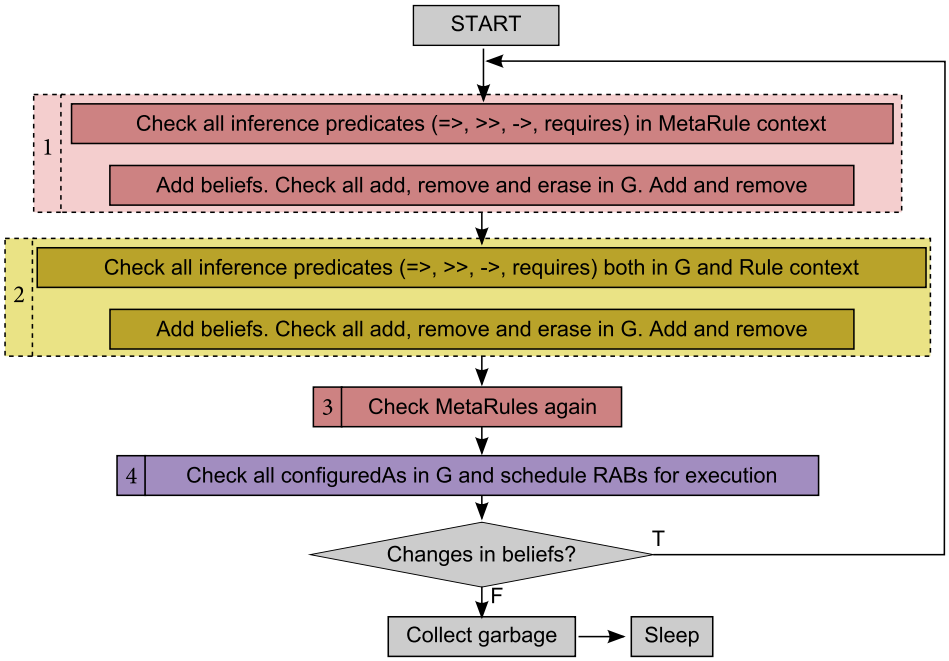


FIGURE 30 Ubiware agent's lifecycle

2.10.6 Belief types

As mentioned earlier in the text, there are several types of beliefs. Some are just pure data and others are commitments, rules, RAB calls, etc. In this section we will discuss the most important ones. The first type of belief is a RAB call. Whenever this type of belief appears in the general G context, the engine will call the specified RAB with the specified parameters. The belief blueprint looks like following:

```
{sapl:I sapl:do [RABURI]}
  sapl:configuredAs {
    [PARAM1] sapl:is [VALUE1] .
    [PARAM2] sapl:is [VALUE2] .

    [PARAMn] sapl:is [VALUEn] .
  } .
```

Parameter `[RABURI]` represents a URI from the `java:` namespace with qname equal to the full class name to a Java class implementing the behavior. Parameters `[PARAM1]` and `[VALUE1]` represent the parameter name and parameter value that will be given to the executed RAB. In general, a parameter value can be a literal, URI or a container. For a concrete RAB the expected value types are described in the RAB technical documentation. One of the most common RABs is a message-sending RAB, which was also used in the example in Figure 27 (last 8 lines).

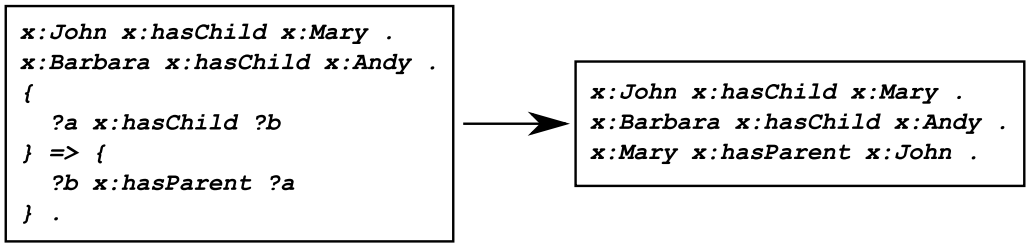


FIGURE 31 Execution of a conditional commitment and its effects on the belief structure

The second type of belief is an implication. Implications always have their subject and object as a context container. The predicate is a URI from the `sapl:` namespace . Since these implications are beliefs, by definition they may exist in various context containers. However, they are evaluated by the engine only if they are in one of these three containers – general context G, the MetaRule context and the Rule context. Based on what kind of context they are in and what kind of predicate they use, they can be categorized as conditional commitment, behavioral rule, conditional action, inference rule and meta rule. We will concentrate on the first three.

A conditional commitment is a triple in the general context with `sapl:-implies` as its predicate. The subject container represents the condition (left side) of the commitment and the object container represents the result (effect) of the commitment (right side). The condition is met if and only if one can find such a substitution in the general context that the condition becomes true. If such a substitution is found, the conditional commitment is removed and the right side of the commitment is inserted into the general context with respect to the substitution. If it is not found, the conditional commitment stays in the belief storage. In the case of the conditional commitment, only the first found substitution is used, unless specified otherwise. An example of the commitment execution and its effects can be seen in Figure 31. Note that instead of `sapl:implies` we used `=>`. Since implications are one of the most used types of beliefs, S-APL introduces these special "arrows" for the programmer to quickly identify the type of implication being used.

A conditional action is very similar to the conditional commitment. It uses the `sapl:impliesNow` or `->` predicate and it is also located in the general context. The mechanism of triggering the action is almost identical with one exception – if the engine cannot find a substitution for the left side, it deletes the conditional action from its belief structure. If it does find a substitution, it behaves the same way as the conditional commitment. Therefore one can be always sure that a conditional action "survives" only one cycle in the belief storage no matter what the result.

The third type of implication is a behavioral rule. Syntactically it is a conditional commitment in the Rule container instead of the general context. The interpretation is similar to the conditional commitment with one exception – no matter if the substitution is found or not, the rule always stays in the belief storage

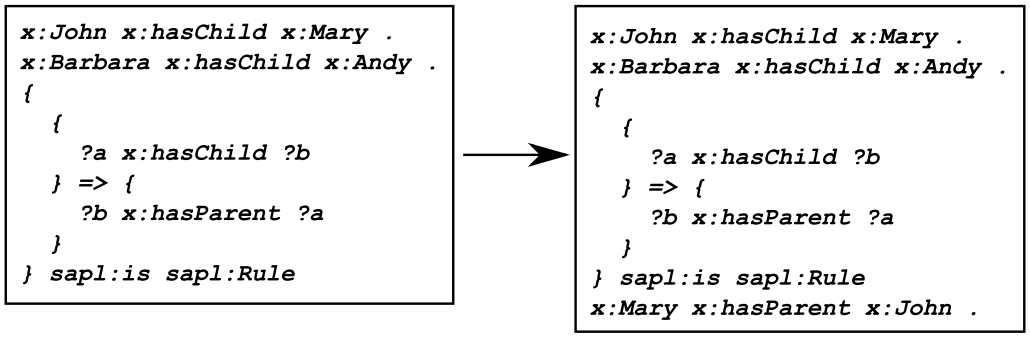


FIGURE 32 Execution of a rule and its effects on the belief structure

(more precisely in the Rule context). An example is shown in Figure 32. This particular case, the rule would be executed every cycle, because the condition of the rule is always fulfilled (we can find `?a` and `?b` such that `?a x:hasChild ?b`). The agent would be endlessly executing this rule and it would never enter the sleep state. This could lead to a livelock.

Table 7 lists five most commonly used implication types in S-APL. The table should be read in the following way. The first column contains the implication name as it is mentioned in the official Ubiware and S-APL documentation. The second and third column identify the predicates used in the implication. Both versions are equivalent and will be interpreted the same way by the engine. The "container" column specifies the container in which the implication should be positioned in order to be recognized by the engine as the type described in the first column. Note that the first three rows identify three different implication types. Even though they use the same predicate, they differ in the container which they reside in. In other words, `{...} => {...}` in the general context and `{...} => {...}` in the `sapl:Rule` context are two different implications. Therefore one should always consider both the container and the predicate type when trying to specify a certain implication type. The last two columns describe what happens to the implication itself once a match is found or not found. The letter D stands for "deleted" and letter S stands for "stays". Note that this table does not specify where in the agent's lifecycle the implication is executed. For example a rule and a metarule seem to be almost identical according to this table. However, when consulted with Figure 30, one can notice that a metarule is triggered in different steps of the agent's lifecycle than a rule. The combination of Figure 30 and Table 7 is the best "cheat sheet" for a Ubiware programmer working with implications.

TABLE 7 The most commonly used implication types in S-APL

implication type	predicate	symbol	container	match found	match not found
conditional commitment	sapl:implies	=>	G	D	S
rule	sapl:implies	=>	Rule	S	S
metarule	sapl:implies	=>	MetaRule	S	S
conditional action	sapl:impliesNow	->	G	D	D
inference rule	sapl:infers	==>	Rule	S	S

2.11 Short summary of terms

agent	Agent is a hardware or software-based system that has 4 properties – autonomy, social ability, reactivity and proactiveness
sensor	A means to collect the information about agent's environment
actuator	A means to affect the agent's environment
context container	A group of SPO statements that can be used as a subject or an object in another statement
RAB	Reusable Atomic Behavior – Java-based component that encapsulates a behavior, sensor or actuator
context	Context is any information that can be used to describe the situation of an entity
internal context	(in autonomic computing) Contextual information from the body of the software
external context	(in autonomic computing) Contextual information from the software's environment
ontology	A way to explicitly represent the meaning of terms in vocabularies and the relationships between those terms
self-management	Closed-loop system with feedback from the internal and external context.

3 SMART ADAPTIVE FRAMEWORK (SAF)

In this section we are going to introduce a middleware framework for self-managed systems called *Smart Adaptive Framework* (SAF). Firstly, a discussion about the nature of pervasive computing (PerComp) environments is provided. Secondly, a series of requirements for self-managed software in such an environment is presented. Furthermore, we introduce SAF as conceptual framework for adaptation. Lastly, we discuss the possibility of using the Ubiware platform as an implementation candidate for SAF.

3.1 Various aspects of pervasive computing environments

3.1.1 A pervasive computing environment

The main goal of a pervasive computing environment is to improve people's lives by providing computer-based support to their daily activities in a transparent way (Weiser, 1991). Therefore humans as computer users are in the center of this environment. A typical pervasive computing (PerComp) environment is a distributed computing system consisting of various devices . When looking at the environment from a hollistic point of view, one can distinguish three main elements in such an environment – humans, devices and the computer network. A device is any kind of hardware capable of communication that can be plugged into the environment. The computer network allows the devices to interact with each other. Finally, a human is capable to communicate with some of the devices directly through a user interface (UI). However, not all the devices provide a user interface. Figure 33 shows an example of a pervasive computing environment in form of a graph. The nodes represent either humans or devices. Some devices have a UI and some do not. The network is represented by edges of the graph.

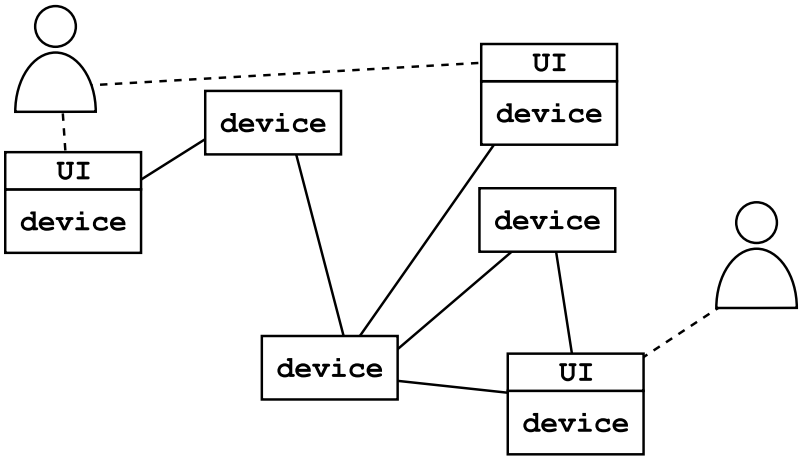


FIGURE 33 A typical pervasive computing environment

3.1.2 Properties of pervasive computing environments

Simply put, the goal of pervasive computing is to make computing technology transparent to the users while retaining its benefits. We argue that in order to achieve the true transparency of the computing technology, the environment should be as open as possible. The openness of the environment has several implications.

Firstly, an open environment is shaped not only by one, but many stakeholders. The original idea of ubiquitous computing considers an environment where most of the devices (pads, tabs and boards) are blank and not owned by a particular user. Only when a person acquires a device, the device starts to serve the operator. While this approach is perfectly reasonable in some environments, it might not be feasible in all the situations. There are cases when it is unsafe to let people take devices freely and start using them (e.g. in a shopping mall). Also, nowadays almost everybody owns a mobile computer (smartphone, tablet, laptop, etc.) and carries it with him-/herself. Carrying such a device has become a part of our everyday lives. We base our idea on the assumption that every device is owned by some entity (human, organization, etc.). One entity (owner) can own several devices, but each device is owned just by one entity.

In general, every device can have a different owner and thus different goals. Yet still these devices should be able to engage in communication with each other. An example of such an environment is a shopping mall full of people. People come to the mall with different goals – to shop, to rest, to meet friends, etc. Shop owners are trying to make profit by selling products and services. The mall is trying to keep the area clean, safe, rent the space to the shop owners, etc. It is clear that some of these goals are in conflict with each other. All these stakeholders own devices that should behave according to the owner's goals. Despite these goals being in conflict, the devices should be capable of communicating with each other and provide services to each other. It is one big *computing marketplace*. On one hand, there has to be a certain degree of flexibility. On the other hand, too much

flexibility might lead to chaos.

One might conclude that if it so, then a device should always prioritize the individual goals over the group goals. This is however not true. There are situations when fulfilling a group goal is of higher priority. Such situations include cases when the punishment for not obeying a group contract is severe, e.g. loss of reputation or loss of certain rights. Therefore there must be a way to evaluate the priority of each goal and action under a given context .

Secondly, an open environment by definition cannot put too many restrictions on the devices entering it. Therefore the devices operating in such a space will be heterogeneous. Heterogeneity makes the problem of conflicting goals even more difficult. For example, some devices will have a constant power supply, while others will be battery-powered. Naturally, in a battery-powered device the power consumption is critical, while in a constantly-powered device it is not. Heterogeneity also complicates the communication. Not all devices are capable of communicating with each other. Not all devices provide the same quality of communication in terms of throughput, latency, etc. Another manifestation of heterogeneity is in the form of various software and hardware. A system trying to tie these devices together should allow communication among as many device types as possible, ranging from a simple light sensor, through a smartphone, to a server.

Moreover, an open distributed environment is changing rapidly. Pervasive computing systems are dynamic – each device can leave or enter freely at any time. Also, many devices are portable. Usually the fluctuation is high in comparison to non-mobile distributed systems. The situation in dynamic environments changes quickly. The information that was true a moment ago might not be true anymore. Also, the devices know only what they can obtain from their sensors or from other devices. Due to the size and changing nature of the environment, they cannot know everything. Thus the devices have to assume that there is some unknown information. In other words, if one sees only five cars around himself/herself, one cannot assume that there are only five cars in the whole world. Therefore, if one asks “How many cars are there in the world?”, one must answer “I do not know”, rather than answering “There are five cars”. This is called the *open world assumption*. On the contrary, according to the closed world assumption any belief not known by an entity is automatically considered false by this entity.

To summarize, a typical pervasive computing environment is open. It consists of various stakeholders, each trying to achieve a combination of individual and group goals. The appropriate actions taken by these stakeholders depend on these goals and the context in which they find themselves. Moreover, devices operating in such environments are heterogeneous in their software and hardware nature. Lastly, pervasive computing environments are highly dynamic.

3.1.3 Conflicting goals and various stakeholders

We consider a human to be yet another resource in the environment. Since humans require an interface in order to be a part of a pervasive computing environment,

they are represented by an agent running on some kind of device (usually portable). In many cases, a good candidate for such a device could be a mobile phone, since it is exclusively owned by a single person and in most cases it is being transported with it. That way one can assume that the owner of the phone is wherever the phone is. In some cases a different device might be used, e.g. ID badge, wrist band, or even a sub-dermal chip/computer. Not all environments provide this possibility. However, this assumption may significantly simplify the system development process.

One thing to keep in mind when developing a system with human components is the fact that there can be considerable differences between people's expectations. The software should be designed for people who want only minimal human-computer interaction (HCI), as well as for people who want to be engaged in decision making and thus want a highly customizable software. Naturally, giving a user a wider variety of options can be dangerous, especially if the user does not understand the consequences of his/her actions. Since autonomous software with an external approach to adaptation by definition provides a configuration interface, user preferences can easily be incorporated into the adaptation cycle. It is our belief that the group of people demanding highly customizable software will grow in the future due to fact that the young population is well versed in handling of a computer. Therefore the ability to incorporate user preferences into the adaptation process will increase in significance over time.

3.1.4 Device heterogeneity

A pervasive computing environment consists of wide variety of devices in terms of their computing capabilities. According to Smith (2013), between 77% and 90% of U.S. citizens of age 18-29 own a smartphone. This number is more likely to increase over the next few years. An example of a mid-end smartphone at the time of writing (October 2013) is Samsung Galaxy S4 mini (also called Samsung I9190), which features a dual-core ARMv7 1.7 GHz processor and 1.5 GB of operating memory (Samsung, 2013). In comparison, the first consumer desktop dual-core processor by Intel was Intel Core 2 released in July 2006, 7 years ago (Intel, 2006). In 1993, around the time when Weiser's original vision was introduced, Galaxy S4 mini would qualify as one of the top 500 supercomuters in the world in terms of computing power (according to the historical data provided by TOP500 (2013)).

In the recent years many low-priced single-board computers appeared. An example would be Raspberry Pi offered in several specifications. As of time of writing (November 2013), the weakest specification (Model A) with end price approximately 25 EUR [1] has a 700MHz ARM11 processor with 256MB of operational memory. This is powerful enough to run a middleware framework without significantly influencing other software processes.

On the other side of the spectrum, there are devices that do not posses such computational abilities. A variety of smaller, less powerful and more energy-efficient devices is used in the area of automation. These sensors and actuators

[1] Notice that this is the end price. The production costs are significantly lower.

can be used for the environmental control, each of them providing a networking interface. Some of the companies already provide fully integrated systems with a shared communication bus. In most cases these sensors and actuators are benevolent by nature, i.e. they accept all other components' commands without any deliberation. Such components alone do not posses any computing abilities.

Heterogeneity of devices manifests itself also in the way the devices communicate. Since pervasive computing environments contain a number of mobile devices, the prevalent form of communication will be wireless. However, in some cases wired connections may still be used, e.g. for simple low-cost stationary sensors. Several wireless communication technologies can be used. Smartphones and other devices (e.g. tablets or book readers) offer cellular data service based on protocols such as GPRS (General Packet Radio Service), EDGE (Enhanced Data rates for GSM Evolution), LTE (Long Term Evolution), etc. Then there are WLANs (Wireless Local Area Networks) based on IEEE 802.11 protocol – often called WiFi™. Another type of wireless data network are WPANs (Wireless Personal Area Networks), mostly based on the IEEE 802.15 standard. WPANs are often ad-hoc and have a more limited reach than WLANs. The three most well-known WPAN networks are Bluetooth, IrDA (Infrared Data Association) and ZigBee. The hardware assumptions can be summarized as follows:

1. Computing capabilities of devices range from very powerful computers, single-board computers, microcontrollers, to sensors and actuators with no computing capabilities.
2. Some devices have a UI and some do not.
3. Devices are capable of communicating with each other over various forms of wired and wireless networks.
4. Some devices are stationary and some are mobile.
5. Some devices are battery-powered and some are constantly-powered.

3.1.5 Dynamicity of the environment

The environment consists of a mixture of stationary and mobile devices. Mobile devices mostly communicate using wireless technologies and are battery-powered. Their location changes and so changes the environment in which they reside. We believe that mostly mobile devices are the ones that influence the dynamicity [2] and the level of openness of PerComp environments. This makes the problem of communication and decision making inherently more difficult to tackle.

Generally speaking, the environment of a device is observed by the device purely through its sensors. This implies that the state of the environment as it is perceived by the device is determined by the sensory data – both current and historical. The environment is dynamic due to the fact that the sensory data changes over time. The rate at which the data is changing determines the

[2] The word dynamism is used to refer to something that has a dynamic quality. We use the word *dynamicity* to describe the extent into which something is dynamic, thus the exent of dynamism.

dynamicity as perceived by the device. Note that the rate at which the environment is changing objectively might be different than the rate at which the device senses (perceives) these changes. This is due to fact that device's sensing abilities are limited, e.g. some sensors might be missing or they are not precise enough. As an example, let us consider a tablet that has only two sensors – a light sensor and a sound sensor (microphone). If the tablet is in an environment with a constant level of ambient light and ambient sound, then the environment is perceived as static. Changes in the atmospheric pressure, temperature or other parameters are not perceived by the tablet. Such an evironment is objectively dynamic, but subjectively it is perceived as static by the tablet.

Depending on the reason why the environment is perceived as changing, we identify three main factors that influence the dynamicity as perceived by a device – dynamicity of the physical environment, dynamicity of the social environment and the rate of location change (Figure 34). As the name suggests, the dynamicity of the physical environment is determined by the rate at which the properties of the physical environment are changing over time. For example, a relatively static environment is a museum at night. If one would put the tablet mentioned previously into such an environment, the tablet would perceive the environment as nearly static, because it is quiet and dark over a long period of time. An example of a dynamic environment would be an urban environment, where the ambient sound, light, temperature and other parameters change more often over time.

The second factor is the dynamicity of the social environment. Under the term *social environment* of a device D we understand a set of devices (neighbors) that the device D can potentially communicate with using a computer network. The social environment of a device does not have to correspond to its physical environment. The existence of WANs makes it possible to be connected to virtually any device in the world. Naturally, LANs as well increase the range at which devices can communicate, but one can still assume that the end nodes are in the same geographical area. The difference between a physical location and social location of a neighbor is smallest in the case of PANs, since they operate in a very limited space. Generally speaking, the social environment is dynamic also due to the fact that the communication medium (computer network) is constantly changing. Moreover, if a device moves in space, then its network status (e.g. the amount and quality of network connections) may change as well. Therefore there is a connection between the physical and social environment.

Lastly, the rate of physical location change also influences the dynamicity as perceived by a device. Even if the physical environment is static in time, it might vary in space. Thus the change of the location of the device is perceived as an environmental change. This is determined mostly by the environmental diversity and device's movement. Under the term diversity we understand the rate of informational change in space. It might be the change in information about the physical environment or the social environment. The device's movement is determined by its trajectory, velocity, etc.

For the purpose of this publication, we will call the "Dynamicity as perceived by a device" *perceived dynamicity*. Perceived dynamicity is a subjective measure of

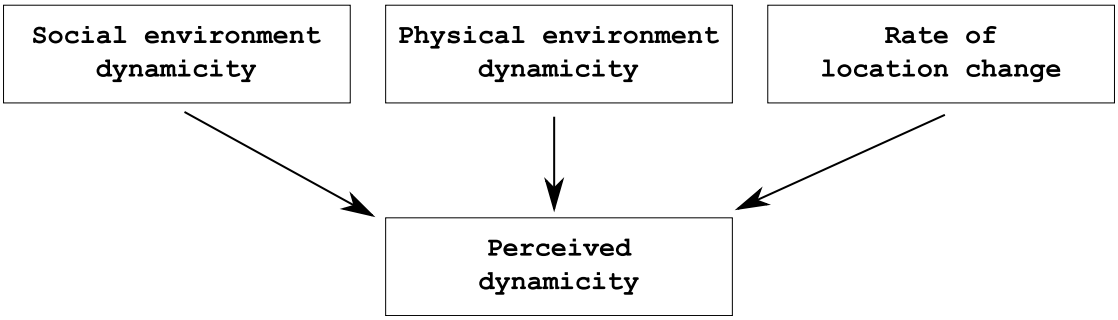


FIGURE 34 Dynamicity of the environment as it is perceived by an agent in a pervasive computing environment

time change of agent's beliefs. The only way for an agent to perceive is through its sensors. Therefore a situation might arise, when objectively the environment might be changing more dynamically or less dynamically than the agent perceives. This might happen for example if some sensor is faulty, wrongly configured or not available. It is also closely related to the problem of observability of the agent's environment.

3.2 Self-adaptive software in pervasive computing environments

3.2.1 Requirements

As mentioned in the previous chapter, self-adaptive software is a closed-loop system with a feedback from the internal and external context. The combination of self-managed software and a pervasive computing environment brings new challenges.

Firstly, the autonomic software has to take into consideration other entities in the environment. This not only increases the sensory load, but also creates new problems. In pervasive computing environments not all services are available to all devices. Therefore devices have to interact (cooperate) with each other and their actions must be coordinated. Competition among the devices has to be expected due to potentially conflicting individual goals. Therefore the adaptation software should be capable of **making decisions based on both individual and group goals**.

Secondly, the heterogeneity of the devices entering the environment makes the communication more complicated. The problem is not only in the heterogeneity of the media, but also in the heterogeneity of communication languages. The machine-to-machine (device-to-device) interaction is one of the most important features of pervasive computing. Devices interact using communication languages. As a conclusion, the self-managed software should be able to **adapt to various interaction methods** depending on the communication context (e.g. type of communication partner, type of medium, desired quality).

Third, a constantly changing environment calls for reconfiguration more often than a static environment. Every self-managed software must deal with the problem of reconfiguration cost versus its benefits. If too much computing power is spent on reconfiguration (or adaptation in general), then there is no time left for the actual application logic. An autonomic software in such an environment should try to **minimize the adaptation overhead**.

Moreover, a highly dynamic environment requires the adaptive software to be loosely coupled. Since not all components might be available at all times, components need to be replaced often. Therefore the adaptive software should support both **the parameter adaptation and the compositional adaptation**. Also, tightly coupled systems scale worse than loosely coupled systems.

Furthermore, service discovery and reuse by different applications has a very high priority in ubiquitous computing (Kindberg and Fox, 2002). These dynamic and heterogeneous environments can cause the service availability changes rapidly. Since pervasive computing environments are open, an agent must also count with the possibility of being cheated. This forces the device to consider trust and reputation as a part of service discovery process. In order to solve these issues, the autonomic software must provide **dynamic service discovery** methods.

3.2.2 Approach

In order to improve the scalability of the system, we use an **external approach to the adaptation** in a form of an adaptation middleware[3]. Using the external approach, one can separate the adaptation logic (middleware) and the application logic (middleware application). Using this approach application developers do not have to be concerned with self-management to such an extent as it is in the case of the internal approach. Instead of making the application adaptive, they make it only adaptable.

Many authors use a control loop as a method to achieve self-adaptivity. Oreizy et al. (1999) describe a loop for self-adaptive software, that consists of 4 steps (Figure 35a). In the first step, the system monitors and evaluates the software's running parameters. Secondly, the planner constructs an adaptation plan based on the evaluation. In the third step, the change descriptions are deployed. Lastly, the change descriptions are used to replace and/or reconfigure the components. Kephart and Chess (2003) propose a structure of an autonomic manager based on so-called **MAPE-K cycle**. MAPE-K is very similar to the approach by Oreizy et al. (1999). The acronym stands for Monitor, Analyze, Plan, Execute with shared Knowledge. The similarity between these two models can be seen in Figure 35, where the left side (a) describes the model by Oreizy et al. (1999) and the right side (b) describes the model by Kephart and Chess (2003). Dobson et al. (2006) introduce another autonomic control loop based on four steps – collect, analyze, decide and act. These four steps are very similar to the MAPE-K cycle. We believe that the usage of a control loop might be beneficial due to the fact that it reasonably divides the tasks performed by the self-adaptive system and clearly

[3] See Section 2.9.5 (page 59) for more details

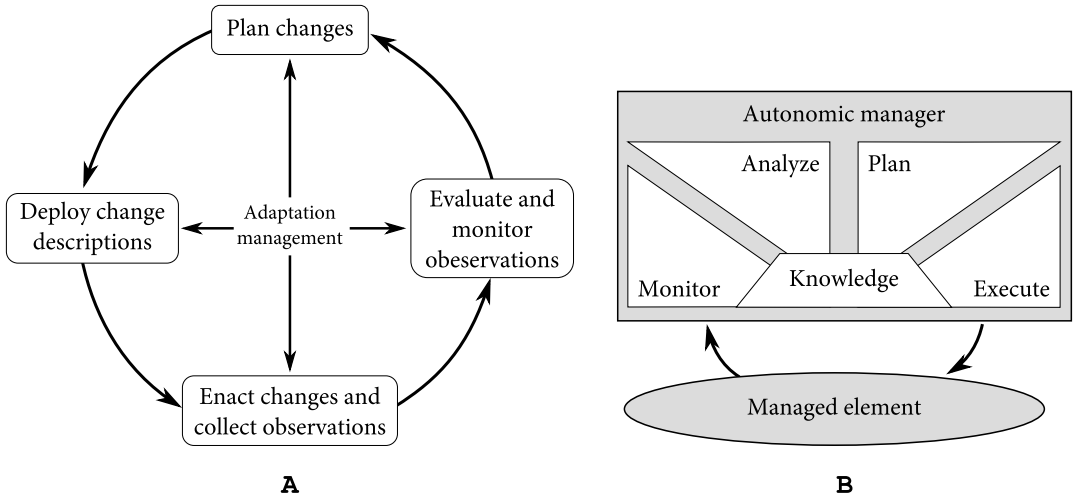


FIGURE 35 Two closed loop examples: (a) Closed loop by Oreizy et al. (1999), (b) MAPE-K closed loop by Kephart and Chess (2003)

defines the information flow. However, we argue that the planning stage is not always necessary due to its high computation costs.

Our approach is also based on the **three-layered architecture for adaptation** by Kramer and Magee (2007)[4]. The three-layered model however does not specify the internal implementation of each layer. By combining this approach with the MAPE-K cycle, it is possible to avoid costly planning by allowing shorter sub-cycles within the main cycle. By utilizing three layers with increasing processing capabilities, the information is travelling to the upper layers only when it cannot be handled by the current level. We believe that this model is suitable for the area of pervasive computing, because it decreases the computation time, memory usage, energy consumption, etc.

The nature of device interactions within a pervasive computing environment strongly resembles a **multi-agent system** (MAS). These two systems share several similarities. Firstly, MAS are by nature distributed, same as pervasive computing environments. Secondly, in both cases the usefulness of the system increases with the increasing number of interactions among the elements (agents or devices). Moreover, both systems engage in similar types of social activities, such as cooperation, competition and negotiations. The area of MAS is a field with a variety of problem solving techniques that can also be applied in pervasive computing environments.

In our experience, the heterogeneity problem can be solved by utilizing **semantic technologies**. Using semantic technologies can help in discovery and utilization of heterogeneous resources (Katasonov and Terziyan, 2008). Also, they can be used for behavioral control among various resources (Lassila, 2005). Table 8 summarizes the requirements for self-managed software in pervasive computing environments. It also provides our approach to these problems.

[4] See Section 2.9.6 (page 60) for more details

TABLE 8 Requirements for self-managed software in pervasive computing environments and our approach

Requirement		Our approach
R_1	Adaptation decisions are based on both individual and group goals.	MAS
R_2	Ability to adapt to heterogeneous interaction methods.	Semantic web technologies
R_3	Minimize the adaptation overhead.	Architectural approach (three-layered model)
R_4	Support both the parameter adaptation and the compositional adaptation.	External approach
R_5	Support dynamic service discovery.	MAS and Semantic Web technologies

3.3 Smart Adaptive Framework

In this section we are going to introduce an adaptive middleware for pervasive computing environments called Smart adaptive framework (SAF). SAF is based on the four previously mentioned approaches. Since SAF consists of several components with various functions, we divided this section into the following subsections. Firstly, we introduce the conceptual architecture of the framework. Each framework processing element is described. Secondly, we describe various types of sensors and actuators that the platform uses for actuation and context provisioning. Moreover, we elaborate on possible reconfiguration triggers, which we call incidents. Lastly, various software and platform profiles are discussed.

3.3.1 Conceptual architecture

Smart adaptive framework (SAF) is a middleware based on a hybrid architecture using the three-layered architectural approach (Kramer and Magee, 2007) and a MAPE-K cycle (Oreizy et al., 1999; Kephart and Chess, 2003). The platform architecture can be seen in Figure 36. The platform consists of a shared knowledge base (KB), service facilitator, various processing elements and resources.

The letter K in MAPE-K denotes a knowledge base, which can be seen on the left hand side of the figure. The shared knowledge base (KB) contains all contextual information, plans, goals, capability descriptions, etc. At any moment any processing element may access the base and fetch information that it needs for its operation. The rest of the loop (MAPE) can be seen in the middle as a modified circle consisting of the processing elements. The name *element* was chosen intentionally, because the word *component* refers to a component of the managed software on top of the platform. The monitoring phase is embodied in

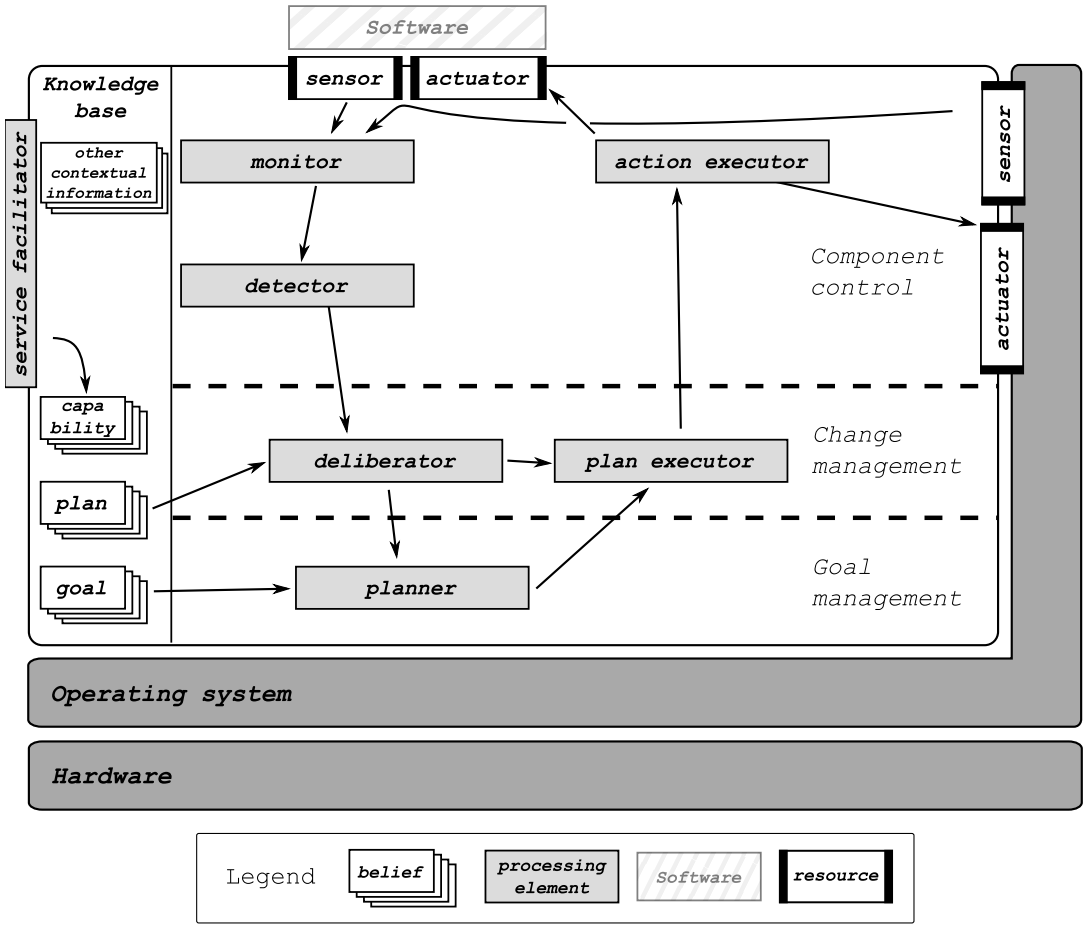


FIGURE 36 SAF architecture

the monitor processing element. The analyse phase is implemented by the detector. Instead of the MAPE planning phase, SAF contains two elements – deliberator and planner. This is due to our previously mentioned argument that planning is not always necessary. Finally, the execute phase is implmented in two processing elements – plan executor and action executor.

When looking at the figure, the three layers are in the opposite order than they were depicted in the original model. The logically lowest layer (Component control) is in the top and it consists of sensors, actuators, monitor, detector and action executor. The goal of this layer is only to monitor and affect the software and the environment. The logically second layer (Change management) is in the middle and it consists of deliberator and plan executor. This layer is concerned with best plan selection and its execution. The logically topmost layer (Goal management) is in the bottom and it is represented by the planner element.

The depiction of the framework contains two main types of sensors and actuators. On the top, sensors and actuators are responsible for the internal context and software actuation. The sensors and actuators on the right are related to the external context and actuation.

The monitor reads the information from various sensors, which will be discussed later in a separate section. The sensory information is forwarded to the detector, which classifies the sensory input. Based on the user's and software's profile, the detector decides what should happen. It either ignores the change (if it is irrelevant) or contacts the deliberator in the second layer. A relevant change is any sensory information that either breaks a policy or it can contribute to a solution of an already broken policy.

The deliberator is responsible for checking of the existing plans and finding the best solution among them. If no plan is suitable, the third layer is contacted, where the planner takes over. The planner checks the goals of the system together with user preferences and creates a plan. This plan is then sent to the plan executor, which determines the sequence of actions, which in turn are executed by the action executor. Table 9 contains a summary of processing elements, their relationship to the MAPE portion of the MAPE-K cycle and the three-layered architecture.

Each device in a pervasive computing environment may run several applications, each having different capabilities and providing various services. These services and capabilities need to be discovered and properly accessed. In the SAF framework, it is the responsibility of the service facilitator (SF). The facilitator collects the information about all available services and provides it to other components. SF is also responsible for the trust and reputation management. Some services may be provided by more than one application. However, the quality of the provided service may vary. In order to determine the quality of the provided service there has to be some metric. We suggest the use of utility functions as metric providers due to their superiority over simple action and goal policies (White et al., 2004; Kephart and Walsh, 2004; Kephart and Das, 2007; Walsh et al., 2004).

Each device in the environment has an instance of SAF on top of which it runs middleware applications. The devices communicate with each other through

TABLE 9 Summary of SAF processing elements' roles

Layer	Processing component	Original MAPE	Role
Component control	Monitor	M	Reading context-relevant sensory data
	Detector	A	Policy consistency check
Change management	Deliberator	P	Incident discovery and classification
			Best incident recovery plan selection among existing plans
	Action executor	E	Action execution
	Plan executor	E	Incident recovery plan execution
Goal management	Planner	P	Planning for incident recovery

their SAF instances. The human-computer interaction (HCI) between a device and its user is facilitated through an application running on top of SAF. It is not our intention to tackle the problem of HCI in pervasive computing environments. However, the framework provides enough flexibility for a UI application to be developed on top of it. Through such an application the user may change the global (platform-wide) preferences, manage the device, configure sensors, etc. We do not impose any restrictions on the type of UI and HCI methods, e.g. graphical user interface, haptic interface, command-line interface, voice user interface, etc. A sample pervasive computing environment based on SAF is shown in Figure 37.

3.3.2 Sensors and actuators

There are various criteria according to which we may categorize sensors and actuators. Based on where the data is collected from, we distinguish between software (SW), hardware (HW), operating system (OS) and environmental sensors (Figure 38). We consider these four to be the most important aspects of the system in terms of context provisioning.

The software sensors sense data from the adaptable software. Naturally, the software developer must make this data available. Since every software running on top of SAF middleware can have a different application logic, there should be a way to provide own classification of software sensors.

The hardware sensors sense data about physical components of the device that the software is running on. This could include CPU usage, memory usage, battery level, Bluetooth status, screen brightness, disk temperature, etc. The OS sensors sense the information about the operating system and its running parameters. This can include number of active processes, virtual memory size, number of active users, etc.

Finally, the environmental sensors pick up information about the device's

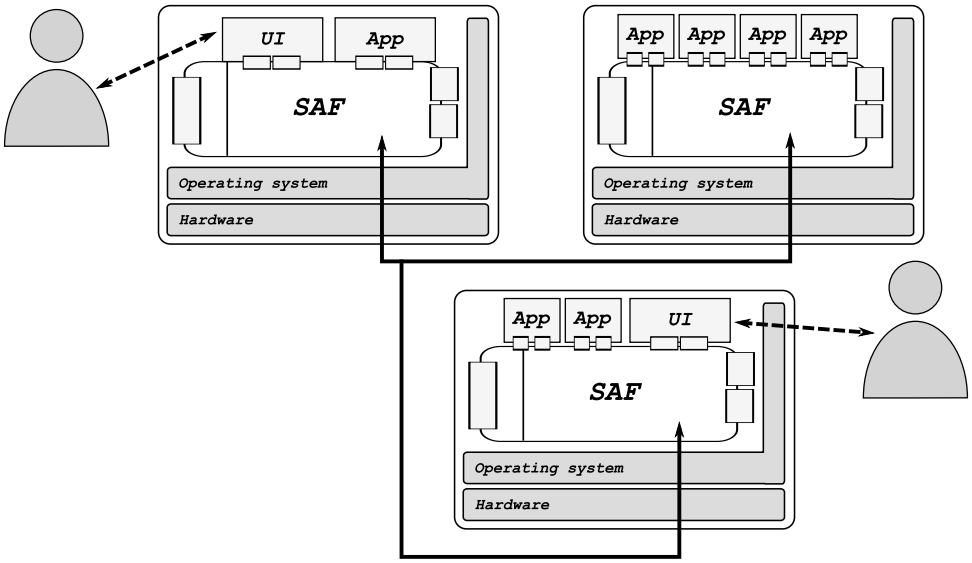


FIGURE 37 SAF in pervasive computing environments

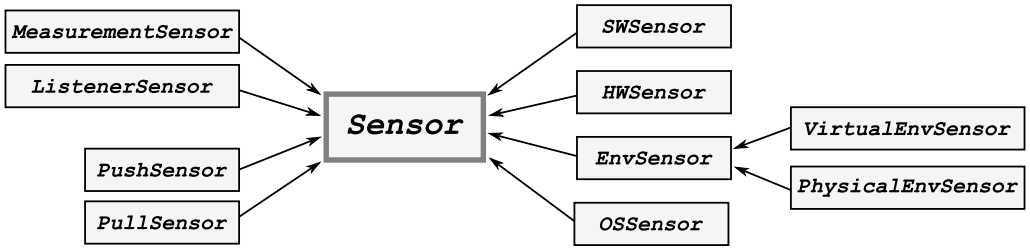


FIGURE 38 Sensor classification

environment. The environmental sensors are divided into to two categories – virtual and physical environmental sensors. Physical environmental sensors (PES) measure data from the device's surroundings such as ambient temperature, humidity, loudness, location, speed, yaw angle, pitch angle, etc. Virtual environmental sensors (VES) measure data related to the devices presence on the Internet and/or other computer networks. An example of such a sensor is a social media sensor, news feed sensor, etc.

The second sensor classification is based on the method used to provide the data. In general there are pull and push sensors. A pull sensor is a passive sensor that provides the data only on request. In the case of the framework, it means that the agent has to actively ask the sensor for the information. A push sensor is an active sensor. Typically, a push sensor keeps track of parties that are interested in the data and whenever there is a change in the reading, the parties are provided with the new data. Often it is possible to provide a wrapper that would expose a pull sensor as a push sensor. A similar method is used to notify the detector by the monitor, which will be discussed later.

The third method to classify the sensors is based on their mode of operation. There are measurement sensor and listener sensors. Measurement sensors are

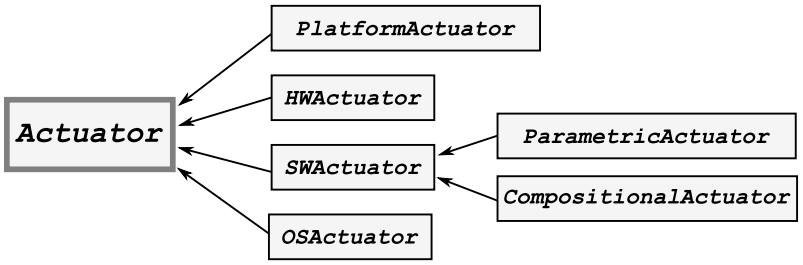


FIGURE 39 Actuator classification

measuring the value of a certain variable, such as temperature, CPU usage, virtual memory size, etc. The result of such a sensor is a value. Usually they are implemented as pull sensors. Listener sensors are listening for certain events, rather than just providing a value. The result of such a sensor is not only a value, but a whole event description. We base this classification on our experience in the SCOPE project . In the Work Package 2 we used both types of sensors for gathering and classification of information about life threatening events (e.g. natural disasters), cultural events, marketing messages, etc. All these were then integrated into a single mobile application and provided to the user based on his/her preferences and location.

There are four main types of actuators (Figure 39). Firstly, platform actuator can change the configuration of the SAF platform. Secondly, hardware actuator is responsible for manipulation of hardware components such as network interface card, camera, speakers, etc.This type of actuator would be used for actions such as taking a picture, playing a sound or calibrating a sensor. Moreover, operating system actuator is used to influence the operating system, e.g. chaning the memory space for a certain application. Finally, software actuator changes the adaptive software running on SAF. Parametric actuator is used for parametric adaptation and compositional actuator is used for compositional adaptation (e.g. component replacement).

3.3.3 Incident classification

The self-adaptive software is capable of autonomously changing its behavior if it does not operate within specified parameters. In order to do this, such a software must be able to sense all relevant parameters that might influence its operation. Not all of the changes are relevant to the operation of the software. There must be some way to distinguish between relevant and irrelevant ones. We call the relevant changes *incidents*. Not all incidents are equally important. Some of them might be more severe than others. Therefore some kind of classification should be performed. We suggest the classification depicted in Figure 40. Note that even though the word incident refers to something unpleasant or unwanted, developers can use incidents as indicators of a significant change. We decided to use a word different from change, because not all changes in the sensory information are necessarily context-relevant. Therefore we may say that an incident is a context-

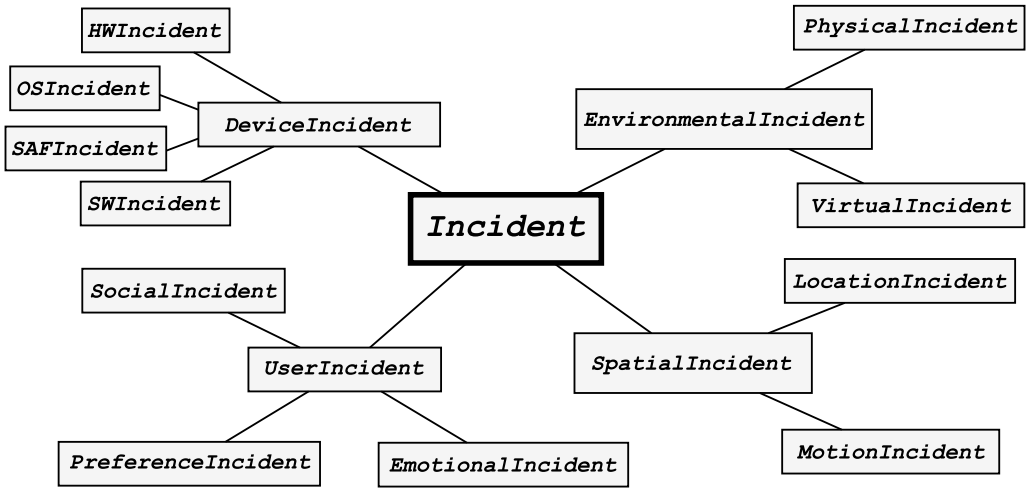


FIGURE 40 Incident classification

relevant change.

As mentioned previously, we assume that a significant portion of devices in the environment will be mobile. It is reasonable to assume that the change in the position and motion-related data (e.g. speed, acceleration, pitch, yaw) might be relevant to mobile devices. We call these spatial incidents.

Another type of incidents are user incidents. An adaptive software must by definition react to user preference changes. A preference incident is used to refer to such a change. Humans are one of the three elements of a pervasive computing environment. A social incident refers to any change in terms of other people's proximity to the device. For example if a mall customer comes close to an information screen, it plays a commercial. Lastly, an emotional incident is related to the field of emotionally intelligent HCI (Pantic and Rothkrantz, 2003). If a device has sensors for measuring user's affective states, an emotional incident can indicate a context-relevant change in user's emotions.

Device incidents are related to changes in the device itself. As the name suggests, HWIncident is a hardware incident (e.g. high disk temperature), OSIncident is related to changes in the operating system (e.g. drop in free virtual memory space), SAFIncident is a framework-related incident (e.g. new software deployed) and finally, SWIncident refers to changes in the internal context of the application.

The last type of an incident is an environmental incident. The physical environment incident determines a change in the physical environment of the device, e.g. ambient temperature, humidity, amount of light, etc. The virtual environment incident indicates a context-relevant change in the area of available services, capabilities and other devices. This type of change is relevant for example in cases when a device cooperates with other devices/services that are not in its physical proximity (e.g. a web service). Finally, same as in the case of software sensors, there should be a simple way to provide own classification of software incidents.

3.3.4 Configurations and profiles

3.3.4.1 Software profile

In general, every software requires a different way of adaptation. However, many types of actions associated with adaptation share certain similarities. By providing generic configurable adaptation techniques in form of a middleware, the software developer does not have to reimplement these techniques inside every software, as it is in the case of the internal approach to self-management. Instead of that, the developer provides a configuration of his/her software, which can be read by the platform.

The adaptability of the software is achieved through two elements. Firstly, the developer defines the *adaptation profile* of the application. The adaptation profile is a set of policies that the adaptation engine should follow for this particular software. There should be two types of policies – hard and soft. If a hard policy is broken, the system is considered non-functional. Hard policies are often associated with mission-critical portions of the application logic. A soft policy represents the optimal state. However, if a soft policy is broken, the system still operates, it is only less effective. There should also be a hierarchy of soft policies, which reflects their importance. This is useful in cases where the system has to choose between braking one or the other policy.

Secondly, the *structural profile* describes the structure of the application in terms of components, functions, modules, etc. Using the structural profile, the adaptation middleware can make decisions related to component reconfiguration and replacement. Many modern programming languages are delivered with introspection tools and libraries that can be used to partially or fully automate the process of structural profile creation. For example, in the case of Java applications one can use custom annotations for this purpose.

3.3.4.2 Platform profile

The use of configurable middleware also allows a greater control over the adaptation process. This might be beneficial for managing a larger numbers of devices falling under one organization. The administrator can restrict certain behavior by configuring each device's platform. Also, the platform might be configured to facilitate adaptation for specific types of software.

In order for the platform to operate properly, the available sensors must be configured. This is especially true for the external context sensors, since they can serve all the applications running on top of the middleware. Moreover, certain capabilities might be used platform-wide. Lastly, the platform configuration profile might contain some generic plans, e.g. component replacement or sensor calibration.

3.4 Self-* properties

Kephart and Chess (2003) present four properties of self-managed software. Self-configuration is the ability to autonomously configure systems and components following high-level policies. Mckinley et al. (2004) recognize two approaches to adaptation – parameter adaptation and compositional adaptation. SAF provides both types through software actuators that can modify software parameters. A parameter change or component change can be expressed as an action with certain input parameters. Such an action is then executed by the action executor element. Actions can be composed into plans by the planner element. The policies are provided in the software adaptation profile and are incorporated into the planning process.

Self-healing autonomously finds, diagnoses and repairs localized software and hardware faults. In SAF, the monitor reads sensory data and notifies the detector. The detector is capable of detecting a fault. If a fault is detected, an incident is created that is then forwarded to the deliberator. Depending on the availability of a plan, either deliberator or planner finds a solution to the problem, which is then enforced through the plan executor and action executor as a set of repair actions for the actuator.

Self-protection is the ability to autonomously defend against malicious attacks or cascading failures. In the case of pervasive computing environments, the greatest danger of malicious attacks comes from the other devices in the environment. It is important to recognize potential attackers among the communication partners. This is achieved through the service facilitator (SF). SF is responsible for trust and reputation evaluation. Also, policies (especially hard ones) in the software adaptation profile define the operating parameters of the software. As mentioned earlier, the planner considers policies in the decision-making process.

Self-optimization drives the components to continuously seek possibilities to improve their operating parameters. In SAF, this ability is achieved through utility functions.

3.5 Ubiware as a candidate for SAF implementation

Based on the requirements and the approach to self-management described previously, the SAF framework should be an agent-based middleware based on the hybrid model of adaptation. It should be capable of effective resource discovery and communication in heterogeneous environment. We base our approach on two main hypotheses. Firstly, we assume that semantic technologies improve the interoperability among various heterogeneous devices in pervasive computing environments. Secondly, we believe that agent technologies are a suitable tool model and implement interactions among various actors in such environments.

These two hypotheses are also the base of the Global Understanding Environ-

ment (GUN) vision (Terziyan, 2003, 2005; Kaykova et al., 2005). The Ubiware agent platform was created as an effort toward the GUN vision (Terziyan, 2008, 2011). Another important contribution towards the vision is the creation of the Semantic Agent Programming Language (S-APL). S-APL is a declarative language based on Semantic Web technologies. The language is used as an agent communication language (ACL), agent programming language (APL) and for the representation of agent's beliefs. S-APL can be used to tackle the problem of heterogeneity of the environment. Naturally, this is not a coincidence, since Ubiware was developed with pervasive computing environments in mind.

The three layered architecture from Figure 36 can be implemented as a Ubiware agent. This follows the Ubiware and GUN philosophy, where a Ubiware agent acts as an adapter to the resource. The knowledge base contains beliefs in form of SPO triples. Since in the case of a Ubiware agent everything is a belief, also plans and goals are just special beliefs. The knowledge base also contains other types of beliefs that will be mentioned later. The framework components can be implemented using the S-APL language. Following the Ubiware philosophy, the sensors and actuators can be implemented as RABs. Finally, one of the benefits of using S-APL is the fact that it is based on the RDF family of standards. This allows us to utilize already existing reasoning and modeling tools for added benefit. In order to implement SAF using Ubiware and S-APL, several issues must be solved. The next chapter introduces three new features that should extend the range of Ubiware's capabilities and make it more suitable for implementation of SAF framework. For each of these improvements, we provide the motivation that drove us to implement it.

4 UBIWARE PLATFORM IMPROVEMENTS

In this chapter we provide three Ubiware platform improvements necessary for a Ubiware-based implementation of SAF. For each improvement, we provide the motivation with regards to SAF together with the implementation in terms of S-APL constructs. In case new constructs or RABs are introduced, they are described as well. Also, for each implementation an OWL ontology is provided – both as a graph visualization and as an OWL file in Turtle notation.

Note that while the provided ontologies are formally defined and consistent with respect to the OWL specification, they are used in an atypical way. Since S-APL uses beliefs as a way to define both data and application logic elements (e.g. implications or RAB calls), the ontologies act as belief blueprints. This is similar to the role of a class in object-oriented programming.

4.1 Utility functions

4.1.1 Motivation

Russell and Norvig (2003) describe four main types of agents – simple reflex agents, model-based reflex agent, goal-based agent and utility-based agent. A simple reflex is the simplest type of agent that makes decisions only based on the current sensory input. A model-based reflex agent deliberates based on a model of the environment. This model is gradually built by incorporating historical observations into the decision-making process. The third type is a goal-based agent that makes such decisions that should progressively bring it into the desired state (goal). The choice of actions leading to this state is determined by a plan. Finally, the most sophisticated agent is a utility-based agent, which, apart from being able to find a plan, is also capable of evaluating the usefulness of each plan or action by using utility functions.

According to Russell and Norvig (2003), utility function maps agent's state onto a real number, which expresses the degree of agent's happiness. Utility

functions are a suitable problem-solving tool for two types of problems. Firstly, in the case of several conflicting goals, the utility function can determine an appropriate tradeoff. Secondly, if there are several unreachable goals, the utility function can help choose a goal that is the most likely to be achieved.

Kephart and Walsh (2004) argue that in complex environments such as autonomic systems, a combination of reflex, goal-based and utility-based agent behavior may improve the overall performance of the system. The current set of S-APL constructs does not allow a simple representation of utility functions. Therefore we introduce a set of new constructs to overcome this problem.

4.1.2 Implementation

In Nagy (2012) we discussed the issue of business-to-customer (B2C) multi-channel communication. We proposed two approach to this problem. Firstly, for modeling of the domain, we suggest an ontology-based approach. Secondly, for decision making (e.g. best channel selection), we suggest an approach based on utility function. We developed this idea further in Nagy (2013) and presented a framework for multi-channel communication. The process of best channel selection based on utility functions on top of semantic data was described in more detail. This approach had one limitation – the utility functions were not described in form of RDF data. In this section we present an improved version of the original utility functions for semantic data. In our approach, utility functions are described using RDF statements and therefore this approach follows the Ubiware philosophy "everything is a belief".

By describing utility functions in form of RDF statements, the following benefits are gained. Firstly, it is possible to make statements about utility functions and their elements. This allows us to build taxonomies. Secondly, it is possible to reason about the utility functions and thus check their consistency and/or infer new facts about them. Moreover, it is possible to query utility functions and their properties. For example, one can ask "Show me all the utility functions where person's age matters".

Figure 41 shows an example of a utility function `u:uf1`, where cost c and speed s of a certain route is taken into account. Mathematically, the function can be expressed as follows:

$$F_u = \frac{10 \cdot s}{c}$$

Utility function `u:uf1` has a query associated with it. The query acts as a filter over the data. In the example it is asking for all the routes with their speeds and costs. The query is written in S-APL, however the transformation to SPARQL is trivial. The second parameter is describes which element is being evaluated (`u:usesEvalElement`). In the example, a route `r` is being evaluated. The result of a query is a result set with one binding for each found route. This result set is then later used in the mathematical function `u:f0`. Each function is described by its type (e.g. multiplication, addition, etc.) and its operands. In the example, one can see two binary functions, where `u:f1` acts as a part

```
u:uf1 rdf:type u:UtilityFunction
   ; u:hasQuery {
      ?r rdf:type x:Route
       ; x:cost ?c
       ; x:speed ?s .
   }
   ; u:usesEvalElement "r"
   ; u:hasFunction u:f0 .

u:f0 rdf:type u:Function, u:BinaryFunction, u:FunctionDiv
   ; u:op1 u:f1
   ; u:op2 [ rdf:type u:Variable ; u:hasVarName "c" ] .

u:f1 rdf:type u:Function, u:BinaryFunction, u:FunctionMul
   ; u:op1 [ rdf:type u:Variable ; u:hasVarName "s" ]
   ; u:op2 [ rdf:type u:Value ; u:hasValue "10"^^xsd:integer ] .
```

FIGURE 41 An example of a utility function description

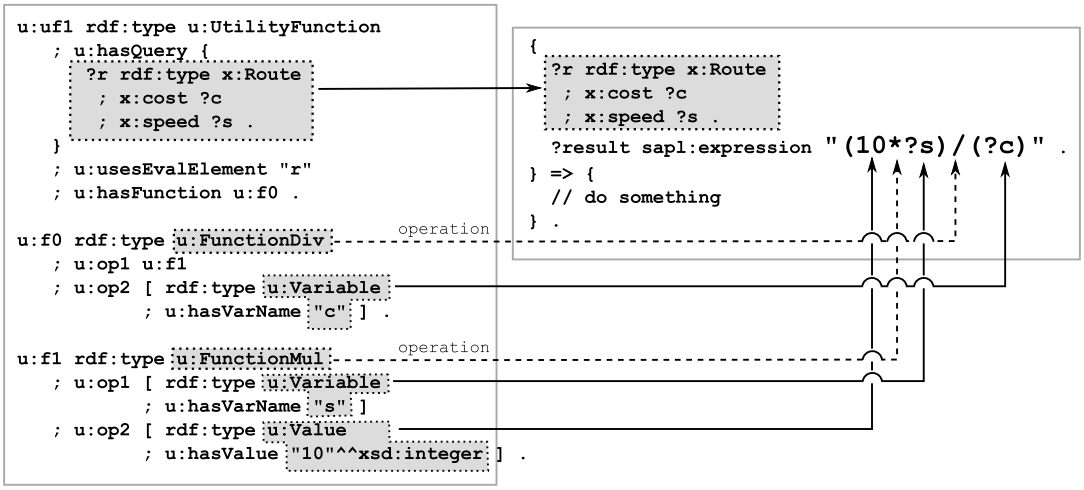


FIGURE 42 Conversion of a utility function description into a conditional commitment

of `u:f0`. Such functions can easily be transformed into a `sapl:expression` command as shown in Figure 42. We do not provide a formal algorithm, since this transformation is trivial.

We have shown how a utility function can be described and how it relates to S-APL language constructs. The next step is to evaluate the function on top of some RDF data. Figure 43 shows an evaluation annotation (lines 1-3). The evaluation resource `u:eval1` is has the mandatory property `u:useFunction` that points to the function that should be used for the evaluation. The parameter `u:useDataContainer` is optional and it specifies the context container, from which the data should be queried. If no container is defined, then the general context `G` is queried. Currently, the evaluation is performed by executing an experimental RAB called `UtilityEvaluator` (lines 5-8). However, in the future a simpler call can be used (line 10). The reason for not using the simpler call is the

```
u:eval1 rdf:type u:Evaluation
       ; u:useFunction u:uf1
       ; u:useDataContainer ?dataCont .

{ sapl:I sapl:do java:ubiware.experimental.UtilityEvaluator }
sapl:configuredAs {
        u:eval sapl:is u:eval1
} .

sapl:I sapl:evaluate u:eval1 .
```

FIGURE 43 The evaluation of a utility function

DATA

```
x:rA rdf:type x:Route
   ; x:cost "0.25"^^xsd:float
   ; x:speed "0.5"^^xsd:float .

x:rB rdf:type x:Route
   ; x:cost "0.8"^^xsd:float
   ; x:speed "0.6"^^xsd:float .

x:rC rdf:type x:Route
   ; x:cost "0.1"^^xsd:float
   ; x:speed "0.3"^^xsd:float .

x:rD rdf:type x:Route
   ; x:cost "0.6"^^xsd:float
   ; x:speed "0.9"^^xsd:float .
```

RESULTS

```
u:eval1 u:result (
[rdf:type u:ResultElement ; u:hasResourceURI x:rC ; u:hasEvalValue "30"^^xsd:float]
[rdf:type u:ResultElement ; u:hasResourceURI x:rA ; u:hasEvalValue "20"^^xsd:float]
[rdf:type u:ResultElement ; u:hasResourceURI x:rD ; u:hasEvalValue "15"^^xsd:float]
[rdf:type u:ResultElement ; u:hasResourceURI x:rB ; u:hasEvalValue "7.5"^^xsd:float]
) .
```

FIGURE 44 Result of the utility function execution on top particular data

fact that the simpler call would require changes in the Ubiware engine, while the use of a RAB is inherently supported.

The result of both calls is the same. Figure 44 shows the data consisting of four routes named A, B, C and D. Each route is annotated with its speed and cost. The result is an ordered list of result elements. Each result element describes the evaluation value (score) and the resource's URI. According to this data and function, we may conclude that the best route is the route C with score of 30.

We also provide an ontology for checking the consistency of utility function definitions, calls and results. The OWL ontology can be found in Appendix 1.1 and its visualization is available in Figure 45.

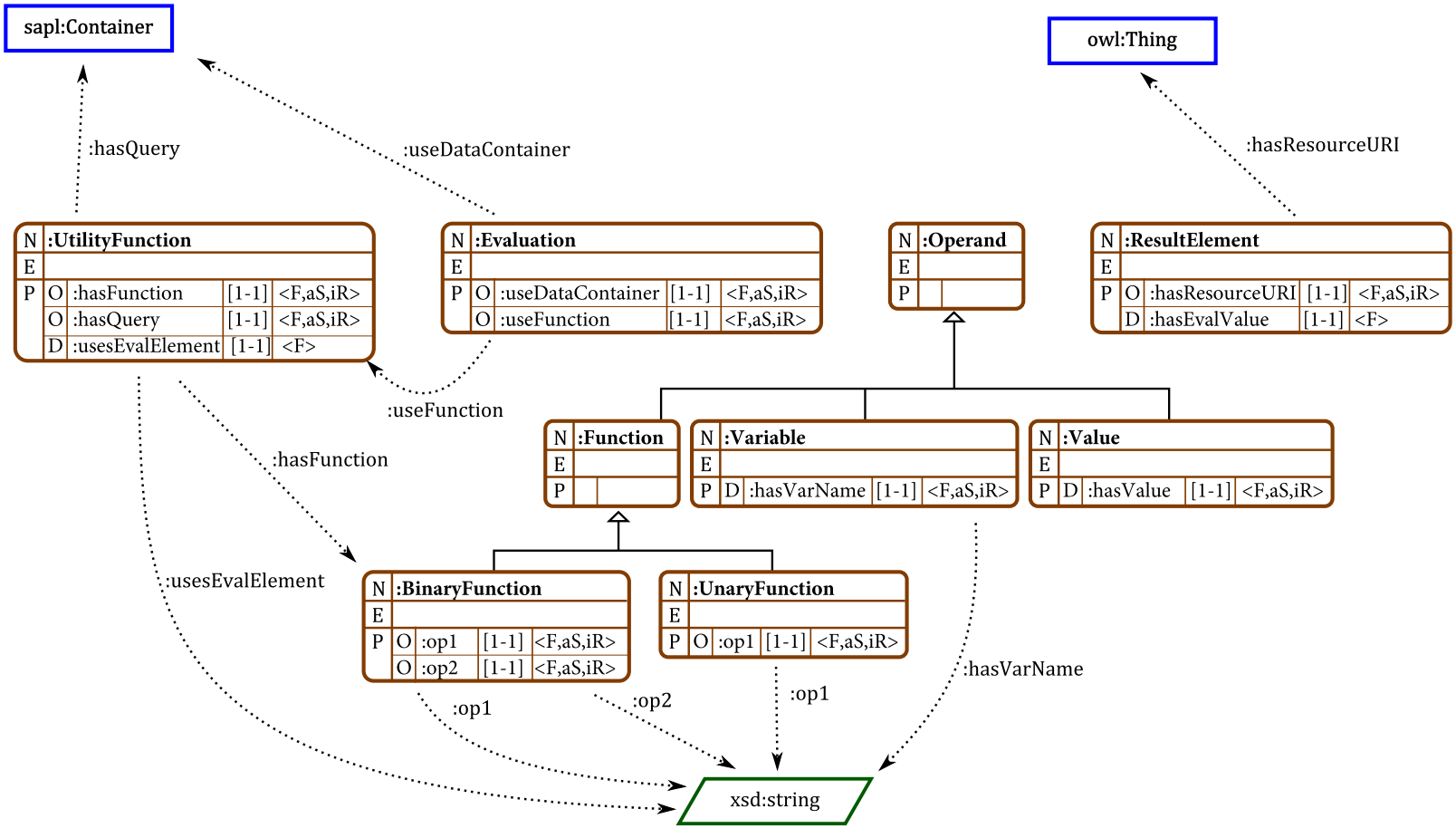


FIGURE 45 Visualization of the utilify function ontology

```
u:eval3 u:result {
{[rdf:type u:ResultElement] u:resourceURI x:rC ; u:evalValue "30"} x:order "1" .
{[rdf:type u:ResultElement] u:resourceURI x:rA ; u:evalValue "20"} x:order "2" .
{[rdf:type u:ResultElement] u:resourceURI x:rD ; u:evalValue "15"} x:order "3" .
{[rdf:type u:ResultElement] u:resourceURI x:rB ; u:evalValue "7.5"} x:order "4" .
} .

{
  u:eval3 u:result {
    {?res u:resourceURI ?x} x:order "3" .
  } .
  u:eval3 u:result {
    {* rdf:type u:ResultElement} x:order ?ord .
  } .
  ?m sapl:max ?ord
} => {
  { sapl:I sapl:do java:ubiware.shared.PrintBehavior } sapl:configuredAs {
    p:print sapl:is "There are ?m elements and the third element is ?x"
  } .
} .
```

FIGURE 46 A variation of the utility function result and a conditional commitment showing two operations on top of the result

4.1.3 Other improvements

When representing an ordered list of result elements, we used a method based on containers as described in the RDFS specification (Guha and Brickley, 2004). However, it is possible to use other methods as well. Melnik and Decker (2001) describe seven methods for representing the order in RDF. All of these methods have advantages and disadvantages when it comes to operations on top of ordered lists, in particular operations in S-APL. Therefore we introduce a different approach resembling the *Order by Ordinal Properties* method. The elements of the list are stored in a context container, where statements about them indicate their order. Figure 46 shows the method together with a S-APL conditional commitment querying the total number of elements in the list and the 3rd element. It is also possible to perform other statistical operations, such as the average evaluation value in the list. In general, utility functions in S-APL can support several methods for result representation. This can be done by adding a property to the `u:Evaluation` resource, which would indicate the desire result list type. The developer would determine the most desirable list type based on the operations that he/she wants to perform with it later in the code.

Another improvement could the introduction of meta-functions. Meta-functions are classes of functions. It is also possible to look at them as function providers. An example could be a multiplication meta-function with one argument n. An instance of this meta-function would an n-ary multiplication function of type `u:Function`. By associating a certain execution logic to the meta-function, a new fully-defined function could be created on the fly. Figure 47 shows an example of this approach. When the Ubiware engine detects an unknown function class (`x:TernaryMul`) on line 4, it tries to find its definition by looking for its class (function's metaclass). In the case of `x:MulMetafunction`, the defintion is

```
x:TernaryMul rdf:type u:Function, x:MulMetafunction
           ; x:hasArity "3" .

x:fx rdf:type x:TernaryMul
   ; u:operand1 [ rdf:type u:Variable ; u:hasVarName "x" ]
   ; u:operand2 [ rdf:type u:Variable ; u:hasVarName "y" ]
   ; u:operand3 [ rdf:type u:Value ; u:hasValue "4"^^xsd:integer ] .
```

FIGURE 47 Meta-function example in S-APL

primitive: expect the amount of operands determined by the arity and multiply them. In this particular example it means that $f_{TernaryMul}(x, y, z) = x \cdot y \cdot z$. Note that (as mentioned earlier) metaclasses are only allowed in OWL Full. OWL DL does not support them.

4.2 Belief safeguards

4.2.1 Motivation

In 2010, a new policy system was introduced in Ubiware prototype version 3.0 (Terziyan et al., 2010b). This policy system restricts agent's external actions. A policy check is triggered whenever an unconditional commitment statement is found in the global context G. The implementation is based on the idea that every agent has a policy checker object that restricts its behavior and there is a policy checking agent on the platform. Every time an agent performs an action, it first asks the policy checker object if the action is allowed. If yes, it performs it. If no, it redirects its request to the policy agent, which gives the definite answer based on the global set of policies. We will call this system *action policies*.

In our experience, when defining agent's behavior, there are some critical states that the agent should not achieve, because it may be in conflict with its goals. Also, there might be some desired states that an agent should always be in. The set of desired or undesired states depends on the tasks that the agent performs. In the case of a Ubiware agent, the state of the agent can be determined by its beliefs, therefore we are interested in a set of *desired and undesidered beliefs*.

In the current implementation of Ubiware, a Ubiware developer can utilize implications to check for existence or nonexistence of certain beliefs. The check (S-APL query) would be performed in the left-hand side of the implication and the action would provided in the right-hand side. This does not differ from any other implication, e.g. a conditional commitment. The existence or nonexistence of such beliefs is critical in terms of agent's operation and it should be treated with a higher priority than simple implications. Therefore it would be helpful to the Ubiware developer to have a simple method to check for existence or nonexistence of certain beliefs. We will call these methods *belief safeguards*.

In general, some belief safeguards might be more important than others. For

example, an agent controlling a smartphone does not wish to be in a situation when it is without any Internet connection. On the other hand, it must not get into a situation that its battery is below 15% and it still consumes more than 0.1 Watts of energy. Clearly, the second state is more dangerous than the first state. Thus, the decision of turning off the Internet connection when the battery is below 15% is permissible, even though it breaks the first safeguard. Therefore, we believe that each safeguard should have a severity level associated with it.

There are several reasons why safeguards are not called policies. First of all, an agent with hard policies must not enter a state that would break any of the hard policies. An agent looks at the consequences of its actions and if the consequence breaks a policy, the action is not performed. A safeguard is "weaker" in its protecting ability. A safeguard can be broken if a more important safeguard must be protected. Also, we base the idea of safeguards on the fact that an agent cannot always know all the consequences of its actions. If the safeguard is broken, only after that the agent performs corrective actions. For example, it makes little sense to put a policy guarding the fact that the smartphone should not be operated in temperatures below $-30\,^{\circ}\mathrm{C}$, since it might not be in the power of the phone to affect this parameter. Secondly, a Ubiware agent is based on RDF and RDF is based on the open world assumption. Therefore, if a certain belief does not exist in KB, it does not mean that this belief is false. Falsehood must be annotated explicitly. The definition of a safeguard is not based on the truth and falsehood of a belief, but rather on the existence or nonexistence of the belief.

4.2.2 Implementation

We introduce two safeguard types – existence safeguard and nonexistence safeguard. As the name suggests, an existence safeguards is guarding a set of beliefs that should always exist. A nonexistence safeguard is guarding beliefs that should never exist. The existence of beliefs is determined by a S-APL query. Figure 48 shows the example of S-APL code describing two safeguards mentioned earlier.

The safeguard checking mechanism is implemented with the help of S-APL metarules (see Appendix 2.1). The metarules were chosen, because they are triggered before all the other implications in the G context. This way the safeguards are given a higher priority than the rest of the code. In the case of a nonexistence safeguard, a metarule is used to check that the beliefs guarded by the prohibition safeguard do not exist. For this purpose the `sapl:doNotBelieve` construct is used. In the case of an existence safeguard, the condition in its original form is used. If a safeguard has been broken, then a special ticket object is created describing the situation. The ticket is used to resolve the problem and it goes through several states. The state diagram is shown in Figure 49.

Once the ticket is detected, it enters the *detected* (DET) state. The Ubiware developer may register a handler to handle the ticket. In that case the state moves to *resolution in progress* (PRO). The resolution process might finish in three ways. Firstly, the conflict is resolved and the safeguard is not broken anymore. In that case the ticket is put to the *resolved* (RES) state. Secondly, the conflict cannot be

```
@prefix sfg: <http://www.ubiware.jyu.fi/safeguard.owl#> .

x:safeBattery rdf:type sfg:Safeguard, sfg:NonexistenceSafeguard
    ; sfg:hasImportance "10"^^xsd:integer
    ; sfg:hasCondition {
      x:thisDevice x:hasBatteryLevel ?lev .
      ?lev < "15"^^xsd:integer .
      x:thisDevice x:hasConsumption ?con .
      ?con > "0.1"^^xsd:float .
    } .

x:safeInternet rdf:type sfg:Safeguard, sfg:ExistenceSafeguard
    ; sfg:hasImportance "5"^^xsd:integer
    ; sfg:hasCondition {
      x:thisDevice x:hasConnection ?con .
      ?con rdf:type x:InternetConnection .
      ?con x:hasStatus x:active .
    } .
```

FIGURE 48 An example of two safeguards

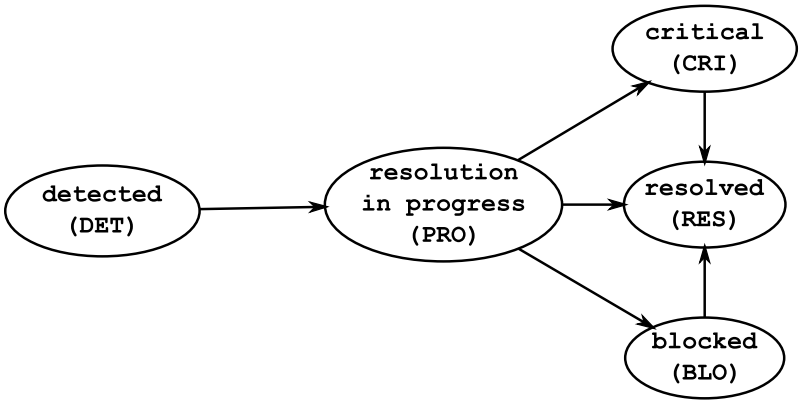


FIGURE 49 State diagram of safeguard ticket states

```
x:t1 rdf:type sfg:Ticket, sfg:TicketBLO
     ; sfg:inState sfg:stateBLO
     ; sfg:handlingSFG x:safeInternet
     ; sfg:blockedBySFG x:safeBattery .
```

FIGURE 50 Example of a safeguard ticket in the blocked state

resolved due to the fact that another safeguard with a higher priority would be broken. This is indicated by moving to the *blocked* (BLO) state. It is also indicated which higher priority safeguard caused the block. Lastly, the agent does not know how to solve the problem and the ticket enters the *critical* (CRI) state. It depends on the programmer what should happen if a ticket in such a state is found. In some cases the program might be terminated, suspended, or the user must resolve the problem. Once the ticket is released from its BLO or CRI state, it enters the RES state and is discarded. In all cases, the ticket always starts its life in the DET state and always ends its life in the RES state. An example of a ticket in the BLO state can be seen in Figure 50. Note that the state of the ticket is indicated both by its class and the `sfg:inState` property indicating the state. The usage of class was chosen, because in OWL it is possible to easily add an extra property for the ticket in a certain state. For example the property `sfg:blockedBySFG` makes sense only in the BLO state. The safeguard ontology is available in Appendix 1.2 and its visualization can be seen in Figure 51.

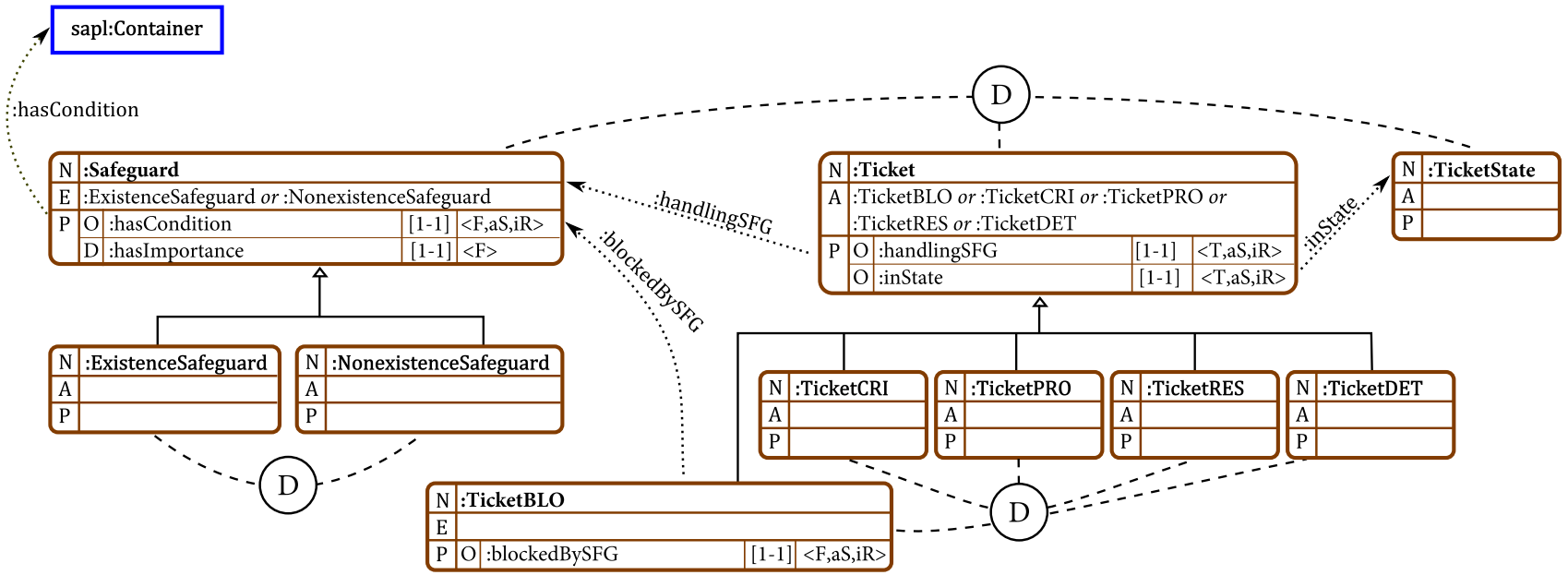


FIGURE 51 Visualization of the safeguard ontology

4.3 Plans and actions

4.3.1 Motivation

As mentioned previously, SAF uses a hybrid approach based on two concepts – MAPE-K cycle and three-layered architecture for self-management. In both these concepts, planning is of great importance. In the case of MAPE-K, planning is one of the steps in the cycle, which is represented by the letter P in the word MAPE. In the case of the three-layered architecture, the top layer called "Goal management" is responsible for plan creation.

Nikitin et al. (2009) introduced the concept of Ontonuts into the Ubiware platform. An ontonut is a semantic software component, whose instance represents a capability with a known input and expected output. An Ontonut is described by providing a precondition, effect and a script. Using a backward chaining planning technique, it is possible to create Ontonut plans. Nikitin (2011) uses Ontonuts as one of the tools to solve the problem of distributed information querying in the industrial domain. Whereas this solution is suitable for this particular domain, we believe that by providing a more general approach to planning we may gain several benefits.

The planning problem has been studied for several decades. Generally speaking, there is a great amount of planning approaches, each having certain benefits and drawbacks. For a given domain and problem, one planner might perform better than the other. For a different domain and problem, the situation might be reversed. There are various outlets, where new or improved planning algorithms are introduced every year (e.g. International Conference on Planning and Scheduling). We believe that instead of creating a planner specifically for Ubiware, it is more benefitial to utilize already existing planners. The Planning Domain Definition Language (PDDL) is a widely accepted problem and domain description language in the area of planning. By transforming the planning problem from S-APL into PDDL we may utilize all planning algorithms capable of accepting PDDL as the input language.

4.3.2 Overview

Planning takes place in an external module that accepts planning problems described in PDDL. Creation of new planning algorithms is outside the scope of this publication. Also, a planner implementation in S-APL language is not provided, because the language is not suitable for such a task. Principally, it could be possible to implement a planner using S-APL, but the implementation would be cumbersome and it would bring no extra benefits in comparison to already existing planners. We rely on an external planner capable of accepting PDDL as the input language.

Typically, a planning problem described in PDDL consists of two files – domain file and problem file. The domain file describes available predicates and

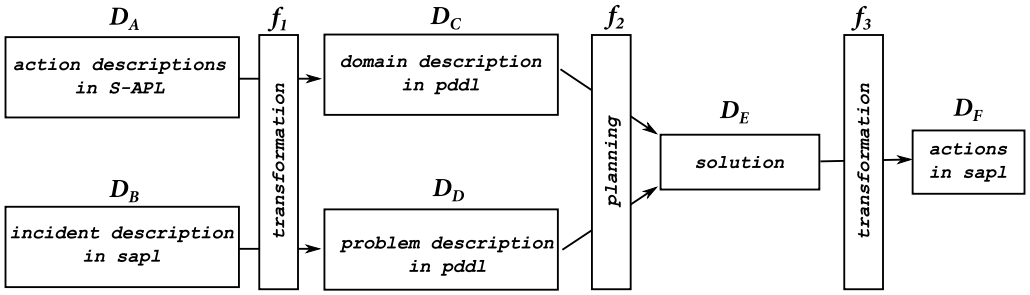


FIGURE 52 Overview of the planning process in Ubiware

available actions. The problem file contains the description of objects, initial state and goals. If the domain does not change, a single domain file can be used to solve various problems described in the problem files. Since in SAF the amount of predicates and actions does not change often, the content of the domain file will in most cases stay unchanged as well. However, for every new problem that the planner is going to face, a new problem file is needed.

The planning process is described in Figure 52. The first step is to convert the action descriptions in S-APL (D_A) into the domain description in PDDL (D_C). This is done by the SAPL-PDDL transformation (f_1), which will be described later. The concrete incident described in S-APL (D_B) is also transformed by (f_1) into a PDDL problem description file (D_D). Both files are then used by the planner (f_2) that provides a solution (D_E). The solution is then transformed by the solution transformation function (f_3) into a set of actions in S-APL (D_F).

The following subsections contain the description of actions and plans both in S-APL and PDDL. Also, the conversion between S-APL descriptions and PDDL descriptions is provided.

4.3.3 Plan ontology

The framework's planning component is responsible for plan creation. Each plan is described as a sequence of actions. Also, the plan itself is an action. This creates a recursion that allows one to create multi-level plans. Each action has preconditions and effects represented by containers consisting of positive and negative S-APL expressions. The ontology is formally described in Appendix 1.3 and visualized in Figure 53.

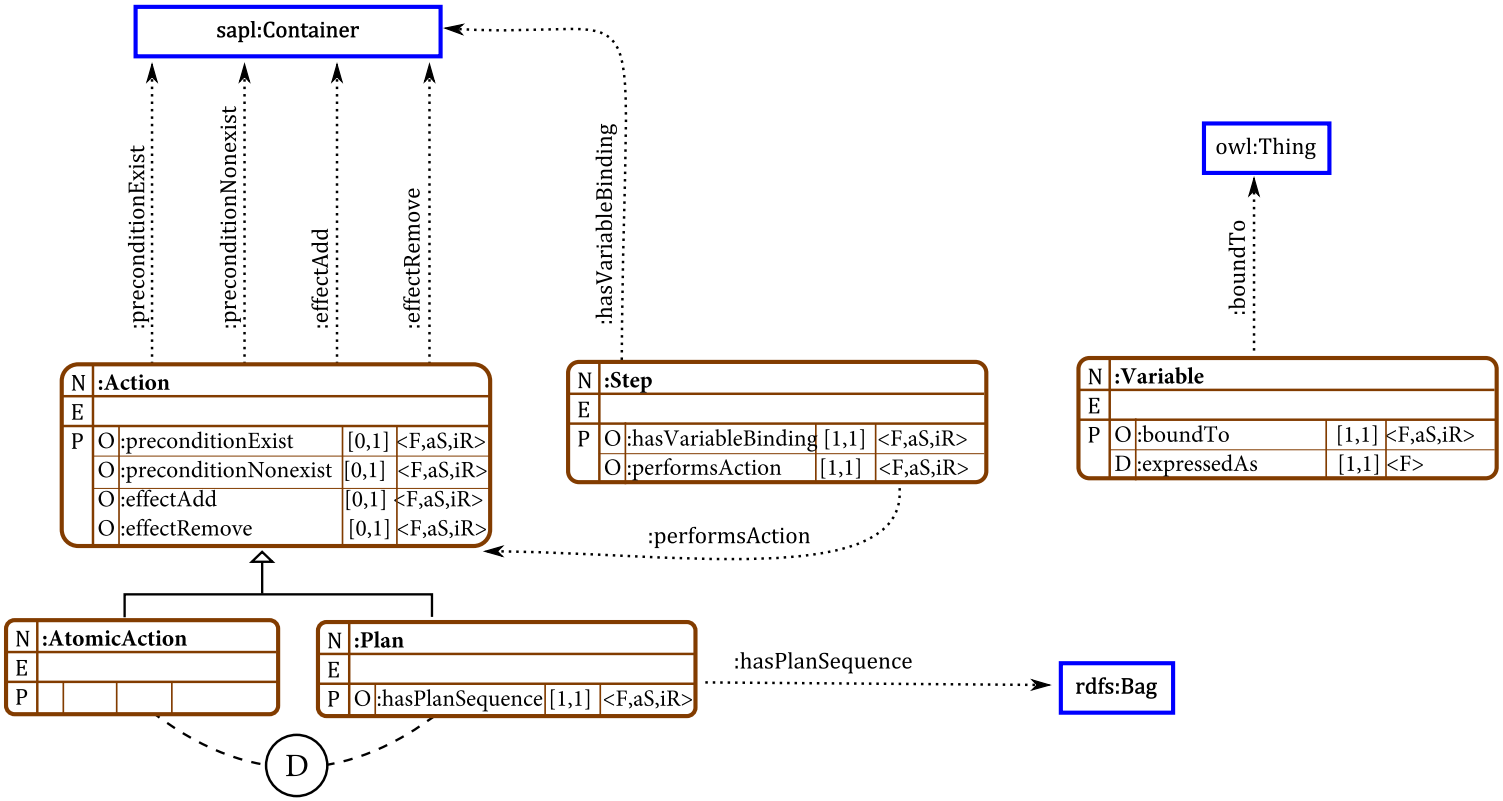


FIGURE 53 Visualization of the planning ontology

4.3.4 Action description in S-APL

An action is described in S-APL using a special resource of type `pla:Action`. The resource URI determines the action name and therefore for every action the URI must be unique. Every action can have a precondition and an effect. The precondition is a logical expression that must be true in order for the action to be applicable. The effect is a logical expression that becomes true once the action has been completed. Therefore we may consider an action a transition from a state described by the precondition into a state described by the effect. Both the effect and the precondition are described in two containers each. Lastly, a set of S-APL commands is associated with the action. These commands determine what should be done in order to perform the action. Figure 54 contains a simple action in S-APL and a set of beliefs.

The precondition is determined by two containers – the exist container (`pla:preconditionExist`) and the nonexist container (`pla:precondition-Nonexist`). The semantics is as follows. If the beliefs in the exist container do exist and the beliefs in the nonexist container do not exist, then the precondition is met. Similarly, there are two effect containers. The add container contains beliefs that will be inserted in the belief storage and the remove container contains beliefs that will be removed from the belief storage. The content of these four containers must be a list of S-APL query beliefs. These beliefs must not contain containers. Only variables, URIs and literals are allowed. Also, each belief must contain at least one and at most two variables.

4.3.5 Plan description in S-APL

As mentioned earlier, every plan is considered a compound action. Therefore every plan has the same properties as an action. Furthermore, a resource describing a plan has a property `pla:hasPlanSequence` which points to a list of so-called steps. A step can be considered an instantiated action. The step resource provides the information about the action that should be performed and the corresponding variable binding (`pla:hasVariableBinding`), where each variable is given a value (`pla:boundTo`). An example of a plan is shown in Figure 55.

4.3.6 SAPL-PDDL domain transformation

The transformation of an action in S-APL into PDDL consists of the following steps. Firstly, the name of the action in PDDL is equal to the URI of the `pla:Action` resource. Secondly, the parameters of an action are determined by going through all the beliefs in all effect and precondition containers and finding all the variables. In the case of the example from Figure 54 there are only two parameters – `?x` and `?y`. Lastly, the preconditions and effects are determined in the same way – by transforming the beliefs in RDF into unary or binary predicates, which will be explained in the next paragraph. The final product of the transformation can be seen in Figure 56.

```
x:getMarried rdf:type pla:Action
    ; pla:preconditionExist {
        ?x x:loves ?y .
        ?y x:loves ?x .
        ?x != ?y .
    }
    ; pla:preconditionNonexist {
        ?x x:marriedTo ?y .
        ?y x:marriedTo ?x .
    }
    ; pla:effectAdd {
        ?x x:marriedTo ?y .
        ?y x:marriedTo ?x .
    }
    ; pla:effectRemove {
        ?x x:loves ?y .
        ?y x:loves ?x .
    } .

x:John x:loves x:Mary .
x:Mary x:loves x:John .
x:Bill x:loves x:Mary .
x:Bill x:marriedTo x:Alice .
```

FIGURE 54 An example of a S-APL action

The first step of the transformation of an SPO triple into a PDDL predicate consists of determining the belief type depending on what elements were used to express it. All beliefs contained in precondition and effect containers of an action are S-APL queries. A S-APL query triple can be expressed using four elements – variables, URIs, literals and containers. Since containers do not have a counterpart in PDDL, statements containing them are not allowed in action descriptions. Let us divide the remaining element types into two categories – variables and non-variable (URIs or literals). Based on this division, there can be eight belief types in total. Out of these eight belief types, two are borderline cases – a belief consisting of three variables and a belief consisting of three non-variables. These two cases are not allowed, because as mentioned earlier, each precondition or effect belief must have at least one and at most two variables. The remaining six belief types and their conversion into PDDL predicates are displayed in Table 10.

The first three columns of Table 10 determine the type of the belief, where V_i means a variable at i-th position and N_i means a non-variable at i-th position. The combination of the first three columns determines the type of the predicate, e.g. the first row describes a VVN predicate, the second row describes a VNV predicate, etc. The fourth column determines the conversion schema. For binary predicates, the predicate name consists of a letter determining the position of the non-variable, semicolon and the URI or literal of that non-variable. The predicate name is then followed by two variables. In the case of unary predicates, the predicate name consists of two elements separated by a dash followed by a variable. The [] brackets denote a string transformation function. The reason for applying such a function

```
x:plan1 rdf:type pla:Plan, pla:Action
    ; pla:hasPlanSequence (x:step1 x:step2)
    ; pla:hasVariables {
        x:varCompX pla:expressedAs "compX" .
        x:varCompY pla:expressedAs "compY" .
    }
    ; pla:preconditionExist {
        ?compX rdf:type conf:Component .
        ?compX conf:implements ?capA .
        conf:thisSoftware conf:requires ?capA .
    }
    ; pla:preconditionNonexist {
        conf:thisSoftware conf:uses ?compX .
    }
    ; pla:effectAdd {
        conf:thisSoftware conf:uses ?compX .
    }

x:step1 rdf:type pla:Step
    ; pla:performsAction x:removeComponent
    ; pla:hasVariableBinding {
        x:varCompX pla:boundTo x:componentOld .
    } .

x:step2 rdf:type pla:Step
    ; pla:performsAction x:addComponent
    ; pla:hasVariableBinding {
        x:varCompX pla:boundTo x:componentNew .
        x:varCapA pla:boundTo x:capabilityX .
    } .
```

FIGURE 55 An example of a S-APL plan

TABLE 10 Types of S-APL query statements

Subject	Predicate	Object	PDDL predicate	Predicate type
V_1	V_2	N_3	([O:N_3] V_1 V_2)	B
V_1	N_2	V_3	([P:N_2] V_1 V_3)	B
V_1	N_2	N_3	([P:N_2-O:N_3] V_1)	U
N_1	V_2	V_3	([S:N_1] V_2 V_3)	B
N_1	V_2	N_3	([S:N_1-O:N_3] V_2)	U
N_1	N_2	V_3	([S:N_1-P:N_2] V_3)	U

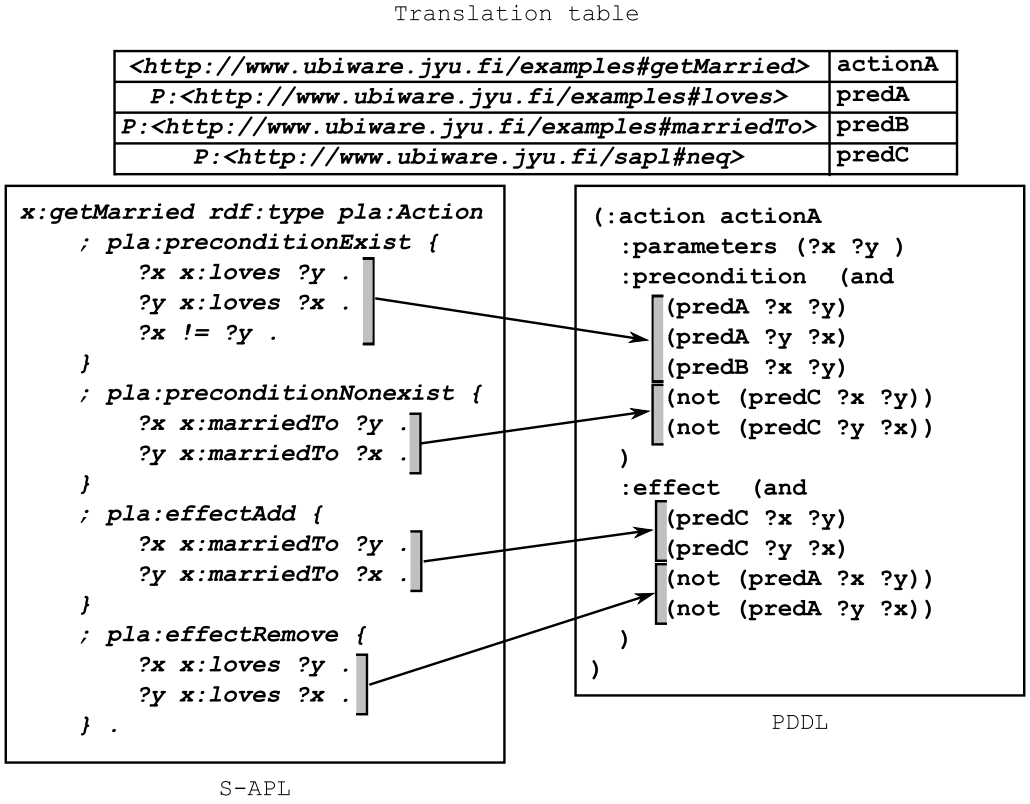


FIGURE 56 Conversion of a S-APL action into PDDL

is the fact that URIs might contain special characters that are not allowed in PDDL predicate names. Therefore a transformation into a PDDL-enabled string space is required. This transformation must be bijective, because a reverse transformation is needed once a plan is found. One of the possible transformation functions is a simple translation table (Figure 56).

There is one important aspect of the action description transformation which we call *effect expansion*. This situation happens if there is a special relationship between a certain binary and a certain unary predicate. We will show this situation on an example. Let `marriedTo(?x ?y)` be a binary VNV predicate corresponding to `?x x:marriedTo ?y`. Let `johnsWife(?z)` be a unary NNV predicate that translates into `x:john x:marriedTo ?z`. Let's assume that a certain action A promisses to marry any two people, which means that the effect of the action is `marriedTo(?x ?y)`. Then this means that A's effect should also include (be expanded to) `johnsWife(?y)`, meaning that it can marry john as well. We say that the predicate `marriedTo(?x ?y)` *expands* the predicate `johnsWife(?y)`. The expansion relationship between each of the 6 predicate types is shown in Figure 57.

A PDDL domain file consists of a set of predicates and a set of actions. An example of a converted PDDL domain file can be seen in Figure 58.

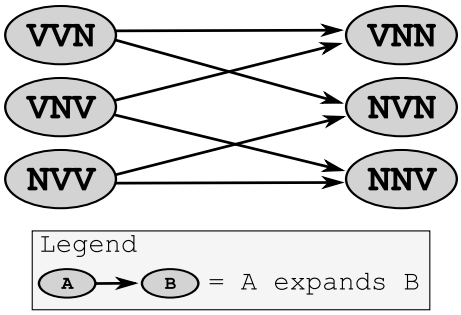


FIGURE 57 The effect expansion

```
(define (domain domain)
  (:predicates (predA ?x ?y) (predB ?x ?y) (predC ?x ?y)
)
  (:action actionA
     :parameters (?x ?y )
     :precondition  (and (predA ?x ?y)
       (predA ?y ?x) (predB ?x ?y)
       (not (predC ?x ?y)) (not (predC ?y ?x))
)
     :effect  (and
       (predC ?x ?y)
       (predC ?y ?x)
       (not (predA ?x ?y))
       (not (predA ?y ?x))
     )
  )
)
```

FIGURE 58 An example of a PDDL domain file

4.3.7 SAPL-PDDL problem transformation

For every incident reaching the planner component a PDDL problem description file must be created. The file is created based on the domain file and the knowledge base of the agent. A typical PDDL problem definition file consists of five elements – problem name, domain, objects, initial state and goal. During the conversion process, the problem name is derived from the incident URI, which is unique for every incident. Since in SAF there is only a single domain, the domain name is irrelevant as long as it matches the domain file name. The objects, initial state and goal are related and can be determined as follows.

The objects and the initial state can be determined together by accessing the domain description file and acquiring the list of predicates. An alternative to the list of predicates is the translation table, if one is used. The list of actions corresponds to a certain query pattern in S-APL language. For each action, the query pattern is used to query the knowledge base. The result of such a query is a result set with variable bindings. Each variable binding together with the predicate corresponds to one initial state statement. The objects are determined simply by collecting all variable values for all the variable bindings.

The algorithm is described in Algorithm 1. Let's consider a scenario from Figure 54. There is one action that has already been included into a PDDL domain description file and a set of four beliefs. The domain description file (Figure 58) states that there are three predicates with corresponding S-APL query statements, therefore the `predicates` variable (algorithm line 3) contains three elements. We will show the algorithm for the first iteration of the for cycle on line 4. Let's assume that p is equal to predicate `predA`. On line 5 the query is performed. The corresponding query of predicate `predA` is `?x x:loves ?y`. Once the query is performed, the result set contains three variable bindings (line 7), each having two variables (?x and ?y). For each variable binding a statement object with corresponding predicate is prepared (lines 9 and 10). For each variable name we determine the variable value (line 12), add the value to the statement object (line 13) and insert it into the result set (line 14). Naturally, for unary predicates the cycle consists of one iteration only. The same happens for the other two remaining predicates. The result is then a set of URIs describing the S-APL resources and a set of statements with bound variables defining the initial state. These resources can then be transformed and used as objects in PDDL problem description file.

The goal is determined by the safeguard condition that caused the incident to occur. Every incident is caused by a ticket, which can be determined by following the `inc:responsibleTicket` property of the incident resource that triggered planning. The safeguard can be determined by following the `sfg:handlingSFG` property of the ticket. An example of a complete PDDL problem file can be seen in Figure 59.

Algorithm 1: PDDL conversion algorithm

```
   input  : KnowledgeBase kb
   input  : File domainFile
   output : URI[] result
   output : Statement[] stats
 1 URI[] resources;
 2 Statement[] stats;
 3 Predicate[] predicates = getAllPredicates (domainFile);
 4 foreach p in predicates do
 5     SAPLQuery query = p.getSAPLQuery ();
 6     String[] varNames = query.getVarNames ();
 7     VariableBinding[] rs = kb.performQuery (query);
 8     foreach vb in rs do
 9         Statement statement;
10         statement.predicate = p;
11         foreach var in varNames do
12             URI resource = vb.getVarValue (var);
13             statement.addVarValue (var,resource) ;
14             resources.add (resource);
15         end
16         stats.add (statement);
17     end
18 end
```

```
(define (problem incidentX)
  (:domain domain)
  (:objects john mary bill alice)
(:init (predA john mary)
       (predA mary john)
       (predA bill mary)
       (predB bill alice)
)
  (:goal (predB john mary))
)
```

FIGURE 59 An example of a PDDL problem file

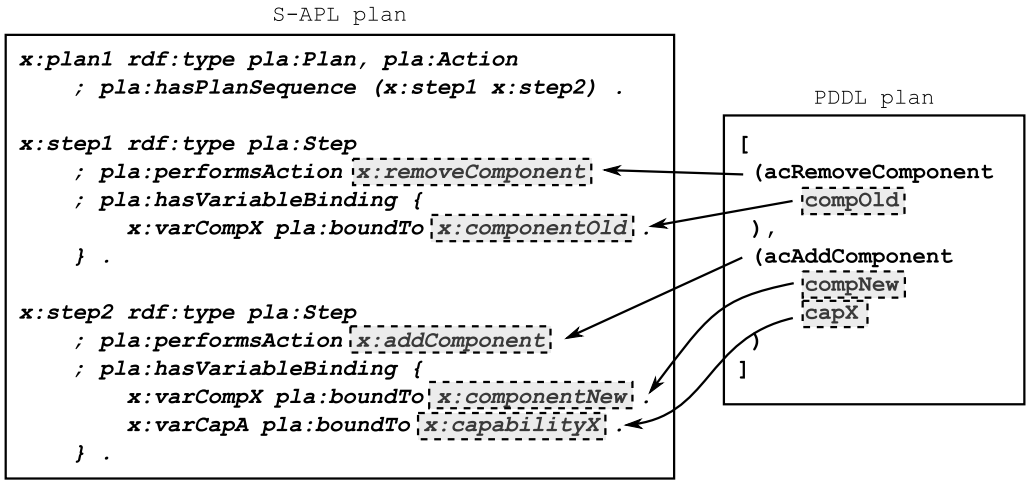


FIGURE 60 Conversion of a PDDL plan into S-APL

4.3.8 Solution transformation into S-APL

The solution to a planning problem is a sequence of actions with bound variables. If each action is performed as defined in sequence, the goal will be reached. A sample action sequence transformation into a S-APL plan is shown in Figure 60. In order to save space, the sequence consists of only two actions and the plan is missing precondition and effect containers. Using a transformation table or a bijective function, the PDDL actions and objects are transformed into URIs. These are then used as action identifiers and variable values for plan steps.

4.4 Related ontologies

Ontology	Prefix	Purpose	Defined in
utility function ontology	`ufn:,u:`	Describes utility functions	Appendix 1.1
safeguard ontology	`sfg:`	Describes existence and nonexistence belief safeguards	Appendix 1.2
planning ontology	`pla:`	Describes actions, plans and their conditions	Appendix 1.3

5 UBIWARE-BASED IMPLEMENTATION OF SAF

Chapter 5 introduces a SAF implementation based on the Ubiware platform enriched by the three new improvements that were introduced in Chapter 4. Firstly, we present ontologies that are used to describe agent's beliefs that are used for knowledge representation and decision-making. Secondly, the platform and software configurations are described. Moreover, we define the service facilitator and related processes (e.g. calculation of reputation). Lastly, each SAF processing element is described in terms of its implementation in Ubiware and S-APL.

5.1 Ontologies

In Chapter 3 we discussed the classification of sensors, actuators and incidents. We presented a simple classification based on subsumption. In this section we introduce a set of formal OWL ontologies in Turtle notation. Due to their length, all OWL files will be provided as Appendices. However, in each of the following subsections we discuss the choice of ontologies and we also provide sample ABox assertions.

Many assertions might appear to be long despite using a relative concise notation (Turtle). We assume that the developer will use some kind of graphical tool (e.g. Protégé[1] or similar) that would allow him/her to conveniently configure the platform, sensors, actuators and other components. The output of the graphical tool would be an S-APL document.

5.1.1 Sensor ontology

The formal OWL ontology is provided in Appendix 1.4 and its visualization in Appendix 3.1. The platform keeps track of available sensors by using `sen:Sensor` resources. According to the Ubiware philosophy, sensors are implemented as RABs and therefore `sen:implementedIn` property points to the RAB that per-

[1] Available at http://protege.stanford.edu/

```
x:sensorCPUTemp rdf:type sen:Sensor, sen:MeasurementSensor,
                         sen:PullSensor, sen:HWSensor
    ; sen:implementedIn java:fi.jyu.ubiware.sensors.CPUSensor
    ; sen:providedObjectType x:CPUTempMeasurement .

x:measurement45356 rdf:type sen:SensorResult, x:CPUTempMeasurement
    ; sen:readFrom x:sensorCPUTemp
    ; sen:hasTimestamp "2013-09-12T14:36:23.556Z"^^xsd:dateTimeStamp
    ; sen:value "25"^^xsd:integer .

sapl:I sen:haveCurrentState x:measurement45356 .
```

FIGURE 61 Example of a sensor assertion

forms this function. Each time some data is sensed, a special resource of type `sen:SensorResult` is created that represents the measured data or event. Each result provides a timestamp using the `sen:hasTimestamp` property together with an event object (for event sensors) or a measured value (for measurement sensors). Since in general there can be many sensors performing the same function, every sensor result contains a link (`sen:readFrom` property) to the sensor that it was obtained from. An example of a sensor with a result is shown in Figure 61.

The timestamp on each measurement is provided for the historical data analysis. The agent always remembers the current sensory value for each sensor. This achieved through the last statement in Figure 61. The `sapl:I` URI refers to the current agent. The object refers to the URI of a measurement resource indicating the last measured value. There can be several statements like this, one for each sensor.

Each sensor can be configured to remember only certain amount of values. In Appendix 2.2 we provide a piece of S-APL code, where a CPU temperature sensor is called. Upon a successful execution a condition starting at line 13 is checked. The condition specifies that if there are more than 10 entries from that particular sensor, the oldest one will be deleted (line 24). This way there are always at most 10 measurements. The amount of remembered values can easily be modified (also at runtime) through a sensor parameter. An example of parametric self-configuration could be a case when thanks to this feature it is possible to change the amount of remembered measurements depending on the available memory. As the amount of free memory changes over time, so does the amount of remembered measurements.

5.1.2 Actuator ontology

The formal OWL ontology can be found in Appendix 1.5 and its visualization in Appendix 3.2. It follows the class hierarchy introduced earlier in Figure 39 (page 89). It also defines several object and datatype properties. Firstly, each software actuator has the property `act:affects`, which indicates the capability of the adaptive software that is influenced by it. In case the actuator is implemented as a RAB, it is referenced by the `act:implementedIn` property. Figure 62 shows an

```
x:camera rdf:type act:Actuator, act:HWActuator
   ; act:implementedIn java:fi.jyu.ubiware.phone.CameraActuator .

x:fontActuator rdf:type act:Actuator, act:SWActuator,
                            act:ParametricActuator
   ; act:implementedIn java:fi.jyu.ubiware.ui.FontActuator
   ; act:affects x:capabilityUI .
```

FIGURE 62 Example of an actuator assertion

example of an actuator that changes the font size in the UI.

5.1.3 Incident ontology

Incident is any context-relevant change in the sensory input. A formal OWL ontology following the classification from Chapter 3 can be found in Appendix 1.6 and its visualization in Appendix 3.3. An incident is created when a policy expressed by a safeguard is broken by some sensory input. Therefore each incident is related to the policy that was broken (`inc:brokenPolicy` property), ticket that deals with the issue (`inc:responsibleTicket` property) and the sensory input that caused it (`inc:responsibleSensorResult` property). An example of an incident can be found later in the text in Figure 75 (page 134).

It is possible to extend the ontology and define own types of events. This can be done by in the application profile. The profile can include the developer's own ontology based on the default event ontology. The developers can also provide categorization rules for event annotation according to their own ontology.

5.1.4 Configuration ontology

The configuration ontology is related to software and platform configuration. The formal OWL ontology is provided in Appendix 1.7 and its visualization in Appendix 3.4. Since this ontology deals with various aspects of configuration, we provide two separate subsections.

5.1.4.1 Structural configuration

We assume that the adaptive software is designed using a component-based approach. We also assume that the component connections are loose enough to allow component replacement. The configuration ontology captures the relationship between the software, components and the service that they provide. Figure 63 shows the basic class hierarchy.

The central concept is the adaptive software represented by `conf:AdaptiveSoftware` class. The adaptive software provides and requires certain capabilities (`conf:Capability`) for its operation. Some of these capabilities might be required mandatorily (`conf:requiresMandatorily`) and some optionally (`conf:requiresOptionally`). While the mandatory capabilities are constantly

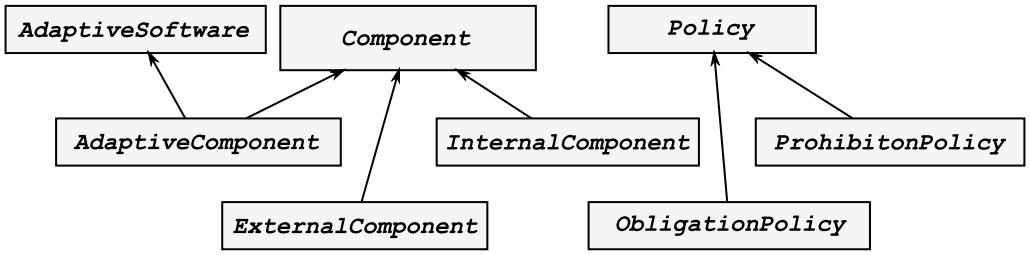


FIGURE 63 Configuration class hierarchy

needed in order for the software to operate, the optional ones do not always have to be available. These two properties indicate the need for a certain capability. This information depends on the application logic and therefore it usually does not change during the life of the software. Capabilities can be understood as abstract component descriptions.

A capability may be provided by a an internal (`conf:InternalComponent`) or external component (`conf:ExternalComponent`). An internal component originates from the same system as the software that uses it. An external component originates from the outside, e.g. is provided by another agent/device. In the case of an external component, the provider agent is annotated using the `conf:providedByAgent` property. The pool of available components is managed by the service facilitator. For every needed capability, the software may choose a component that provides it. This fact is indicated by the `conf:currentlyUtilizes` property. Contrary to the software-capability relationship, the software-component relationship changes depending to the context. There is a number of reasons for component reconfiguration, e.g. a better component was discovered, the used component is out of reach, the policies have changed, etc.

In cases when the adaptive software provides a certain capability, it also becomes a component itself. This might happen if the software advertises itself to other devices. In these situations the resource describing the software is of type `conf:AdaptiveComponent`.

For the purposes of trust and reputation management (discussed later) each component's abilities are evaluated from four different perspectives. The quality of the provided service is expressed by the `conf:hasQuality` property. The response time is described by `conf:hasPromptness`. The degree into which the external component stays true to the negotiated contract is recorded by `conf:hasMisjudgment`. Finally, the `conf:hasExternalFactor` expresses the factors beyond the control of both parties, e.g. a network fault or a power outage. Figure 64 shows a sample external component annotation.

5.1.4.2 Adaptation configuration

An important element of the adaptation are the policies. A policy is represented by the class `conf:Policy` with two subclasses – `conf:ObligationPolicy` and `conf:ProhibitionPolicy`. The importance of the policy is determined by the

```
x:compFrontDoorControl rdf:type conf:Component,
                               conf:ExternalComponent ;
         conf:provides x:capabilityDoorControl ;
         conf:providedByAgent x:agentDoorOperator ;
         conf:hasQuality "0.91" ;
         conf:hasPromptness "150" ;
         conf:hasMisjudgment "0.1" ;
         conf:hasExternalFactor "0.9" .
```

FIGURE 64 External component annotation

```
x:pol2 rdf:type  conf:Policy, conf:ObligationPolicy
    ; conf:hasImportance "10"
    ; conf:hasCondition {
         conf:thisSW conf:requiresMandatorilly ?capA .
         conf:thisSW conf:currentlyUtilizes ?compX .
         ?compX rdf:type conf:Component .
         ?compX conf:implements ?capA .
    } .
```

FIGURE 65 Example of a policy in S-APL

`conf:hasImportance` property. The importance can be expressed through 10 levels, where 10 represents a hard policy and 1-9 represents soft policies. Each policy has a condition associated with it (property `conf:hasCondition`) which is expressed as a series of S-APL query statements. Figure 65 shows a hard policy which tells the framework that all mandatory capabilities must be provided by a component. In other words, if there is a situation, where the software requires a mandatory capability and no component implements it, the policy is broken. This also a good example of a platform-wide policy, since we can assume that mandatory capabilities required by software are mission critical.

5.1.5 Summary

Ontology	Prefix	Visualization	Description in OWL
sensor ontology	`sen:`	Appendix 3.1	Appendix 1.4
actuator ontology	`act:`	Appendix 3.2	Appendix 1.5
incident ontology	`inc:`	Appendix 3.3	Appendix 1.6
configuration ontology	`conf:`	Appendix 3.4	Appendix 1.7

```
x:sensorGPS rdf:type sen:Sensor, sen:MeasurementSensor, sen:PullSensor,
                     sen:EnvSensor, sen:PhysicalEnvSensor
    ; sen:implementedIn java:fi.jyu.ubiware.sensors.GPSSensor
    ; sen:providedObjectType x:GPSLocation .

x:sensorFacebook rdf:type sen:Sensor, sen:EventSensor, sen:PushSensor,
                          sen:EnvSensor, sen:VirtualEnvSensor
    ; sen:implementedIn java:fi.jyu.ubiware.sensors.FBSensor
    ; sen:providedObjectType x:FacebookUpdate .

x:wifiModule rdf:type conf:Component, conf:InternalComponent
    ; conf:provides x:networkProviderCapability .

x:GSMModule rdf:type conf:Component, conf:InternalComponent
    ; conf:provides x:networkProviderCapability .
```

FIGURE 66 An example of a platform configuration file

5.2 Configurations

5.2.1 Platform configuration

In order for the SAF middleware platform to provides support to the software, it must be properly configured. The platform configuration files is a S-APL document consisting of three parts. Firstly, the available sensors are described based on the sensor ontology described earlier. Secondly, the components and capabilities are described based on the configuration ontology. Figure 66 shows a sample platform configuration file. Lastly, generic platform-wide actions are described (e.g. component replacement or sensor calibration). As mentioned earlier, such configuration document in S-APL can be generated from a graphical tool.

5.2.2 Software configuration

5.2.2.1 Software structural profile

The structural profile is a S-APL file that describes software's dependencies on capabilities and components. Since the adaptive software itself is a component as well, it also describes what kind of capabilities are provided by the software. The resource `conf:thisSW` of type `conf:AdaptiveSoftware` refers to the software being described. Figure 67 contains a sample structural description of a software that receives Facebook event updates and shows them to the user. If the user's location is available, then also other information might be provided, e.g. walking distance to the event. The software strictly requires network access and a Facebook sensor. The initial configuration uses a wireless network as the network provider and a Facebook adapter as the event provider. Naturally, this configuration may change over time, since it is an adaptive software. In 2011, as part of the Tivit SHOK Cloud Software Program, we presented a similar Ubiware-

```
conf:thisSW rdf:type conf:AdaptiveSoftware,
                     conf:Component, conf:AdaptiveComponent
   ; conf:provides x:FBEventMashupCapability
   ; conf:requiresMandatorily x:networkProviderCapability
   ; conf:requiresMandatorily x:sensorFacebook
   ; conf:requiresOptionally x:sensorGPS
   ; conf:curentlyUtilizes x:wifiModule
   ; conf:curentlyUtilizes x:sensorFacebook .
```

FIGURE 67 An example of a software structural profile

based application called "Social media mashupper", which collected social media data for a single user and presented them as a mashup (Cochez and Nagy, 2011).

5.2.2.2 Software adaptation profile

The adaptation profile is a S-APL file provided by the software developer and used by the platform to understand what kind of adaptation the software creator is seeking. The profile consists of three parts. Firstly, the policies are described. Each policy (`conf:Policy`) contains a logical expression in S-APL and the severity level. The severity level determines how soft or hard the policies are. Soft policies might be broken if it protects a higher level policies. Hard policies are mission-critical and must never be broken. The policy can either be an obligation or a prohibition. The expressions in obligation policies must always be true in the G context, while the expressions in prohibition policies must never be true in the G context. Everything else is considered to be allowed with respect to the policies.

Policies are also a way to express the goals of the software in terms of its health. A policy defines, which operational parameter values are considered normal (healthy). The information about required sensors from the structural configuration determines the context-relevant sensory data.

The second element of the adaptation profile is the information about software actuators. Actuators are RABs that are used by the platform to change the operating parameters of the software (parameter adaptation) or modify its structure (compositional adaptation). Since the software developers are responsible for creating software that is adaptable, only they can provide a way to invoke reconfiguration.

Lastly, the adaptation profile contains a list of actions. The adaptation framework cannot know how to adapt the software and construct plans unless it is provided with actions and policies. Some generic actions are already provided by the platform. However actions related to the application logic of the software being configured are needed as well. Figure 68 shows an example of a software adaptation profile.

```
x:thisSW conf:hasPolicy x:pol1, x:pol2 .
x:pol1 rdf:type conf:Policy, conf:ProhibitionPolicy
    ; conf:hasImportance "5"
    ; conf:hasCondition {
        conf:thisSW
    } .

x:pol2 rdf:type  conf:Policy, conf:ObligationPolicy
    ; conf:hasImportance "10"
    ; conf:hasCondition {
        conf:thisSW conf:requiresMandatorilly ?capA .
        ?compX rdf:type conf:Component .
        ?compX conf:implements ?capA .
    } .

x:replaceComponent rdf:type pla:Action
    ; pla:hasVariables {
        x:varCompX pla:expressedAs "compX".
        x:varCompY pla:expressedAs "compY".
        x:varCapA pla:expressedAs "capA".
    }
    ; pla:preconditionExist {
        ?compX rdf:type conf:Component .
        ?compY rdf:type conf:Component .
        ?compX conf:provides ?capA .
        ?compY conf:provides ?capA .
        conf:thisSoftware conf:uses ?compX .
    }
    ; pla:preconditionNonexist {
        conf:thisSoftware conf:uses ?compY .
    }
    ; pla:effectAdd {
        conf:thisSoftware conf:uses ?compY .
    }
    ; pla:effectRemove {
        conf:thisSoftware conf:uses ?compX .
    } .
```

FIGURE 68 An example of a software adaptation profile

5.3 Service facilitator – trust and resource discovery

As mentioned earlier, pervasive computing environments are open, dynamic and the devices in them are controlled by different stakeholders. Each device has its own set of goals. Goals of one agent might be in conflict with goals of another one (e.g. use of a resource with exclusive access). This means that the agents have to compete against each other. However, sometimes an agent cannot reach its goals on its own. Sometimes information or action from other agents are needed, which means that the agents also have to cooperate with each other . Therefore we cannot say that pervasive computing environments are purely competitive or purely cooperative. They manifest features of both.

This situation is very similar to the world of humans, therefore we would like to use it as an analogy. The pervasive computing environment is an analogy of a marketplace and agents/devices are an analogy of humans. In the marketplace, each human has its own goals, but sometimes people need help from each other in order to achieve their goal. The question arises: Why would one human help another? In the case of pervasive computing environments the analogous question is: Why would one device (agent) help another one?

5.3.1 Approach

There have been many approaches trying to explain human behavior in such scenarios, many coming from the area of economy. Some authors see humans as selfish entities working only for their own benefit (Arrow, 1980). This may lead to a deceiving behavior such as lying, cheating and stealing (Williamson, 1985). Other authors however believe that humans very often behave according to the principle of reciprocity (Fehr and Gächter, 2000). Reciprocity means that friendly actions will be rewarded with friendly behavior and similarly hostile actions will be rewarded with unfriendly behavior.

Since pervasive computing environments are by definition unregulated and open, agents must count with the possibility of being cheated and lied to. In order for reciprocity principle to work, there must be some way of evaluating actions for each agent in the system. In other words, one must distinguish lies from truth and act accordingly. There are two basic terms in the area of inter-agent trust research – trust and reputation. Since trust has several meanings in the area of Artificial Intelligence, first we would like to define this term. Barber and Kim (2001) define trust as "confidence in the ability and intention of an information source to deliver correct information". We suggest a slightly modified version of the definition:

> Trust is the confidence in the ability and intention of an information source to deliver correct information or perform the promised action

Barber et al. (2003) define reputation as "the measurement of the amount of trust one agent holds for another". For the rest of this publication, we will use these two definitions.

Fullam et al. (2005) argue that there are two capabilities that an agent should have in order to select the most trustworthy partner. Firstly, it must maintain a model of trustworthiness of potential partners and secondly, the trust-based decisions should be based on those models. In pervasive computing environments it is inherently impossible to impose some central authority responsible for reputation management and the only option is a distributed approach.

There are two types of trust modeling – interaction-based (also called experience-based) and recommendation-based. The interaction-based trust modeling relies on building the reputation based on direct interaction with the agents being assessed (Jonker and Treur, 1999). Every action of the trustee is evaluated and assessed from several points of view based on the criteria (components of the trust). This information is then used by the truster in its decision making. This is not a problem when a trust baseline for the trustee has been established. However, a newcoming agent is not yet known and therefore its reputation is unknown as well. How can the truster find it out? One way is to interact with the newcoming trustee and discover the trust associated with it. In order to do this, the new trustee must have a certain default reputation. Then, based on its actions, the reputation increases or decreases in corresponding components. Depending on the initial reputation, this process can be very turbulent in the initial stage of trust discovery. In the later stages, when the baseline is established, the changes in the reputation are not so significant. While the truster is discovering the trust level of the trustee, it also has to keep on performing its tasks. While performing the tasks, the initial stage of newcomer's trust discovery can significantly influence the decisions made by the truster (Barber et al., 2003).

An example of this situation is described in Figure 69. The horizontal axis represents the time and the vertical access represents the trust as a single number between 0 and 1. Agents A_1 and A_2 have been evaluated by the truster before and their value trust is relatively stable. However, at time t_1 a new agent A_3 appears and the trust evaluation algorithm assigns the newcomer trust value of 0.5. This trust value stays stable until the first evaluation round is finished at time t_2, when the value changes to 0.3. Agent A_3 then keeps the value close to 0.25 for the rest of the life. The problem arises when the truster tries to evaluate the trust of all three agents and establish a ranking. In the t_1-t_2 interval the ranking from the best to the worst would be A_3,A_1,A_2. The agent A_3 wins only because the default value of a newcoming agent is 0.5, which in this particular case is higher than the other two agents. However, we can clearly see that after the first evaluation of A_3 (time t_2), agent A_3 drops to 0.3, which is significantly less than other two agents. One of the solutions to this problem is to establish a evaluation period after a new agent enters the capability space of the truster. During this period the truster would collect the information about the new trustee, but it would not use it in the decision process. The evaluation period may expire after the first, second or n-th evaluation of the new trustee. Then the truster behaves normally.

The second problem with interaction-based trust modeling is the fact that it does not work well in cases where there are not many interactions with the trustee or these interactions are sporadic . In those cases the second trust modeling method

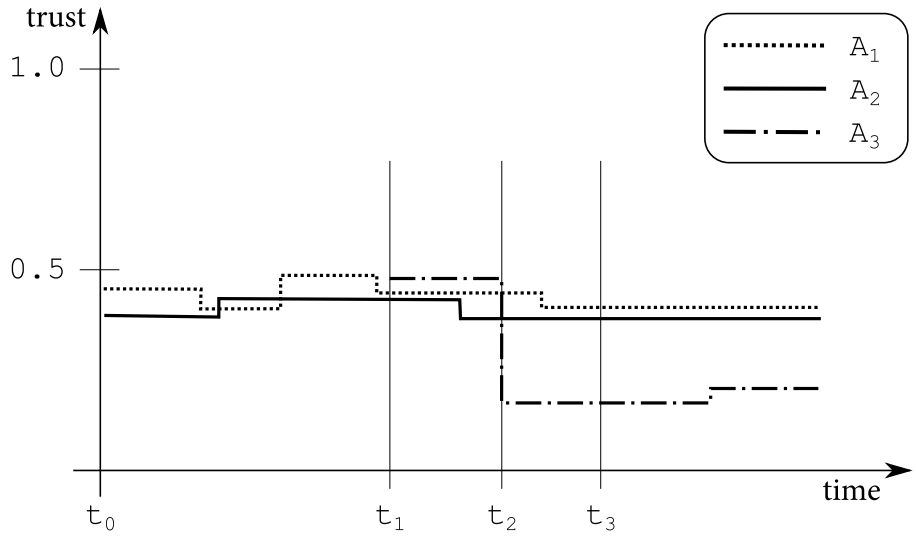


FIGURE 69 The trust development of a newly discovered agent

might be more beneficial. In the recommendation-based modeling method agents communicate with each other and share their experiences with other agents (Yu and Singh, 2002). This way the truster can receive testimonies about an unknown agent from so-called recommenders. Recommenders are agents that are willing and capable to share the trust information about another agent with the truster. This way the truster does not have to engage in a cooperation with an unknown agent.

However, there is still some risk associated with the information it receives from the recommender – the recommender might lie. Therefore it is necessary to assess the trustworthiness of the recommender as well. Moreover, if a new truster enters the environment, it does not know which recommenders it can trust. Similarly, if a new agent joins the system, no recommender knows it and therefore its reputation is unknown. In both cases there has to be some initial trust value, which brings us to a problem similar to the interaction-based trust modeling. There is one difference however. While in interaction-based trust modeling the agent evaluates only based on the trustee's answer, in recommendation-based trust modeling the truster might receive several answers from several recommenders. Even though the recommenders are unknown, this information might be used to establish a default value for trust and continue with interaction-based trust modeling with an evaluation period.

5.3.2 Composite trust metric

A simple way of expressing trust of an agent is assigning a numerical value to it, where 0 means no trust and 1 means full trust (Sen and Sajja, 2002) . According to Barber et al. (2003) this lacks additional description as to the nature of this trust/distrust. Another approach is to model trust as a set of components rather than just a number. One example of such an approach is a metric defined by

Falcone et al. (2002), who propose a set of 5 components – intent, competence, availability, promptness and external factors. Intent is the tendency of an agent to behave honestly. Competence reflects the ability of the agent to perform the desired action or provide the information. Availability measures how free the agent is from other commitments that limit its ability to provide the information or perform an action. Promptness expresses the speed of agent's actions. External factors reflect agent's susceptibility to other uncontrollable factors affecting its performance. They argue that several components give the requesting agent the possibility to find the most suitable performing agent according to the requesting agent's goals. For example, if one is looking for a quick answer, one prioritizes agents with the reputation of being quick, rather than precise. In the context of a single evaluation, the agent being evaluated is called *a trustee* and the agent that evaluates is called *a truster*. . Naturally, in a multi-agent scenario, every agent is a truster and a trustee at the same time, since it evaluates and is evaluated by others.

When one agent asks for an action or information from another agent, there is a risk of failure. The problem is that for an observer it is difficult to distinguish between an agent that failed due to its incapability and an agent that decided not to honor the agreement for its own benefit. In other words, it is difficult to tell apart a deceiving and incapable agent. In some environments this difference can be crucial. If the multi-agent system permits it, an agent with malicious intents can be punished and an incapable agent can just be notified that it failed. This is one of the arguments for a composite trust metric. We argue that this difference is irrelevant. The reason is that in the case of pervasive computing environments it would be difficult to enforce the punishment due to the fact that there is no central authority . Therefore we do not distinguish between these two. We also believe that availability as a metrics component is not necessary, because a poor time management ability is just a special case of incompetence. It is irrelevant to the agent if a task was not performed because the performer was unable to perform it or because it did not calculate the time properly.

Every agent provides certain capabilities and therefore agent's performance and trust should be evaluated with respect to those. An agent can have a capability to provide a certain service with high quality. The same agent can also have a different capability providing a lower quality service. Therefore an agent should not be judged as a whole. Instead, agent's capabilities should be evaluated. The reason for this approach is that agents are primarily interested in information and tasks to be performed. They are looking for the best tradeoff between quality, speed, cost, etc. of an action or information. They are not looking for the best agent necessarily. If one is looking for the fastest runner, one would choose a person with an outstanding running ability, even if it should mean that this person is very bad at all the other things.

A typical service provisioning scenario is shown in Figure 70. The provider agent advertises a certain set of capabilities. The consumer is able to utilize these capabilities in form of external components. The evaluation is happening on the side of the consumer in the service facilitator (SF). SF is responsible for assigning trust value to every external component. External component is only an

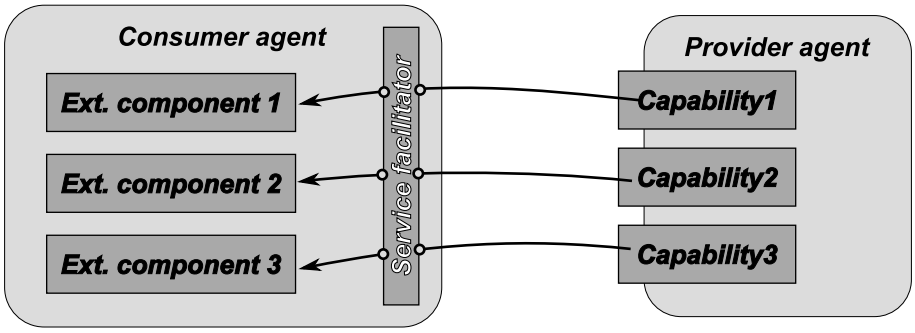


FIGURE 70 Provider-consumer relationship in pervasive computing environments

encapsulation of the capability advertised by the provider.

With respect to the nature of pervasive computing environments, we believe that for the reasons stated above a composite metric is the best solution. However, we do not believe that all five elements as stated by Falcone et al. (2002) are needed. Intent, competence and availability can be integrated into one component – quality. The quality component is a real number between 0 and 1, where 0 represents no quality and 1 represents flawless quality. In the case of an information query, the quality parameter only reflects the correctness of the result. In the case of a task request, it represents the success of the tasks. The quality has both lower and upper bounds, because there is no worse quality than no quality (value 0) and there is no better quality than flawless quality (value 1). This means that based on the value of the quality we may compare different capabilities of the same agent or different agents. If agent A has a capability C_i with quality $Q(A, C_i)$ and a capability C_j with quality $Q(A, C_j)$ and $Q(A, C_i) < Q(A, C_j)$, then we can say that A performs tasks related to C_j better than tasks related to C_i.

The response time is characterized by promptness. The numerical representation of promptness is different in nature than the numerical representation of quality. As mentioned above, the quality has both upper and lower bounds. In the case of reaction time, we know for sure that no action will last shorter or equal to 0 units of time. Thus reaction time has a lower bound. However, for any given reaction time one can always find a longer time. In other words, the reaction time has no upper bound. The trust metric is related to a capability and not to the whole agent. The reaction time differs with the capability, because some actions take naturally longer than others. Being able to run 400 meters in 45 seconds is considered a very good time. Whereas being able to run 100 meters in the same time is considered a poor result. We may compare swiftness only for a given discipline (capability). Formally, if several agents Ai (i=1..n) are providing the same capability C_j, their reaction time $T(A_i, C_j)$ is comparable for all n agents. The fact that $T(A, C_i) < T(A, C_j)$ for $i \neq j$ only tells us that agent A performs C_i faster than it performs C_j. This however, does not tell us if time $T(A, C_i)$ is a good time with respect to the capability C_i. Having this in mind, we believe that the most practical way of expressing the promptness is in form of reaction time. Therefore, the promptness is represented by a non-negative integer.

TABLE 11 Summary of trust metric components

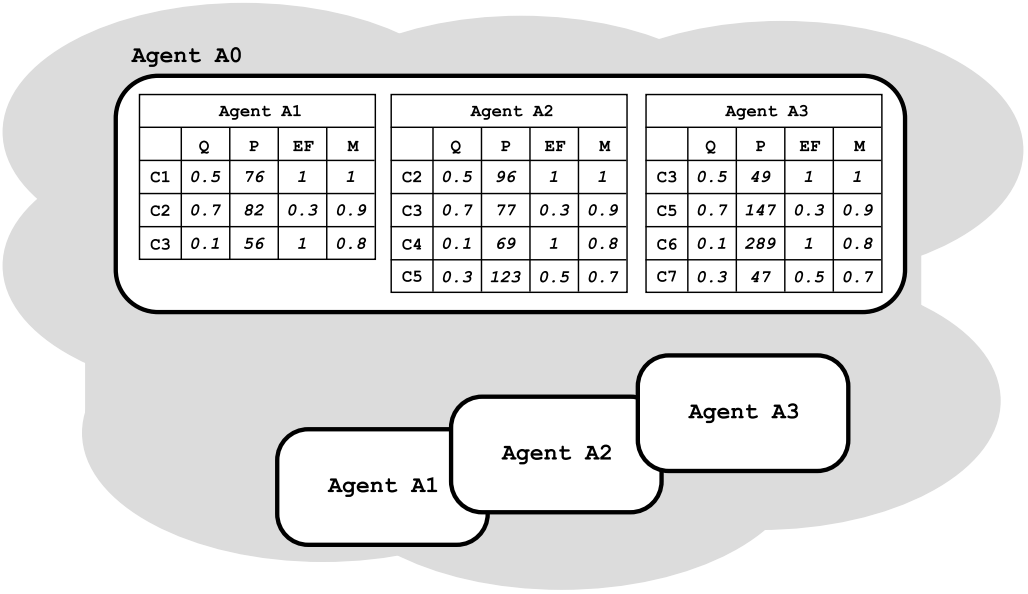
Name	Definition	Interval	Better
quality	A real number representing the degree of quality of the offered service	$\langle 0, 1 \rangle$	+
promptness	A real number representing the response time of the offered service	$(0, \infty)$	-
ext factor	All other factors that are not in the power of the provider or consumer	$\langle 0, 1 \rangle$	+
misjudgment	A real number representing how much the provider overstates or understates	$(-\infty, \infty)$	-

The third component of the composite metric is "external factors". This parameter incorporates all other factors that are not in the power of the performer or the requestor. This may include faulty network connection, HW errors, power outage, etc.

Apart from these three components, we suggest one more – misjudgment. Since some interactions are based on a contract, the estimations stated in the contract might differ from the parameters achieved in real life. The misjudgment reflects how much the agent overstates or understates. Table 11 contains a summary of all metrics components.

Based on the composite metrics, an entry for a particular agent A can be visualized as a table where rows are capabilities and columns are components of the composite evaluation metrics. Figure 71 depicts agent A_0 and its trust database for agents A_1, A_2 and A_3, where each table represents one agent entry. One way of implementing the trust database is by using a single table where each row represents a unique agent-capability combination.

In the case of the example in Figure 71, agent A_0 would choose its transaction partners in the following way. Let us assume that A_0 has only three agents in its environment – agents A_1, A_2 and A_3. Each of these agents is known to A_0 and thus each of them has an entry in A_0's trust table. Agent A_0 is looking for the most suitable agent with capability C_3. It tries to discover which agents can provide this service. The answer in this case is all of them. Then it looks for the criteria of the choice. A_0 is mostly interested in quality. Promptness is secondary. It also does not want any disruptions caused by external factors. The process of capability selection is described later in the text. For now we just assume that A_0 chooses in the following order of priority – firstly quality together with external factors and secondly promptness. Agent A_1 provides a very poor quality service, because $Q(A_1, C_3) = 0.1$ and due to external factors it succeeds in only 80% of cases ($EF(A_1, C_3) = 0.8$). Agent A_2 provides the highest quality $Q(A_2, C_3) = 0.7$ among the agents and also fairly good external factor score $EF(A_2, C_3) = 0.7$. Agent A_3 provides quality $Q(A_3, C_3) = 0.5$ and it has the best possible external factor score of $EF(A_3, C_3) = 1$. Based on the conditions, A_2 is the clear winner. If



Agent A1				
	Q	P	EF	M
C1	0.5	76	1	1
C2	0.7	82	0.3	0.9
C3	0.1	56	1	0.8

Agent A2				
	Q	P	EF	M
C2	0.5	96	1	1
C3	0.7	77	0.3	0.9
C4	0.1	69	1	0.8
C5	0.3	123	0.5	0.7

Agent A3				
	Q	P	EF	M
C3	0.5	49	1	1
C5	0.7	147	0.3	0.9
C6	0.1	289	1	0.8
C7	0.3	47	0.5	0.7

FIGURE 71 Trust evaluation table

the condition would be that there should be no external factor influence at all, the winner would be A_3.

As mentioned earlier, the framework keeps track of all the metric components for every software component. The metric values are stored in the knowledge base as properties of the software component description resource. Figure 72 shows an example of a relatively complicated utility function that determines the best face recognition component. Let us assume that `x:QMul` is a quaternary multiplication function (multiplication with arity 4). Also, let us assume that `x:Threshold` is a binary function with arguments v and t that returns 0 except for $f(v > t) = v$. The utility function `x:bestFace` helps find the best face recognition component according to this function (f_T is the threshold function):

$$F_0(q, p, m, ef) = f_T(q, 0.8) \cdot \frac{1000}{p} \cdot \frac{3}{m} \cdot ef$$

5.3.3 Reputation building process

Based on the discussion above, we suggest the following algorithm for finding the reputation of an unknown agent (Figure 73). We assume that this algorithm is used by the truster A_0 to find out the reputation of an unknown agent Ax. We also assume that A_0 knows exactly m agents that can act as potential recommenders. The algorithm starts with agent A_0 sending reputation request to m potential recommenders. Then agent A_0 waits P_{wt} units of time (e.g. milliseconds) for the answer. We assume that out of m potential recommenders exactly n will answer ($0 \leq n \leq m$). If nobody answered ($n = 0$), then the agent has to revert to interaction-based trust evaluation with an evaluation period, which will be described later in this section. If at least one agent answered, then we divide the

```
x:bestFace rdf:type u:UtilityFunction
   ; u:hasQuery {
      ?fc rdf:type conf:Component ;
         conf:provides x:capabilityFaceRecognition ;
         conf:providedByAgent ?ag ;
         conf:hasQuality ?q ;
         conf:hasPromptness ?p ;
         conf:hasMisjudgment ?m ;
         conf:hasExternalFactor ?ef .
   }
   ; u:usesEvalElement "fc"
   ; u:hasFunction x:f0 .

x:f0 rdf:type u:Function, x:QMul
   ; u:op1 [ rdf:type x:Threshold
            ; u:opV [ rdf:type u:Variable ; u:hasVarName "q" ]
            ; u:opT [ rdf:type u:Value ; u:hasValue "0.8" ]
 ]
   ; u:op2 [ rdf:type u:Div
            ; u:op1 [ rdf:type u:Value ; u:hasValue "1000" ]
            ; u:op2 [ rdf:type u:Variable ; u:hasVarName "p" ]
           ]
   ; u:op3 [ rdf:type u:Div
            ; u:op1 [ rdf:type u:Value ; u:hasValue "3" ]
            ; u:op2 [ rdf:type u:Variable ; u:hasVarName "m" ]
           ]
   ; u:op4 [ rdf:type u:Variable ; u:hasVarName "ef" ] .
```

FIGURE 72 Example of a utility function for best facial recognition component selection

group of n agents that answered into two disjunctive groups – n_k known agents and n_u unknown agents ($n = n_u + n_k$). The group of known agents contains n_k agents that are known to A_0 and thus A_0 knows their reputation. The group of unknown agents consists of n_u agents that do not have any reputation record in A_0's knowledge base. The agent calculates the weighted arithmetic mean for the group of known agents $\bar{x}_k$ and for the group of unknown agents $\bar{x}_u$ using the following formula:

$$\bar{x} = \sum_{i=1}^{n} w_i' x_i = \sum_{i=1}^{n} \frac{w_i}{\sum_{j=1}^{n} w_j} x_i = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{j=1}^{n} w_j} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

These two steps may happen in parallel. After values $\bar{x}_k$ and $\bar{x}_u$ are obtained, the agent calculates the total initial trust of agent Ax using the following formula:

$$T = \frac{n_k}{n} \bar{x_k} + P_{uk} \frac{n_u}{n} \bar{x_u}$$

Then the agent switches into interaction-based mode with no evaluation period and it uses T as the initial trust value for Ax. Please notice that the algorithm uses two parameters – P_{uk} and P_{wt}. As mentioned earlier, P_{wt} determines the wait time for the recommenders to respond. P_{uk} is a number between 0 and 1 that determines how relevant the unknown agents' responses are. Value 1 means that they are equally important to the known agents' responses and value 0 means that they are not important at all.

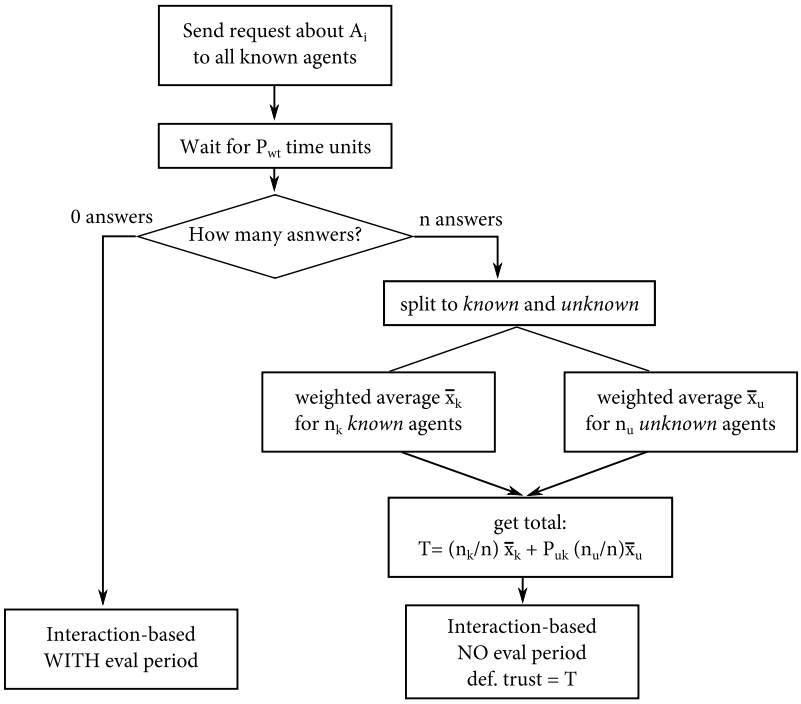


FIGURE 73 Recommendation-based trust modeling

5.4 SAF processing elements

5.4.1 Knowledge base

The system should be capable of storing different types of information about neighboring devices, goals, user preferences, etc. Some of this data is relatively static (e.g. own preferences) and some data is dynamic (e.g. information about neighbors). However, we believe that a single way of representation is favorable due to the need to integrate and reason about all these kinds of data together. Reasoning will take place mostly in two cases – when a message is received and when a message is sent.

We need a way of knowledge modeling that would be expressive enough to describe relationships between heterogeneous entities such as customers, products, services, channels, etc. Moreover, the language must allow reasoning about the facts. This reasoning must be sound, so that it infers only valid facts. It should be complete as well, so that no valid fact is missing. If a valid missing was missing, it would result in a failure to send an outbound message or to properly annotate an inbound message. Lastly, this modeling framework must not be computationally too complex. As mentioned earlier, the discussed system should be usable in real-world production environment. Therefore the existence of a mature tool with abovementioned properties is important.

Based on these requirements we believe that Resource Description Framework (RDF) is the most reasonable way to model our data. It is expressive enough,

```
x:measurement45356 rdf:type sen:SensorResult, x:TempResult
                 ; sen:readFrom x:tempSensor
                 ; sen:value "25"^^xsd:integer .

x:measurement45356 rdf:type sen:SensorResult, x:TempResult
                 ; sen:readFrom x:tempSensor
                 ; sen:hasEvent x:ev32532 .

x:ev32532 rdf:type sen:SensorEvent, x:FaultEvent
        ; x:source x:airConditioning
        ; x:severity x:medium .
```

FIGURE 74 Sample of sensory data

proven and widely supported by a great amount of production-quality tools for modeling (e.g. Protégé (Standford, 2013)), storing and querying data (e.g. Jena framework (Apache, 2013), Sesame (Aduna, 2012)). Also, there are many reasoners that are sound, complete and still computationally inexpensive (e.g. RacerPro (RacerSystems, 2013), HermiT (Hermit, 2013), etc.). RDF is built on top of wide-spread standards such as XML, XML Schema, etc. It is closely related to Web Ontology Language (OWL) which is used to formally define knowledge schemas called ontologies. Thanks to ontologies and reasoners it is possible not only to conclude new facts from existing ones, but to check data consistency as well.

5.4.2 Monitor

As the name suggests, the role of this component is to monitor the contextual data and report any changes. In the software adaptation profile, the application developer specifies which sensory data is considered relevant and thus a part of the context. The monitor then collects this data in form of RDF and stores it in the agent's knowledge base. After the collection, the detector is notified. A sample of collected data is shown in Figure 74. There is a reading from a measurement sensor measuring the ambient temperature. In that case the result is a value. The second example comes from an event sensor and therefore the result is an event. In this case the sensor reads data from the car's Electronic control unit indicating that the air-conditioning is faulty.

5.4.3 Detector

The detector is responsible for detection of incidents. Incidents are changes in sensed parameters and events that may influence the operation of the software. Incident classification is based on software's adaptation profile, where software developer indicates undesired events and optimal operation parameters' values. Once these values are crossed or undesired events are detected, the detector contacts the deliberator.

The incident classification is implemented using safeguards introduced earlier. A safeguard contains a condition that determines the normal working state

```
x:inc5 rdf:type inc:EnvironmentIncident, inc:PhysicalIncident
     ; inc:brokenPolicy x:policyHighTemp
     ; inc:responsibleTicket x:tic72
     ; inc:responsibleSensorResult x:measurement454 .

x:tic72 rdf:type sfg:Ticket, sfg:TicketDET
     ; sfg:inState sfg:stateDET
     ; sfg:handlingSFG x:safeTemp .
```

FIGURE 75 Example of an incident

(in the case of existence safeguards) or the abnormal working state (nonexistence safeguards). Whenever a safeguard is broken, a ticket is created. In the software's adaptation profile, it is specified which condition is related to which incident. For example the profile might specify that the permissible ambient temperature for a smartphone camera flash is between $-20\,^{\circ}\mathrm{C}$ and $+60\,^{\circ}\mathrm{C}$ and whenever it outside this range, the incident is classified as a physical environment incident. A condition might also be more complicated – e.g. a combination of several measurement values and/or events. Once a safeguard breach is found, a ticket is created automatically. This ticket then acts as an accompanying resource to the incident object. A sample incident description together with the accompanying ticket can be found in Figure 75.

5.4.4 Deliberator

The goal of this component is to decide what should happen in order to deal with the incident. First, the deliberator consults the database of existing plans and tries to find a plan that can make the system return to the normal state. If several plans are found, the best one is chosen. If no plan is found, the planner is contacted.

5.4.5 Planner

The planner tries to find a plan based on the goals specified in the software adaptation profile. It is not our goal to provide a new planning algorithm, due to the fact that this domain is outside the scope of this publication. Also, we do not implement a planner using S-APL language, because the language is not suitable for such a task. Principally, it could be possible to implement a planner using S-APL, but the implementation would be cumbersome and it would bring no extra benefits in comparison to already existing planners. Instead of that, we provide a method to convert the problem into the Planning Domain Definition Language (PDDL). There are a number of various planners capable of reading PDDL problem specifications and provide a solution. Such a solution can be converted back to a plan.

Every hard policy (level 10) has to be considered when preparing the problem definition file. For every obligatory hard policy, the condition in its original form become a part of the goal. For every prohibition hard policy, the negated condition

is added to the goal section. This way the system ensures that the plan will never break a hard policy.

If no plan is found and the incident was caused by a soft policy, then the system leaves both the incident and its ticket in KB. This is due to the fact that at some point in the future some sensory data may come, which may help resolve the incident.

5.4.6 Plan executor

This component is responsible for the proper plan execution. It coordinates actions specified in the plan with the involved components. For every action in the plan, it contacts the action executor. When the action is performed, the plan executor determines the next step to be performed. In case some action fails, the plan is considered failed as well. In those cases, the deliberator takes over and treats the situation as a new incident.

5.4.7 Action executor

The action executor receives an action from the plan executor. It performs the action using the specified actuators and action parameter values. Both in the case of a success and failure, it reports back to the plan executor. If the action causes a component replacement or addition, new data might be added or removed. Each component may have an initialization and finalization (clean-up) script.

6 SMART HOSPITAL SCENARIO

The first goal of this chapter is to describe an example of a case, where the Ubiware-based SAF helps achieve both the vision of pervasive computing and decrease the perceived complexity of software used in it. We have chosen the area of healthcare, because it has relatively medium complexity, dynamicity and openness. One seemingly simple scenario with two situations is provided. The SAF adaptation process is described in detail. Note that many parts of the text will refer to S-APL code fragments. Due to limited space we provide most of them as appendices. The second goal is to contrast between the new, Ubiware-based SAF and the old Ubiware implementation without SAF.

Since SAF is a conceptual framework, it can be implemented using any combination of programming languages and platforms assuming that they provide the required features. This chapter deals with a Ubiware-based implementation of SAF introduced in Chapter 5 and therefore when referring to it we will use the term *Ubi-SAF*. Note that Ubi-SAF contains the improvements from Chapter 4, since they are required for the implementation. The old Ubiware implementation without SAF or any of the three improvements will be shortly called *Ubi-0* and it will act as the reference point. For each situation, we provide a comparison between *Ubi-0* and *Ubi-SAF*.

We show that Ubi-SAF provides three types of improvements over Ubi-0. In the order of increasing significance they are as follows. Firstly, Ubi-SAF introduces some simplifications of the S-APL code (I_1). This improves the readability and helps the developer focus on the problem at hand. Secondly, there are cases where both Ubi-SAF and Ubi-0 are capable of performing an action, but Ubi-SAF outperforms Ubi-0 (I_2). Lastly and most importantly we will show cases where Ubi-SAF possesses abilities to solve certain types of problems that Ubi-0 cannot solve (I_3). Table 12 summarizes the improvement types.

TABLE 12 Types of improvements that Ubi-SAF introduces

ID	Improvement	Significance
I_1	S-APL code simplification	+
I_2	Ubi-SAF outperforms Ubi-0	++
I_3	Ubi-SAF can solve problems that Ubi-0 cannot solve	+++

6.1 Case description

The case takes place in a fictional hospital in a developed country. The year is 2025. Computing devices have become an everyday part of people's lives. Over 90% of people own a smartphone. A vast majority of mobile devices has a constant access to the Internet. The RFID technology is being used in the supply chain management, hospitals, homes, etc. Some of the people already have sub-dermal chips. Smart spaces are becoming more prevalent.

6.1.1 Human actors

As mentioned earlier, ubiquitous or pervasive computing is about humans and their relationship to the technology. Therefore we first introduce the human actors. Human actors are described using the concept of personas. Personas are widely used in the field of interaction design to function as certain representatives of the target group . They provide a unified view of the users and help make the product design more customer-focused (Preece et al., 2011). There are two personas in this scenario.

Anna (35) has been a nurse for over 15 years. She works for the University Hospital at the Intensive Care Unit (ICU). She is very thorough in her work and she does not like surprises. She likes to check on her patients often to make sure that everything is fine. Anna has a high sense of responsibility. Charles (63) is one of Anna's patients.

Several hours ago Charles was hit by a car when he was trying to cross the road. When the ambulance came to the crash site, the crew administered first aid. As a part of the procedure, he received a GPS- and RFID-enabled wristband that is constantly checking his vital signs. The ambulance took him to the Emergency Room (ER), where they discovered internal bleeding in the brain. He was immediately taken to the operating room, where they conducted surgery to release the pressure in his head. After the operation, Charles was taken to the ICU, where Anna started to take care of him. Charles was in an artificially induced coma and he was connected to a ventilator that was fully breathing for him. His bed with all the sensors was put into the cubical number 2 in the ICU ward.

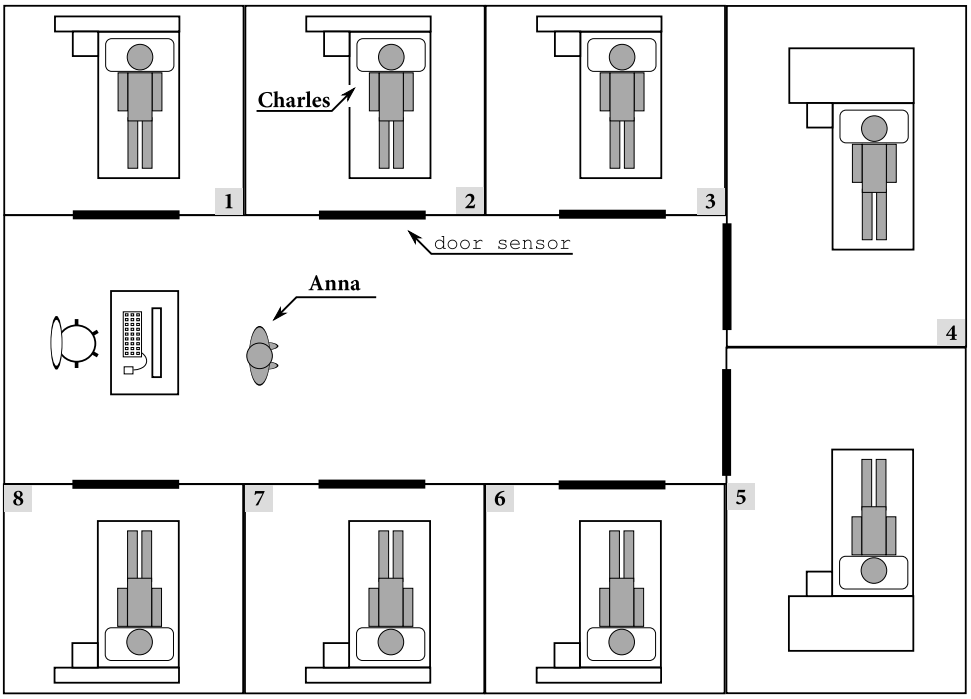


FIGURE 76 The spatial setting of the ICU ward

6.1.2 Spatial setting

The scenario will take place in a hospital, in the ICU. The ICU in our case is one large room with a nurse station in the center. Around the station there are 8 cubicals separated by a soundproof plexiglass that can be opened and closed. Each cubical contains at most one patient. This way the nurse can see all the patients from one place. The station has a large screen with all patients' vital signs displayed on it. Figure 76 shows the spatial setting.

6.1.3 Devices

Each hospital employee not having a sub-dermal chip is wearing a bracelet with an RFID chip, GPS and a low-powered wireless network interface card. Each patient is wearing a bracelet with all the previous mentioned functionalities, which also measures patient's vital signs. The hospital personnel is allowed to carry personal phones if their phones follow the hospital's policies. Often, nurses and doctors carry a tablet-like device with them when visiting patients. Tablets are mostly used to present patient's data. The default policy for hospital's tablets is to connect to the nearest patient information source in case only one patient is located within 2 meters from the device. If more patients are present, then the nurse has to choose one of them. Moreover, every patient's bed has a controller that is connected to various sensors and collects all the sensory data related to patient's health. Finally, each room has a room controller, a small UI-less computer that takes care of the room's sensors and actuators.

6.2 Scenario: Anna checking on Charles

Anna's shift just started. As one of the first, she starts to check on patients by visiting their cubicals. Each cubical has a motion sensor and an RFID sensor. Upon entering the cubical, the blinds on both sides close to provide privacy to the patient. The light turns on slightly to illuminate the room. Anna's tablet detects the presence of a patient in the room. According to the roster, she is currently taking care of him. The tablet automatically shows the patient's information depending on Anna's profile. In this scenario there are three devices (Anna's tablet, Charles' bed, room controller) and two humans (Anna, Charles).

6.2.1 Configurations

6.2.1.1 Room controller configuration

The room controller is running a control software on top of SAF. It does not have any user interface. The controller has an RFID scanner (`c2:senRFIDDoor`) around the door frame. Every time somebody enters the room, the scanner can pick up the RFID code belonging to the person. The scanner is a push sensor (of type `sen:PushSensor`). It means that every time it detects an RFID tag inside the door, it notifies the controller by sending an event of type `m:DoorEvent`, which contains the RFID identifier.

Also, the device has a soft policy (`c2:pol1`) saying that at all circumstances it should know the person being inside the room with the patient. It however does not say how the device should proceed once this is not true. The policy only describes a desire or undesired state. It is up to the device to find the right solution to the problem. Both the policy and sensors' descriptions are a part of the software configuration. For a full description in S-APL see Appendix 4.2.

Currently, the software and platform configuration offer 3 actions. Two are generic and can be used platform-wide – initialize component (`m:initComp`) and find component (`m:findComp`). One action is specific to the application logic – find human (`m:findHuman`). As the name suggests, the component finder is capable of finding a component implementing a certain capability (Appendix 4.2, line 33). The action is implemented as a service facilitator call. When a component is found, the information about it is stored into the knowledge base. The component initializator action performs a compositional reconfiguration of the software. A specific unused component is initialized and as a result the platform starts to use it (Appendix 4.2, line 50). Finally, the human finder action performs a query in the database of people trying to find the identity of a person based on an RFID identifier (Appendix 4.2, line 14).

6.2.1.2 Tablet configuration

The tablet belongs to the hospital and nurses use it to visualize patient's information. The tablet has an RFID sensor (`m:bedSensor`) that of type `sen:PushSensor`.

```
c2:ev532 rdf:type sen:SensorEvent, m:DoorEvent
      ; sen:hasValue "354186465463"
      ; sen:hasTimestamp "2013-09-10T08:15:50.547Z"^^xsd:dateTimeStamp
      ; sen:readFrom c2:senRFIDDoor
      ; m:source c2:door .
sapl:I m:cannotIdentifyHuman "354186465463" .
```

FIGURE 77 Example of a door event

It is a more sophisticated sensor than the door RFID reader from the previous example. Whenever a bed is detected next to the device, an event with a bed URI is generated.

The software structural profile specifies that the tablet optionally requires a capability that allows patient's monitoring (`m:capPatientData`). The adaptation profile contains no plan and only one action `sw:attachBed` that can provide the patient's monitor component if the bed's URI is provided. The tablet has one soft obligation policy `tab:polA`, which specifies that it should always be connected to a patient. The configuration can be found in Appendix 4.3.

6.2.2 Situation 1: Identification of Anna

6.2.2.1 Initial state

We assume the following initial state:

1. There is only one person in the room – Charles
2. The room controller is aware of Charles
3. The room controller does not have a connection to any database of people

6.2.2.2 Chain of events

Anna enters the room wearing her RFID bracelet. The door RFID scanner picks up the code. Since the software is configured to listen to this sensor, the information is caught by the SAF monitor and stored into the knowledge base as a new event. Also, the sensor is configured so that also a special belief `sapl:I m:cannotIdentifyHuman <ID>` is inserted into KB whenever a previously unencountered person is met. This is the first time Anna is entering the room, therefore the belief is inserted. Figure 77 shows the event with the special belief. The SAF detector detects that the policy `c2:pol1` is broken, because an unknown person entered the room. An incident with the corresponding ticket is created.

The SAF deliberator is contacted to resolve the situation. The first step is to look into the knowledge base and find a suitable plan or action. Since there is no plan and only 3 actions, which are unsuitable, the deliberator has to contact the SAF planner. Note that findHuman action is unusable, because it requires the software to have a human database component.

The SAF planner starts with the conversion from S-APL into PDDL. In order

TABLE 13 Translation table for actions

Predicate	V_1	V_2	Type	S-APL query
implements	?x	?y	VNV	?x conf:implements ?y
humanDBComp	?x		VNN	?x conf:implements m:capHumanDB
isComp	?x		VNN	?x rdf:type conf:Component
using	?x		NNV	m:thisSW conf:currentlyUtilizes ?x
unknown	?x		NNV	sapl:I m:cannotIdentifyHuman ?x

```
  (:action findComp
     :parameters (?comp ?cap )
     :precondition  (and
   (not (isComp ?comp))
   (not (implements ?comp ?cap))
)
     :effect  (and
   (isComp ?comp)
   (implements ?comp ?cap)
   (humanDBComp ?comp)
     )
  )
```

FIGURE 78 PDDL version of the find component action

to transform the actions, the predicates must be converted. The conversion table is shown in Table 13. For convenience, the table shows predicate names that resemble the meaning of the original S-APL query statement. In reality the predicate names would be converted into a seemingly random string with no meaning. Once the conversion is finalized, it is crucial to mark which binary predicates expand unary predicates. In this particular case the implements predicate expands humanDBComp, which is consistent with the expansion graph introduced earlier in Figure 57. This means that if there is an action with the implements predicate in the effect, also the humanDBComp predicate should be included.

Once the predicate conversions are known, an action can be constructed. There are 3 actions in the configuration. Therefore there will be 3 actions in the PDDL domain description file. Figure 78 shows the findComp action together with the effect expansion. The incident indicates that the policy was broken. Upon examining the condition of the policy, the SAF planner finds out that the conflicting statement is `sapl:I sw:cannotIdentifyHuman "354186465463"`. There is no hard policy in the system, therefore the problem file indicates that the desired goal is `not (unknown 354186465463)`.

The planner runs the external planning algorithm with both the problem and the domain PDDL file as the input. In general, the planner can provide several plans. In this particular case, only one plan is provided. The visual representation of the plan is shown in Figure 79. It is possible to see the expansion in the findComp

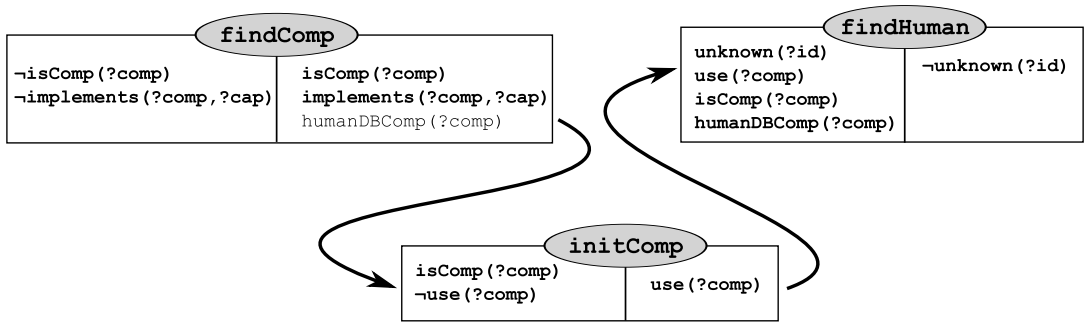


FIGURE 79 Visual representation of a plan

action. The plan is converted into S-APL (see Appendix 4.1) and delivered to the SAF plan executor.

For each step in the plan, the SAF executor contacts the SAF action executor with a single action to perform (e.g. a RAB call). Upon a successful action completion, the next step action is performed. This repeats until the goal has been reached. Once the goal has been reached, the incident with its ticket is removed from KB. In this particular case the first step was to find a human database component. The second step was to initialize the found component. Lastly, the component was used to find out who entered the room. The plan was saved into KB. The next time Anna enters the room, no safeguard will be broken, because the person-ID mapping is already present in KB. If some other unknown person enters the room, the safeguard will be broken, which will cause an incident. However, this time the planner will be skipped, because the deliberator will find a suitable plan in KB.

6.2.2.3 Scenario modification

In some cases a certain combination of sensory inputs within the given time frame may cause an incident. Figure 80 shows how such a condition could be modeled. The condition shows two door events for the same door. Event A comes from the RFID door sensor and event B is sent from the motion sensor indicating movement in the door area. The condition also specifies that the event B occurred at most 1 second after event A (lines 14-16). Finally, the last condition says that the human cannot be identified.

6.2.2.4 Comparison between Ubi-0 and Ubi-SAF

There are two steps (moments) in the deliberation process, where Ubi-SAF improves Ubi-0. The first step is the moment when an unknown person entered the room and the detector detected that the policy `c2:pol1` was broken. As described earlier, Ubi-0 contains a policy system for RAB-related actions. However, it does not have a policy system for agent's internal beliefs. In Ubi-SAF this is achieved using safeguards. Whenever a safeguard is broken, a precisely specified series of events happens. As an indication of a broken policy, a ticket is created and it goes through five states until it is resolved. A similar approach could be implemented

```
1  // two events detected for the same door
2  ?eventA rdf:type m:DoorEvent
3        ; sen:hasValue ?RFIDValue
4        ; sen:hasTimestamp ?timeA
5        ; sen:readFrom c1:senRFIDDoor
6        ; m:source c1:door .
7
8  ?eventB rdf:type m:DoorEvent
9        ; sen:hasTimestamp ?timeB
10       ; sen:readFrom c1:senMotionRoom
11       ; m:source c1:door .
12
13 // one second apart
14 ?timeDiff sapl:expression "?timeB-?timeA" .
15 ?timeDiff < 1000 .
16 ?timeDiff > 0 .
17
18 sapl:I m:cannotIdentifyHuman ?RFIDValue .
```

FIGURE 80 Modification of the incident condition

in Ubi-0 as a series of implications. By externalizing the safeguard break detection, ticket creation and state transition process, the agent developer does not have to repeat the same set of implications for each safeguard. Instead, he/she just specifies the condition and the safeguard module (sub-system) will take care of the process. This improvement can be classified as I_1.

The second improvement can be identified in the situation when the deliberator finds out that no plan or action is available. In Ubi-SAF a planner can be contacted and together with the newly introduced ability to plan, the planner can provide a series of actions that lead to a ticket resolution. In Ubi-0 there is no ticket, planner, deliberator or any other adaptation component. All adaptation-related actions have to be implemented using implications and RABs. If some component or portion of the code in Ubi-0 detects a problem, it has to rely only on the implications to solve it. In the situation described in the scenario, Ubi-0 would fail due to its inability to plan. Therefore in this case Ubi-SAF provides an improvement of type I_3.

Apart from these two moments in the deliberation process, Ubi-SAF outperforms Ubi-0 in the following way. Ubi-SAF has the ability of semi-automatic memory management, which lowers the risk of memory overflow and improves the performance. In general, the speed of agent's deliberation is determined by the speed of S-APL implication execution. The speed of the implication execution mostly depends on the size of the knowledge base (KB) and the complexity of conditions in the left-hand side of implications. When looking at the adaptation cycle, the highest data flow (belief flow) is the one entering the monitor and detector components. This flow comprises of sensory data (beliefs). In Ubiware all the beliefs stay in KB until an explicit action to delete them is detected (`sapl:remove` or `sapl:erase`). In Ubi-0 the developer must provide these actions manually. In Ubi-SAF we provide an automatic belief cleanup mechanism for sensory data

(introduced in Section 5.1.1). Due to these improvements, the historical sensory data can be deleted either depending on the amount of stored values or depending on the age of the measurement.

Another problem of a potential memory overflow and performance impediment is in cases when a developer creates beliefs and then forgets to remove them. This is a well known problem in software engineering and it has led to the implementation of automatic garbage collection methods into various programming languages (e.g. Java, Python, C#, etc.). In Ubi-SAF, by externalizing the adaptation process, the developer does not have to perform a cleanup manually. Instead of that, the cleanup is a part of the adaptation cycle. An example of such a process can be seen in the incident lifecycle management. An incident is created by SAF, held in KB throughout the resolution process and once the incident has been solved, SAF automatically removes the incident and the corresponding ticket.

Depending on the point of view, this improvement can be seen as of type I_1, because it simplifies the development process by automating the memory management. Also, it is a type I_2 improvement, because it lowers the size of KB and thus it improves the performance of the whole system. Lastly, it can be seen as a type I_3 improvement, because it allows Ubi-SAF to partially solve the problem of memory overflow.

6.2.3 Situation 2: Anna's tablet autonomously connects to patient's bed

6.2.3.1 Initial state

We assume the following initial state:

1. Anna has just entered Charles' room
2. Anna has already been detected and identified by the room controller
3. The tablet is not connected to any bed

The policy `m:polA` (Appendix 4.3, line 12) specifies that there should always be a connection to a bed providing patient's data. However, in the initial state, there is no connection to any bed. Also, no component is used as the patient's data provider. There is no action that can help solve the issue. Therefore, in this particular case, the scenario starts with a broken soft policy. So far we have shown only cases when in the beginning the policy condition was holding and then a disrupting even arrived. Since a soft policy is broken, a ticket in PRO (in progress) or BLO (blocked) must exist. In this case, no other policy is blocking `m:polA` and therefore some ticket (let us call it `m:tA`) in the PRO state exists in the agent's (tablet's) belief storage.

6.2.3.2 Chain of events

The RFID sensor picks up Charles' bed. Since Charles is in the room alone, only his bed is picked up by the tablet. The sensory input can be seen in Figure 81. This sensor provides a more sophisticated information than the door sensor in the

```
x:ev327 rdf:type m:BedEvent
      ; sen:hasValue c2:bed
      ; sen:hasTimestamp "2013-09-10T08:15:52.239Z"^^xsd:dateTimeStamp
      ; sen:readFrom c1:senRFIDTablet .
```

FIGURE 81 Sample bed event

previous example. The event directly contains the bed URI. There is no need to look it up based on the RFID identifier.

The SAF monitor detects the event and stores it into the knowledge base. The detector does not notice any new policy inconsistency. However, since the policy `m:polA` has been broken and the `m:tA` ticket exists, the detector also checks if the newly available sensory input solves `m:tA`. In fact, the detector finds out that the new event may contribute to the solution of `m:tA`. The deliberator is contacted. This time the deliberator finds the action `sw:attachBed` that can solve the situation. Instead of contacting the planner, the plan executor is contacted. Since it is a simple action, the plan executor directly contacts the action executor. The component representing the bed is connected and the application can start to download patient's data and display it to the nurse. Note that Anna did not have to do anything. She just approached the bed and the data was displayed automatically. The same would happen if she entered another room with a different patient in it.

6.2.3.3 Comparison between Ubi-0 and Ubi-SAF

Thanks to the existence of the safeguard sub-system in Ubi-SAF, a SAF application developer can easily specify various safeguards with different importance levels. Whenever a safeguard is broken, SAF automatically creates a ticket, which is then kept in the KB until it is resolved. The developer does not have to explicitly specify the adaptation logic. He/she only specifies a set of policies, actions and plans as a part of the software configuration. SAF takes care of the problem resolution process automatically. Traditionally, in Ubi-0 one would have to use a set of implications and RABs to implement the adaptation logic that is already contained in Ubi-SAF. This new feature of Ubi-SAF can be classified as type I_1.

6.3 Conclusion

We have provided a sample scenario with two situations. In the first situation S_1 Anna enters Charles' cubical and the room controller detects her for a credential check. The second situation S_2 describes how Anna's tablet autonomously connects to Charles' bed. While S_1 requires planning to resolve the situation, in S_2 a suitable action is found and planning is completely avoided, saving resources. Another difference is that S_1 starts in a state when no policy is broken. S_2 starts with a broken policy. Thanks to a new sensory input the solution is found. Lastly, S_1 uses a prohibition policy and S_2 uses an obligation policy.

7 CONCLUSION AND DISCUSSION

Chapter 7 finalizes the thesis by providing an analysis of the related work, summary of conclusions and a discussion on future work. Firstly, the related work in the area of self-managed systems both in and out the area of pervasive computing is examined. Several systems are presented and compared to SAF in terms of capabilities. Secondly, the three research questions asked in the beginning of this dissertation are answered. Lastly, we elaborate on possible future research in this area.

7.1 Related work

This work presents a self-management middleware called SAF (Smart Adaptive Framework), first as a conceptual framework (Chapter 3) and then as a Ubiware-based implementation (Chapter 5). The related work in this section will be presented with respect to the effort described in Chapter 5, which also includes the concept of SAF (Chapter 3) and a set of Ubiware improvements (Chapter 4). This allows us to cover the whole extent of the work demonstrated in this dissertation.

7.1.1 Work that SAF is based on

In the general description of SAF we mention four main technologies that the framework is based on – multi-agent systems (MAS) (Shoham, 1993; Wooldridge, 1997), semantic technologies (Berners-Lee et al., 2001; Lassila and Adler, 2003; Lassila, 2005), an external (middleware-based) approach to adaptation (White et al., 2004) and the three-layered model for adaptation (Kramer and Magee, 2007). The first three are also related to the vision of the Global Understanding Environment (GUN) (Terziyan, 2003; Terziyan and Katasonov, 2009), which served as a motivation for the development of the Ubiware platform and the S-APL language (Katasonov et al., 2008).

In the area of MAS we base our work on the idea of Agent Oriented Program-

ming (AOP) by Shoham (1993) and Agent-Oriented Software Engineering (AOSE) by Wooldridge (1997). The Ubiware platform and thus also the Ubiware-based implementation of SAF (Ubi-SAF)utilize an approach similar to the Belief-Desire-Intention (BDI) architecture by Rao and Georgeff (1995), OMNI by Vázquez-Salceda et al. (2005) and Gaia by Wooldridge et al. (2000).

The research in the area of Semantic Web and related technologies contributed to the idea of utilizing a semantic approach to tackle the problem of interoperability in heterogeneous environments (Lassila and Adler, 2003; Lassila, 2005). Since this work deals with software engineering problems as well, the adoption of widely-used standards such as RDF (Miller and Manola, 2004), OWL (Miller and Manola, 2004) and Notation3 (Berners-Lee and Connolly, 2011) became important as well.

The work in the field of self-management architectures such as White et al. (2006) and Kramer and Magee (2007) helped us better understand the processes that an adaptive middleware has to deal with. The work by Oreizy et al. (1999) and Kephart and Chess (2003) improved our understanding of self-management cycles.

Finally, our work on various industrial projects helped us understand the real-world needs and also provided a testbed for partial solutions that were presented in this work. Namely, the work on idustrial cases within the Ubiware project showed us how various events and measurement data can be classified using S-APL and formal ontologies. Also, the work on various core components of the Ubiware platform broadened our view on its usability in self-management (Terziyan et al., 2010a). This work has been further improved in the SCOPE project. Lastly, our work within the Tivit SHOK Cloud Software program helped us develop a method for utility function description in S-APL (Nagy, 2012, 2013).

7.1.2 Qualitative comparison to other approaches

We provide a qualitative comparison between the Ubiware-based SAF (Ubi-SAF) and other self-management architectures. There is a number of surveys in the area of autonomic computing, self-managed systems, context-aware architectures and other related fields. Each of the surveys provides a different view on the topic. For example, Bradbury et al. (2004) provide a survey of formal specifications in self-management software architectures. Parashar and Hariri (2005) present an overview of mostly commercial autonomic computing solutions for different application domains such as distributed data storage management, database systems management or server farms. Kjæ r (2007) provides a survery of context-aware middleware. Note that while context-awareness is a necessary condition of self-management, it is not a sufficient condition of self-management. Salehie and Tahvildari (2009) present a relatively broad overview of research projects dealing with self-management.

Pervasive computing environments have been studied for over a decade. Roman et al. (2002) describe a meta-operating system called Gaia [1] that runs on top

[1] Do not confuse with Gaia – Analysis and design methodology for agent-based systems

of a normal operating system. The goal of the system is to provide a distributed middleware architecture for coordination of software entities in physical spaces. The middleware simplifies the resource discovery and utilization in heterogeneous environments. However, it does not fully address the issue of autonomicity (Sharmin et al., 2006). On the other hand, Ubi-SAF is based on the MAPE-K cycle and thus it features the adaptation logic necessary to achieve autonomicity.

MARKS (Middleware Adaptability for Resource Discovery, Knowledge Usability and Self-healing) is an effort to provide a middleware for pervasive computing environments (Sharmin et al., 2005, 2006). Its emphasis is on the resource discovery in open and dynamic environments using the notion of knowledge usability (Ahmed et al., 2005).

The self-healing portion of MARKS is called ETS, which stands for Efficient, Transparent, and Secured (Shameem et al., 2007). The ETS healing process consists of six steps – Fault Detection, Fault Notification, Faulty Device Isolation, Alteration, Information Distribution and Fault Healing. A fault is detected by comparing a set of historical measurement data with a predefined threshold. In Ubi-SAF a fault is called an incident and it is detected if a policy is broken. First of all, Ubi-SAF supports events as well. Secondly, Ubi-SAF is capable of reasoning and thus detecting indirect faults. Moreover, using a policy one can provide a more complex condition than "value over a threshold". A policy can use a combination of several measurement values, events and internal beliefs (agent's states). Finally, according to the architecture presented by Shameem et al. (2007), ETS does not support internal context provisioning. The fault notification phase of ETS is similar to the service provided by the SAF detector element. The last four phases roughly correspond to the deliberator-planner-executor portion of the SAF deliberation cycle. While Ubi-SAF can deal with unforseen situations due to the planning ability, the ETS implementation does not mention any planning process.

One of the similar approaches to SAF is Niche (Al-Shishtawy et al., 2009). Niche, same as SAF, is trying to achieve self-management of component-based applications in dynamic distributed environments. Niche relies on the replication of a service using finite state machine (Al-Shishtawy et al., 2010). The machine is then used as a replica for analysis and decision-making. Niche performs well in large-scale distributed applications, where system nodes cooperate to achieve the self-management properties. However, to our knowledge Niche does not address the issue of heterogeneous resource discovery. Also, it does not mention the issue of trust and reputation. Both of these are important in PerComp environments. In Ubi-SAF, both of these problems are solved in the service facilitator element. The use of semantic technologies makes the resource discovery less complicated and the use of utility functions lowers the complexity of reputation management.

Saxena et al. (2010) introduce a self-management method based on the DESERT self-management framework. The framework utilizes Model Integrated Computing (MIC), which is based on a method of applying computer-based modeling approaches to the problem solution. DESERT symbolically encodes the space of valid system states. When a goal changes or a fault is detected, a constraint describing the problem is created and used to resolve the issue. Similarly to Niche,

DESERT performs well when self-configuration and self-healing is expected in the cooperative environements. However, it does not provide any planning capability and thus it is only able to deal with predefined states.

Hochstatter et al. (2008) describe the CArAM project (Context Architecture for Autonomic Management) that uses a similar approach to self-management as Ubi-SAF. The main idea is to provide a system that acts as a middleware between policy-based service management and various context providers. CArAM shares some similarities with Ubi-SAF. Firstly, the adaptation process is achieved through an adaptation manager implementing a MAPE-K-like cycle. Secondly, the context provisioning is achieved using semantic technologies (Buchholz et al., 2004). Lastly, the adaptation management is based on policies. While CArAM utilizes various languages for different purposes, Ubi-SAF uses S-APL to provide all the features of platform. Moreover, to our knowledge CArAM does not use utility functions in the decision-making process.

Kumar et al. (2007) present a self-adaptive middleware based on utility functions, overlay networks and data-stream processing techniques. The system provides self-configuration and self-optimization features in form of a three-layered architecture consisting of application, autonomic and underlay layers. The application layer collects and organizes the data using an SQL-like language. The autonomic layer then performs the utility calculation and optimization. Finally, the underlay layer is responsible for maintaining a hierarchy of physical nodes, which are clustered according to system attributes. Even though the proposed system is not dealing with the area of pervasive computing, there are several similarities to the Ubi-SAF approach. It is possible to look at the three-layered model as a MAPE-like cycle, where the application layer monitors and analyzes, the autonomic layer decides and the underlay layer executes.

Probably the thematically closest effort to Ubi-SAF is the Hydra project (Sarnovsky et al., 2007), which stands for a Networked Embedded System Middleware for Heterogeneous Physical Devices in a Distributed Architecture. The goal was, among others, to create a self-managed middleware platform for the realization of the AmI vision. Despite being two separate efforts, Hydra and Ubi-SAF share several similarities. Firstly, according to Zhang and Hansen (2008a), the context in Hydra is modelled using semantic technologies. Secondly, both Hydra and Ubi-SAF utilize the three-layered model of adapatation by Kramer and Magee (2007). Lastly, both of the visions are targeting the area of pervasive computing.

Hydra was an FP6 IST Programme integrated project that lasted between the years 2006 and 2010 with a budget over 12 million EUR. This overlaps with the efforts of the SmartResource project (2004-2008) and the Ubiware project (2008-2011), where Ubiware platform and S-APL as GUN (Global Understanding Environment) enablers were developed.

There are several differences between Ubi-SAF and Hydra. Firstly, Ubi-SAF utilizes an agent-based approach to self-management achieved through the Ubiware platform. Secondly, a typical Hydra programmer uses a combination of various languages (e.g. OWL, SWRL, WSDL, Alloy modelling language, etc.) in

the application development process. This might be due to the fact that the project consortium consisted of various research institutions, each providing solutions based on their own approach and research background. Ubi-SAF is based on Ubiware and therefore the software developer uses mostly S-APL. In some cases, he/she might need to use Java to provide a new RAB functionality. However, for most applications, the database of available RABs is sufficient. Moreover, Hydra and Ubi-SAF use a different knowledge representation. Furthermore, to our knowledge Hydra does not use utility functions in the decitionsmaking process. Finally, in Ubi-SAF semantic technologies are used not only for context modelling as in Hydra, but also to solve the problem of interoperability in heterogeneous environments.

7.2 Conclusions

Chapter 1 proposed three research questions that we are trying to answer in this dissertation. Firstly, we wanted to identify the key elements for adaptive software set in pervasive computing environments. Secondly, we were trying to find an approach to implementation of such software. Lastly, we wanted to show how it can be used in pervasive computing environments.

Chapter 2 introduced several scientific visions related to the topic of this dissertation. Moreover, several key technologies were mentioned as well. Lastly, the Ubiware platform and S-APL language were described.

7.2.1 Answer to Q1

The first stated question is: *What are the key elements of a middleware for adaptive software?* In Chapter 3 we first discussed various aspects of pervasive computing environments. Such an environment consists of three elements – humans, machines (devices) and a communication network. Pervasive computing environments are open and consist of various stakeholders, each trying to achieve a combination of individual and group goals. The appropriate actions taken by these stakeholders depend on these goals and the context in which they find themselves. Moreover, devices operating in such environments are heterogeneous in their software and hardware nature. Lastly, pervasive computing environments are highly dynamic.

We introduced five requirements for software trying to achieve self-management in these environments. Along these five requirements we introduced four technologies that are able to provide the solutions. Firstly, adaptation decisions should be based on both individual and group goals. This can be achieved by using the multi-agent systems (MAS), which are a suitable tool to model and implement competitive and cooperative distributed systems. Secondly, the software should have the ability to adapt to heterogeneous interaction methods among the devices. As a solution we suggest the use of semantic technologies, which

have proven to be an effective tool to solve the heterogeneity problem. Thirdly, the adaptation overhead should be minimized, so that the software can spend most of its computing time on providing the actual service it was designed to provide. This can be achieved by dividing the adaptation process into three layers with increasing computation complexity. Moreover, self-managed software should support both the parameter adaptation and the compositional adaptation in an unobtrusive manner. The answer to this problem could be an external approach to self-management, where the adaptable software is built on top of an adaptation middleware providing basic adaptation services. Lastly, such software should support dynamic service discovery due to the dynamism of the environment it operates in. The solution could be a combination of MAS and semantic technologies. Table 8 (on page 84) summarizes these requirements.

We also introduced the concept called Smart Adaptive Framework (SAF), which is a middleware platform for development of self-managed applications for pervasive computing environments. The framework is based on the aforementioned approaches and consists of a shared knowledge base (KB), service facilitator, various processing elements and resources. We utilize a modified MAPE-K cycle, which is implemented in six processing elements. The first element, the monitor, reads all sensory data and annotates it according to the sensor ontology. Secondly, the detector is responsible for policy consistency checks. Third, the deliberator decides which actions should be taken in case a plan is available. If no plan is available, it contacts the fourth element, the planner, which performs planning. Moreover, the plan executor is responsible for the implementation of the recovery plan. Lastly, the action executor performs the actions specified in the plan by triggering the actuators. Table 9 (on page 87) summarizes the processing elements of SAF.

SAF and each software built on top of SAF have a configuration file that specifies various parameters. The platform configuration file specifies the platform administrator's policies, actions and plans. The software configuration and adaptation profile specifies various attributes of the adaptation logic.

7.2.2 Answer to Q2

The second question asks: *How can such a middleware be implemented?* To answer the question we utilize the Ubiware platform, which is based both on agent technologies and semantic technologies. Using this platform we show how SAF can be implemented in terms of S-APL constructs and Ubiware components.

In order to make the implementation possible, the platform has to be extended with additional functionality and constructs. Chapter 4 provides three improvements, namely belief safeguards, utility functions and planning. Safeguards are special S-APL constructs that allow the Ubiware developers to describe certain desired or undesired states in the agent's belief structure. The Ubiware engine is able to automatically detect if a safeguard has been broken and create a special object called ticket that can be used to resolve the issue. Utility functions allow the developer to define a resource evaluation method based on the properties

of the resources being evaluated. Utility function definitions are provided as RDF statements, which allows the engine to reason about them. A utility evaluation method is described as well. Lastly, the planning process is described by converting S-APL actions and goals into the Planning Domain Definition Language (PDDL) and utilizing an external PDDL planner.

The second portion of the answer to Q2 can be found in Chapter 5, where we provide a Ubiware-based implementation of SAF. We provide a concrete way to describe sensors, actuators, configurations, actions and plans in terms of Ubiware and S-APL constructs. Also, in form of appendices, several OWL ontologies are provided. Moreover, the service facilitator and its way to deal with reputation and trust management is explained. Finally, SAF processing elements are described.

7.2.3 Answer to Q3

The last question is: *How can such a middleware be used in the domain of pervasive computing?* The answer to the question is partially provided in Chapter 5, since the Ubiware-based SAF implementation is already based on the assumption that the devices running SAF will interact in pervasive computing environments.

The second part of the answer is provided in Chapter 6, where a sample scenario is introduced. In this scenario SAF helps to achieve both the vision of pervasive computing and decrease the perceived complexity of software used in it. As an example of such an environment we have chosen a hospital from the near future. The scenario features a nurse that takes care of a patient with the help of various technologies embedded into her environment in a transparent way. Thanks to SAF, the nurse is automatically recognized whenever she enters the room. In fact, using the same principle any hospital personnel can be recognized in any patient room. Also, the patient data is automatically displayed on the screen of the nurse's tablet depending on the nurse's profile and patient's diagnosis.

7.3 Limitations and future research

We see 3 areas, where SAF and its Ubiware-based implementation can be improved. The first area is related to the problem of better and simpler utilization of the historical sensory data. While in the current implementation of Ubiware it is possible to perform certain statistical operations on top of the historical sensory data, it is cumbersome. For example if one wants to express a condition "if the current temperature drops bellow one hour average, ...", one has to use a set of complicated statements. A more elegant solution could improve the readability of the code and potentially add new statistical capabilities. Also, some method of trend discovery would improve the proactive capabilities of the agent and thus help prevent it from falling into a certain undesired state.

Secondly, the model of interactions among agents is relatively simple in SAF. A more expressive and detailed way of modeling can improve the self-configura-

tion abilities of the framework. Also, by understanding the relationship between agents can help them plan their actions more efficiently.

Lastly, the self-optimization capabilities of SAF should be improved. While a utility-based approach provides a solution to the problem, we should conside other approaches as well (e.g. mathematical optimization). A combination of several approaches may balance out drawbacks of each individual approach.

YHTEENVETO (FINNISH SUMMARY)

Nykyään useimmilla on käytössään erilaisia suorituskykyisiä kannettavia laitteita kuten älypuhelimia ja tablettitietokoneita. Toisaalta nämä laitteet ovat yhä monimutkaisempia ja kokonaisuus voi muuttua yksittäiselle käyttäjälle hallitsemattomaksi.

Tarvitaan uudentyyppisiä ohjelmistoja, jotka kykenevät hallitsemaan itseään ilman merkittävää käyttäjän panosta. Kutsumme tällaista ohjelmistoa itseohjautuvaksi (tai adaptiiviseksi). Tällaisten uudentyyppisten ohjelmistojen mahdollistamiseksi Ubiware-projektissa luotiin uusi väliohjelmistoalusta. Alusta yhdistää uudella tavalla agettipohjaisen ohjelmistotekniikan semantiikkaan pohjautuvaan deklaratiiviseen ohjelmointikieleen, jota kutsutaan nimellä S-APL (Semantic Agent Programming Language).

Tässä työssä esitetään viitekehys itseohjautuvuudelle (SAF, Smart adaptive Framework - älykäs adaptiivinen viitekehys). Tämä viitekehys yhdistää adaptiivisten ohjelmistojen kolmitasoisen mallin ja itsenäisen suljetun suunnittelusyklin. Viitekehys antaa ohjelmistosuunnittelijalle uusia tekniikoita ja työkaluja kuvata tavoitteita, politiikkoja, tietorakenteita ja toimintasuunnitelmia ohjelmistokehityksen aikana. Näiden kuvausten avulla SAF huolehtii ohjelmiston toiminnan aikaisesta ohjauksesta itsenäisesti tai tuetusti.

Työssä tarkastellaan myös SAF:n toteuttamista Ubiware-väliohjelmiston laajennuksen avulla. Tässä yhteydessä esitellään kolme tarvittavaa laajennusta väliohjelmistoon ja sen S-APL-kieleen: hyödyllisyysfunktiot, uskomusten valvojat ja suunnittelijat. Laajennetun S-APL:n avulla kuvataan viitekehyksen eri modulien toiminta sekä tekniikat, joilla ohjelmistoa ja viitekehystä voidaan konfiguroida. Lisäksi kuvataan viitekehyksen toiminnassa tarvittava ontologia.

Lopuksi esitetään esimerkkiskenaario, jossa demonstroidaan, miten SAF voisi auttaa sairaalaympäristössä hoitohenkilöstön työvälineiden ja potilaan valvontalaitteistojen välisen yhteistyön automatisointia. SAF voisi tukea hoitajan tiedonhallintatehtävien automaattista tai puoliautomaattista suorittamista eri toimijoiden asettamien korkean tason tavoitteiden ja politiikkojen mukaan.

REFERENCES

ACM 2012. The 2012 ACM Computing Classification System. ⟨URL:http://dl.acm.org/ccs_flat.cfm⟩. Accessed on 2013–10–13.

Aarts, E. & Appelo, L. 1999. Ambient Intelligence: thuisomgevingen van de toekomst. ITMonitor 9, 7–11.

Aarts, E. & Eggen, B. 2002. Ambient intelligence in HomeLab. ⟨URL:http://andre.meyer-vitali.com/documents/ambientintelligence.pdf⟩. Accessed on 2013–10–13.

Aarts, E. & Grotenhuis, F. 2009. Ambient Intelligence 2.0: Towards Synergetic Prosperity. In Proceedings of the European Conference on Ambient Intelligence. Berlin, Heidelberg: Springer-Verlag. AmI '09, 1–13. doi:10.1007/978-3-642-05408-2_1. ⟨URL:http://dx.doi.org/10.1007/978-3-642-05408-2_1⟩.

Aarts, E., Harwig, R. & Schuurmans, M. 2002. Ambient intelligence. In P. J. Denning (Ed.) The invisible future. New York, NY, USA: McGraw-Hill, Inc., 235–250. ⟨URL:http://dl.acm.org/citation.cfm?id=504949.504964⟩.

Aarts, E. 2003. Ambient Intelligence: BUILDING THE VISION. ⟨URL:http://www.research.philips.com/technologies/download/homelab_365.pdf⟩. Accessed on 2013–10–13.

Aduna 2012. OpenRDF Sesame. ⟨URL:http://www.openrdf.org/⟩. Accessed on 2013–10–13.

Ahmed, S., Ahamed, S. I., Sharmin, M. & Hasan, C. S. 2009. Self-healing for Autonomic Pervasive Computing. In A. V. Vasilakos, M. Parashar, S. Karnouskos & W. Pedrycz (Eds.) Autonomic Communication. Springer US, 285–307. doi:10.1007/978-0-387-09753-4. ⟨URL:http://www.springerlink.com/index/10.1007/978-0-387-09753-4⟩.

Ahmed, S., Sharmin, M. & Ahamed, S. I. 2005. Knowledge Usability and its Characteristics for Pervasive Computing. In PSC, 206–209.

Al-Shishtawy, A., Fayyaz, M. A., Popov, K. & Vlassov, V. 2010. Achieving Robust Self-Management for Large-Scale Distributed Applications. In Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on, 31–40. doi:10.1109/SASO.2010.42.

Al-Shishtawy, A., Vlassov, V., Brand, P. & Haridi, S. 2009. A Design Methodology for Self-Management in Distributed Environments. In Computational Science and Engineering, 2009. CSE '09. International Conference on, Vol. 1, 430–436. doi:10.1109/CSE.2009.301.

Alahuhta, P., Hert, P. D., Delaitre, S., Friedewald, M., Gutwirth, S., Lindner, R., Maghiros, I., Moscibroda, A., Punie, Y., Schreurs, W., Verlinden, M., Vildjiounaite, E. & Wright, D. 2005. Dark scenarios on ambient intelligence: Highlighting risks and vulnerabilities. European Commission. ⟨URL:http://is.jrc.ec.europa.eu/pages/TFS/documents/SWAMI_D2_scenarios_Final_ESvf_003.pdf⟩.

Alani, H. 2003. TGVizTab: An Ontology Visualisation Extension for Protégé. In Knowledge Capture (K-Cap'03), Workshop on Visualization Information in Knowledge Engineering. ⟨URL:http://eprints.soton.ac.uk/258326/⟩.

Amann, B. & Fundulaki, I. 1999. Integrating Ontologies and Thesauri to Build RDF Schemas. In S. Abiteboul & A.-M. Vercoustre (Eds.) Research and Advanced Technology for Digital Libraries, Vol. 1696. Springer Berlin Heidelberg. Lecture Notes in Computer Science, 234–253. doi:10.1007/3-540-48155-9_16. ⟨URL:http://dx.doi.org/10.1007/3-540-48155-9_16⟩.

Apache 2013. Apache Jena. ⟨URL:http://jena.apache.org/⟩. Accessed on 2013–10–13.

Apple 2011. iPod nano (6th generation) - Technical Specifications. ⟨URL:http://support.apple.com/kb/SP593⟩. Accessed on 2013–10–13.

Arrow, K. 1980. Discrimination in the Labour Market. Readings in Labour Economics.

Ashton, K. 2009. That ' Internet of Things ' Thing. RFID Journal, 1–2. ⟨URL:http://www.rfidjournal.com/article/view/4986⟩.

Auto-IDLabs 2012. Auto-ID Labs. ⟨URL:http://www.autoidlabs.org/⟩. Accessed on 2013–10–13.

Barber, K. S., Fullam, K. & Kim, J. 2003. Challenges for trust, fraud and deception research in multi-agent systems. In Proceedings of the 2002 international conference on Trust, reputation, and security: theories and practice. Berlin, Heidelberg: Springer-Verlag. AAMAS'02, 8–14. ⟨URL:http://dl.acm.org/citation.cfm?id=1762128.1762130⟩.

Barber, K. S. & Kim, J. 2001. Belief Revision Process Based on Trust: Agents Evaluating Reputation of Information Sources. Trust in Cybersocieties 2246, 73–82. ⟨URL:http://www.springerlink.com/index/WHPTD63CXVLBA3JP.pdf⟩.

Bates, J. 1994. The role of emotion in believable agents. Communications of the ACM 37 (7), 122–125. doi:10.1145/176789.176803. ⟨URL:http://portal.acm.org/citation.cfm?doid=176789.176803⟩.

Bellifemine, F., Poggi, A. & Rimassa, G. 1999. JADE–A FIPA-compliant agent framework. Proceedings of PAAM 99 (97-108), 33.

Bellifemine, F., Poggi, A. & Rimassa, G. 2000. Developing Multi-agent Systems with JADE. In ATAL, 89–103.

Berners-Lee, T. & Connolly, D. 2011. Notation3 (N3): A readable RDF syntax. W3C. W3C Team Submission. ⟨URL:http://www.w3.org/TeamSubmission/n3/⟩.

Berners-Lee, T., Hendler, J. & Lassila, O. 2001. The Semantic Web. Scientific American 284 (5), 34–43. doi:10.1038/scientificamerican0501-34. ⟨URL:http://www.nature.com/doifinder/10.1038/scientificamerican0501-34⟩.

Berners-Lee, T. 1993. A Brief History of the Web. ⟨URL:http://www.w3.org/DesignIssues/TimBook-old/History.html⟩. Accessed on 2013–10–13.

Berners-Lee, T. 1997. Links and Law: Myths. ⟨URL:http://www.w3.org/DesignIssues/LinkMyths.html⟩. Accessed on 2013–10–13.

Berners-Lee, T. 1999. The original design and ultimate destiny of the world wide web by its inventor. Harper Collins, chapter 12.

Berners-Lee, T. 2012. Start of the web: Influences. ⟨URL:http://www.w3.org/People/Berners-Lee/FAQ.html#Influences⟩. Accessed on 2013–10–13.

Berners-Lee, T., Fielding, R. & Masinter, L. 2005. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD). ⟨URL:http://www.ietf.org/rfc/rfc3986.txt⟩. Accessed on 2013–10–13.

Bradbury, J. S., Cordy, J. R., Dingel, J. & Wermelinger, M. 2004. A survey of self-management in dynamic software architecture specifications. In Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems. ACM, 28–33.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. & Mylopoulos, J. 2004. Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems 8 (3), 203–236. doi:10.1023/B:AGNT.0000018806.20944.ef. ⟨URL:http://dx.doi.org/10.1023/B:AGNT.0000018806.20944.ef⟩.

Buchholz, T., Krause, M., Linnhoff-Popien, C. & Schiffers, M. 2004. CoCo: Dynamic Composition of Context Information. In MobiQuitous, 335–343.

Buckley, J. 2006. From RFID to the Internet of Things, 32. ⟨URL:ftp://ftp.cordis.europa.eu/pub/ist/docs/ka4/au_conf670306_buckley_en.pdf⟩.

Carothers, G. & Prud'hommeaux, E. 2013. Turtle. W3C. Candidate Recommendation.

Carothers, G. 2013. N-Triples. W3C. W3C Working Draft.

Carroll, J., Herman, I. & Patel-Schneider, P. F. 2012. OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition). W3C. W3C Recommendation.

Cederlof, H. & Pyotsia, J. 1999. Advanced Diagnostics Concept Using Intelligent Field Agents. In ISA TECH. University of Jyväskylä.

Cheng, B. H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D. & Whittle, J. 2009. Software Engineering for Self-Adaptive Systems. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi & J. Magee (Eds.) Software Engineering for Self-Adaptive Systems. Berlin, Heidelberg: Springer-Verlag, 1–26. doi:10.1007/978-3-642-02161-9_1. ⟨URL:http://dx.doi.org/10.1007/978-3-642-02161-9_1⟩.

Cochez, M. & Nagy, M. 2011. WP1: Mashupper agent-enabled social cloud. University of Jyväskylä.

Cochez, M. 2012. Semantic Agent Programming Language: use and formalization. University of Jyväskylä, Finland. Master's Thesis.

Dürst, M. & Suignard, M. 2005. Internationalized Resource Identifiers (IRIs). ⟨URL:http://www.ietf.org/rfc/rfc3987.txt⟩. Accessed on 2013–10–13.

Dey, A. K. 2001. Understanding and Using Context. Personal Ubiquitous Comput. 5 (1), 4–7. doi:10.1007/s007790170019. ⟨URL:http://dx.doi.org/10.1007/s007790170019⟩.

Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N. & Zambonelli, F. 2006. A survey of autonomic communications. ACM Trans. Auton. Adapt. Syst. 1 (2), 223–259. doi:10.1145/1186778.1186782. ⟨URL:http://doi.acm.org/10.1145/1186778.1186782⟩.

Ducatel, K., Bogdanowicz, M., Scapolo, F. & Leijten, J. 2001. Scenarios for ambient intelligence in 2010 : final report. European Commission. ⟨URL:http://www.ist.hu/doctar/fp5/istagscenarios2010.pdf⟩.

Elsevier 2013. Pervasive and Mobile Computing Journal. ⟨URL:http://www.journals.elsevier.com/pervasive-and-mobile-computing/⟩. Accessed on 2013–10–13.

Ernst, N. A. & Storey, M. A. 2003. A Preliminary Analysis of Visualization Requirements in Knowledge Engineering Tools. University of Victoria.

FIPA 2002a. FIPA00001 Abstract Architecture Specification. FIPA, 75. ⟨URL:http://www.fipa.org/specs/fipa00001/SC00001L.pdf⟩.

FIPA 2002b. FIPA00037 Communicative Act Library Specification, 45.

FIPA 2002c. FIPA00061 ACL Message Structure Specification, 11. ⟨URL:http://www.fipa.org/specs/fipa00061/SC00061G.pdf⟩.

FIPA 2003. FIPA00025 Interaction Protocol Library Specification, 25. ⟨URL:http://fipa.org/specs/fipa00025/DC00025F.pdf⟩.

FIPA 2004. FIPA00023 Agent Management Specification, 40. ⟨URL:http://www.fipa.org/specs/fipa00023/SC00023K.pdf⟩.

FIPA 2012. Foundation for Intelligent Physical Agents main web page. ⟨URL:http://www.fipa.org/⟩. Accessed on 2013–10–13.

Falcone, R., Pezzulo, G. & Castelfranchi, C. 2002. Quantifying Belief Credibility for Trust-based Decision. In Proceedings of the Autonomous Agents 2002 Workshop on Deception, Fraud, and Trust in Agent Societies. AAMAS '02, 41–48.

Fehr, E. & Gächter, S. 2000. Fairness and Retaliation: The Economics of Reciprocity. Journal of Economic Perspectives 14 (3), 159–181. doi:10.1257/jep.14.3.159. ⟨URL:http://pubs.aeaweb.org/doi/abs/10.1257/jep.14.3.159⟩.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. & Berners-Lee, T. 1999. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard). ⟨URL:http://www.ietf.org/rfc/rfc2616.txt⟩. Accessed on 2013–10–13.

Fokoue, A., Kershenbaum, A., Ma, L., Schonberg, E. & Srinivas, K. 2006. K.: The Summary Abox: Cutting Ontologies Down to Size. In In: Proc. ISWC-06. Volume 4273 of LNCS. Springer Berlin Heidelberg, 343–356. ⟨URL:http://dx.doi.org/10.1007/11926078_25⟩.

Fullam, K. K., Klos, T. B., Muller, G., Sabater, J., Schlosser, A., Topol, Z., Barber, K. S., Rosenschein, J. S., Vercouter, L. & Voss, M. 2005. A specification of the Agent Reputation and Trust (ART) testbed: experimentation and competition for trust in agent societies. In Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems. New York, NY, USA: ACM. AAMAS '05, 512–518. doi:10.1145/1082473.1082551. ⟨URL:http://doi.acm.org/10.1145/1082473.1082551⟩.

Ganek, A. G. & Corbi, T. A. 2003. The dawning of the autonomic computing era. IBM Syst. J. 42 (1), 5–18. doi:10.1147/sj.421.0005. ⟨URL:http://dx.doi.org/10.1147/sj.421.0005⟩.

Gat, E., Bonnasso, R. P., Murphy, R. & Press, A. 1997. On Three-Layer Architectures. In Artificial Intelligence and Mobile Robots. AAAI Press, 195–210.

Genesereth, M. R. & Ketchpel, S. P. 1994. Software Agents. Commun. ACM 37 (7), 48–53.

GeorgiaTech 2012. Aware Home Research Initiative. ⟨URL:http://www.awarehome.gatech.edu/⟩. Accessed on 2013–10–13.

Gershenfeld, N., Krikorian, R. & Cohen, D. 2004. The Internet of Things. Scientific American 291 (4), 46–51.

Ghosh, D., Sharman, R., Raghav Rao, H. & Upadhyaya, S. 2007. Self-healing systems - survey and synthesis. Decis. Support Syst. 42 (4), 2164–2185. doi:http:

//dx.doi.org/10.1016/j.dss.2006.06.011. ⟨URL:http://dx.doi.org/10.1016/j.dss.2006.06.011⟩.

Google 2011. Greater choice for wireless access point owners. ⟨URL:http://googleblog.blogspot.fi/2011/11/greater-choice-for-wireless-access.html⟩. Accessed on 2013–10–13.

Guha, R. V. & Brickley, D. 2004. RDF Vocabulary Description Language 1.0: RDF Schema. W3C. W3C Recommendation.

Hansen, T., Hardie, T. & Masinter, L. 2006. Guidelines and Registration Procedures for New URI Schemes. RFC 4395 (Best Current Practice). ⟨URL:http://www.ietf.org/rfc/rfc4395.txt⟩. Accessed on 2013–10–13.

Hansmann, U., Merk, L., Nicklous, M. & Stober, T. 2003. Pervasive Computing, 448.

van Harmelen, F. & McGuinness, D. 2004. OWL Web Ontology Language Overview. W3C. W3C Recommendation.

Harris, S. & Seaborne, A. 2013. SPARQL 1.1 Query Language. W3C. W3C Recommendation.

Hayes, P., Patel-Schneider, P. F. & Horrocks, I. 2004. OWL Web Ontology Language Semantics and Abstract Syntax. W3C. W3C Recommendation.

Hayes, P. 2004. RDF Semantics. W3C. W3C Recommendation.

Hermit 2013. hermit. ⟨URL:http://hermit-reasoner.com/⟩. Accessed on 2013–10–13.

Hinchey, M. G. & Sterritt, R. 2006. Self-managing software. doi:10.1109/MC.2006.69. ⟨URL:http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1597102⟩.

Hochstatter, I., Dreo, G., Serrano, M., Serrat, J., Nowak, K. & Trocha, S. 2008. An architecture for context-driven self-management of services. In INFOCOM Workshops 2008, IEEE, 1–4. doi:10.1109/INFOCOM.2008.4544629.

Hoffman, P. 2005. The telnet URI Scheme. RFC 4248 (Proposed Standard). ⟨URL:http://www.ietf.org/rfc/rfc4248.txt⟩. Accessed on 2013–10–13.

Horrocks, I., Parsia, B. & Sattler, U. 2012. OWL 2 Web Ontology Language Direct Semantics (Second Edition). W3C. W3C Recommendation.

Intel 2006. Intel Unveils World's Best Processor. ⟨URL:http://web.archive.org/web/20070403081121/http://www.intel.com/pressroom/archive/releases/20060727comp.htm?cid=rss-83642-c1-135841⟩. Accessed on 2013–10–13.

Jennings, N. R. & Wooldridge, M. 1995. Applying agent technology. Applied Artificial Intelligence 9 (4), 357–369.

Jennings, N. R. 1999. Agent-Oriented Software Engineering. In Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: MultiAgent System Engineering. London, UK, UK: Springer-Verlag. MAAMAW '99, 1–7. ⟨URL:http://dl.acm.org/citation.cfm?id=646910.710805⟩.

Jennings, N. R. 2001. An agent-based approach for building complex software systems. Communications of the ACM 44 (4), 35–41. ⟨URL:http://portal.acm.org/citation.cfm?id=367250⟩.

Jennings, N. R. 2000. On Agent-Based Software Engineering. Artificial Intelligence 117 (2), 277–296. ⟨URL:http://eprints.ecs.soton.ac.uk/3741/⟩.

Jessup, L. M. & Robey, D. 2002. The relevance of social issues in ubiquitous computing environments. Commun. ACM 45 (12), 88–91. doi:10.1145/585597.585621. ⟨URL:http://doi.acm.org/10.1145/585597.585621⟩.

Jonker, C. & Treur, J. 1999. Formal Analysis of Models for the Dynamics of Trust Based on Experiences. In F. Garijo & M. Boman (Eds.) Multi-Agent System Engineering, Vol. 1647. Springer Berlin Heidelberg. Lecture Notes in Computer Science, 221–231. doi:10.1007/3-540-48437-X_18. ⟨URL:http://dx.doi.org/10.1007/3-540-48437-X_18⟩.

Katasonov, A., Kaykova, O., Khriyenko, O., Nikitin, S. & Terziyan, V. Y. 2008. Smart Semantic Middleware for the Internet of Things. In ICINCO-ICSO, 169–178.

Katasonov, A., Nagy, M. & Cochez, M. 2012. Ubiware application developer guide, 32. ⟨URL:http://www.cs.jyu.fi/ai/OntoGroup/ubidoc/Application_developer.pdf⟩.

Katasonov, A. & Terziyan, V. 2008. Semantic Agent Programming Language (S-APL): A Middleware Platform for the Semantic Web. In Proceedings of the 2008 IEEE International Conference on Semantic Computing. Washington, DC, USA: IEEE Computer Society. ICSC '08, 504–511. doi:10.1109/ICSC.2008.82. ⟨URL:http://dx.doi.org/10.1109/ICSC.2008.82⟩.

Katifori, A., Halatsis, C., Lepouras, G., Vassilakis, C. & Giannopoulou, E. 2007. Ontology Visualization Methods – a Survey. ACM Comput. Surv. 39 (4). doi:10.1145/1287620.1287621. ⟨URL:http://doi.acm.org/10.1145/1287620.1287621⟩.

Kaykova, O., Khriyenko, O., Kovtun, D., Anton, N., Terziyan, V. & Zharko, A. 2005. General Adaption Framework. International Journal on Semantic Web and Information Systems 1 (3), 31–63.

Kephart, J. O. & Walsh, W. E. 2004. An Artificial Intelligence Perspective on Autonomic Computing Policies. In POLICY, 3–12.

Kephart, J. O. & Chess, D. M. 2003. The vision of autonomic computing. doi:10.1109/MC.2003.1160055. ⟨URL:http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055⟩.

Kephart, J. O. & Das, R. 2007. Achieving Self-Management via Utility Functions. Internet Computing, IEEE 11 (1), 40–48. doi:10.1109/MIC.2007.2.

Khriyenko, O. 2008. Adaptive semantic web based environment for web resources. University of Jyväskylä. Ph. D. Thesis, 193. ⟨URL:http://urn.fi/URN:ISBN:978-951-39-3446-0⟩.

Kindberg, T. & Fox, A. 2002. System Software for Ubiquitous Computing. IEEE Pervasive Computing 1 (1), 70–81. doi:10.1109/MPRV.2002.993146. ⟨URL:http://dx.doi.org/10.1109/MPRV.2002.993146⟩.

Kjæ r, K. E. k. 2007. A SURVEY OF CONTEXT-AWARE MIDDLEWARE. In Context. ACTA Press, 148–155. doi:10.1109/MPRV.2002.1012334. ⟨URL:http://portal.acm.org/citation.cfm?id=1332069⟩.

Krötzsch, M., Patel-Schneider, P., Rudolph, S., Parsia, B. & Hitzler, P. 2012. OWL 2 Web Ontology Language Primer (Second Edition). W3C.

Kramer, J. & Magee, J. 2007. Self-Managed Systems: an Architectural Challenge. In 2007 Future of Software Engineering. Washington, DC, USA: IEEE Computer Society. FOSE '07, 259–268. doi:10.1109/FOSE.2007.19. ⟨URL:http://dx.doi.org/10.1109/FOSE.2007.19⟩.

Kumar, V., Cooper, B. F., Cai, Z., Eisenhauer, G. & Schwan, K. 2007. Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows. Cluster Computing 10 (4), 443–455.

Laddaga, R. 1999. Creating robust software through self-adaptation. Intelligent Systems and their Applications, IEEE 14 (3), 26–29. doi:10.1109/MIS.1999.769879.

Lassila, O. & Adler, M. 2003. Semantic gadgets: Device and information interoperability. In In the working notes of the Workshop of Ubiqutous Computing Environment.

Lassila, O. 2005. Using the Semantic Web in Mobile and Ubiquitous Computing. In M. Bramer & V. Terziyan (Eds.) Industrial Applications of Semantic Web, Vol. 188. Springer US. IFIP – The International Federation for Information Processing, 19–25. doi:10.1007/0-387-29248-9_1. ⟨URL:http://dx.doi.org/10.1007/0-387-29248-9_1⟩.

Lee, J. S. M., Katari, G. & Sachidanandam, R. 2005. GObar: a gene ontology based analysis and visualization tool for gene sets. BMC bioinformatics 6 (1), 189. doi:10.1186/1471-2105-6-189. ⟨URL:http://www.biomedcentral.com/1471-2105/6/189⟩.

Liebig, T. 2004. OntoTrack: Combining browsing and editing with reasoning and explaining for OWL Lite ontologies. In In Proceedings of the 3rd International Semantic Web Conference (ISWC) 2004. Springer, 244–258.

Lyytinen, K. & Yoo, Y. 2002. Issues and Challenges in Ubiquitous Computing. Commun. ACM 45 (12), 62–65. doi:10.1145/585597.585616. ⟨URL:http://doi.acm.org/10.1145/585597.585616⟩.

MIT 2012. AUTO-ID Labs at MIT. ⟨URL:http://autoid.mit.edu/cs/⟩. Accessed on 2013–10–13.

Mühl, G., Werner, M., Jaeger, M. A., Herrmann, K. & Parzyjegla, H. 2007. On the Definitions of Self-Managing and Self-Organizing Systems. In T. Braun, G. Carle & B. Stiller (Eds.) Proceedings of the KiVS 2007 Workshop Selbstorganisierende Adaptive Kontextsensitive verteilte Systeme SAKS 2007. VDE Verlag, 291–301. ⟨URL:http://www.kbs.cs.tu-berlin.de/publications/fulltext/kivs2007_2.pdf⟩.

Mamei, M. & Zambonelli, F. 2006. Field-Based Coordination for Pervasive Multiagent Systems. Springer. Springer Series on Agent Technology. ⟨URL:http://books.google.se/books?id=IQJR3J-5HpwC⟩.

Max-neef, M. A., Hopenhayn, M. & Hamrell, S. 1991. Development and Human Needs. In HUMAN SCALE DEVELOPMENT CONCEPTION. The Apex Press, 13–54. ⟨URL:http://www.max-neef.cl/download/Max-neef_Human_Scale_development.pdf⟩.

McFarlane, D., Sarma, S., Chirn, J. L., Wong, C. Y. & Ashton, K. 2003. Auto ID systems and intelligent manufacturing control. Engineering Applications of Artificial Intelligence 16 (4), 365–376. doi:10.1016/S0952-1976(03)00077-0. ⟨URL:http://www.sciencedirect.com/science/article/pii/S0952197603000770⟩.

McFarlane, D. 2002. Auto ID based control systems-an overview. In Systems, Man and Cybernetics, 2002 IEEE International Conference on, Vol. 3, 6 pp. vol.3. doi:10.1109/ICSMC.2002.1176120.

Mckinley, P. K., Sadjadi, S. M., Kasten, E. P. & Cheng, B. H. C. 2004. A Taxonomy of Compositional Adaptation. East 2004 (May). ⟨URL:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.7262&rep=rep1&type=pdf⟩.

Melnik, S. & Decker, S. 2001. Representing Order in RDF. ⟨URL:http://infolab.stanford.edu/~stefan/daml/order.html⟩.

Merriam-Webster 2013. Merriam-Webster dictionary.

Miller, E. & Manola, F. 2004. RDF Primer. W3C. W3C Recommendation.

Nagy, M., Katasonov, A., Khriyenko, O., Nikitin, S., Szydlowski, M. & Terziyan, V. 2009. Challenges of Middleware for the Internet of Things. In Automation Control - Theory and Practice. InTech, 247–270.

Nagy, M. 2012. On the Problem of Multi-Channel Communication. In ICTERI, 128–133. ⟨URL:http://ceur-ws.org/Vol-848/ICTERI-2012-CEUR-WS-paper-40-p-128-133.pdf⟩.

Nagy, M. 2013. A Multi-channel Communication Framework. In V. Ermolayev, H. Mayr, M. Nikitchenko, A. Spivakovsky & G. Zholtkevych (Eds.) ICT in Education, Research, and Industrial Applications, Vol. 347. Springer Berlin Heidelberg. Communications in Computer and Information Science, 72–88. doi:10.1007/978-3-642-35737-4_5. ⟨URL:http://dx.doi.org/10.1007/978-3-642-35737-4_5⟩.

Nikitin, S., Katasonov, A. & Terziyan, V. Y. 2009. Ontonuts: Reusable Semantic Components for Multi-agent Systems. In ICAS, 200–207.

Nikitin, S. 2011. Dynamic aspects of industrial middleware architectures. University of Jyväskylä. Ph. D. Thesis.

Noy, N. F., Fergerson, R. W. & Musen, M. A. 2000. The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility. In Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management. London, UK, UK: Springer-Verlag. EKAW '00, 17–32. ⟨URL:http://dl.acm.org/citation.cfm?id=645361.650855⟩.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S. & Wolf, A. L. 1999. An architecture-based approach to self-adaptive software. Intelligent Systems and their Applications, IEEE 14 (3), 54–62. doi:10.1109/5254.769885.

Pantic, M. & Rothkrantz, L. J. M. 2003. Toward an affect-sensitive multimodal human-computer interaction. Proceedings of the IEEE 91 (9), 1370–1390. doi:10.1109/JPROC.2003.817122.

Parashar, M. & Hariri, S. 2005. Autonomic Computing : An Overview. Unconventional Programming Paradigms 3566, 247–259. doi:10.1007/11527800_20. ⟨URL:http://www.springerlink.com/index/8JWVM292E2N5NPMG.pdf⟩.

Parsia, B., Motik, B. & Patel-Schneider, P. 2012. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C. W3C Recommendation.

Patel-Schneider, P. & Horridge, M. 2012. OWL 2 Web Ontology Language Manchester Syntax (Second Edition). W3C. W3C Note.

PervasiveHealth 2013. 8th International Conference on Pervasive Computing Technologies for Healthcare.

Preece, J., Rogers, Y. & Sharp, H. 2011. Interaction Design (3rd edition). New York, NY, USA: John Wiley & Sons, Inc.

Prud'hommeaux, E. & Seaborne, A. 2008. SPARQL Query Language for RDF. W3C. W3C Recommendation.

Quitadamo, R. & Zambonelli, F. 2008. Autonomic communication services: a new challenge for software agents. Autonomous Agents and Multi-Agent Systems 17

(3), 457–475. doi:10.1007/s10458-008-9054-9. ⟨URL:http://dx.doi.org/10.1007/s10458-008-9054-9⟩.

RacerSystems 2013. Renamed ABox and Concept Expression Reasoner (RACER). ⟨URL:http://www.racer-systems.com/products/racerpro/⟩. Accessed on 2013–10–13.

Rao, A. S. & Georgeff, M. P. 1995. BDI Agents : From Theory to Practice. Practice 95 (Technical Note 56), 312–319. doi:10.1.1.51.9247. ⟨URL:http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:BDI+Agents+:+From+Theory+to+Practice#0⟩.

Rivadeneira, W. 2003. A Study of Search Result Clustering Interfaces: Comparing Textual and Zoomable User Interfaces. University of Maryland HCIL, 8.

Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H. & Nahrstedt, K. 2002. A middleware infrastructure for active spaces. Pervasive Computing, IEEE 1 (4), 74–83. doi:10.1109/MPRV.2002.1158281.

Ronzani, D. 2009. The Battle of Concepts: Ubiquitous Computing, Pervasive Computing and Ambient Intelligence in Mass Media. Ubiquitous Computing and Communication Journal 4, 9–19. ⟨URL:http://www.ubicc.org/files/pdf/paper-146_20070929.pdf_146.pdf⟩.

Russell, S. J. & Norvig, P. 2003. Artificial Intelligence: A Modern Approach (2nd edition). Pearson Education.

Saha, D. & Mukherjee, A. 2003. Pervasive computing: a paradigm for the 21st century. Computer 36 (3), 25–31. doi:10.1109/MC.2003.1185214.

Salehie, M. & Tahvildari, L. 2009. Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4 (2), 1–42. doi:10.1145/1516533.1516538. ⟨URL:http://portal.acm.org/citation.cfm?doid=1516533.1516538⟩.

Salehie, M., Li, S. L. S., Asadollahi, R. & Tahvildari, L. 2009. Change Support in Adaptive Software: A Case Study for Fine-Grained Adaptation. doi:10.1109/EASe.2009.11. ⟨URL:http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4839203⟩.

Samsung 2013. Galaxy S4 mini. ⟨URL:http://www.samsung.com/uk/consumer/mobile-devices/smartphones/android/GT-I9195ZKABTU-spec⟩. Accessed on 2013–10–13.

Sarnovsky, M., Butka, P., Kostelnik, P. & Lackova, D. 2007. HYDRA – Network Embedded System Middleware for Ambient Intelligent Devices. In ICCC2007: Proceedings of 8th Inter-national Carpathian Control Conference, 4.

Saxena, T., Dubey, A., Balasubramanian, D. & Karsai, G. 2010. Enabling Self-Management by Using Model-Based Design Space Exploration. In Engineering of Autonomic and Autonomous Systems (EASe), 2010 Seventh IEEE International Conference and Workshops on, 137–144. doi:10.1109/EASe.2010.22.

Schreiber, G. & Dean, M. 2004. OWL Web Ontology Language Reference. W3C. W3C Recommendation. ⟨URL:http://www.w3.org/TR/2004/REC-owl-ref-20040210/⟩.

Sen, S. & Sajja, N. 2002. Robustness of reputation-based trust: boolean case. In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1. New York, NY, USA: ACM. AAMAS '02, 288–293. doi:10.1145/544741.544808. ⟨URL:http://doi.acm.org/10.1145/544741.544808⟩.

Sestinim, F. 2004. Situated and Autonomic Communications. European Commission.

Sestinim, F. 2006. Situated and autonomic communication an EC FET European initiative. SIGCOMM Comput. Commun. Rev. 36 (2), 17–20. doi:10.1145/1129582.1129587. ⟨URL:http://doi.acm.org/10.1145/1129582.1129587⟩.

Shameem, A., Moushumi, S. & Sheikh, I. A. 2007. ETS (Efficient, Transparent, and Secured) Self-healing Service for Pervasive Computing Applications. I. J. Network Security 4 (3), 271–281.

Sharmin, M., Ahmed, S. & Ahamed, S. I. 2005. SAFE-RD (secure, adaptive, fault tolerant, and efficient resource discovery) in pervasive computing environments. In Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on, Vol. 2, 271–276 Vol. 2. doi:10.1109/ITCC.2005.249.

Sharmin, M., Ahmed, S. & Ahamed, S. I. 2006. MARKS (Middleware Adaptability for Resource Discovery, Knowledge Usability and Self-healing) for Mobile Devices of Pervasive Computing Environments. In Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on, 306–313. doi:10.1109/ITNG.2006.88.

Shneiderman, B. 1992. Tree Visualization with Tree-maps: 2-d Space-filling Approach. ACM Trans. Graph. 11 (1), 92–99. doi:10.1145/102377.115768. ⟨URL:http://doi.acm.org/10.1145/102377.115768⟩.

Shoham, Y. 1991. AGENT0: A Simple Agent Language and Its Interpreter. In AAAI, 704–709.

Shoham, Y. 1993. Agent-oriented programming. Artificial Intelligence 60 (1), 51–92. doi:10.1016/0004-3702(93)90034-9. ⟨URL:http://linkinghub.elsevier.com/retrieve/pii/0004370293900349⟩.

Shoham, Y. 1997. Software agents. In J. M. Bradshaw (Ed.) Software agents. Cambridge, MA, USA: MIT Press, 271–290. ⟨URL:http://dl.acm.org/citation.cfm?id=267985.268004⟩.

Smith, A. 2013. Smartphone Ownership 2013, 12. ⟨URL:http://pewinternet.org/~/media//Files/Reports/2013/PIP_Smartphone_adoption_2013_PDF.pdf⟩.

Standford 2013. Protege ontlogy editor. ⟨URL:http://protege.stanford.edu/⟩. Accessed on 2013–10–13.

Sterritt, R., Parashar, M., Tianfield, H. & Unland, R. 2005. A concise introduction to autonomic computing. Adv. Eng. Inform. 19 (3), 181–187. doi:10.1016/j.aei.2005.05.012. ⟨URL:http://dx.doi.org/10.1016/j.aei.2005.05.012⟩.

Storey, M. A., Musen, M. A., Silva, J., Best, C., Ernst, N., Fergerson, R. & Noy, N. F. 2001. Jambalaya: Interactive visualization to enhance ontology authoring and knowledge acquisition in Protege. In Workshop on Interactive Tools for Knowledge Capture, K-CAP-2001. Victoria, B.C. Canada: . ⟨URL:http://www.cs.toronto.edu/~nernst/papers/storey-kcap2001.pdf⟩.

TOP500 2013. Performance Development. ⟨URL:http://www.top500.org/statistics/perfdevel/⟩. Accessed on 2013–10–13.

Terziyan, V. & Katasonov, A. 2009. Global Understanding Environment: Applying Semantic and Agent Technologies to Industrial Automation. In M. D. Lytras & P. Ordóñez de Pablos (Eds.) Emerging Topics and Technologies in Information Systems. IGI Global. doi:10.4018/978-1-60566-222-0. ⟨URL:http://www.igi-global.com/chapter/global-understanding-environment/10190/⟩.

Terziyan, V., Nagy, M., Cochez, M., Pilli-Sihvola, V., Kesäniemi, J. & Khriyenko, O. 2010a. Deliverable D3.4 UBIWARE Platform Prototype v. 3.1. Industrial Ontologies Group - Agora Center, University of Jyväskylä. Deliverable.

Terziyan, V., Nagy, M., Cochez, M., Pulkkis, A., Kesäniemi, J., Khriyenko, O. & Nikitin, S. 2010b. Deliverable D3.3: UBIWARE Platform Prototype v.3.0, 45.

Terziyan, V. & al., E. 2007. SmartResource Project Final Report. SmartResource Tekes Project - Agora Center, University of Jyväskylä. Deliverable.

Terziyan, V. 2003. Semantic Web Services for Smart Devices in a Global Understanding Environment. In R. Meersman & Z. Tari (Eds.) On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops, Vol. 2889. Springer Berlin Heidelberg. Lecture Notes in Computer Science, 279–291. doi:10.1007/978-3-540-39962-9_37. ⟨URL:http://dx.doi.org/10.1007/978-3-540-39962-9_37⟩.

Terziyan, V. 2005. Semantic Web Services for Smart Devices Based on Mobile Agents. International Journal of Intelligent Information Technologies 1 (2), 43—-55.

Terziyan, V. 2008. SmartResource - Proactive Self-Maintained Resources in Semantic Web: Lessons Learned. International Journal of Smart Home 2 (2), 33–58.

Terziyan, V. Y. 2011. Global Understanding Environment: Towards Self-managed Web of Everything. In GPC Workshops, 1–2.

Tesler, J. & Strasnick, S. 1992. FSN: The 3D file system navigator. Silicon Graphics, Inc., Mountain View, CA.

Vázquez-Salceda, J., Dignum, V. & Dignum, F. 2005. Organizing Multiagent Systems. Autonomous Agents and Multi-Agent Systems 11 (3), 307–360.

W3C 2009. OWL 2 Web Ontology Language Document Overview. W3C.

W3C 2013. Literals as Subjects. ⟨URL:http://www.w3.org/2001/sw/wiki/Literals_as_Subjects⟩. Accessed on 2013–10–13.

Walsh, W. E., Tesauro, G., Kephart, J. O. & Das, R. 2004. Utility functions in autonomic systems. In Autonomic Computing, 2004. Proceedings. International Conference on, 70–77. doi:10.1109/ICAC.2004.1301349.

Weiser, M. 1991. The Computer for the Twenty-First Century. Scientific American 265 (3), 94–104.

Weiser, M. 1993. Some computer science issues in ubiquitous computing. Commun. ACM 36 (7), 75–84. doi:10.1145/159544.159617. ⟨URL:http://doi.acm.org/10.1145/159544.159617⟩.

Weyns, D., Haesevoets, R., Van Eylen, B., Helleboogh, A., Holvoet, T. & Joosen, W. 2008. Endogenous versus exogenous self-management. In Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems. New York, NY, USA: ACM. SEAMS '08, 41–48. doi:10.1145/1370018.1370027. ⟨URL:http://doi.acm.org/10.1145/1370018.1370027⟩.

White, S. R., Hanson, J. E., Whalley, I., Chess, D. M. & Kephart, J. O. 2004. An Architectural Approach to Autonomic Computing. In ICAC, 2–9.

White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., Segal, A. & Kephart, J. O. 2006. Autonomic computing: Architectural approach and prototype. Integrated Computer-Aided Engineering 13 (2), 173–188. ⟨URL:http://iospress.metapress.com/content/FFMYG7A5H3F0M0LA⟩.

Wildstrom, S. 2000. Can Oxygen Turn Sci-Fi into Reality? Business Week Magazine. ⟨URL:http://www.businessweek.com/2000/00_29/b3690062.htm⟩.

Williamson, O. E. 1985. The Economic Institutions of Capitalism: Firms, Markets, Relational Contracting. Free Press. ⟨URL:http://books.google.se/books?id=lj-6AAAAIAAJ⟩.

Wolf, P. 1996. Three-Dimensional Information Visualisation: The Harmony Information Landscape. Technische Universitat Graz. Master's Thesis.

Wooldridge, M., Jennings, N. R. & Kinny, D. 2000. The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems 3 (3), 285–312.

Wooldridge, M. 1997. Agent-based software engineering. IEE Proceedings - Software 144 (1), 26–37.

Wooldridge, M. 2002. Intelligent agents: The key concepts. In V. Marík, O. Štepánková, H. Krautwurmová & M. Luck (Eds.) MultiAgent Systems and Applications II, Vol. 2322. Springer. Lecture Notes in Computer Science, 3–43. doi:10.1007/3-540-45982-0_1. ⟨URL:http://www.springerlink.com/index/e34fyq4mnejb0pa6.pdf⟩.

Wooldridge, M. J. & Jennings, N. R. 1995a. Agent theories, architectures, and languages: a survey. In Agent theories, architectures, and languages: a survey. Springer-Verlag, 1–39.

Wooldridge, M. J. & Jennings, N. R. 1995b. Intelligent Agents: Theory and Practice. Knowledge Engineering Review 10 (2), 115–152. ⟨URL:http://eprints.soton.ac.uk/252102/⟩.

Wu, Z., Horrocks, I., Motik, B., Fokoue, A. & Grau, B. C. 2012. OWL 2 Web Ontology Language Profiles (Second Edition). W3C. W3C Recommendation.

Yu, B. & Singh, M. P. 2002. An evidential model of distributed reputation management. In AAMAS. ACM, 294–301.

Zhang, W. & Hansen, K. M. 2008a. Towards self-managed pervasive middleware using owl/swrl ontologies. In Fifth International Workshop on Modelling and Reasoning in Context. MRC 2008.

Zhang, W. & Hansen, K. M. 2008b. Semantic Web Based Self-Management for a Pervasive Service Middleware. In Self-Adaptive and Self-Organizing Systems, 2008. SASO '08. Second IEEE International Conference on, 245–254. doi:10.1109/SASO.2008.14.

Zhong, S., Storch, K.-F., Lipan, O., Kao, M.-C., Weitz, C. & Wong, W. 2004. GoSurfer. Applied Bioinformatics 3 (4), 261–264. doi:10.2165/00822942-200403040-00009. ⟨URL:http://dx.doi.org/10.2165/00822942-200403040-00009⟩.

APPENDIX 1 OWL ONTOLOGIES

APPENDIX 1.1 Utility function ontology

```
@prefix : <http://iog.jyu.fi/utility-functions.owl#> .
@prefix u: <http://iog.jyu.fi/utility-functions.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://iog.jyu.fi/utility-functions.owl> .
<http://iog.jyu.fi/utility-functions.owl> rdf:type owl:Ontology .

:hasFunction rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:range :Function ;
rdfs:domain :UtilityFunction .

:hasQuery rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :UtilityFunction .

:hasResourceURI rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :ResultElement ;
rdfs:range owl:Thing .

:op1 rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :Function ;
rdfs:range :Operand .

:op2 rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :BinaryFunction ;
rdfs:range :Operand .

:useDataContainer rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :Evaluation .
```

```
50
51 :useFunction rdf:type owl:AsymmetricProperty ,
52 owl:FunctionalProperty ,
53 owl:IrreflexiveProperty ,
54 owl:ObjectProperty ;
55 rdfs:domain :Evaluation ;
56 rdfs:range :UtilityFunction .
57
58 :hasEvalValue rdf:type owl:DatatypeProperty ,
59 owl:FunctionalProperty ;
60 rdfs:domain :ResultElement ;
61 rdfs:range xsd:float .
62
63 :hasValue rdf:type owl:DatatypeProperty ,
64 owl:FunctionalProperty ;
65 rdfs:domain :Value .
66
67 :hasVarName rdf:type owl:DatatypeProperty ,
68 owl:FunctionalProperty ;
69 rdfs:domain :Variable ;
70 rdfs:range xsd:string .
71
72 :usesEvalElement rdf:type owl:DatatypeProperty ,
73 owl:FunctionalProperty ;
74 rdfs:domain :UtilityFunction ;
75 rdfs:range xsd:string .
76
77 :BinaryFunction rdf:type owl:Class ;
78 rdfs:subClassOf :Function .
79
80 :Evaluation rdf:type owl:Class .
81
82 :Function rdf:type owl:Class ;
83 rdfs:subClassOf :Operand .
84
85 :FunctionAdd rdf:type owl:Class ;
86 rdfs:subClassOf :BinaryFunction .
87
88 :FunctionDiv rdf:type owl:Class ;
89 rdfs:subClassOf :BinaryFunction .
90
91 :FunctionMul rdf:type owl:Class ;
92 rdfs:subClassOf :BinaryFunction .
93
94 :FunctionSquare rdf:type owl:Class ;
95 rdfs:subClassOf :BinaryFunction .
96
97 :FunctionSub rdf:type owl:Class ;
98 rdfs:subClassOf :BinaryFunction .
99
100 :Operand rdf:type owl:Class .
101
102 :ResultElement rdf:type owl:Class .
103
```

```
:UnaryFunction rdf:type owl:Class ;
rdfs:subClassOf :Function .

:UtilityFunction rdf:type owl:Class .

:Value rdf:type owl:Class ;
rdfs:subClassOf :Operand .

:Variable rdf:type owl:Class ;
rdfs:subClassOf :Operand .
```

APPENDIX 1.2 Safeguard ontology

```
@prefix : <http://www.ubiware.jyu.fi/safeguard.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://www.ubiware.jyu.fi/safeguard.owl> .
<http://www.ubiware.jyu.fi/safeguard.owl> rdf:type owl:Ontology .

:blockedBySFG rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:InverseFunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:range :Safeguard ;
rdfs:domain :TicketBLO .

:handlingSFG rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:range :Safeguard ;
rdfs:domain :Ticket .

:hasCondition rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :Safeguard .

:inState rdf:type owl:AsymmetricProperty ,
owl:FunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :Ticket ;
rdfs:range :TicketState .

:hasImportance rdf:type owl:DatatypeProperty ;
```

```
rdfs:domain :Safeguard ;
rdfs:range xsd:integer .

:ExistenceSafeguard rdf:type owl:Class ;
rdfs:subClassOf :Safeguard ;
owl:disjointWith :NonexistenceSafeguard ,
:Ticket ,
:TicketBLO ,
:TicketCRI ,
:TicketDET ,
:TicketPRO ,
:TicketRES ,
:TicketState .

:NonexistenceSafeguard rdf:type owl:Class ;
rdfs:subClassOf :Safeguard ;
owl:disjointWith :Ticket ,
:TicketBLO ,
:TicketCRI ,
:TicketDET ,
:TicketPRO ,
:TicketRES ,
:TicketState .

:Safeguard rdf:type owl:Class ;
owl:equivalentClass [ rdf:type owl:Class ;
owl:unionOf ( :ExistenceSafeguard
:NonexistenceSafeguard
)
] ;
owl:disjointWith :Ticket ,
:TicketBLO ,
:TicketCRI ,
:TicketDET ,
:TicketPRO ,
:TicketRES ,
:TicketState .

:Ticket rdf:type owl:Class ;
owl:equivalentClass [ rdf:type owl:Class ;
owl:unionOf ( :TicketBLO
:TicketCRI
:TicketDET
:TicketPRO
:TicketRES
)
] ;
owl:disjointWith :TicketState .

:TicketBLO rdf:type owl:Class ;
rdfs:subClassOf :Ticket ;
owl:disjointWith :TicketCRI ,
:TicketDET ,
:TicketPRO ,
```

```
:TicketRES ,
:TicketState .

:TicketCRI rdf:type owl:Class ;
rdfs:subClassOf :Ticket ;
owl:disjointWith :TicketDET ,
:TicketPRO ,
:TicketRES ,
:TicketState .

:TicketDET rdf:type owl:Class ;
rdfs:subClassOf :Ticket ;
owl:disjointWith :TicketPRO ,
:TicketRES ,
:TicketState .

:TicketPRO rdf:type owl:Class ;
rdfs:subClassOf :Ticket ;
owl:disjointWith :TicketRES ,
:TicketState .

:TicketRES rdf:type owl:Class ;
rdfs:subClassOf :Ticket ;
owl:disjointWith :TicketState .

:TicketState rdf:type owl:Class .

:stateBLO rdf:type :TicketState ,
owl:NamedIndividual .

:stateCRI rdf:type :TicketState ,
owl:NamedIndividual .

:stateDET rdf:type :TicketState ,
owl:NamedIndividual .

:statePRO rdf:type :TicketState ,
owl:NamedIndividual .

:stateRES rdf:type :TicketState ,
owl:NamedIndividual .
```

APPENDIX 1.3 Plan ontology

```
@prefix : <http://www.ubiware.jyu.fi/plan.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix pla: <http://www.ubiware.jyu.fi/plan.owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
```

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@base <http://www.ubiware.jyu.fi/plan.owl> .
<http://www.ubiware.jyu.fi/plan.owl> rdf:type owl:Ontology .

:boundTo rdf:type owl:ObjectProperty ;
rdfs:domain :Variable ;
rdfs:range owl:Thing .

:effectAdd rdf:type owl:ObjectProperty ;
rdfs:domain :Action ;
rdfs:range sapl:Container .

:effectRemove rdf:type owl:ObjectProperty ;
rdfs:domain :Action ;
rdfs:range sapl:Container .

:hasPlanSequence rdf:type owl:ObjectProperty ;
rdfs:domain :Plan ;
rdfs:range rdfs:Bag .

:hasVariableBinding rdf:type owl:ObjectProperty ;
rdfs:domain :Step ;
rdfs:range sapl:Container .

:performsAction rdf:type owl:ObjectProperty ;
rdfs:range :Action ;
rdfs:domain :Step .

:preconditionExist rdf:type owl:ObjectProperty ;
rdfs:domain :Action ;
rdfs:range sapl:Container .

:preconditionNonexist rdf:type owl:ObjectProperty ;
rdfs:domain :Action ;
rdfs:range sapl:Container .

:expressedAs rdf:type owl:DatatypeProperty ;
rdfs:domain :Variable ;
rdfs:range xsd:string .

:Action rdf:type owl:Class .

:AtomicAction rdf:type owl:Class ;
rdfs:subClassOf :Action ;
owl:disjointWith :Plan .

:Plan rdf:type owl:Class ;
rdfs:subClassOf :Action .

:Step rdf:type owl:Class .

:Variable rdf:type owl:Class .

sapl:Container rdf:type owl:Class .
```

```
62
63 rdfs:Bag rdf:type owl:Class .
```

APPENDIX 1.4 Sensor ontology

```
1  @prefix : <http://www.ubiware.jyu.fi/sensor.owl#> .
2  @prefix owl: <http://www.w3.org/2002/07/owl#> .
3  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4  @prefix xml: <http://www.w3.org/XML/1998/namespace> .
5  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
6  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
7  @prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
8  @base <http://www.ubiware.jyu.fi/sensor.owl> .
9  <http://www.ubiware.jyu.fi/sensor.owl> rdf:type owl:Ontology .
10
11 :hasEvent rdf:type owl:AsymmetricProperty ,
12 owl:FunctionalProperty ,
13 owl:IrreflexiveProperty ,
14 owl:ObjectProperty ;
15 rdfs:domain :EventResult ;
16 rdfs:range :SensorEvent .
17
18 :implementedIn rdf:type owl:AsymmetricProperty ,
19 owl:FunctionalProperty ,
20 owl:IrreflexiveProperty ,
21 owl:ObjectProperty ;
22 rdfs:range sapl:RAB ;
23 rdfs:domain :Sensor .
24
25 :providedObjectType rdf:type owl:AsymmetricProperty ,
26 owl:FunctionalProperty ,
27 owl:IrreflexiveProperty ,
28 owl:ObjectProperty ;
29 rdfs:domain :Sensor ;
30 rdfs:range owl:Class .
31
32 :readFrom rdf:type owl:AsymmetricProperty ,
33 owl:FunctionalProperty ,
34 owl:IrreflexiveProperty ,
35 owl:ObjectProperty ;
36 rdfs:range :Sensor ;
37 rdfs:domain :SensorResult .
38
39 :hasTimestamp rdf:type owl:DatatypeProperty ,
40 owl:FunctionalProperty ;
41 rdfs:domain :SensorResult ;
42 rdfs:range xsd:dateTimeStamp .
43
44 :hasValue rdf:type owl:DatatypeProperty ,
45 owl:FunctionalProperty ;
46 rdfs:domain :MeasurementResult .
```

```
47
48 sapl:RAB rdf:type owl:Class .
49
50 :EnvSensor rdf:type owl:Class ;
51 rdfs:subClassOf :Sensor ;
52 owl:disjointUnionOf ( :PhysicalEnvSensor
53 :VirtualEnvSensor
54 ) .
55
56 :EventResult rdf:type owl:Class ;
57 rdfs:subClassOf :SensorResult ;
58 owl:disjointWith :MeasurementResult .
59
60 :HWSensor rdf:type owl:Class ;
61 rdfs:subClassOf :Sensor .
62
63 :ListenerSensor rdf:type owl:Class ;
64 rdfs:subClassOf :Sensor .
65
66 :MeasurementResult rdf:type owl:Class ;
67 rdfs:subClassOf :SensorResult .
68
69 :MeasurementSensor rdf:type owl:Class ;
70 rdfs:subClassOf :Sensor .
71
72 :OSSensor rdf:type owl:Class ;
73 rdfs:subClassOf :Sensor .
74
75 :PhysicalEnvSensor rdf:type owl:Class ;
76 rdfs:subClassOf :EnvSensor .
77
78 :PullSensor rdf:type owl:Class ;
79 rdfs:subClassOf :Sensor .
80
81 :PushSensor rdf:type owl:Class ;
82 rdfs:subClassOf :Sensor .
83
84 :SWSensor rdf:type owl:Class ;
85 rdfs:subClassOf :Sensor .
86
87 :Sensor rdf:type owl:Class ;
88 owl:disjointWith :SensorEvent ,
89 :SensorResult ;
90 owl:disjointUnionOf ( :EnvSensor
91 :HWSensor
92 :OSSensor
93 :SWSensor
94 ) ,
95 ( :ListenerSensor
96 :MeasurementSensor
97 ) ,
98 ( :PullSensor
99 :PushSensor
100 ) .
```

```
101
102 :SensorEvent rdf:type owl:Class ;
103 owl:disjointWith :SensorResult .
104
105 :SensorResult rdf:type owl:Class ;
106 owl:disjointUnionOf ( :EventResult
107 :MeasurementResult
108 ) .
109
110 :VirtualEnvSensor rdf:type owl:Class ;
111 rdfs:subClassOf :EnvSensor .
112
113 owl:Class rdf:type owl:Class .
```

APPENDIX 1.5 Actuator ontology

```
1  @prefix : <http://www.ubiware.jyu.fi/actuator.owl##> .
2  @prefix owl: <http://www.w3.org/2002/07/owl#> .
3  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4  @prefix xml: <http://www.w3.org/XML/1998/namespace> .
5  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
6  @prefix conf: <http://www.ubiware.jyu.fi/config.owl#> .
7  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
8  @prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
9  @base <http://www.ubiware.jyu.fi/actuator.owl#> .
10 <http://www.ubiware.jyu.fi/actuator.owl#> rdf:type owl:Ontology ;
11 owl:imports <http://www.ubiware.jyu.fi/configuration.owl> .
12
13 :affects rdf:type owl:AsymmetricProperty ,
14 owl:IrreflexiveProperty ,
15 owl:ObjectProperty ;
16 rdfs:domain :SWActuator ;
17 rdfs:range conf:Capability .
18
19 :implementedIn rdf:type owl:AsymmetricProperty ,
20 owl:FunctionalProperty ,
21 owl:IrreflexiveProperty ,
22 owl:ObjectProperty ;
23 rdfs:domain :SWActuator ;
24 rdfs:range sapl:RAB .
25
26 :Actuator rdf:type owl:Class .
27
28 :CompositionalActuator rdf:type owl:Class ;
29 rdfs:subClassOf :SWActuator .
30
31 :HWActuator rdf:type owl:Class ;
32 rdfs:subClassOf :Actuator .
33
34 :OSActuator rdf:type owl:Class ;
35 rdfs:subClassOf :Actuator .
```

```
36
37 :ParametricActuator rdf:type owl:Class ;
38 rdfs:subClassOf :SWActuator .
39
40 :PlatformActuator rdf:type owl:Class ;
41 rdfs:subClassOf :Actuator .
42
43 :SWActuator rdf:type owl:Class ;
44 rdfs:subClassOf :Actuator .
```

APPENDIX 1.6 Incident ontology

```
1  @prefix : <http://www.ubiware.jyu.fi/incident.owl#> .
2  @prefix inc: <http://www.ubiware.jyu.fi/incident.owl#> .
3  @prefix owl: <http://www.w3.org/2002/07/owl#> .
4  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
5  @prefix sen: <http://www.ubiware.jyu.fi/sensor.owl#> .
6  @prefix sfg: <http://www.ubiware.jyu.fi/safeguard.owl#> .
7  @prefix xml: <http://www.w3.org/XML/1998/namespace> .
8  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
9  @prefix conf: <http://www.ubiware.jyu.fi/config.owl#> .
10 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
11 @base <http://www.ubiware.jyu.fi/incident.owl#> .
12 <http://www.ubiware.jyu.fi/incident.owl#> rdf:type owl:Ontology ;
13 owl:imports <http://www.ubiware.jyu.fi/config.owl> ,
14 <http://www.ubiware.jyu.fi/safeguard.owl> ,
15 <http://www.ubiware.jyu.fi/sensor.owl> .
16
17 :brokenPolicy rdf:type owl:ObjectProperty ;
18 rdfs:range conf:Policy ;
19 rdfs:domain :Incident .
20
21 :responsibleSensorResult rdf:type owl:ObjectProperty ;
22 rdfs:domain :Incident ;
23 rdfs:range sen:SensorResult .
24
25 :responsibleTicket rdf:type owl:ObjectProperty ;
26 rdfs:domain :Incident ;
27 rdfs:range sfg:Ticket .
28
29 :DeviceIncident rdf:type owl:Class ;
30 rdfs:subClassOf :Incident .
31
32 :EmotionalIncident rdf:type owl:Class ;
33 rdfs:subClassOf :OwnerIncident .
34
35 :EnvironmentIncident rdf:type owl:Class ;
36 rdfs:subClassOf :Incident .
37
38 :HWIncident rdf:type owl:Class ;
39 rdfs:subClassOf :DeviceIncident .
```

```
:Incident rdf:type owl:Class .

:LocationIncident rdf:type owl:Class ;
rdfs:subClassOf :SpatialIncident .

:MotionIncident rdf:type owl:Class ;
rdfs:subClassOf :SpatialIncident .

:OwnerIncident rdf:type owl:Class ;
rdfs:subClassOf :Incident .

:PhysicalIncident rdf:type owl:Class ;
rdfs:subClassOf :EnvironmentIncident .

:PreferenceIncident rdf:type owl:Class ;
rdfs:subClassOf :OwnerIncident .

:SWIncident rdf:type owl:Class ;
rdfs:subClassOf :DeviceIncident .

:SocialIncident rdf:type owl:Class ;
rdfs:subClassOf :OwnerIncident .

:SpatialIncident rdf:type owl:Class ;
rdfs:subClassOf :Incident .

:VirtualIncident rdf:type owl:Class ;
rdfs:subClassOf :EnvironmentIncident .
```

APPENDIX 1.7 Configuration ontology

```
@prefix : <http://www.ubiware.jyu.fi/config.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sen: <http://www.ubiware.jyu.fi/sensor.owl#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@base <http://www.ubiware.jyu.fi/config.owl> .
<http://www.ubiware.jyu.fi/config.owl> rdf:type owl:Ontology ;
owl:imports <http://www.ubiware.jyu.fi/sensor.owl> .

:currentlyUtilizes rdf:type owl:ObjectProperty ;
rdfs:domain :AdaptiveSoftware ;
rdfs:range :Component .

:dependentOntology rdf:type owl:AsymmetricProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
```

```
20 rdfs:domain :Capability ;
21 rdfs:range owl:Ontology .
22
23 :hasCondition rdf:type owl:AsymmetricProperty ,
24 owl:FunctionalProperty ,
25 owl:IrreflexiveProperty ,
26 owl:ObjectProperty ;
27 rdfs:domain :Policy .
28
29 :hasInVar rdf:type owl:AsymmetricProperty ,
30 owl:IrreflexiveProperty ,
31 owl:ObjectProperty ;
32 rdfs:domain :Capability ;
33 rdfs:range :InVariableDescription .
34
35 :hasOutVar rdf:type owl:AsymmetricProperty ,
36 owl:IrreflexiveProperty ,
37 owl:ObjectProperty ;
38 rdfs:domain :Capability ;
39 rdfs:range :OutVariableDescription .
40
41 :ofType rdf:type owl:AsymmetricProperty ,
42 owl:FunctionalProperty ,
43 owl:IrreflexiveProperty ,
44 owl:ObjectProperty ;
45 rdfs:domain :VariableDescription .
46
47 :providedByAgent rdf:type owl:AsymmetricProperty ,
48 owl:FunctionalProperty ,
49 owl:IrreflexiveProperty ,
50 owl:ObjectProperty ;
51 rdfs:domain :ExternalComponent ;
52 rdfs:range sapl:Agent .
53
54 :provides rdf:type owl:AsymmetricProperty ,
55 owl:IrreflexiveProperty ,
56 owl:ObjectProperty ;
57 rdfs:range :Capability ;
58 rdfs:domain :Component .
59
60 :requires rdf:type owl:IrreflexiveProperty ,
61 owl:ObjectProperty ,
62 owl:TransitiveProperty ;
63 rdfs:domain :AdaptiveSoftware ;
64 rdfs:range [ rdf:type owl:Class ;
65 owl:unionOf ( :Capability
66 sen:Sensor
67 )
68 ] .
69
70 :requiresMandatorily rdf:type owl:ObjectProperty ;
71 rdfs:subPropertyOf :requires .
72
73 :requiresOptionally rdf:type owl:ObjectProperty ;
```

```
74 rdfs:subPropertyOf :requires .
75
76 :cardinalityMax rdf:type owl:DatatypeProperty ,
77 owl:FunctionalProperty ;
78 rdfs:domain :VariableDescription ;
79 rdfs:range xsd:integer .
80
81 :cardinalityMin rdf:type owl:DatatypeProperty ,
82 owl:FunctionalProperty ;
83 rdfs:domain :VariableDescription ;
84 rdfs:range xsd:integer .
85
86 :hasExternalFactor rdf:type owl:DatatypeProperty ;
87 rdfs:domain :Component ;
88 rdfs:range xsd:float .
89
90 :hasImportance rdf:type owl:DatatypeProperty ,
91 owl:FunctionalProperty ;
92 rdfs:domain :Policy ;
93 rdfs:range xsd:integer .
94
95 :hasMisjudgment rdf:type owl:DatatypeProperty ;
96 rdfs:domain :Component ;
97 rdfs:range xsd:float .
98
99 :hasPromptness rdf:type owl:DatatypeProperty ;
100 rdfs:domain :Component ;
101 rdfs:range xsd:float .
102
103 :hasQuality rdf:type owl:DatatypeProperty ;
104 rdfs:domain :Component ;
105 rdfs:range xsd:float .
106
107 :AdaptiveComponent rdf:type owl:Class ;
108 owl:equivalentClass [ rdf:type owl:Class ;
109 owl:intersectionOf ( :AdaptiveSoftware
110 :Component
111 )
112 ] .
113
114 :AdaptiveSoftware rdf:type owl:Class .
115
116 :Capability rdf:type owl:Class .
117
118 :Component rdf:type owl:Class .
119
120 :ExternalComponent rdf:type owl:Class ;
121 rdfs:subClassOf :Component ;
122 owl:disjointWith :InternalComponent .
123
124 :InVariableDescription rdf:type owl:Class ;
125 rdfs:subClassOf :VariableDescription .
126
127 :InternalComponent rdf:type owl:Class ;
```

```
128 rdfs:subClassOf :Component .
129
130 :ObligationPolicy rdf:type owl:Class ;
131 rdfs:subClassOf :Policy .
132
133 :OutVariableDescription rdf:type owl:Class ;
134 rdfs:subClassOf :VariableDescription .
135
136 :Policy rdf:type owl:Class .
137
138 :ProhibitionPolicy rdf:type owl:Class ;
139 rdfs:subClassOf :Policy .
140
141 :VariableDescription rdf:type owl:Class .
142
143 sapl:Agent rdf:type owl:Class .
144
145 owl:Ontology rdf:type owl:Class .
```

APPENDIX 2 S-APL CODE FRAGMENTS

APPENDIX 2.1 Safeguard metarules

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@prefix java: <http://www.ubiware.jyu.fi/rab#> .
@prefix p: <http://www.ubiware.jyu.fi/rab_parameters#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix x: <http://www.ubiware.jyu.fi/examples#> .
@prefix sfg: <http://www.ubiware.jyu.fi/safeguard.owl#> .

{
{
  ?sfg rdf:type sfg:ExistenceSafeguard
    ; sfg:hasCondition ?condContainer .
  sapl:I sapl:doNotBelieve {?condContainer sapl:is sapl:true} .
  sapl:I sapl:doNotBelieve {* rdf:type sfg:Ticket ;
                                  sfg:handlingSFG ?sfg} .
  ?ticket sapl:expression "ID()" .
} => {
  ?ticket rdf:type sfg:Ticket, sfg:TicketDET
     ; sfg:inState sfg:stateDET
     ; sfg:handlingSFG ?sfg
} .
} sapl:is sapl:MetaRule .

{
{
  ?sfg rdf:type sfg:NonexistenceSafeguard
    ; sfg:hasCondition ?condContainer .
  ?condContainer sapl:is sapl:true .
  sapl:I sapl:doNotBelieve {* rdf:type sfg:Ticket
                                ; sfg:handlingSFG ?sfg} .
  ?ticket sapl:expression "ID()" .
} => {
  ?ticket rdf:type sfg:Ticket, sfg:TicketDET
     ; sfg:inState sfg:stateDET
     ; sfg:handlingSFG ?sfg
} .
} sapl:is sapl:MetaRule .
```

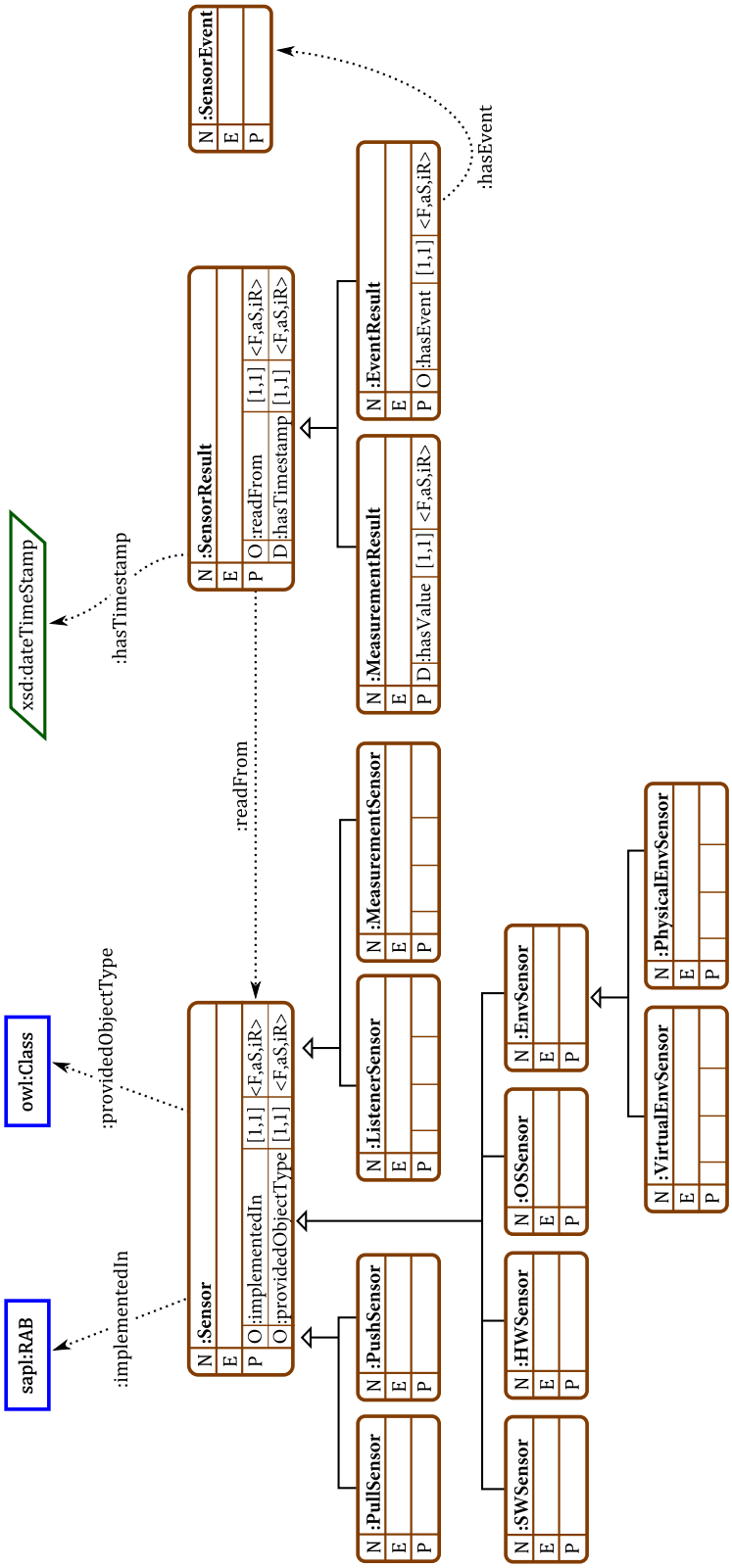
APPENDIX 2.2 Sensory data garbage collection

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@prefix java: <http://www.ubiware.jyu.fi/rab#> .
@prefix p:    <http://www.ubiware.jyu.fi/rab_parameters#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
```

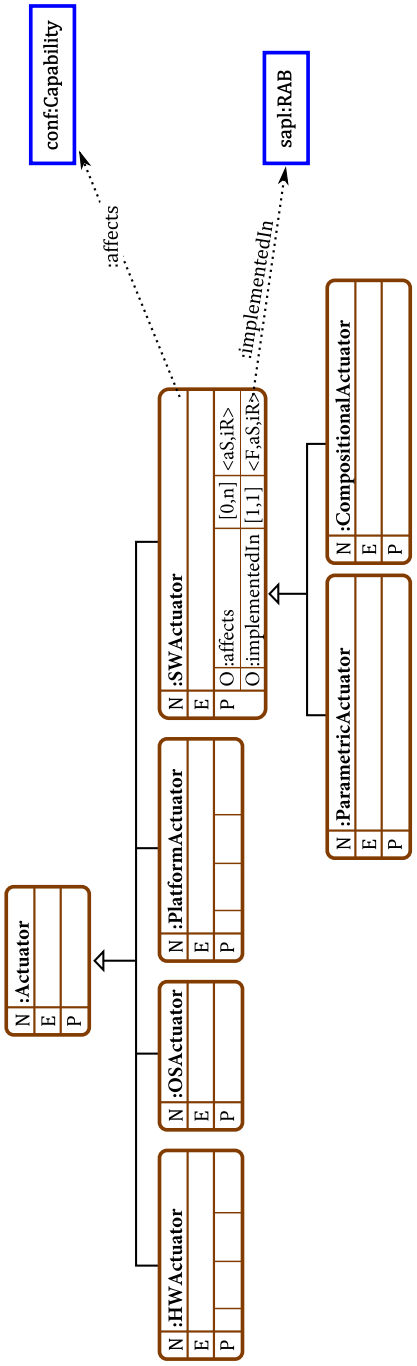
```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sen:  <http://www.ubiware.jyu.fi/sensor.owl#> .

{sapl:I sapl:do java:fi.jyu.ubiware.sensors.CPUSensor}sapl:configuredAs {
  p:resultURI sapl:is ?resURI
  sapl:Success sapl:add {
    {
      {
        ?measurement rdf:type x:CPUTempMeasurement
                     ; sen:readFrom x:sensorCPUTemp
                     ; sen:hasTimestamp ?timeStmp .
        ?count sapl:count ?measurement .
        ?oldest sapl:min ?timeStmp .
        ?count > 10 .
      } sapl:All ?measurement .
    } => {
      {
        ?remID sen:hasTimestamp ?oldest .
      } -> {
        sapl:I sapl:remove {?remID * *} .
      } .
      {sapl:I sapl:do java:ubiware.shared.PrintBehavior} sapl:configuredAs
      {p:print sapl:is "More than 10 measurements (oldest ?oldest)"} .
    } .
  }
} .
```

APPENDIX 3 ONTOLOGY VISUALIZATIONS

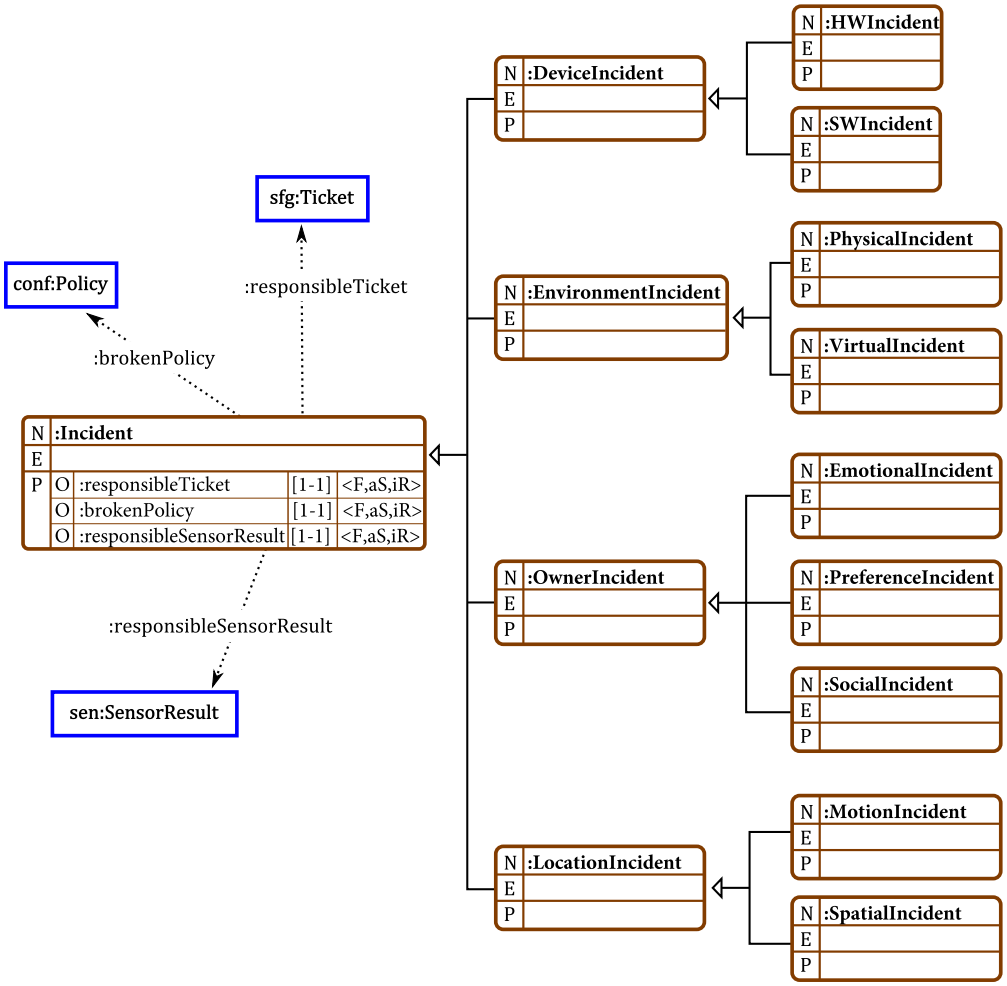
APPENDIX 3.1 Visualization of the sensor ontology



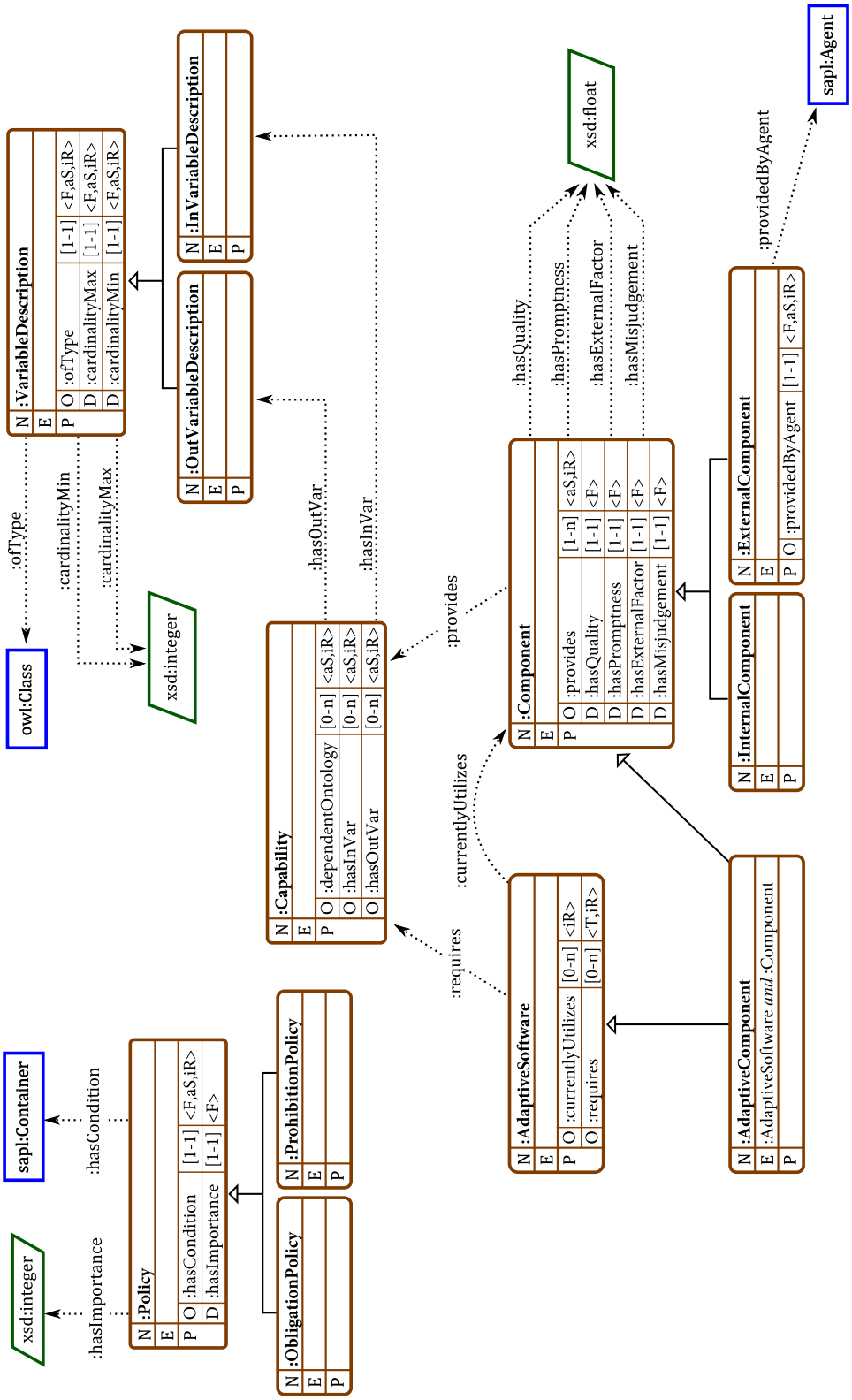
APPENDIX 3.2 Visualization of the actuator ontology



APPENDIX 3.3 Visualization of the incident ontology



APPENDIX 3.4 Visualization of the configuration ontology



APPENDIX 4 SMART HOSPITAL SCENARIO

APPENDIX 4.1 Plan 1

```
sw:plan1 rdf:type pla:Plan, pla:Action
; pla:hasPlanSequence (sw:step1 sw:step2 sw:step3)

sw:step1 rdf:type pla:Step
; pla:performsAction sw:findComp
; pla:hasVariableBinding {
   x:varCap pla:boundTo sw:capHumanDB .
   x:varComp pla:boundTo ?x .
} .

x:step2 rdf:type pla:Step
; pla:performsAction sw:initComp
; pla:hasVariableBinding {
   x:varComp pla:boundTo ?x .
} .

x:step3 rdf:type pla:Step
; pla:performsAction sw:findHuman
; pla:hasVariableBinding {
   x:varRFIDVal pla:boundTo "354186465463" .
   x:varHumanDB pla:boundTo ?x .
} .
```

APPENDIX 4.2 Room controller's configuration

```
conf:thisSW rdf:type conf:AdaptiveSoftware,
                     conf:Component, conf:AdaptiveComponent
  ; conf:provides m:RoomControl
  ; conf:requiresMandatorily m:DoorSensor
  ; conf:requiresOptionally m:HumanHandle
  ; conf:requiresOptionally m:capHumanDB
  ; conf:curentlyUtilizes m:DoorSensor .


// Finding human's indentity based on the RFID code
// var: I have an RFID code and a database component
// pre: I cannot identify a person
// eff: I found the person
m:findHuman rdf:type pla:Action
; pla:hasVariables {
  m:varRFIDVal pla:expressedAs "RFIDValue".
  m:varHumanDB pla:expressedAs "humanDB".
}
; pla:preconditionExist {
  sapl:I sw:cannotIdentifyHuman ?RFIDValue .
  conf:thisSW conf:currentlyUtilizes ?humanDB .
  ?humanDB rdf:type conf:Component
         ; conf:implements m:capHumanDB .
}
; pla:effectRemove {
  sapl:I m:cannotIdentifyHuman ?RFIDValue .
} .

// Finds a component implementing the given capability
// var: I have a capability URI
// pre: I don't have any information about a component with that capability
// eff: I have this information
```

```
m:findComp rdf:type pla:Action
; pla:hasVariables {
  m:varCap pla:expressedAs "cap".
}
; pla:preconditionNonExist {
  ?comp rdf:type conf:Component
        ; conf:implements ?cap .
}
; pla:effectAdd {
  ?comp rdf:type conf:Component
        ; conf:implements ?cap .
} .

// Initializes a component
// var: I have a component URI
// pre: I know of this component, but I am not using it
// eff: I am using the component
m:initComp rdf:type pla:Action
; pla:hasVariables {
  m:varComp pla:expressedAs "comp".
  m:varCap pla:expressedAs "cap".
}
; pla:preconditionExist {
  ?comp rdf:type conf:Component.
}
; pla:preconditionNonExist {
  conf:thisSW conf:currentlyUtilizes ?comp .
}
; pla:effectAdd {
  conf:thisSW conf:currentlyUtilizes ?comp .
} .


m:pol1 rdf:type conf:Policy, conf:ProhibitionPolicy
; conf:hasImportance "9"
; conf:hasCondition {
  ?eventA rdf:type m:DoorEvent
      ; sen:hasValue ?RFIDValue
      ; sen:hasTimestamp ?timeA
      ; sen:readFrom c2:senRFIDDoor
      ; m:source c2:door .
  sapl:I m:cannotIdentifyHuman ?RFIDValue .
} .
```

APPENDIX 4.3 Tablet's configuration

```
conf:thisSW rdf:type conf:AdaptiveSoftware,
                     conf:Component, conf:AdaptiveComponent
; conf:provides m:dataVisualization
; conf:requiresMandatorily m:bedSensor
; conf:requiresOptionally m:capPatientData
; conf:curentlyUtilizes m:bedSensor .

m:bedSensor rdf:type sen:Sensor, sen:PushSensor,
                     sen:EventSensor .

// policy - always stay connected to the patient
m:polA rdf:type conf:Policy, conf:ObligationPolicy
; conf:hasImportance "8"
; conf:hasCondition {
  ?event rdf:type m:BedEvent
       ; sen:hasValue ?bedURI .
  conf:thisSW conf:currentlyUtilizes ?compBed .
  ?compBed conf:implements m:capPatientData
```

```
19          ; m:fromSource ?bedURI .
20  } .
21
22  // Provides patient data component from bed URI
23  // var: I have a bed URI
24  // pre: I know of a patients bed
25  // eff: I am using the patient data component
26  m:attachBed rdf:type pla:Action
27  ; pla:hasVariables {
28    m:varBedURI pla:expressedAs "bedURI" .
29    m:varCompBed pla:expressedAs "compBed" .
30  }
31  ; pla:preconditionExist {
32    ?bedURI rdf:type m:PatientsBed
33  }
34  ; pla:effectAdd {
35    conf:thisSW conf:currentlyUtilizes ?compBed .
36    ?compBed conf:implements m:capPatientData
37           ; m:fromSource ?bedURI .
38  } .
```